SQL Autocompletion System based on Query Logs

Jiannan Chen
University of Massachusetts Amherst
140 GOVERNORS DR
Amherst, Massachusetts MA, 01003
jiannanchen@umass.edu

Ping Lin
University of Massachusetts Amherst
140 GOVERNORS DR
Amherst, Massachusetts MA, 01003
pinglin@umass.edu

Ruisi Zhang
University of Massachusetts Amherst
140 GOVERNORS DR
Amherst, Massachusetts MA, 01003
ruisizhang@umass.edu

Abstract

This paper presents a SQL Autocompletion System based on Query Logs (SASQL system) which can provide interactive, instant, context-aware assistance in composing SQL queries. The goals of SASQL are to help non-expert database users to perform complex analysis on their datasets and to simplify the composing process for frequent users. As the user inputs partial queries, the SASQL system produces query-snippet suggestions based on past queries in the query logs. Our system borrows much from the SnipSuggest system [1] and extends it by adding a Reranking process, using the geometric method for query session detection, and building a web platform on the top of the recommendation system. By now, the capabilities of the SASQL system includes suggesting table names in the **FROM** clause, attribute names in the **SELECT** clause, and simple predicates in the **WHERE** clause. And the web-interface provides the users with interactive service for query recommendations.

For the evaluation part, we test our system on the Sloan Digital Sky Survey (SDSS) database. The experiments show that the SASQL system is able to reach up to 60% average precision with a response time below 2ms. We also show that our system outperforms the Random Suggestion algorithm and has a faster response speed than the SnipSuggest system.

Keywords SQL, query auto-completion, query session detection, reranking

ACM Reference format:
Jiannan Chen, Ping Lin, and Ruisi Zhang. 2017. SQL Autocompletion System based on Query Logs. In *Proceedings of CS645 Final Project, Amherst , MA USA, May 2017 (CS645'17),* 8 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The growing study and analysis in big data have brought great changes to the scientific, industry and commercial files. Scientists, commercial enterprises, and other organizations or persons are kept exploring data sets and finding many facts that they did not ever expect before. The new findings can help both scientific research and business to improve their work or services. The advantages that data analysis brought about attract a great many of users rushing into the data analysis field and other relative fields. Many of the new users are in need of the ability to operate the data while they have not been trained enough. The December 2016 TDWI Best Practices Report shows that there are 40% of the respondents admitted that they lack the understanding of big data analysis [2]. The lack of understanding techniques is becoming a barrier to the utilization of data analysis these days.

Databases are helpful tools for the data analysis progress. Users can extract and analyze information from the database with the help of queries. But database users should have the knowledge of the SQL language. To help non-expert users perform complex analysis on their large-scale datasets through helping them write SQL queries, as well as ease query composition for experts, we designed and developed a SQL Autocompletion System based on Query Logs (SASQL) to provide query suggestions.

SASQL system can be considered as an extension to SnipSuggest system [1]. According to user partial queries, SASQL system can recommend K most likely query snippets for a clause based on conditional probabilities of features as SnipSuggest system does. However, we independently developed a web-based system to provide interactive query recommendations with PostgresSQL, Django Python Web Framework [3] and jQuery EasyUI, as well as extending query suggestion algorithms by introducing several algorithms from other papers to improve recommendation quality and efficiency. The extended algorithms include:

- Personalized recommendation based on short-term history features [4];
- Query session detection using geometric method [5];
- Re-ranking algorithm in query recommendation.

We evaluate the quality and efficiency of our system suggestion algorithm over a small sample of query logs from Sloan Digital Sky Survey (SDSS) [6]. From the results, our SASQL system can improve the averaged accuracies by 113% compared to random suggestion algorithm, the re-ranking algorithm can further improve average accuracies by 8.9% and more than 90% suggestions can be provided under 2ms whenever we provide 3 or 10 suggestions for each query. The overall average accuracies are not so perfect is due to our highly biased data set and fixed number of suggestions in our experiments. Overall, our SASQL system can effectively provide interactive query suggestions for users.

2 Related Work

Much work has been devoted to finding ways to help non-expert users extract and analyze information from the database using SQL queries. N.Agrawal et al. propose a way using graph mining to provide auto-completion suggest that suites to all application environments [7]. The method is based on the extracting neighboring words from the graph structure. The complete graph structure is created based on the extracted keyword. So it is different from this our system's goal of providing relevant snippets from a structured SQL query.

Arnab Nandi et al provide another auto-completion suggestion using FussyTree data structure [8]. They aim to predict phrase suggestion rather than word. However, it is clear that the phrase-based suggestion is more suitable for helping the users with web-based search rather than writing database query.

Doug Downey et al believe that the study of users'searching and browsing behavior can help build user searching models [9]. They assume that the predictive user behavior models which are generated by the large-scale behavioral data can help researchers to figure out more concrete ways to offer query suggestion. Their team also use a large-scale query logs to analysis the distribution of queries and goals in Web search [10]. By checking the popularity of queries and information goals from the query logs, they find that the best search results are achieved when the frequency of the query and destination URL are similar. So they suggest that the ranking of the auto-complete suggestions should take the query frequency into consideration. Our system does not utilize the conclusion from this paper because we cannot make sure whether their findings are suitable to the SQL query suggestion. But this could be a good direction for improvements as the study in auto-completion suggestion in web-search filed sheds light on the study of SQL auto-completion analysis.

N.Khoussainova et al. propose a different approach called SnipSuggest [1]. They come up a way to develop a system which can generate a list of ranked query snippets based on query logs and complete the current partial query with the user input. The SnipSuggest system models the queries as a workload Directed Acyclic Graph (DAG) and takes partial queries as the vertices in the DAG. Our system borrows much from the SnipSuggest system such as the definition of DAG and the way to search possible query suggestions based on conditional probabilities. On the basis of these ideas, we build our own query suggestion system and it differs from the SnipSuggest system as follows. First of all, while generating suggestions, the SnipSuggest system drops obsolete queries by the Query Eliminator in order to reduce the response time. So it doesn't explicitly put different weights on short-term features and long-term features. However, our system is much faster so we choose to focus on improving the quality of generated suggestions. Milad Shokouhi has proved that short-term features extracted from queries in the same query session are effective in producing better suggestions. Thus our system gives the suggestions generated from long-term features and short-term features different weights and then rerank the merged suggestion list. Secondly, both the Query Eliminator in the SnipSuggest system and our Reranking process need to detect query sessions. The Reranking process uses the geometric method [5] instead of the machine learning methods applied in the Query Eliminator.

There are also many systems which try to provide an interface above the classical DBMS system to help the untrained users, such as visual query building tools. One example is a visual feature search engine [11] which can support queries based on visual features. These methods, however, is more like a basic SQL query tutorial tool, which is more helpful to users on constructing the simple queries but are not that helpful to construct a complex query. X.Yang et al propose another method [12] that could provide the recommendation to users by summarizing the previous methods on the relational schemas. But the recommendations are limited to the table names and attribute names. The third kind of method to help non-expert users is to develop a natural language processing interface that can transfer the natural language to SQL queries. M. Choudhary et al. propose such a system [13] that can provide

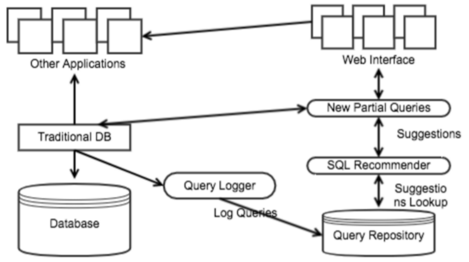


Figure 1. System Architecture

bilingual service, which means users can enter their query needs in their most familiar natural languages. Nevertheless, only a limited features are included in that system currently. The goal of our system is to provide the users with query-snippet suggestions service, and the generated recommendations should include table names in the **FROM** clause and attribute names in the **SELECT** clause as well as other kinds of snippets such as the predicates in the **WHERE** clause. With the Reranking algorithm and the web platform, users can interactively ask for query snippet suggestions as they input partial queries.

3 Methodology

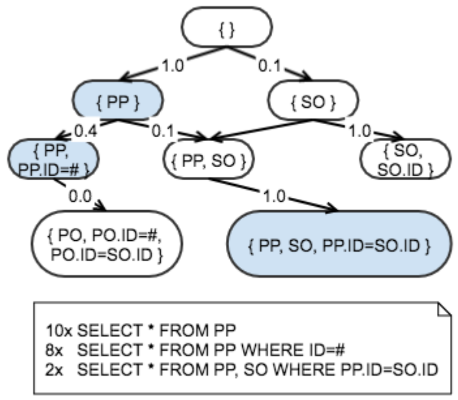
Different from the system in [1], the SASQL system comprises a web application layer as well as a middle-layer on top of a PostgresSQL database system. As shown in Figure 1, there are four main components in the system architecture, Query Logger, Query Repository, SQL Recommender and the Web Interface. The Query Logger logs the users' query inputs from other applications, then the Query Repository extracts features from the logs in the logger. The repository maintains two relations, QueryFeatures and QuerySessions. Upon a recommendation request from the Web Interface, the SQL Recommender searches snippet suggestions in the repository. When the search is finished, the suggestions are returned to the Web Interface to the users. Meanwhile, the newly generated query is stored in the query logs if it is selected by the user. In the following part, we will introduce the three components in detail.

3.1 Query Logger and Repository

Our recommender is based on SQL features, which are fragments of SQL queries. For example, the attributes in the select clause, the table names in the where clause, or the predicates in the where clause. These features are extracted from the logs of the Query Logger and then stored into the Query Repository.

When collecting features from the queries, we use Python and regular expressions [14] to do mapping and extraction. We also utilize some natural language processing toolkits such as the NLTK [15]. The extracted features are then stored in a PostgreSQL database system.

Two relations are maintained in the Query Repository, QueryFeatures and QuerySessions. QueryFeatures contains some important information about features, such as id, feature description, timestamp, query session it belongs to, and original query it belongs to and so on. The QuerySession table records which query belongs to which query session.



```
10x SELECT * FROM PP
8x  SELECT * FROM PP WHERE ID=#
2x  SELECT * FROM PP, SO WHERE PP.ID=SO.ID
```

Figure 2. Find the most popular features among queries that share *m* features with partial query *q*. *qsid* is used only when reranking is applied.

3.2 SQL Recommender

In the process of composing a SQL query, a user may want to get some suggestions about a certain clause. Our system is designed for such a scenario. When a user makes a request, the written partial query and the clause information is transferred to the recommender. And the recommender will try to find the best k snippets that satisfy the user in that clause.

To give a more intuitive explanation of how the recommender produces its recommendations, we use the idea of Workload Directed Acyclic Graph (DAG) [1]. The space of queries is viewed as a DAG shown in Figure 2. It models each query as a set of features and each vertex in the DAG represents a possible set of features (i.e. a possible query). Upon a request, the system models the input partial query into a set of features which maps to a node in the DAG. Each outgoing edge of this node represents the addition of a feature. So the problem can be translated as ranking the outgoing edges based on users'written partial queries. The system approximates the user's intended query with the set of all queries in the Query Repository that are descendants of the current vertex in the DAG, which are referred to as *potential_goals* for the partial query. With these *potential_goals*, we would rank them by their popularities among all queries in the repository.

The workflow of the SQL recommender can be described as follows: the history queries are modeled into features in the Repository, and each query are assigned with a query session id based on the Query Session Detection algorithm. When a written partial query has entered, the recommender produces two ranked suggestion lists with long-term features and short-term features respectively. With these lists, a new reranked list is generated using the Reranking algorithm. In the end, the reranked suggestion list is returned to the user. As long as the system is not terminated, the repository keeps getting new queries from the Query Logger and conducts the analysis in the first step.

The following are some important definitions for the Get Suggestion stage, we will not devote too much in introducing them as most of them are well described in [1].

3.2.1 Definitions

Definition 3.1. A **feature** f is a function that tells whether a certain property holds on a query.[1]

For example, f_{PP}^{FROM} represents if the table PP appears in the query's FROM clause.

Definition 3.2. The **feature set of a query** q [1] is defined as:

$$features(q) = f|f(q) = true \tag{1}$$

When the SASQL system produces recommendation snippets, it actually is advising the user to add the snippets which satisfy $f(q) = true$. For instance, the suggestion f_{PP}^{FROM} indicates that the recommender advises the user to add table **PP** to the **FROM** clause.

Definition 3.3. The dependencies of a feature f is the set of features that must be in the query so that it is syntactically correct when f is added. [1]

The $dependencies(f_{PP.ID=SO.ID}^{FROM}) = \{f_{PP}^{FROM}, f_{SO}^{FROM}\}$. The system only suggests a feature f for a partial query q if $dependencies(f) \subseteq features(q)$.

Definition 3.4. A feature set F_2 is a successor of a feature set F_1 if $\exists f$ where $F_2 = F_1$ and $dependencies(f) \subseteq F_1$. [1]

If a feature set F_2 can be reached from another feature set F_1 by adding a single feature, then we say F_2 is a successor of F_1.

Definition 3.5. The conditional probability [1] of a feature f given a feature set F is defined as:

$$P(F) = \frac{|\{q \in W | F \subseteq features(q)\}|}{|W|} \tag{2}$$

With these definitions, we can define the workload DAG [1]. Set F is the set of all features including those which don't appear in the workload.

Definition 3.6. The **workload DAG** $T = (V, E, \omega, \chi)$ for a query workload W is constructed as follows:

- Add vertices for all syntactically correct subset of F, which means we add a feature set as long as the query it represents is syntactically valid.
- If F_2 is a successor of F_1, we add an edge (F_1, F_2). We denote the additional feature of F_2 by **addlFeature(**(F_1, F_2)**)**$=f$, where $F_2 = F_1 \cup f$
- ω is the weight of each edge and is given by conditional probability.
- χ is the color of each vertex. $\chi(q)$ is blue if $q \in W$, otherwise it is white.

In Figure 2, we have an example workload DAG with 20 queries. The full queries correspond to the blue nodes, which means that the features the node represent are contained in the workload W. The edge (PP, PP, SO) indicates that we will have a chance of 10% to have $PP.ID = SO.ID$ in the query if we have already had PP in the query. As long as a partial query is syntactically correct, it will appear in the workload DAG because there's a vertex for every valid subset of F. The SASQL system's goal is to lead the user to the corresponding vertex given a partial query q by recommending snippets. As the system suggests one snippet at one time, the problem is equivalent to ranking the outgoing edges of q.

3.2.2 Baseline Model

Due to the lack of time, we choose the Random recommender–a most naive algorithm as our baseline model. The random recommender ranks the outgoing edges randomly and randomly picks a snippet from the repository if there're no outgoing edges. Compared to the Get Suggestion Algorithm, the Random recommender doesn't exploit the information available in the workload DAG. For example, the weight ω would tell us which features are likely to appear in the intended query given the current partial query. What's more, it is meaningless for the Random recommender to detect query sessions since it just produces suggestions randomly. We expect our system based on conditional probabilities to outperform the Random recommender.

3.2.3 Get Suggestion Algorithm (GS Algorithm)

The GS algorithm is a Context-Aware algorithm, which means it conducts searches based on the current context. Here, we borrow the definition of *potential_goals* from [1].

Definition 3.7. Given a workload DAG and a vertex q, define: $potential\_goals(q = v | v \text{ is reachable from } q)$.

In general, the *potential_goals* of q is the set of queries which have features which could possibly appear in the user's intended query. Now we can have a more precise definition of the Snippet Suggestion Problem. Given a worklaod DAG G , and a partial query q , recommend a set of k outgoing edges, $e_1, ..., e_k$, form q that maximizes:

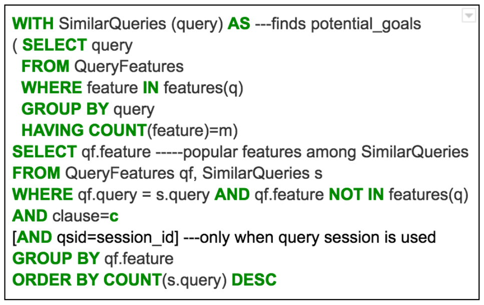
$$\sum_{i=1}^{k} P(addlFeature(e_i)|q) \qquad (3)$$

This means that our algorithm aims to give helpful features in the top k results as many as possible. To achieve this, the algorithm will select the outgoing edges with highest conditional probabilities in the DAG. We can get such features through a single SQL query as shown in 3. In the first half of this query, it tries to find queries which share the same features with the input partial query. Then the query select features with highest conditional probabilities in the specified clause from these *potential_goals*. Here a higher conditional probability is equivalent to a larger frequency. If we want to make use of query session, we need to specify which session the written partial query belongs to.

What might happen is that the input query is not in the workload DAG, which means the SimilarQueries(query) table is empty. For example, we have a partial query "SELECT PHOTOPRIMARY.G, PHOTOPRIMARY.Z FROM PHOTOPRIMARY", but we only have "SELECT PHOTOPRIMARY.G FROM PHOTOPRIMARY"in the DAG. In this case, the *potential_goals* would be $\emptyset$. To avoid that, it relaxes the conditions by set $i \leftarrow i-1$ at the end of every round in the *while* loop of Algorithm 1. In other words, the algorithm will search the upper level of the DAG if it does g n't get a match in the designated level.

3.2.4 Query Session Detection Algorithm

Before we enter this part, let's introduce the notion of query session. Wen and Zhang [16] provided the definition for query session as 'made up of the query and the subsequent activities the user performed'. More simply, we can say a query session is a sequence of queries written by the same user during a single task. The SnipSuggest system[1] used a component called Query



```
WITH SimilarQueries (query) AS ---finds potential_goals
( SELECT query
  FROM QueryFeatures
  WHERE feature IN features(q)
  GROUP BY query
  HAVING COUNT(feature)=m)
SELECT qf.feature -----popular features among SimilarQueries
FROM QueryFeatures qf, SimilarQueries s
WHERE qf.query = s.query AND qf.feature NOT IN features(q)
AND clause=c
[AND qsid=session_id] ---only when query session is used
GROUP BY qf.feature
ORDER BY COUNT(s.query) DESC
```

Figure 3. Find the most popular features among queries that share m features with partial query q. $qsid$ is used only when reranking is applied.

Algorithm 1 Get Suggestion Algorithm

```
Input: query $q$, number of suggestions $k$, clause $c$,
$session\_id$(default=None)
Output: a ranked list of snippet features
$i \leftarrow |features(q)|$
$suggestions \leftarrow []$
while $|suggestions| < k$ : do
    if $session\_id$ is not None then
        $S \leftarrow$ execute SQL above($m \leftarrow i, qsid \leftarrow session\_id$)
    else
        $S \leftarrow$ execute SQL above($m \leftarrow i$)
    for all $s \in S$ do

        if $s \notin suggestions$ and clause(s)=c then
            $suggestions \leftarrow suggstions, s$
    $i \leftarrow i - 1$
return $suggestions$
```

Eliminator to limit the size of the Query Repository and to accelerate the response process. However, the SASQL system doesn't have such problems even though we have a larger dataset according to the result in [wait for cite]. Thus we mainly focus on improving the suggestion quality instead of speeding up the system. The features generated from queries in the same session can be taken as short-term features and have been proved to be effective in improving the quality of query recommendation[4]. So our goal is to make use of query session to provide more helpful suggestions to the users. By doing so, our system would acquire advantages as following:

- The Query Repository in the SASQL system only needs to maintain two relations, while the SnipSuggest system needs to maintain three relations.
- The Query Eliminator component drop obsolete queries by executing two phases, segmentation and stitching. But our system complete similar jobs by a single step.
- In their system, labeled training data and query results are required for session detection. By comparison, our system saves us from these works because it just needs the query logs to work with.

The geometric method[17] has been proved to be a simple and relatively reliable method for query session detection. It takes two factors into account, text similarity and the elapsed time between the queries. Given two queries q and q' and their submission times

Algorithm 2 Query Session Detection Algorithm

```
1: Input: $query_1, query_2$, and their submission time $t_1, t_2$, period $p$
2: Output: if the two queries belongs to the same query session
3: $f_{cos}$ = CosineSimilarity($query_1, query_2$)
4: $t_{offset} = |t_1 - t_2|$
5: $f_{time}$ = max($0, 1 - \frac{t_{offset}}{p}$)
6: if $\sqrt{(f_{cos})^2 + (f_{time})^2} \geq 1$ then
7:     return True
8: else
9:     return False
```

t and t', the geometric method decides whether they belong to the same session as in **algorithm 2**. The cosine similarity f_{cos} and the time feature f_{time} are computed, the method votes for a session continuation iff $\sqrt{(f_{cos})^2 + (f_{time})^2} \geq 1$.

3.2.5 Reranking algorithm

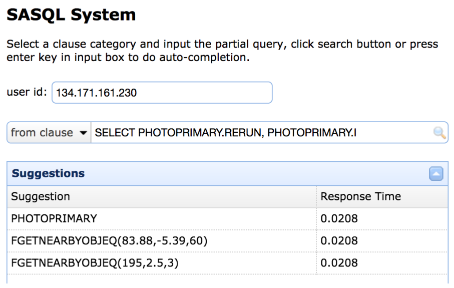
On the basis of the Get Suggestion algorithm and the Query Detection algorithm, we can generate suggestions from long-term features (i.e. not in the same query session) or from short-term features (i.e. in the same session). Then we can do reranking with these two suggestion lists. Given two suggestion lists *suggestions*$_1$ and *suggestions*$_1$ and a weight parameter α, we firstly give each suggestion in these lists a score according to their ranks. For example, in a suggestion list with a size of 3, we give the top suggestion a score of 3, the second suggestion a score of 2, and the last suggestion a score of 1. Then will generate a new reranked suggestion list with the parameter αas in **algorithm 3**. The parameter α is decided based on how important the long-term features and short-term features are in our minds.

Algorithm 3 Reranking Algorithm

```
1: Input: $suggestions_1, suggestions_2, \alpha$
2: Output: a reranked list of snippet features
3: $dictionary_1 \leftarrow []$
4: $dictionary_2 \leftarrow []$
5: $vocab \leftarrow []$
6: $length_1 \leftarrow$ length($suggestions_1$)
7: $length_2 \leftarrow$ length($suggestions_2$)
8: $all\_features \leftarrow suggestions_1 + suggestions_2$
9: for sin$suggestions_1$ do
10:     $dictionary_1[s] \leftarrow length_1$
11:     $length_1 - = 1$
12: for sin$suggestions_2$ do
13:     $dictionary_2[s] \leftarrow length_2$
14:     $length_2 - = 1$
15: for sin$all\_features$ do
16:     $vocab[s] = \alpha \cdot dictionary_1[s] + (1 - \alpha) \cdot dictionary_2[s]$
17: $reranked\_suggestions$ = sorted($vocab.values$).reverse
18: return $reranked\_suggestions$
```

3.3 Web Platform

On the top of the SQL recommender component, we build a web-based system which can provide interactive SQL suggestion service for the users. The system is developed using Django Python Web Framework[3] and jQuery EasyUI Library. The interface comprises a search-box, a text-box, and a datagrid as shown in Figure 4. The **user-id** text-box is for user's IP address, which will be replaced by automatic extraction in the release edition. To get suggestions, the user will input his partial query in the search-box, and choose the clause he wants to get suggestions about. After submission, the suggestions returned by the SQL recommender will be displayed in the **Suggestions** datagrid.



SASQL System

Select a clause category and input the partial query, click search button or press enter key in input box to do auto-completion.

user id: 134.171.161.230

from clause ▾ SELECT PHOTOPRIMARY.RERUN, PHOTOPRIMARY.I

Suggestions

Suggestion	Response Time
PHOTOPRIMARY	0.0208
FGETNEARBYOBJEQ(83.88,-5.39,60)	0.0208
FGETNEARBYOBJEQ(195,2.5,3)	0.0208

Figure 4. The Web Platform Interface

4 Dataset

The data set we used for our experiments is small sample of queries logs from Sloan Digital Sky Survey (SDSS) [6]. SDSS database logs queries submitted by real users, varying from the general public to real scientists. This small sample of queries logs consists of queries to SDSS data from Nov. 28th to 30th, 2004, with the size of 32MB and 127,461 rows. Table 1 lists the attributes of each tuple in this queries log.

Due to time constraints, we only evaluate SASQL's performance in **SELECT**, **FROM**, and **WHERE** clauses. So, we remove all queries that contain other clauses and aggregation operation. Through this data processing, the data set is reduced to around 90K rows. We use around 60K tuples as our training data and the remainder as our testing data.

Since all query logs is a full SQL query that submit by real users for execution. In order to evaluate our recommendation algorithms based on partial queries, we randomly remove some part of the full queries as partial queries for different clauses. We use a 3-fold cross-validation to evaluate SASQL's performance. Since generating partial queries and providing suggestions for each partial queries takes some time, and the original test set is highly unbalanced, we only randomly select 2000 tuples from a rebalanced test set in order to acquire better diversity as well as a reasonable experiment time.

5 Evaluation

We use two criteria to evaluate our system's performance: the quality of recommendation and the efficiency of recommendation.

5.1 Quality of recommendations

We use average precision (AP) to measure the quality of recommendation. AP is a widely-used measure to evaluate the ranking accuracy of recommendation and also used in SnipSuggest [1] system to evaluate its recommendation quality.

According to AP definition in the paper [1], average precision at k for recommendation Lq is defined as follows.

$$AP_{@k}(q, L_q) = \frac{\sum_{i=1}^{k}(P(q, L_q, i) \cdot rel(q, L_q[i]))}{|features(fullQuery(q) - features(q))|} \quad (4)$$

L_q is a ranked recommendation list provided by our recommendation system. $P(q, L_q, k)$ is the precision of the top-k recommendations, and defined as

$$P(q, L_q, k) = \frac{\sum_{i=1}^{k} rel(q, L_q[i])}{k} \quad (5)$$

$rel(q, L_q[i]) = 1$ if $L_q[i]$ is correct, and 0 otherwise. $L_q[i]$ is correct for a given partial query q is correct if and only if Lq[i] is not included in features of partial queries but in full queries. Features in full queries should account the features in a full query, not just features in a particular clause.

We then evaluate the quality of recommendation of our algorithm by comparing with the random algorithm, then evaluate recommendation quality for our algorithms before and after re-ranking algorithm.

5.1.1 Our suggestions without re-ranking algorithm vs. random suggestions

In our experiment, we always to provide 3 suggestions for each partial queries. We get $AP_{@3}$ for each partial queries, then get their average. Table 1 shows the average precision at 3 for both our suggestion algorithm without re-ranking algorithm and random suggestion algorithm.

Table 1. AP comparison between our algorithm without re-ranking and random suggestion

	Random suggestion	Our algorithm	Improvements
$AP_{@3}$	0.0787	0.1674	113%

Then, we measure the distribution of average $AP_{@3}$ over different number of features in partial queries Figure 5 presents the overall distribution of average $AP_{@3}$ for all queries including **SELECT**, **FROM**, and **WHERE** clauses. Figure 6, Figure 7 and Figure 8 show the distribution of average $AP_{@3}$ for **SELECT**, **FROM**, and **WHERE** clause respectively.

The results show that our suggestion algorithms provide much better suggestions compared to the random algorithm. However, the AP of our suggestion algorithms are still not very high, mostly below 35%, and the averaged accuracy does not strictly increase as users input more features in the partial queries. There are several reasons contributing to the problem.

- According to AP equation, the precision should be divided by $|features(fullQuery(q) - features(q))|$. In our experiment, we only test the scenario that the system gives 3 suggestions under a particular clause, not cover more than one clause, thus even all 3 suggestions are provided correctly, the average precision is not good, especially in a long query. Actually, 62% queries in our test dataset have more than 20 features, which causes low accuracies in our experiments, especially for **SELECT** and **FROM** clause. For example, if a partial query is like "select rowc_g" and the full query is "select rowc_g,colc_g from BESTDR3..PhotoPrimary where objID = 587726015076958360", we only get $AP_{@3}(q, Lq) = \frac{1}{|4-1|} = 0.33$ in the scenario of providing 3 suggestions, even our algorithm correctly suggest the rest attribute in **SELECT** clause.
- Our data set is highly biased and unbalanced. We list the top-5 features obtained from our training dataset in Table 2. Which cause our average accuracies do not increase strictly when users input more features in partial queries.

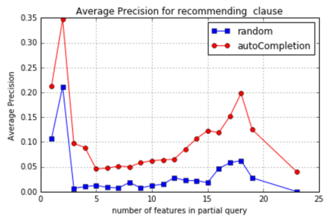


Figure 5. Overall Average Precision of Our Algorithm and Random Algorithm for All Clauses

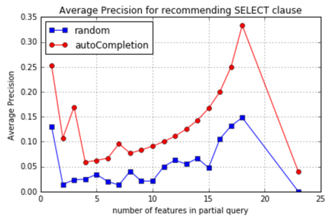


Figure 6. Average Precision of Our Algorithm and Random Algorithm for SELECT Clauses

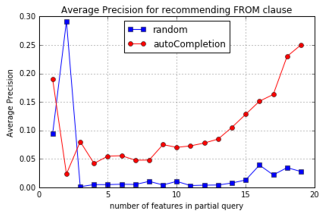


Figure 7. Average Precision of Our Algorithm and Random Algorithm for FROM Clauses

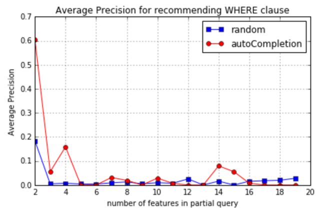


Figure 8. Average Precision of Our Algorithm and Random Algorithm for WHERE Clauses

Table 2. Top-5 features in our training data set and their frequencies

	TOP 1	TOP 2	TOP 3	TOP 4	TOP 5
SELECT Clause	TROWC_G: 56732	COLC_G: 56732	PHOTOPRI-MARY.ERR_I: 2528	PHOTOPRI-MARY.I: 2528	PHOTOPRI-MARY.ERR_G: 2528
FROM Clause	BESTDR3.. PHO-TOPRIMARY: 56732	PHOTOPRI-MARY: 2528	FGETNEARBY-OBJEQ(#,#,#): 2526	BESTDR3..FIELD: 606	DBCOLUMNS: 16
WHERE Clause	OBJID #: 56732	FGETNEARBY-OBJEQ(#,#,#). OBJID = PHO-TOPRIMARY. OBJID: 2526	RUN=#: 606	SPECOBJID=#: 38	OBJID=#: 34

5.1.2 Our suggestions with re-ranking vs. without re-ranking

We also test and compare the quality of recommendations before and after re-ranking algorithm. Because the query session detection requires the queries input to be temporally correlated, we select 6000 tuples in a specific period and take 4000 of them as the training set, 2000 of them as the test set. Table 3 show the overall average accuracies of the two techniques. Figure 9 - Figure 12 show the average precision of our suggestion algorithm before and after re-ranking for overall clause, **SELECT** clause, **FROM** clause and **WHERE** clause.

From the results, the suggestion with re-ranking technique achieves 8.9% improvement compared with the algorithm without re-ranking

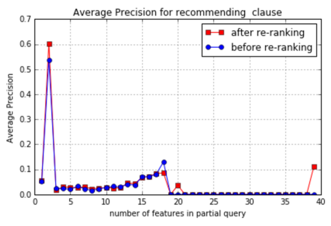


Figure 9. Average Precision with and without Re-ranking Technique for All Clauses

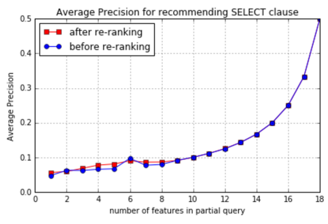


Figure 10. Average Precision with and without Re-ranking Technique for SELECT Clauses

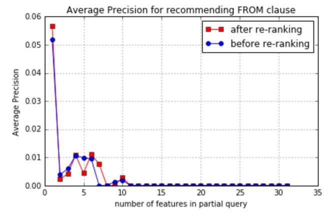


Figure 11. Average Precision with and without Re-ranking Technique for FROM Clauses

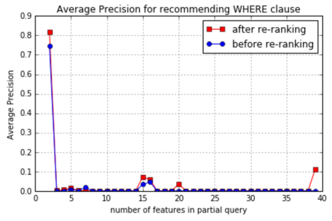


Figure 12. Average Precision with and without Re-ranking Technique for WHERE Clauses

technique. We can conclude that re-ranking technique is effective for providing suggestions. The reasons of low accuracies are the same as presented in section 5.1.1. Besides, the queries used for this experiment has removed the dominant query, most of the rest testing data are long queries, which further results in big denominators, thus the accuracies are low especially for **SELECT** and **FROM** clause in the case that we only test for providing 3 suggestions.

Table 3. Average precision of our algorithm before and after reranking

	Before Reranking	After Reranking	Improvements
$AP_{@3}$	0.1067	0.1161	8.9%

5.2 Efficiency of Recommendations

We test response time for providing suggestions as an indicator of the efficiency of recommendations.

Figure 13 shows the mean of response time for our suggestion algorithm without re-ranking technique and random suggestion algorithm over different number of features in partial queries. Figure 14 is the box plot of response time. From the results, the response time of our system is only a little bit larger than that of random algorithm, and 94.2% recommendations except some outliers are given under 2ms, largely lower than 100ms. The research [18] shows that the response time up to 100ms is considered interactive. Therefore, our system can provide interactive suggestions.

To evaluate the efficiency of our algorithm, we also test the response time when providing 10 suggestions (k = 10). Figure 15 and Figure 16 respectively show the mean and box distribution of response time for our algorithm and random algorithm. From the results, we can observe that the response time is not impacted much when we provide more suggestions. This response time of

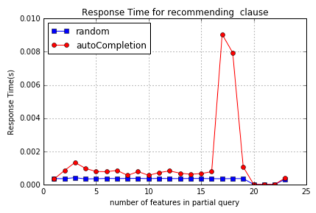


Figure 13. The Mean of Response Time (Our Algorithm vs. Random Algorithm)

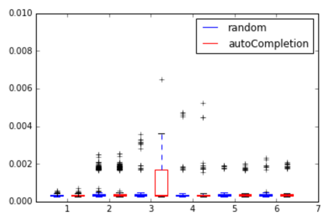


Figure 14. The Box Distribution of Response Time (Our Algorithm vs. Random Algorithm)

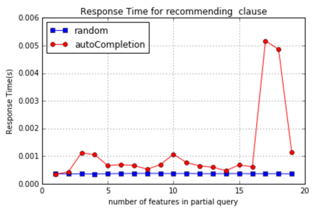


Figure 15. The Mean of Response Time When Providing 10 Suggestions

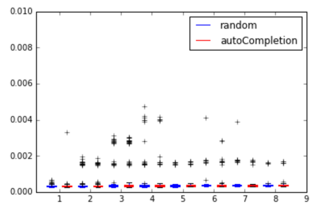


Figure 16. The Box Distribution of Response Time When Providing 10 Suggestions

our algorithm (around 2ms) is much better than the experimental results in SnipSuggest [1] system which response time is around 100ms.

We also measure the response time of our algorithm in two scenarios: with re-ranking and without re-ranking algorithm. Table 4 shows the response time of these two technique of our algorithm. We can conclude that re-ranking has little impact to the response time of recommendation, but can significantly improve the accuracies.

Table 4. Response time for our algorithms with and without reranking

	Before Reranking	After Reranking	Increased Rate
Mean of response time	0.000805	0.000806	0.17%
Percentage of response time below 2ms	97%	97%	

6 Conclusion

In this paper, we present the SASQL, a context-ware SQL query auto-completion system with a web interface. Its main goals are to assist non-expert users in writing SQL queries and help frequent users to simplify the composition process when they try to perform complex analysis on their datasets, and it achieves that by suggesting snippets based on short-term and long-term query histories. The SASQL system extends the SnipSuggest system by introducing a Reranking process, applying another query-session detection method, and adding a web interface. We have shown that our system has the capabilities of producing relatively good recommendations with a very short response time.

For furture work, we will firstly re-run our experiments on a larger and more diverse dataset and meanwhile pay more attention to the preprocessing of SQL queries. Secondly, the SASQL system only has the ability to produce suggestions about table names, attribute names, and predicates. We will improve the system by enabling it to generate more types of recommendations. What's more, we will also compare our Reranking algorithm with the Query Eliminator algorithm in the SnipSuggest system in terms of suggestion quality. Besides, we will try to improve the query session methods for our system. Lastly, the current system is completely based on query logs, but we believe it will expand the system's application scope if we integrate the recommendation system with traditional DBMS where the real data resides such as the SDSS dataset.

References

[1] Nodira Khoussainova, YongChul Kwon, Magdalena Balazinska, and Dan Suciu. Snipsuggest: Context-aware autocompletion for sql. *Proceedings of the VLDB Endowment*, 4(1):22–33, 2010.

[2] Fern Halper. Data science and big data âĂŞ enterprise paths to success, 2017.

[3] Django Software Foundation. Django (version 1.5) [computer software], 2013.

[4] Milad Shokouhi. Learning to personalize query auto-completion. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*, pages 103–112. ACM, 2013.

[5] Matthias Hagen, Benno Stein, and Tino Rüb. Query session detection as a cascade. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 147–152. ACM, 2011.

[6] Franco D Albareti, Carlos Allende Prieto, Andres Almeida, Friedrich Anders, Scott Anderson, Brett H Andrews, Alfonso Aragon-Salamanca, Maria Argudo-Fernandez, Eric Armengaud, Eric Aubourg, et al. The thirteenth data release of the sloan digital sky survey: First spectroscopic data from the sdss-iv survey mapping nearby galaxies at apache point observatory. *arXiv preprint arXiv:1608.02013*, 2016.

[7] Neeraj Agrawal and Mrutyunjaya Swain. Auto complete using graph mining: A different approach. In *Southeastcon, 2011 Proceedings of IEEE*, pages 268–271. IEEE, 2011.

[8] Arnab Nandi and HV Jagadish. Effective phrase prediction. In *Proceedings of the 33rd international conference on Very large data bases*, pages 219–230. VLDB Endowment, 2007.

[9] Doug Downey, Susan T Dumais, and Eric Horvitz. Models of searching and browsing: Languages, studies, and application. In *IJCAI*, volume 7, pages 2740–2747, 2007.

[10] Doug Downey, Susan Dumais, Dan Liebling, and Eric Horvitz. Understanding the relationship between searchers' queries and information goals. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 449–458. ACM, 2008.

[11] John R Smith and Shih-fu Chang. Querying by color regions using the visualseek content-based visual query system. *Intelligent multimedia information retrieval*, 7(3):23–41, 1997.

[12] Xiaoyan Yang, Cecilia M Procopiuc, and Divesh Srivastava. Summarizing relational databases. *Proceedings of the VLDB Endowment*, 2(1):634–645, 2009.

[13] Munisha Choudhary, Mohit Dua, and Zorawar S Virk. A web-based bilingual natural language interface to database. In *Image Information Processing (ICIIP), 2015 Third International Conference on*, pages 433–438. IEEE, 2015.

[14] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009.

[15] Steven Bird. Nltk: the natural language toolkit. In *Proceedings of the COLING/ACL on Interactive presentation sessions*, pages 69–72. Association for Computational Linguistics, 2006.

[16] Ji-Rong Wen, HongJiang Zhang, et al. Query clustering in the web context., 2003.

[17] Daniel Gayo-Avello. A survey on session detection methods in query logs and a proposal for future evaluation. *Information Sciences*, 179(12):1822–1843, 2009.

[18] Stuart K Card, George G Robertson, and Jock D Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human factors in computing systems*, pages 181–186. ACM, 1991.