Workload-Aware Query Recommendation Using Deep Learning

Eugenie Y. Lai
Massachusetts Institute of Technology
Cambridge, Massachusetts, USA
eylai@mit.edu

Zainab Zolaktaf
The University of British Columbia
Vancouver, British Columbia
Canada
zolaktaf@cs.ubc.ca

Mostafa Milani
The University of Western Ontario
London, Ontario, Canada
mostafa.milani@uwo.ca

Omar AlOmeir
The University of British Columbia
Vancouver, British Columbia
Canada
oomeir@cs.ubc.ca

Jianhao Cao
The University of British Columbia
Vancouver, British Columbia
Canada
jhcao@cs.ubc.ca

Rachel Pottinger
The University of British Columbia
Vancouver, British Columbia
Canada
rap@cs.ubc.ca

ABSTRACT

Users interact with databases by writing sequences of SQL queries that are are often stored in query workloads. Current SQL query recommendation approaches make little use of query workloads. Our work presents a novel workload-aware approach to query recommendation. We use deep learning prediction models trained on query pairs extracted from large-scale query workloads to build our approach. Our algorithms suggest contextual (query fragments) and structural (query templates) information to aid users in formulating their next query. We evaluate our algorithms on two real-world datasets: the Sloan Digital Sky Survey (SDSS) and SQLShare. We perform a thorough analysis of the workloads and then empirically show that our workload-aware, deep-learning approach vastly outperforms known collaborative filtering approaches.

ACM Reference Format:
Eugenie Y. Lai, Zainab Zolaktaf, Mostafa Milani, Omar AlOmeir, Jianhao Cao, and Rachel Pottinger. 2023. Workload-Aware Query Recommendation Using Deep Learning. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (EDBT '23).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

With increasing amounts of data being produced, even if the share of relational databases is falling, the total number of people relying on SQL to access their data is rising substantially. This includes non-expert users. However, writing SQL queries is tricky, especially for non-expert users, as it requires knowledge about the application domain, understanding the database schema, and SQL skills; users often lack at least one of these knowledge sets.

Query recommendation is one way to help users formulate complex SQL queries [2, 11, 21, 22]. Query recommenders predict either the entire query [1, 3, 11, 33] or parts of the query, such as table names, column names, and function names [21, 22]. Recommending an entire query is often not desired or practical for several reasons. First, there is usually limited knowledge about the user's intention. Second, suggesting an entire query may not necessarily help the user compose the next query, especially if the next query has a very complex syntax. Third, recommending an entire query requires methods that can generate queries free from both syntactic and local errors, which is very challenging.

In this paper we split the query prediction problem into two sub-problems: query template recommendation (where we recommend the structure of the next query) and query fragment recommendation (where we help users complete the predicted template). Query templates are a composite of SQL keywords (Figure 5), while query fragments are database-dependent, e.g., table and column names. This allows users to have more guided help in writing their next query than just presenting them with a fully-specified query.

Our solution is different from the existing recommenders and addresses some of their key limitations, including that current query recommenders make limited use of the preceding queries in the session. Since users often take more than one try to write a final query for a particular need, looking at pairs or even longer sequences of queries can contain valuable information in revealing user intent and making recommendations. To illustrate, we use an example from the SQLShare [19], a database-as-a-service platform where human users upload their data and write queries to interact.

Example 1. Changes in queries in a session tell a story. Figure 1 shows a session from the SQLShare workload with queries over a database about genomics experiments. The user starts by counting the number of unique experiment types (Q_1), then explores the gene and type used in each experiment (Q_2), and ends the sequence by asking the number of genes used in each type of experiment, where the gene count is greater than a threshold (Q_3). □

Most existing recommenders, including those that use collaborative filtering, ignore or make little use of the preceding query or queries. For example, QueRIE [11] represents user vectors using query sessions and compares the vectors to find users with similar behaviours. To generate user vectors, it aggregates the query vectors in a session while ignoring the query sequences. SnipSuggest [22] recommends fragments of queries as the user writes queries. While it uses the sequence of query fragments within queries, it ignores the query sequences in query sessions.

Another common limitation of existing query recommendation systems is that the existing approaches rely on predefined syntactic features to represent query statements. Such human intervention leads to inevitable information loss when representing a query's semantic meaning and user intent. For example, the QueRIE framework uses hand-picked features in its query vector representation, SnipSuggest entirely relies on predefined features consisting of SQL keywords, tables, attributes, functions,

EDBT '23, March 23–31, 2023, Ioannina, Greece

ACM ISBN 978-3-89318-088-2.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Q_1: SELECT COUNT(DISTINCT type) FROM Experiment

Q_2: SELECT gene, type FROM Experiment ORDER BY type

Q_3: SELECT type, COUNT(gene) FROM Experiment
GROUP BY type HAVING COUNT(gene) > 1

Figure 1: A sequence of queries in the SQLShare workload.

Q_4: SELECT DISTINCT obj.specclass, obj.zConf
FROM SpecObj obj, SpecLine ln
WHERE obj.specobjid = ln.specobjid
AND obj.specclass = 3

Q_5: SELECT top 10 pt.specclass, pt.z FROM SpecPhoto pt
WHERE pt.zErr NOT IN
(SELECT DISTINCT p.zErr
FROM SpecPhoto p, SpecLine ln
WHERE p.specobjid = ln.specobjid)

Q_6: SELECT top 5 obj.specclass, obj.zConf FROM SpecObj obj
WHERE obj.z NOT IN
(SELECT DISTINCT j.z FROM SpecObj j, SpecLine ln
WHERE j.specobjid = ln.specobjid)

Figure 2: Q_5 and Q_6 are structurally similar as both are nested top-k queries, but Q_4 and Q_6 are more similar since they both access SpecObj **but Q_5 queries** SpecPhoto**.**

and conditions. SQLSugg [13] is an interactive query recommendation framework that works in similar phases as our solution by recommending a template and query fragments to complete the query. However, they also entirely rely on predetermined syntactic features such as tables and attributes to generate and rank queryable templates.

We argue that query representations based on predetermined syntactic features lose critical information about the semantics and user intent. Hence recommendations based on such representations can be improved upon for two reasons. First, queries can have distinct meanings even if they have similar selected features and vice versa. Second, these query representations are oblivious to relationships between the words in queries. To demonstrate, we use an example from the Sloan Digital Sky Survey (SDSS) database [37, 38] workload, which stores images, spectra, and catalogue data for more than three million astronomical objects.

Example 2. Figure 2 shows queries Q_4 and Q_5 from two different sessions in SDSS, and the query Q_6 in the current user session. The QuerRIE framework compares queries based on fragments such as table names, and therefore finds Q_6 more similar to Q_4 rather than Q_5 as almost the same set of tables and attributes appear in Q_6 and Q_4. However, considering the structure of the queries, Q_6 is more similar to Q_5. They only differ in a pair of tables, SpecObj and SpecPhoto, which means Q_5's session may be more useful for query recommendations for the current user compared to Q_4's session. □

Example 2 shows that using predefined syntactic features, such as table name, to compare queries can result in second-rate recommendations. In this example, we also need to compare queries using their structural properties to be able to give a preferred recommendation. The example shows that it is not trivial which features summarize the syntactic properties of queries. In general, the choice of query features depends on the application and the type of queries in the workload. As such, this calls for solutions with automatic feature selection that can adapt to different workloads and queries.

We model the query recommendation problem as a next query prediction task for a given query. We split next query prediction into next template prediction and next fragment prediction. We propose a **data-driven**, **workload-aware** approach to address the concerns in the existing work. Specifically, we use deep learning prediction models to leverage the query statements in SQL query workloads, a collective knowledge exploration history of past users in the form of query sessions. In addition, our approach uses the **sequential information** by considering the preceding query.

Our approach is inspired by the use of deep learning techniques in natural language processing (NLP) [23, 26, 31, 35]. Neural networks eliminate human intervention such as feature selection since they can learn from large volumes of data at different levels [26], e.g., word, sentence, paragraph. Recently, neural networks have been used to model query statements, such as query representation learning [18, 20] and query execution prediction [55].

Specifically, we apply two deep learning techniques, sequence-to-sequence (seq2seq) architecture [15, 25, 45, 48] and fine-tuning [17, 24, 46]. Fine-tuning has been abundantly used for transfer learning in NLP and has proved its effectiveness in text summarization [32], text classification [17], and text generation [10].

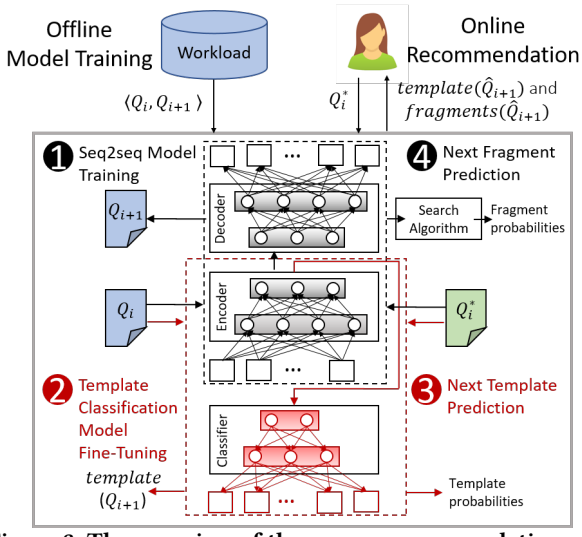

Figure 3: The overview of the query recommendation system (arrows: data flow). We train the encoder and decoder in step one and the classifier in step two. In step three we use the encoder and classifier to solve the next template prediction problem. In step four we use the encoder and the decoder to solve the next fragment prediction problem.

As shown in our overall architecture (Figure 3), there are three components in our approach, an encoder, a decoder, and a classifier. We use the three components to build two deep learning models: a seq2seq model and a template classification model. We first train the seq2seq models to extract information about query prediction from sequences of query statements. We then apply the fine-tuning technique to build the template classification model, where we reuse the encoder as a pre-trained model to fine-tune the classifier. We apply the template classification model to solving next template prediction and the seq2seq model for next fragment prediction.

There are four main contributions in our work:

- We define a new approach that guides database users to write next-step queries (Section 2).
- We adapt two deep learning techniques to solve the two sub-problems (Section 4).

```
SELECT j.target, CAST(j.estimate AS VARCHAR) AS estimate
FROM Jobs j, Status s,
    (SELECT DISTINCT target, queue FROM Servers r
     WHERE r.queue = 'FULL')
WHERE j.outputtype LIKE '%QUERY%'
```

Figure 4: Sample query Q

```
SELECT Column, Function(Column AS VARCHAR)
FROM Table, Table,
    (SELECT DISTINCT Column, Column FROM Table
     WHERE Column = Literal)
WHERE Column LIKE Literal
```

Figure 5: A recommended template statement

- We conduct a thorough analysis of two real-world datasets at the workload, query session, and pair level (Section 5).
- We compare the different models to baselines and known approaches on real-world data and empirically show the effectiveness of our approach (Section 6).

We also give model background (Section 3), discuss related work (Section 7), and conclude (Section 8).

2 PROBLEM DEFINITION

Informally, our ultimate goal is to help users compose queries. We break the query prediction problem into two sub-problems: template prediction and fragment prediction. We assume that the user has started a session by posing an initial query. We also assume that a user's goal is to write a complex query. The initial query is unlikely to be the final query, either because the user is building up the query in stages, or because the user will make errors. We help the user by providing two recommendations. First, we predict the structure of the next query—i.e., we predict the *template* that the query will use. This is the *template prediction problem*. Second, we predict what the parts are that will fill in the query template. In other words, we predict the *fragments* that will be needed to fill in the template. This is the *fragment prediction problem*.

Example 3. Assuming that the user's goal is to write a query Q in Figure 4, we recommend a template statement shown in Figure 5, and then suggest fragments, such as *Jobs*, *Status*, and *Servers* to fill Table and *target*, *estimate*, *queue*, *outputtype* to replace Column. □

Existing query recommendation solutions are based on Next Query Prediction (NQP); they either recommend the next entire query statement [1, 3, 11] or recommend some fragments of the next query [21, 22, 33]. We follow the second approach, which we believe is both more useful and more practical. Suggesting an entire query may not necessarily be useful in helping the user compose the next query, especially if the predicted next query is very complex. Practically, we often have limited knowledge about the user's intention, making it impossible to predict the next full query. Recommending an entire query requires models that can generate queries free from both syntactic and logical errors, which is a challenging task. Therefore, to make our query prediction problem tractable and simplify the recommended information for users, we split our query prediction problem into query template prediction and query fragment prediction.

To formally define the template predication and fragment prediction problems, we first present some basic concepts.

Definition 1. [Query] We represent a query statement Q as a sequence of tokens $(t_1, ..., t_{|Q|})$ with each token t_i from a vocabulary V. We consider a vocabulary that contains words from SQL queries. We define v to denote the size of V and $\mathcal{Q}$ to denote the collection of all queries over V. We use Q to refer to both a query statement and its vector representation when it is clear from the context. □

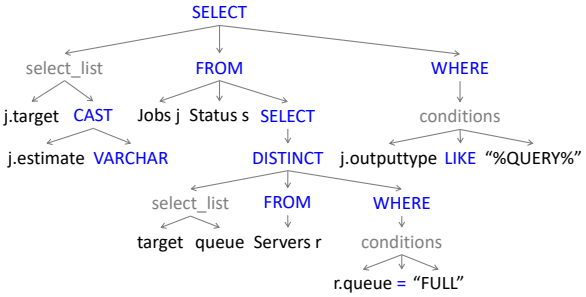

Figure 6: The AST of Q from Figure 4

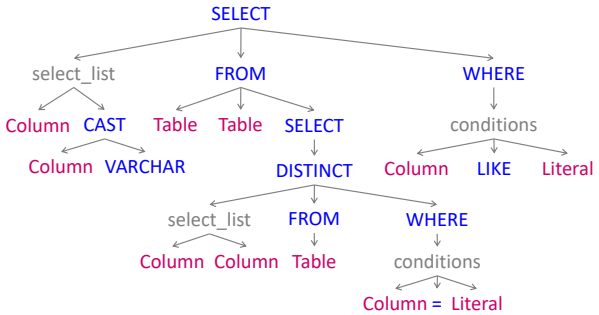

Figure 7: The template of Q from Figure 4

Example 4. Q_1 = (SELECT, ∗, FROM, PhotoTag) is the vector representation of query statement "SELECT ∗ FROM PhotoTag". □

Definition 2. [One-hot Query Representation] In the one-hot representation of a query $Q = (t_1, ..., t_{|Q|})$, we represent each token t_i by its one-hot encoding $\boldsymbol{e}_i \in \{0, 1\}^v$, i.e. a vector of bits that correspond to the tokens in V; the bit that corresponds to t_i is 1 and all the other bits are 0. □

Example 5. The one-hot representation of Q w.r.t. $V = \{t_1, t_2, t_3, t_4\}$ is $(e_1, ..., e_4)$, where e_i is the one-hot encoding of t_i, e.g., $\boldsymbol{e}_1 = [1\ 0\ 0\ 0]$ and $\boldsymbol{e}_2 = [0\ 1\ 0\ 0]$ are the one-hot encodings of t_1 = SELECT and t_2 = *. □

Definition 3. [Session, Query Pair, and Workload] A (user) session $\mathcal{S} = (Q_1, ..., Q_{|\mathcal{S}|})$ is a sequence of queries. A query pair refers to a pair $\langle Q_i, Q_{i+1} \rangle$ of consecutive queries in a session. A query workload $\mathcal{W}$ is a set of sessions $\{\mathcal{S}_1, ..., \mathcal{S}_m\}$. □

In our query recommendation setting, we assume a user poses a sequence of queries that are recorded in a session $\mathcal{S}^*$. The goal is to help the user to write the next query, Q^*_{i+1}, considering the previously posed queries, $Q^*_1, \ldots, Q^*_i$ in $\mathcal{S}^*$.

Definition 4. [Query Fragment] Given a query Q, a *query fragment* is either a table, a column, or a function in Q. We use *tables*(Q), *columns*(Q), *functions*(Q), and *literals*(Q) to respectively refer to the sets of these fragments in Q. [1] □

We represent query templates using *Abstract Syntax Trees* (ASTs) of SQL queries, which allows us to ignore some non-structural differences between query statements, e.g., indentation, spacing and order of some SQL phrases such as select conditions.

[1] Functions refer to built-in functions, e.g., AVG and SUM, and user-defined functions. Literals are fixed values.

Symbol	Description
$\mathcal{W}, \mathcal{S}, Q$	Workload, session and query
Q_i, $\langle Q_i, Q_{i+1} \rangle$	i-th query in a session, query pair
Q^*_{i+1}, $\hat{Q}_{i+1}$	Next query, next model predicted query
$tables(Q)$, $columns(Q)$	Tables and columns in query Q
$functions(Q)$, $literals Q)$	Functions and literals in query Q
$fragments(Q)$	Fragment set of Q
$template(Q)$	Template of query Q
$fragments_N$, $templates_N$	N-fragments set and N-templates set

Table 1: Notation.

Definition 5. [Query Template] The template of Q, denoted by $template(Q)$, is a tree obtained from Q by replacing query fragments, i.e., tables, columns, function names, and literals, in the abstract syntax tree (AST) of Q with placeholders, i.e., Table, Column, Function, and Literal resp., and removing aliases. □

Example 6. Figure 7 shows $template(Q)$, the template of the query Q in Figure 4. The fragments of Q are $tables(Q) = \{Jobs, Status, Servers\}$, $columns(Q) = \{target, estimate, queue, outputtype\}$, $functions(Q) = \{CAST, MIN\}$, and $literals(Q) = \{\%QUERY\%\}$. We use template statement to refer to the SQL statement obtained from a template, e.g., the statement in Figure 5 is obtained from the template in Figure 7. □

The notion of query fragment can be extended to represent combinations of terms or phrases in queries, e.g., frequent conditions in the where clauses or a frequent list of tables in the selection clauses (cf. features in SnipSuggest [22]). Our solutions can be easily adapted to such extensions. We postpone further discussion about template and fragment definition to future work.

Definition 6. [Template Prediction] Let $\mathcal{W} = \{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m\}$ be a workload over a database D, and $\mathcal{S}^* = (Q^*_1, Q^*_2, \ldots, Q^*_i)$ be the sequence of queries in the current session where Q^*_i is the last query the user posed to D. Let Q^*_{i+1} be the next query that the user will pose. The query template prediction problem is to predict $template(Q^*_{i+1})$, given $\mathcal{S}^*$ and $\mathcal{W}$. □

Our template prediction problem can be seen as a variation on classification; the classes are the existing templates in the given query workload. The classification problem is thus to assign a template (class) to a given query. The query represents the current query and the template refers to the structure of the next query. Alternatively, one could synthesize new templates not seen in the workload and recommend them to the user. We focus on the former problem because our SDSS and SQLShare workload analysis shows that most templates cover multiple queries, which means users rarely write queries with a completely new structure. We leave the alternative template prediction problem for future work.

Definition 7. [Fragment Prediction] Given a workload $\mathcal{W}$ and the current user session $\mathcal{S}^* = (Q^*_1, Q^*_2, \ldots, Q^*_i)$ over a database D, we define two versions of fragment prediction:(1) **Fragment-set prediction** predicts the sets of fragments $tables(Q^*_{i+1})$, $columns(Q^*_{i+1})$, $functions(Q^*_{i+1})$, and $literals(Q^*_{i+1})$.(2) For a user-specified positive integer N, N-**fragments prediction** is to predict N fragments from the sets of fragments $tables(Q^*_{i+1})$, $columns(Q^*_{i+1})$, $functions(Q^*_{i+1})$, and $literals(Q^*_{i+1})$. □

We defined two versions of fragment prediction because of their applications. In query recommendation, we recommend a limited number of fragments that users can easily comprehend and select from. Our workload analysis shows that queries in SDSS and SQLShare can contain hundreds of tables; predicting and presenting the user with the entire set of fragments, i.e., fragment-set prediction, is not useful. Therefore, we consider the N-fragments prediction problem which limits the number of fragments. This is simpler than fragment-set prediction; we can use the latter to predict the entire set of fragments and select N fragments to solve the former problem for applications such as database performance tuning.

Definitions 6 and 7 assume the predictions are made using all the past queries posed by the user in the current session, i.e., $\mathcal{S}^* = (Q^*_1, \ldots, Q^*_i)$. In our solution (Section 4), we only use Q^*_i, i.e., the last preceding query, to predict the next query's template and fragments. This is because our experimental results in Section 5 and our baseline prediction model in Section 6 show that the immediate successor, i.e., Q^*_i, encodes most of the necessary information for predicting Q^*_{i+1}. In Section 5, we report statistics about the similarity between Q_{i+1} and Q_i in the two workloads to highlight the importance of Q_i in predicting Q_{i+1}. Our solution using seq2seq models can be easily extended to work with all the queries $Q^*_1, \ldots, Q^*_i$: one can concatenate multiple queries to generate a single sequence and provide as input to the seq2seq models.

3 PRELIMINARIES

As described in the introduction, we rely on deep learning techniques in our architecture (Figure 3). Readers familiar with sequence-to-sequence models and fine-tuning can skip this section; we describe the novel aspects of our approach in Section 4.

Sequence-to-sequence (seq2seq) models are widely used in NLP [48]. because seq2seq models map the input sequence ($\boldsymbol{x_1}$, ..., $\boldsymbol{x_n}$) to the target sequence ($\boldsymbol{y_1}$, ..., $\boldsymbol{y_m}$). Seq2seq also has more general purposes related to sequence mapping, such as next sentence prediction [5, 39] and representation learning [18, 23]. In our problem, we model query statements as a sequence of word tokens. Seq2seq models' sequence-mapping ability well suits our purpose, where we aim to predict the next query using the preceding query.

A seq2seq model consists of an encoder and a decoder. In training, the encoder reads the input, computes hidden states $\boldsymbol{h_1}, ..., \boldsymbol{h_n}$, and outputs the context vector $\boldsymbol{c} = g(\boldsymbol{h_1}, ..., \boldsymbol{h_n})$. $\boldsymbol{c}$ is a numeric representation of the input sequence and is input to the decoder. The decoder uses $\boldsymbol{c}$ to generate an output sequence by finding $\boldsymbol{y_i}$s that maximize the joint probability. Our work assesses three types of seq2seq models. Details of the Recurrent Neural Network (RNN) seq2seq model [4] are in the full version of the paper.

Instead of sequentially processing a sequence token-by-token as in RNNs, the transformer reads tokens in parallel [48]. The transformer computes more meaningful representations by scoring tokens based on their relatedness to others [25, 48].

Convolutional seq2seq models (ConS2S) use convolutional neural networks (CNNs) in the hidden layers. CNNs are feed-forward neural networks that extract local patterns or features in data using convolving filters [15, 47]. Applying CNNs in NLP enables the seq2seq model to identify n-grams in sequences and create a semantic representation. Similar to the transformer, ConS2S is independent of the input length and allows parallel computation [15].

Fine-tuning is commonly used for transfer learning. The goal is to transfer the knowledge learned from a source task to get a head start on a related target task. E.g., BERT [9] is a NLP

language representation model. Fine-tuning trains BERT with additional datasets to add specific context [27] and/or augment it with other models for specific tasks such as text classification [17]. We explain how we apply fine-tuning in next template prediction in Section 4.1.2.

4 METHODOLOGY

We are now ready to describe the details of our query recommendation methodology (Figure 3). This section first describes the offline model training of the encoder, decoder, and classifier (Section 4.1); then we solve the next fragment prediction and next template prediction problem using online recommendation (Section 4.2).

4.1 Offline Model Training

One of our contributions is to adapt the deep learning techniques in Section 3 to next template prediction and next fragment prediction. There are two steps in this stage: step 1 uses the seq2seq models to extract information from query sequences. Step 2 applies the fine-tuning techniques using the trained encoder from step 1.

4.1.1 The Workload-Aware Approach. Step 1 leverages the sequence mapping feature of seq2seq models to extract information for query prediction using the preceding query (Figure 3). Our main contribution is the two sequence-related aspects: we first model query statements as a sequence of word tokens; we then build the seq2seq models using the preceding query and the next query as query pairs $\langle Q_i, Q_{i+1} \rangle$ (Definition 3) to capture the natural sequences in human data exploration recorded in the workloads.

Our seq2seq model has two components, an encoder and a decoder. We use input $(\boldsymbol{x_1}, ..., \boldsymbol{x_n})$ to represent a sequence of word tokens $(t_1, ..., t_n)$ in a query Q_i where each $\boldsymbol{x_i} = \boldsymbol{e_i}$ is the one-hot representation of token t_i. Similarly, we use target output $(\boldsymbol{y_1}, ..., \boldsymbol{y_m})$ to represents the next query Q_{i+1}. In step 1, Q_i is the input to the encoder, and Q_{i+1} is the target output of the decoder. Through training, the encoder learns next-query representation by becoming better at extracting features from queries that are relevant to query prediction, while the decoder becomes better at generating the next query using the given next-query representation.

Previously, seq2seq was used as an autoencoder in [18, 20] for query representation learning. Our work concerns fundamentally different problems. To solve our query prediction problem defined in Section 2, we map the preceding query Q_i and the next query Q_{i+1}. For this purpose, seq2seq models are a natural architecture due to their sequence generative and mapping features. Our approach is motivated by the differences across workloads (Section 5.3.1) and the syntactic differences between Q_i and Q_{i+1} (Section 5.3.2 and 5.3.3). We later show experimentally that this approach is a success, but the details are non-trivial due to the differences above. Next we explain how we handle the unique challenges that come with this data-driven, workload-aware query recommendation.

4.1.2 Template Classification Model Fine-Tuning. Another of our contributions is to apply fine-tuning to solve the challenges in next template prediction. Since model-generated queries can contain syntactic errors, we extract template classes from workload queries and model the template prediction problem as a classification task (Definition 6). Since the AST template (Definition 5) is a part of query statements, we consider next template prediction a related downstream task of step 1 and apply fine-tuning. To do so we leverage the learned query sequences from step 1, where the encoder learns next-query representation by becoming better at extracting features from queries that are relevant to query prediction.

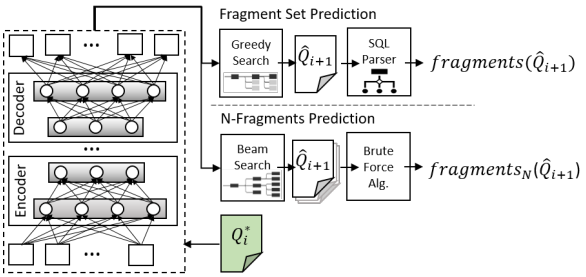

Figure 8: Step 4 next fragment prediction zoom-in.

Step 2 in Figure 3 shows the overview of how we apply fine-tuning to next template prediction. Our template classification model has two components: the trained encoder from step 1 and a classifier. We use the encoder from the seq2seq model as a pre-trained model and augment a standard two-layer classifier in NLP [17]. We take a supervised approach to train the template classification model. For every $\langle Q_i, Q_{i+1} \rangle$, we extract the AST template of Q_{i+1} and label Q_i with $template(Q_{i+1})$ (Definition 5). In training, Q_i is the input sequence, and $template(Q_{i+1})$ is the target output of the classification model. In comparison to the fine-tuning approach to next template classification, we also include a classification model without the pre-trained encoder (Section 6.2.3).

4.2 Online Recommendation

Subsequently, we apply the trained the template classification model to next template prediction (Section 4.2.1) and the seq2seq model to next fragment prediction (Section 4.2.2).

4.2.1 Next Template Prediction. Step 3 applies the trained template classification model from step 2 to next template prediction by finding AST templates that are more likely to appear in the next query statement. In online recommendation, the model takes Q_i^* as input and generates N-templates $templates_N(\hat{Q}_{i+1})$ as output. We rank the predicted $templates_N(\hat{Q}_{i+1})$ based on the probabilities. We evaluate the accuracy on top-1 template prediction as well as N-templates prediction with a rank-aware metric (Section 6.4).

4.2.2 Next Fragment Prediction. Step 4 applies the trained seq2seq model from step 1 to next query fragment prediction (Figure 8) by finding fragments that are more likely to appear in the next query statement. Same as next template prediction, the model takes Q_i^* as input. Our approach differs slightly for the fragment-set prediction and the N-fragments prediction.

Fragment-Set Prediction. In fragment-set prediction (Definition 7), we apply the common greedy decoding strategy to the seq2seq models where the decoder selects the most likely term in each step of decoding. Decoding stops when the model-generated sequence reaches the end-of-file term and returns a query statement $\hat{Q}_{i+1}$. As shown in Figure 8, the model outputs $fragments(\hat{Q}_{i+1})$, which is the predicted fragment set of the next query parsed $\hat{Q}_{i+1}$.

N-Fragments Prediction. Rather than an arbitrary number of fragments, N-fragment prediction recommends a fixed-sized list.

However, the need of decoding N fragments cannot be met with the greedy strategy. To solve this problem, we assess three types of beam search strategies in the decoder to generate multiple queries and achieve more diverse fragment recommendations.

- **Beam search:** Beam search is a decoding method that generates top-B candidate output queries with the highest score at each step; where B is known as the beam width [14, 50, 51].
- **Diverse beam search:** Vanilla beam search can result in queries that are very similar. We apply diverse beam search with the default dissimilarity setting to encourage diversity between the explored beams at each step [30, 49].
- **Stochastic decoding (sampling):** Diverse beam search may select tokens with a low score which can lead to irrelevant queries [16]. The idea of stochastic decoding is to select the tokens by sampling from the probabilities at each step to support diversity. We follow the technique in [16] but we set the probability of the tokens with a low score to zero.

These search strategies generate partial search trees in which each leaf node is an end-of-file token that marks the end of a query statement. A complete search tree is obtained by considering every word token at each decoding step. Every token in the tree has a probability attached. To obtain $fragments_N(\hat{Q}_{i+1})$ from these $\hat{Q}_{i+1}$'s, we use the probabilities of tokens in a search tree and apply a brute-force algorithm to compute a probability for the fragments to appear in the next query. We compute these probabilities as we build the search trees. When a fragment appears for the first time in a path from the roof to a leaf, its probability is the same as the token probability. If the fragment appears in multiple paths, we use the sum of the probability to appear in different paths. We then select the top N fragments with the highest probabilities to recommend.

Notice that computing the exact probability of a fragment to appear in the next query is intractable as it requires exploring the entire search tree, and applying the same brute force algorithm. The search strategies in this section generate a partial tree and as a result the probability that we compute for each fragment is an estimation of the actual probability that the fragment appears in the next query. At this point, we have described the methodology that we used to solve both the fragment prediction problem and the next template prediction problem. We are now ready to describe the workloads that we trained and tested our approach on.

5 WORKLOADS AND ANALYSIS

In order to understand our results and also show why there are reasons to believe why they are generally applicable, we extensively analyzed the workloads to provide context. This is in line with the existing applied ML literature [6, 12, 41–44] which also largely takes into account the specific domain context of the model applications.

5.1 SDSS Workload

The Sloan Digital Sky Survey (SDSS) stores images, spectra, and catalog data for more than three million astronomical objects [37]. For each SQL record, we obtained the raw **query statement** Q and **session class label** following [55]. From Q, we extracted **query template** $template(Q)$ (Definition 5) by replacing $tables(Q)$, $columns(Q)$, $functions(Q)$, and $literals(Q)$ with the corresponding tokens in the raw query statement. We obtained **query pairs** $\langle Q_i, Q_{i+1} \rangle$ (Definition 3) as follows. We first extracted the query **start time** from the "SqlLog.theTime" and **session ID** from "SessionLog.sessionID". Per session, we grouped queries by session ID and sorted queries by start time. In each $\langle Q_i, Q_{i+1} \rangle$, the two queries are from the same session and are consecutive based on the start time, where Q_i has an earlier start time than Q_{i+1}.

Statistics	SDSS	SQLShare
Total pairs	814,855	16,452
Unique pairs	187,762	15,710
Unique queries	15,094	15,792
Sessions	28,395	2,697
Datasets	1	64
Vocabulary	4,648	7,761
Tables	56	1,722
Columns	3,756	4,564
Functions	110	455
Literals	636	685
Templates	2,975	3,485

Table 2: Workload-level statistics.

5.2 SQLShare Workload

The SQLShare workload is collected from SQLShare [19], a database-as-a-service platform [19] where users upload data and write queries to interact. Besides the raw **query statement** [55], we extracted the **start time** from the "start_time" attribute and the **session ID** using SDSS's definition for query sessions [37, 38]. We also obtained the **query template** and **query pairs** as described above.

Rather than a single schema as in the SDSS workload, the SQLShare workload contains query sequences over short-term, user-uploaded datasets in various domains, including biomedical to ocean sciences. Since users upload their own datasets to SQLShare, the query sessions can operate on their individual datasets and be oblivious to the datasets used in other sessions by other users, which makes the SQLShare workload a collection of individual workloads with a variety of schemas. This difference in the SDSS and SQLShare service is reflected in the analysis (Section 5.3).

5.3 Workload Analysis

The objective of our workload analysis is to (1) provide evidence that the query pairs and the sessions in the query workloads contain useful information for our prediction tasks and (2) help explain the experiment results. We analyze the workload data at three levels, workload, session, and query pair (Sections 5.3.1–5.3.3, respectively). The findings lead to the implications in Section 5.4.

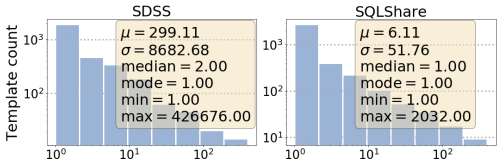

Figure 9: Number of queries covered per template.

5.3.1 Workload-Level Analysis. Overall, SDSS has 50 times more query pairs and 12 times more unique pairs than SQLShare (Table 2). The two datasets have almost the same number of unique queries while SDSS has a smaller vocabulary. SDSS has more query samples, making it easier for deep learning models to learn patterns from them [44], while SQLShare represents a harder scenario.

A key consideration for fragment prediction is the diversity of each fragment type in the data, indicated by the number of unique columns, literals, functions and tables. In SDSS, the ordering from most to least was columns, literals, functions, and tables. In contrast, SQLShare has more tables, making its ordering from most to least be columns, tables, literals, functions. Sample size and the number of appearances of tokens have a great effect on deep learning models [44]. In our setting, given the same number of query statements, tokens from a more diverse fragment type are likely to be used less often, and thus are more difficult to learn. For this reason we expect the difficulty for each type of fragment prediction in the same order. In SDSS, column prediction is expected to be the hardest, followed by literal, function, and table prediction. In SQLShare, column and table prediction is expected to be harder than function and literal.

For template prediction, SDSS and SQLShare have a similar number of unique templates, while SDSS has 50 times more data that the models can learn from. Thus SQLShare would also be a harder dataset than SDSS. Such findings align with the differences in the collection of the SDSS and SQLShare workload (Section 5.2).

5.3.2 Session-Level Analysis. Users start a session with intent. At the session level, we empirically show that users often pose a sequence of different queries to fulfill the session intent, which motivates our workload-aware approach. We grouped the workload data by session and extracted the following statistics (Figures 10 and 11 (a)–(e)) to show how much query statements vary per session: number of queries, number of unique queries, and number of sequential changes, which is the number of times a successive query differs from the one preceding it. For templates per session, we obtained the number of unique templates and number of template changes.

In SDSS and SQLShare, over 70% of the sessions have at least two unique queries. Sessions also use a variety of query templates. In SDSS, 79% of the sessions use at least two unique templates, and 64% of the sessions change templates at least twice. The number is 68% and 55% respectively for SQLShare. Overall, the session-level statistics show frequent changes in queries per session; the majority of users use a variety of queries and templates in sequence to achieve session intent. These findings provide the ground for our work to capture sequential changes in sessions.

On average, per session, SDSS has many more sequential changes than SQLShare, compared to the small difference in unique queries. Note that sequential changes indicate changes in fragments, templates, or both. These session-level query statistics align with our workload-level analysis, which indicates that SQLShare is also a harder dataset for next template prediction.

5.3.3 Pair-Level Analysis. We use query pairs $\langle Q_i, Q_{i+1} \rangle$ (Definition 3) to capture the sequential changes in query fragments and template. We analyze query syntactic properties to show how syntactically different Q_{i+1} is from Q_i. For each Q, we use the ANTLR parser to get its AST and extract six properties [55]: table count, selected columns, predicate count, predicate columns, function count, and word count. Then we show the count difference between Q_i and Q_{i+1} for each property. We use change in template to show whether $template(Q_i)$ is different than $template(Q_{i+1})$.

In SDSS (Figure 10 (g)–(l)), 8% of Q_{i+1} use more tables than Q_i, 14% select more tables and attributes, 10% use more functions, and 16% become longer. In SQLShare (Figure 11 (g)–(l)), 5% of Q_{i+1} use more tables than Q_i, 12% select more attributes, 9% use more functions, and 13% become longer. The percentage is similar for the pairs that decrease in our syntactic measurements. Overall in terms of fragment counts, Q_{i+1} is more similar to Q_i in SQLShare. However, the differences of fragment counts in Q_i and Q_{i+1} still suggest that Q_i as-is may not be a good prediction for Q_{i+1}.

For templates in the pair level (Figure 10, 11 (f)), over 40% of the Q_{i+1} has a different template than Q_i in SDSS, while the percentage goes up to 62% in SQLShare. Thus $template(Q_i)$ alone may also not be a good prediction for $template(Q_{i+1})$, and SQLShare is also a harder scenario than SDSS for template prediction.

5.4 Analysis Implications

5.4.1 Implication on Data Pre-Processing. Our SDSS workload analysis (Table 2) shows evidence of duplicate queries and query pairs within and across sessions, which means that some use of queries and patterns naturally occur more often than others. We keep all query pairs in the SDSS and SQLShare datasets. Further, we need a collection of sensible word tokens as vocabulary since our approach uses model-generated queries. For each Q, we used the ANTLR parser to get its AST. We replaced the numerical literals with a <NUM> token to control the vocabulary size. Since aliases are an instance of a table or view, they encode implicit information about the schema and intent, so we replaced aliases with the corresponding table name.

For template prediction, we assigned a template class (Definition 5) to each Q. The templates capture the query structure using SQL keywords. ASTs can distinguish the type of fragments. We replace fragments in ASTs with their corresponding tokens. We keep templates that appear at least 3 times in the datasets and get 830 template classes in SDSS and 552 in SQLShare.

5.4.2 Implication on Evaluation. The analysis also affects our baseline selection. From our workload-level statistics (Table 2), we see queries repeat within and across sessions, especially in SDSS. This means that some fragments are frequently used. For example, one or more of the 56 tables appear in the 15,094 unique SDSS queries. For templates, the long tail in Figure 9 suggests that some are more frequently used than others. Therefore we include a **baseline *popular*** that predicts the most popular N-fragments and N-templates.

From our pair-level analysis (Section 5.3.3), although we see changes in the fragment count, some of the fragments used in the next query may remain the same. Figures 10 and 11 (f) show that query pairs can share the same template as well. Specifically, over 50% of Q_{i+1} in SDSS and 40% in SQLShare use the same template as Q_i. Hence, we include a second baseline, **naïve Q_i**, that uses the fragment set of the current query for fragment-set prediction, and the template of Q_i for template prediction.

6 EXPERIMENTAL EVALUATION

6.1 Evaluation Objectives

[2]We have four questions: Does the preceding query matter?Is the data-driven approach effective? Between the deep learning architectures, the transformer and ConS2S, does one outperform the other? And do the models perform differently on the workloads? We evaluate our solution from four aspects.

(1) **Effectiveness**: We define effectiveness based on the accuracy measures (Table 4) for each sub-problem.
(2) **Deep learning model effect**: Traditional query recommendation approaches often rely on human-selected features

[2]We made our script, experiment data, and trained models publicly available.

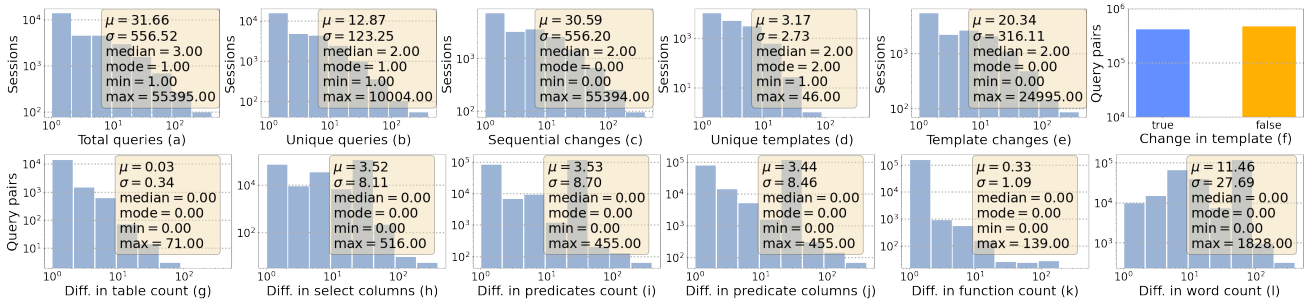

Figure 10: Session and pair level analysis for SDSS.

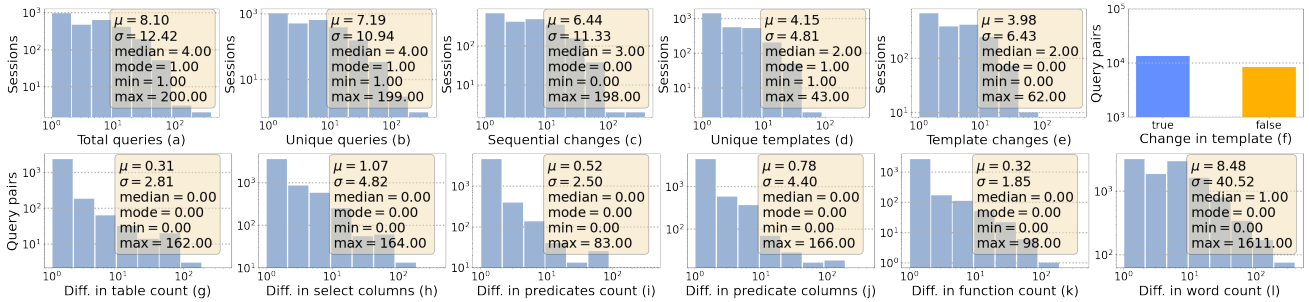

Figure 11: Session and pair level analysis for SQLShare.

such as table names. We compare against the QueRIE framework [1, 11] to evaluate the data-driven approach.

(3) **Sequence effect**: To isolate the effect of the preceding query, we build seq-less deep learning models using the same architecture. Seq-aware models are trained with a query prediction task, where given a query pair $\langle Q_i, Q_{i+1} \rangle$, we used Q_i as the model input sequence and Q_{i+1} as the target sequence. Seq-less models are trained with a reconstruction task using $\langle Q_i, Q_i \rangle$, where the input sequence is the same as the target sequence.

(4) **Fine-tuning effect**: The fine-tuning approach shows to be beneficial to the downstream tasks in NLP as discussed in Section 3. Since our problem is closely related to NLP, we evaluate whether the learned space from the trained encoder is helpful for our downstream task, next template prediction.

6.2 Setup

6.2.1 Data Split. We used the SDSS and SQLShare workload (Table 2) to evaluate the method performance for next fragment prediction and next template prediction. For both datasets, we used an (80/10/10) random split for the train, validation, and test sets.

6.2.2 Evaluation Settings. We have two settings for next fragment prediction: fragment-set and N-fragments. In fragment-set prediction, the methods are evaluated on whether they can predict all the fragments in the next query. In N-fragments prediction, we extract N fragments from multiple beams, where each beam is a model-generated query (Section 4.2.2). For next template prediction, we evaluate the methods on the top-1 and N-templates prediction.

6.2.3 Methods Compared. Our selection of methods compared is derived from the objective (2)–(4) in Section 6.1. We evaluate the effectiveness of the combination of deep learning models and query pairs in query recommendation. We have $\langle Q_i^*, Q_{i+1}^* \rangle$ in the test set, where Q_{i+1}^* is the ground truth next query of Q_i^*. In evaluation, except for the two baselines defined in Section 5.4.2, all other methods take query Q_i^* as input. We compared the following models.

- **Baseline *popular***: It uses the most popular fragments in the workload for N-fragments prediction and the most popular templates for N-templates prediction.
- **Naïve Q_i**: It uses the fragment-set in Q_i^* as-is for fragment prediction and $template(Q_i^*)$ for template prediction.
- **QueRIE framework**: We include the binary fragment-based collaborative filtering approach in the QueRIE framework [1, 11] to measure objective (2) in Section 6.1. Given Q_i^*, the framework constructs a vector based on the tables and attributes. Then it uses cosine similarity to recommend queries in the workload that are the closest to the user input. We parse the output queries to get the fragment-set and N-templates. We note that QueRIE was designed for a different query recommendation problem, and while we have adapted it as fairly as possible, it should not be expected to perform as well here as it did for its original problem.
- **Sequence-less (seq-less) models**: We include another set of models to decouple the deep learning model effect and the sequence effect in Section 6.1. The seq-less models are trained using query pair $\langle Q_i, Q_i \rangle$. The other steps remain the same.
- **Sequence-aware (seq-aware) models**: We trained the seq-aware models with a prediction task using $\langle Q_i, Q_{i+1} \rangle$.
- **Template classification model without fine-tuning**: We apply fine-tuning to solve next template prediction. To evaluate the effect of fine-tuning, we include a template classification model that uses the classifier component without the trained encoder.

We considered experimentally evaluating our work against SQLSugg [13] and SnipSuggest [22] but found this to be inappropriate.

To begin with, both SQLSugg and SnipSuggest solve different problems than the one that we solve—both help users translate their intention or requests to SQL queries. This means that they cannot propose new fragments that are not related to user input, which means that they will necessarily perform similarly to naïve Q_i, which we thoroughly experimentally analyzed. As we show in

Model		SDSS			SQLShare		
		T_{train}	T_{infer}	Size	T_{train}	T_{infer}	Size
seq-less	cons2s	13.12	0.20	8,003,869	1.05	0.81	41,337,294
	tfm	36.24	0.53	72,867,301	0.76	0.51	15,026,682
seq-aware	cons2s	12.5	0.20	8,024,036	2.12	0.83	41,547,994
	tfm	49.48	0.52	73,073,192	3.73	0.46	32,307,382

Table 3: Model statistics, where T_{train} is training time in hour and T_{infer} is inference time per query in second. We denote transformer as tfm.

Problem	Metric	Definition
Fragment prediction	Precision	$\frac{1}{\lvert R\rvert}\sum_{Q_i^*}\frac{\lvert fragments_N(\hat{Q}_{i+1})\cap fragments(Q_{i+1}^*)\rvert}{\lvert fragments_N(\hat{Q}_{i+1})\rvert}$
	Recall	$\frac{1}{\lvert R\rvert}\sum_{Q_i^*}\frac{\lvert fragments_N(\hat{Q}_{i+1})\cap fragments(Q_{i+1}^*)\rvert}{\lvert fragments(Q_{i+1}^*)\rvert}$
Template prediction	Accuracy	$\frac{1}{\lvert R\rvert}\sum_{Q_i^*}\mathbb{1}_{templates_N(\hat{Q}_{i+1})}(template(Q_{i+1}^*))$
	MRR	$\frac{1}{\lvert R\rvert}\sum_{Q_i^*}\frac{1}{rank(template(Q_{i+1}^*),templates_N(\hat{Q}_{i+1}))}$

Table 4: Evaluation metrics.

Section 6.3.1, our approach greatly outperforms naïve Q_i because it cannot suggest new fragments.

Secondly, both methods assume the availability of resources that we do not have. SQLSugg requires schema information and access to the database, neither of which we have in our datasets. SnipSuggest requires human intervention to guide the query completion. Without any intervention, which we would need to assume in order to do a fair comparison, their results would be even more limited than the existing fragments that they are able to produce.

Therefore, since both methods are going to necessarily do worse when testing against our metrics, we have tested against naïve Q_i, which provides very similar results, and they require resources that we did not have, we did not experimentally evaluate against them.

6.2.4 Model Training and Hyper-Parameter Tuning. We use the standard cross-entropy loss, which decreases as the predicted probability converges to the actual word token for a given step. We use Adam as the optimizer following [5, 9, 48].

Deep learning models need to be configured correctly. Since our workload analysis (Section 5) shows many differences in the SDSS and SQLShare datasets, we separately tuned the hyper-parameters for each dataset. Following [4, 9, 15, 25, 48, 55], we mostly preserved the model architecture and constrained the range of the training dynamics to make the tuning tractable. For the transformer models, we assessed the number of attention heads in [8, 16], hidden size in [512, 1024], and number of layers in [2, 12] as in [25, 48]. For ConS2S, we fixed the hyper-parameters as [15]. We tested the batch size in [16, 64] for both architectures. For template prediction, we also tuned the classification models and tested the hidden size in [300, 2000]. We assessed dropout in [0.0, 0.3] and learning rate in $[1e-4, 1e-6]$ for all models. The hyper-parameters are selected based on the best validation loss using early stopping.

6.2.5 Performance Metrics. Table 4 shows our evaluation metrics. For fragment prediction, we evaluate the methods in the fragment-set setting and N-fragments setting, where N is in $[1, 5]$, and R is the test dataset of pairs $\langle Q_i^*, Q_{i+1}^*\rangle$. $fragments_N(\hat{Q}_{i+1})$ is the model-predicted N-fragments in $\hat{Q}_{i+1}$'s generated using beam search strategies (Section 4.2.2). In the fragment-set setting, $fragments_N(\hat{Q}_{i+1})$ is replaced with $fragments(\hat{Q}_{i+1})$, where $\hat{Q}_{i+1}$ is the model predicted query generated by greedy decoding. We report the test F-measure of individual fragment types. Hence *fragments* in the metrics can be replaced by the four specific types of fragments. For N-templates prediction, we report the test average accuracy as a rank-less metric, where $templates_N(\hat{Q}_{i+1})$ is the list of N predicted templates. We also evaluate the methods with two rank-aware metrics, mean reciprocal rank (MRR) and normalized discounted cumulative gain (NDCG). In the definition of accuracy, $\mathbb{1}_{templates_N(\hat{Q}_{i+1})}(template(Q_{i+1}^*))$ is an indicator function that returns 1 if $template(Q_{i+1}^*)$ is in the list of templates $templates_N(\hat{Q}_{i+1})$ and 0 otherwise. In MRR, $rank(template(Q_{i+1}^*), templates_N(\hat{Q}_{i+1}))$ is a function that returns the rank of $template(Q_{i+1}^*)$ in the list of templates $rank(templates_N(\hat{Q}_{i+1})$ (it returns infinite if $template(Q_{i+1}^*)$ does not appear in the list). Due to the similarity in our results, we only include NDCG in the full version.

6.3 Next Fragment Prediction

6.3.1 Fragment-Set Prediction. In this setting, given Q_i^*, the methods output all the fragments that are likely to appear in Q_{i+1}^*. Since naïve Q_i directly outputs $fragments(Q_i^*)$, it shows the similarity in the fragment-sets in Q_i^* and Q_{i+1}^*. Hence naïve Q_i sets the bar for the fragment prediction task and specifies its difficulty. For SDSS (Table 5), both seq-aware deep learning models outperform the baselines, indicating a strong sequence effect in table, column, and function prediction. In terms of architectures, the transformer generally outperforms ConS2S in both seq-less and seq-aware cases. Specifically, seq-aware transformer performs the best in table, column, and function prediction, while seq-less transformer performs well in literal prediction. For each type of fragment-set, comparing the models to naïve Q_i, we observe the most improvement in table and function prediction. One possible reason is the diversity of word tokens in the data. From our workload statistics (Table 2), SDSS has a lower count of unique tables (56) and functions (110) than columns (3,756) and literals (636). This implies that tables and functions may occur more frequently in the dataset as a whole. More occurrences can help the deep learning models recognize functions and literals than tables and columns in SQLShare [44].

For SQLShare (Table 5), seq-less models outperform others by far, indicating a weak sequence effect. We attribute this to the nature of the dataset for two possible reasons. First, we note that naïve Q_i performs better in SQLShare than SDSS. Naïve Q_i's high performance indicates a higher similarity between $fragments(Q_i)$ and $fragments(Q_{i+1})$ in SQLShare (Section 5.3.3). Since our sequence effect is captured by $\langle Q_i, Q_{i+1}\rangle$, we observe that the sequence effect gets weaker as this similarity increases. Second, regardless of the sequential features in data, it is possible that the potential of seq-aware models is constrained by the data size, as neural networks generally benefit from larger datasets [44]. The seq-aware models may be more sensitive to the data size since more variance is introduced as they consider $\langle Q_i, Q_{i+1}\rangle$, as opposed to $\langle Q_i, Q_i\rangle$ for seq-less models. This observation can also be backed up by the superior performance of seq-aware models on SDSS, which has the same number of unique query pairs but more samples (Table 2).

Next we look at the model performance on each type of fragment set for SQLShare (Table 5). Function and literal prediction have the most improvement from naïve Q_i. Similar to SDSS, this

Method		SDSS				SQLShare			
		table	column	function	literal	table	column	function	literal
naïve Q_i		0.53	0.47	0.08	0.13	0.61	0.68	0.54	0.40
QueRIE		0.47	0.26	0.02	0.07	0.16	0.24	0.25	0.06
seq-less	cons2s	0.50	0.45	0.27	**0.18**	0.70	0.73	0.68	0.54
	tfm	0.50	0.45	0.33	**0.18**	**0.71**	**0.76**	**0.73**	**0.56**
seq-aware	cons2s	**0.65**	0.56	0.34	0.13	0.46	0.55	0.62	0.48
	tfm	**0.65**	**0.58**	**0.37**	0.14	0.64	0.66	0.68	0.54

Table 5: Fragment-set prediction f-measure.

Method		SDSS	SQLShare
baseline *popular*		0.67	0.07
naïve Q_i		0.62	0.45
seq-less	cons2s-tuned	0.72	0.48
	tfm-tuned	0.73	**0.55**
	untuned	0.73	0.52
seq-aware	cons2s-tuned	0.77	0.48
	tfm-tuned	**0.81**	0.50
	untuned	0.80	0.45

Table 6: Top-1 template prediction accuracy.

result could also be explained by the diversity of functions and literals in SQLShare. From Table 2, there are fewer unique functions (455) and literals (685) than tables (1,722) and columns (4,564) in SQLShare. As discussed in Section 5.3.1, SQLShare has more unique tables and columns since the service allows user-uploaded datasets.

6.3.2 N-Fragments Prediction. In this setting, given Q_i^*, the methods output N-fragments that are likely to appear in Q_{i+1}^*. We assess N in [1, 10] and report N in [1, 5] since the results for $N > 5$ are similar to $N = 5$. For SDSS (Figure 12), the general trend stays consistent with the fragment-set setting, where the seq-aware models vastly outperform the seq-less models and baseline *popular*. In literal prediction, all deep learning models outperform the baseline *popular*, with seq-less transformer being the best. Between the two deep learning architectures, the transformer generally outperforms ConS2S on all fragment types. The trend is the same for SQLShare (Figure 12), where seq-less models sightly outperform seq-less transformer in table, column, and function prediction, while seq-aware transformer performs the best in literal prediction. In terms of architectures, the transformer performs better than ConS2S.

Across the datasets, baseline *popular* performs drastically better on SDSS than SQLShare. This is due to the differences in the data explained in Section 5.3.1. Table 2 shows that all SDSS users share one dataset and schema, while there are 64 datasets in the SQLShare workload, where users typically upload their own datasets and have limited access to datasets uploaded by other users [19]. Hence the most popular fragments may not be accessible to all SQLShare users and thus may only appear in limited queries, which leads to low performance of baseline *popular* in SQLShare.

6.3.3 Discussion. Using the baseline naïve Q_i as an anchor, it is relatively easier for the deep learning models to improve on table and column prediction. This is due to co-occurrence data such as n-grams in NLP. In SQL query statements, tables and columns are used in a certain position to comply with the grammar, e.g., tables often appear as a group in the where clauses. Neural networks utilize this feature well [53, 54]. Information about the relative or absolute position of the word tokens in a query is crucial for the transformer [48], whereas ConS2S is sensitive to local n-gram patterns [15] (Section 3). For these reasons, tables and columns are more likely to be captured by the deep learning models.

In next fragment prediction, the deep learning models in general outperform the baselines and known existing method on both datasets, which indicates the effectiveness of our data-driven approach in predicting all the fragments of the next query.

The effectiveness of the data-driven approach on next fragment prediction varies depending on the workload, reflecting the importance of workload-aware recommendation. Further, sequence effect also varies between the workloads. In general, seq-aware models trump in SDSS, while seq-less models perform better in SQLShare. Seq-aware models' superior performance on SDSS indicates that their learning could benefit if there were more query pair samples in SQLShare.

In next fragment prediction overall, the transformer architecture outperforms ConS2S. One possible reason is that given long queries, the transformer can relate the word tokens in a query regardless of how far they are apart from each other [48]. This characteristic enables the transformer models to learn and preserve information better than ConS2S, especially when the data is more diverse and local n-gram patterns are less pronounced.

6.4 Next Template Prediction

In this section we discuss the effectiveness of the data-driven approach, sequence effect, and the fine-tuning technique. We first analyze the accuracy results of the methods on each dataset and then consider a rank-aware metric to analyze model performance in the N-templates setting. We acknowledge that the QueRIE framework does not consider SQL structure, which places it at a disadvantage.

6.4.1 Top-1 Template Prediction. Table 6 shows the template prediction accuracy of the methods. Given Q_i^*, the methods output the AST template most likely to appear in Q_{i+1}^*. Overall, all the deep learning models improve upon the baselines. Naïve Q_i again sets up a baseline to show the difficulty of the problem. SQLShare is a harder dataset due to its more diverse templates and unique queries (Table 2). For SDSS, all the seq-aware models drastically outperform their seq-less counterparts, indicating a stronger sequence effect on SDSS than SQLShare. In terms of the architecture, transformer-tuned model consistently achieves the best accuracy on both workloads. For the fine-tuning effect, the deep learning architectures obtain different results. While ConS2S-tuned model seems less effective than the untuned model, transformer-tuned model generally performs the best.

6.4.2 N-Templates Prediction. The methods output N ranked templates to predict $template(Q_{i+1}^*)$. We assess N in [1, 10] and report N in [1, 5] and include a rank-aware metric MRR, described in Section 6.2.5. For SDSS (Figure 13), seq-aware transformer-tuned model consistently outperforms other methods in both accuracy and MRR. In addition, although seq-aware transformer-tuned only slightly outperforms seq-aware ConS2S-tuned in terms of accuracy, the difference grows on the rank-aware metric, meaning that seq-aware transformer-tuned is able to output more relevant template predictions. For SQLShare (Figure 13), although seq-less transformer-tuned performs the best at N = 1, seq-aware models start to pick up when $N > 2$, indicating that the sequence effect becomes more relevant as N increases. In terms of architecture, transformer-tuned model still overall performs the best.

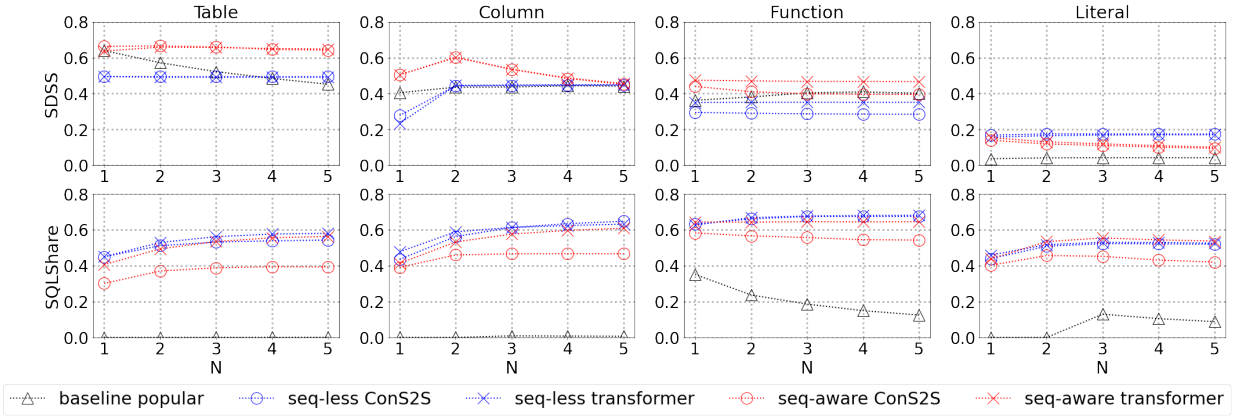

Figure 12: F-measure of N-fragments prediction on SDSS (top) and SQLShare (bottom).

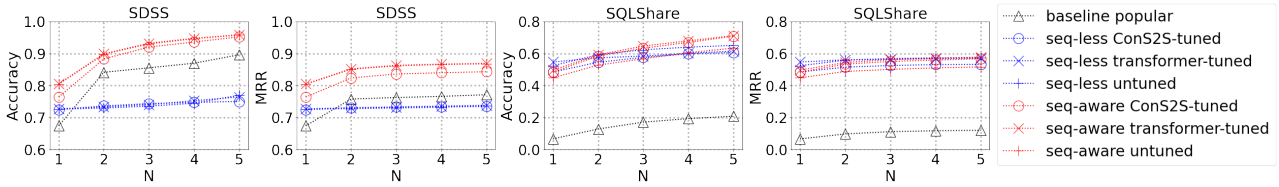

Figure 13: N-templates prediction.

6.4.3 Discussion. For SDSS, the combination of sequence effect and the data-driven approach drastically outperforms the baselines and others. For SQLShare, the seq-aware approach becomes particularly relevant when $N > 2$. These results highlight the importance of the preceding query in next template prediction, especially when the user asks for more than one template recommendation.

For next template prediction overall, seq-aware transformer-tuned achieves the best performance on both rank-less and rank-aware metrics. Though the difference in performance between the transformer-tuned and ConS2S-tuned template classification model depends on the dataset.

In terms of the fine-tuning effect, regardless of sequence-awareness, the transformer architecture can be more effectively tuned than ConS2S for next template prediction. In summary, our results show the effectiveness of the combined use of the data-driven deep learning models and sequential information from the preceding query. The transformer achieves the best performance in both query fragment prediction and query template prediction. The transformer is also shown to work well with the fine-tuning technique.

7 RELATED WORK

7.1 Facilitating Query Composition

Query recommendation is one way to help users write queries. There are many approaches. SnipSuggest [22] models each query as a vertex and uses query popularity to build a directed acyclic graph (DAG); SQLSugg [13] generates undirected graphs as queryable templates; QueRIE [11] uses collaborative filtering to make recommendations based on summarized query sessions. These existing methods use selected features, e.g., tables and attributes, to model query statements, and ignore query sequences in sessions.

Aligon et al. [2] use collaborative filtering to recommend sets of queries as OLAP sessions based on previous sessions. They use whole sessions and full sequences of queries instead of individual queries. Sessions are extracted from workloads, ranked based on similarity, and fitted to create a session that most resembles the user's future steps in the current session. The framework in [34] recommends next steps in a data analysis context, where sessions are modelled as trees. Similar past sessions are compared using a tree edit distance metric. Abstract generalized actions are recommended instead of concrete actions, this is similar in concept to the templates we recommend. These two works are similar in goals but used in different contexts, with different inputs and outputs.

Data-aware query recommendation applies several query similarity measures, including data dependant measures that consider query answers and accessed data by the queries, to find relevant sessions and queries in a workload [3]. This work is similar to QueRIE and relies on pre-determined similarity measures. The most similar work to ours is [33] where the authors use q-learning, i.e., an algorithm based on reinforcement learning, for next query prediction. They apply RNNs and compare these learning algorithms. The main difference with our work is the treatment of next query and the characterization of the next query prediction. While we divide next query prediction into template prediction and fragment prediction, they define the problem as predicting the entire query.

Recent work has been using deep learning techniques for improvements in facilitating query composition from aspects other than query recommendation. In [55] character-level and word-level CNNs and LSTMs are used to predict query performance to guide users to write more efficient queries, while [28] shows query tree visualization to help users understand SQL fragments to formulate queries. While these solutions apply similar techniques to ours, they facilitate query composition from different perspectives.

Other related work, such as [29], addresses a related problem where the intent is to help users, without knowledge of SQL, formulate SQL queries. Users specify the output table as well as the input database instance (or use an existing one). The approach is to interactively show users example results and database instances that can lead them to the correct query. Such work makes a great step towards solving the related problem without using query logs at all. However, the supported class of queries is much

smaller, and the user interaction is more complex. For those reasons, it is very difficult to compare against our query log based methods.

7.2 Sequence Representation Learning

Neural networks are effective tools for learning the underlying explanatory factors and useful representation from data for sequence modelling tasks such as machine translation and natural language understanding [15, 17, 45, 48]. Using neural networks for representation learning has two main advantages. First, it reduces the cost of feature engineering and enables automatic feature learning from raw data in supervised or unsupervised approaches. Second, it allows models to work with heterogeneous data as they do not rely on hand-picked features. These advantages are important in our query recommendation problem. Feature engineering for queries is a challenging task and is not always applicable to different query workloads. [18, 20] employ LSTM autoencoder to express queries in a standardized form for query composition tasks such as error prediction and query similarity comparison. Our work differentiates as discussed in Section 4. We use seq2seq models to capture query sequences for effective query recommendation.

7.3 Deep Learning in Recommender Systems

Recommender systems estimate users' preferences and interests to recommend them favourable items. These systems apply three types of strategies [40]. Content-based systems use items' properties and users' information to match items and users. In collaborative filtering, the recommendations are made by learning from the past user-item interactions, e.g. the history of visited or liked items. Hybrid systems apply a combination of the two strategies.

Recently, plenty of research has been done on deep learning in recommender systems. [8] applies neural networks to recommend videos on YouTube; [7] uses deep learning models to recommend applications on GooglePlay; [36] presents an RNN-based system to recommend Yahoo news (see [52] for a related survey). Deep learning has driven a revolution in recommender systems and the systems that use deep learning have shown significant improvements over traditional recommender systems. Our work is the first to use deep learning in a recommender system for SQL queries.

8 CONCLUSION AND FUTURE WORK

We introduced a workload-aware deep learning query recommendation approach that addresses two main weaknesses of the existing query recommendation systems, namely the limited use of the sequential information from the preceding query and the predefined syntactic features used to compare queries. We applied and compared two state-of-the-art seq2seq models for fragment prediction and tuned them for template prediction. We evaluated the models on two real-world workloads and show major improvements over the existing methods for query recommendation and prediction.

There are many potential extensions to our work. First is incorporating query result information to customize recommendations. Automatic feature selection in these models makes it easier to apply the models in heterogeneous settings. Another research direction is applying and assessing the models in this paper with workloads of different query languages. More complex embedding techniques can also be explored. We are also interested in the transferability of the models. One direction is to train the models on one (or multiple) workload(s) and fine-tune them on a different workload. This could help if the large-scale workload data is unavailable for some DBMSs. A closely-related successful precedent is BERT [9] in NLP. Other architectures such as hierarchical models [31] could be explored to incorporate multi-level session information.

REFERENCES

[1] J. Akbarnejad, G. Chatzopoulou, M. Eirinaki, S. Koshy, S. Mittal, D. On, N. Polyzotis, and J. S. V. Varman. SQL querie recommendations. *PVLDB*, 3(2):1597–1600, 2010.

[2] J. Aligon, E. Gallinucci, M. Golfarelli, P. Marcel, and S. Rizzi. A collaborative filtering approach for recommending OLAP sessions. *Decision Support Systems*, 69:20–30, 2015.

[3] N. Arzamasova and K. Böhm. Scalable and data-aware SQL query recommendations. *Information Systems*, 96:101646, 2021.

[4] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[5] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners, 2020.

[6] C. Chen, K. Lin, C. Rudin, Y. Shaposhnik, S. Wang, and T. Wang. An interpretable model with globally consistent explanations for credit risk, 2018.

[7] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu, and H. Shah. Wide and deep learning for recommender systems. In *DLRS*, page 7–10, 2016.

[8] P. Covington, J. Adams, and E. Sargin. Deep neural networks for youtube recommendations. In *RecSys*, page 191–198, 2016.

[9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[10] S. Edunov, A. Baevski, and M. Auli. Pre-trained language model representations for language generation. *arXiv preprint arXiv:1903.09722*, 2019.

[11] M. Eirinaki, S. Abraham, N. Polyzotis, and N. Shaikh. QueRIE: Collaborative database exploration. *TKDE*, 26(7):1778–1790, 2014.

[12] F.-L. Fan, J. Xiong, M. Li, and G. Wang. On interpretability of artificial neural networks: A survey. *IEEE Transactions on Radiation and Plasma Medical Sciences*, 5(6):741–760, 2021.

[13] J. Fan, G. Li, and L. Zhou. Interactive sql query suggestion: Making databases user-friendly. In *ICDE*, pages 351–362, 2011.

[14] M. Freitag and Y. Al-Onaizan. Beam search strategies for neural machine translation. *arXiv preprint arXiv:1702.01806*, 2017.

[15] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin. Convolutional sequence to sequence learning. *arXiv preprint arXiv:1705.03122*, 2017.

[16] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019.

[17] J. Howard and S. Ruder. Universal language model fine-tuning for text classification. In *ACL*, pages 328–339, 2018.

[18] S. Jain, B. Howe, J. Yan, and T. Cruanes. Query2vec: An evaluation of nlp techniques for generalized workload analytics. *arXiv preprint arXiv:1801.05613*, 2018.

[19] S. Jain, D. Moritz, D. Halperin, B. Howe, and E. Lazowska. SQLShare: Results from a multi-year sql-as-a-service experiment. In *SIGMOD*, page 281–293, 2016.

[20] S. Jain, J. Yan, T. Cruane, and B. Howe. Database-agnostic workload management, 2018.

[21] N. Khoussainova, M. Balazinska, W. Gatterbauer, Y. Kwon, and D. Suciu. A case for A collaborative query management system. In *CIDR*, 2009.

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

[23] R. Kiros, Y. Zhu, R. Salakhutdinov, R. S. Zemel, R. Urtasun, A. Torralba, and S. Fidler. Skip-thought vectors. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *NIPS*, pages 3294–3302, 2015.

[24] S. Kornblith, J. Shlens, and Q. V. Le. Do better imagenet models transfer better? In *CVF*, pages 2661–2671, 2019.

[25] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut. Albert: A lite bert for self-supervised learning of language representations, 2020.

[26] Q. Le and T. Mikolov. Distributed representations of sentences and documents. In *ICML*, pages 1188–1196, 2014.

[27] J. Lee, W. Yoon, S. Kim, D. Kim, S. Kim, C. H. So, and J. Kang. Biobert: a pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics*, 36(4):1234–1240, 2020.

[28] F. Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *PVLDB*, 8(1):73–84, Sept. 2014.

[29] H. Li, C. Y. Chan, and D. Maier. Query from examples: An iterative, data-driven approach to query construction. *Proceedings of the VLDB Endowment*, 8(13):2158–2169, 2015.

[30] J. Li and D. Jurafsky. Mutual information and diverse decoding improve neural machine translation. *arXiv preprint arXiv:1601.00372*, 2016.
[31] J. Li, T. Luong, and D. Jurafsky. A hierarchical neural autoencoder for paragraphs and documents. In *ACL*, pages 1106–1115, 2015.
[32] Y. Liu and M. Lapata. Text summarization with pretrained encoders. *arXiv preprint arXiv:1908.08345*, 2019.
[33] V. V. Meduri, K. Chowdhury, and M. Sarwat. Evaluation of Machine Learning Algorithms in Predicting the Next SQL Query from the Future. *TODS*, 46(1):1–46, 2021.
[34] T. Milo and A. Somech. Next-step suggestions for modern interactive data analysis platforms. In *SIGMOD*, pages 576–585, 2018.
[35] K. Pichotta and R. J. Mooney. Using sentence-level lstm language models for script inference. *arXiv preprint arXiv:1604.02993*, 2016.
[36] M. Quadrana, A. Karatzoglou, B. Hidasi, and P. Cremonesi. Personalizing session-based recommendations with hierarchical recurrent neural networks. In *RecSys*, page 130–137, 2017.
[37] M. J. Raddick, A. R. Thakar, A. S. Szalay, and R. D. C. Santos. Ten years of skyserver i: Tracking web and sql e-science usage. *Computing in Science Engineering*, 16(4):22–31, 2014.
[38] M. J. Raddick, A. R. Thakar, A. S. Szalay, and R. D. C. Santos. Ten years of skyserver ii: How astronomers and the public have embraced e-science. *Computing in Science Engineering*, 16(4):32–40, 2014.
[39] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8), 2019.
[40] F. Ricci, L. Rokach, and B. Shapira. Recommender systems: Introduction and challenges. *Recommender Systems Handbook*, page 1, 2015.
[41] C. Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1(5):206–215, 2019.
[42] C. Rudin, R. J. Passonneau, A. Radeva, H. Dutta, S. Ierome, and D. Isaac. A process for predicting manhole events in manhattan. *Machine Learning*, 80(1):1–31, 2010.
[43] C. Rudin and B. Ustun. Optimized scoring systems: Toward trust in machine learning for healthcare and criminal justice. *INFORMS Journal on Applied Analytics*, 48(5):449–466, 2018.
[44] T. J. Sejnowski. The unreasonable effectiveness of deep learning in artificial intelligence. *PNAS*, 117(48):30033–30038, 2020.
[45] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks, 2014.
[46] N. Tajbakhsh, J. Y. Shin, S. R. Gurudu, R. T. Hurst, C. B. Kendall, M. B. Gotway, and J. Liang. Convolutional neural networks for medical image analysis: Full training or fine tuning? *TMI*, 35(5):1299–1312, 2016.
[47] B. Trevett. Convolutional sequence to sequence learning, 2018.
[48] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
[49] A. K. Vijayakumar, M. Cogswell, R. R. Selvaraju, Q. Sun, S. Lee, D. Crandall, and D. Batra. Diverse beam search: Decoding diverse solutions from neural sequence models. *arXiv preprint arXiv:1610.02424*, 2016.
[50] T. Wang, D. J. Wu, A. Coates, and A. Y. Ng. End-to-end text recognition with convolutional neural networks. In *ICPR*, pages 3304–3308, 2012.
[51] S. Wiseman and A. M. Rush. Sequence-to-sequence learning as beam-search optimization. In *EMNLP*, pages 1296–1306, 2016.
[52] S. Zhang, L. Yao, A. Sun, and Y. Tay. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys*, 52(1):1–38, 2019.
[53] Z. Zhao, T. Liu, S. Li, B. Li, and X. Du. Ngram2vec: Learning improved word representations from ngram co-occurrence statistics. In *EMNLP*, pages 244–253, 2017.
[54] W. Zhu, C. Lan, J. Xing, W. Zeng, Y. Li, L. Shen, and X. Xie. Co-occurrence feature learning for skeleton based action recognition using regularized deep lstm networks. In *AAAI*, 2016.
[55] Z. Zolaktaf, M. Milani, and R. Pottinger. Facilitating sql query composition and analysis. In *SIGMOD*, 2020.