SnipSuggest: Context-Aware Autocompletion for SQL

Nodira Khoussainova, YongChul Kwon, Magdalena Balazinska, and Dan Suciu
Department of Computer Science and Engineering, University of Washington, Seattle, WA, USA
{nodira, yongchul, magda, suciu}@cs.washington.edu

ABSTRACT

In this paper, we present SnipSuggest, a system that provides on-the-go, context-aware assistance in the SQL composition process. SnipSuggest aims to help the increasing population of non-expert database users, who need to perform complex analysis on their large-scale datasets, but have difficulty writing SQL queries. As a user types a query, SnipSuggest recommends possible additions to various clauses in the query using relevant snippets collected from a log of past queries. SnipSuggest's current capabilities include suggesting tables, views, and table-valued functions in the `FROM` clause, columns in the `SELECT` clause, predicates in the `WHERE` clause, columns in the `GROUP BY` clause, aggregates, and some support for sub-queries. SnipSuggest adjusts its recommendations according to the context: as the user writes more of the query, it is able to provide more accurate suggestions.

We evaluate SnipSuggest over two query logs: one from an undergraduate database class and another from the Sloan Digital Sky Survey database. We show that SnipSuggest is able to recommend useful snippets with up to 93.7% average precision, at interactive speed. We also show that SnipSuggest outperforms naïve approaches, such as recommending popular snippets.

1. INTRODUCTION

Recent advances in technology, especially in sensing and high-performance computing, are fundamentally changing many fields of science. By increasing the amount of data collected by orders of magnitude, a new method for scientific discovery is emerging: testing hypotheses by *evaluating queries over massive datasets*. In this setting, spreadsheets and hand-written scripts are insufficient. Consequently, despite the belief still held by many database researchers, scientists today are increasingly using databases and SQL to meet their new data analysis needs [13, 23, 24].

The Sloan Digital Sky Survey (SDSS) [22] is a famous example of this shift toward data-intensive scientific analytics and SQL. SDSS has mapped 25% of the sky, collecting over 30 TB of data (images and catalogs), on about 350 million celestial objects [22]. SDSS has had a transformative effect in Astronomy not only due to the value of its data, but because it made that data accessible via SQL [1]. To date, astronomers and others have submitted over 20.7 million SQL queries to the SDSS database; most queries are submitted through web forms but many forms enable users to modify the submitted queries or even author queries from scratch [23].

There are many more examples of SQL use in science. The SQLShare system [14], developed by the eScience institute [25] at the University of Washington (UW), allows scientists to upload their data (e.g., in Excel files), and immediately query it using SQL. Our own small survey of UW scientists revealed that respondents across the sciences are using SQL on a weekly or even daily basis.

SQL is thus having a transformative effect on science. Authoring SQL queries, however, remains a challenge for the vast majority of scientists. Scientists are highly-trained professionals, and can easily grasp the basic select-from-where paradigm; but to conduct advanced scientific research, they need to use advanced query features, including group-by's , outer-joins, user defined functions, functions returning tables, or spatial database operators, which are critical in formulating their complex queries. At the same time, they have to cope with complex database schemas. For example the SDSS schema has 88 tables, 51 views, 204 user-defined functions, and 3440 columns [22]. One of the most commonly used views, PhotoPrimary, has 454 attributes! The learning curve to becoming an expert SQL user on a specific scientific database is steep.

As a result, many scientists today leverage database management systems (DBMSs) only with the help of computer scientists. Alternatively, they compose their SQL queries by sharing and reusing sample queries: the SDSS website provides a selection of 57 sample queries, corresponding to popular questions posed by its users [1]. Similarly, SQLShare provides a "starter kit" of SQL queries, translated from English questions provided by researchers. *Scientists who write complex SQL today do this through cut and paste.* The challenge with sample SQL queries is that users either have access to a small sample, which may not contain the information that they need, or they must search through massive logs of past queries (if available), which can be overwhelming.

Assisting users in formulating complex SQL queries is difficult. Several commercial products include visual query building tools [4, 7, 15, 19], but these are mostly targeted to novice users who struggle with the basic select-from-where paradigm, and are not used by scientists. More recent work [26] has proposed a method for clustering and ranking relations in a complex schema by their importance. This can be used by an automated tool to recommend tables and attributes to SQL users, but it is limited only to the most important tables/attributes. Scientists are experts in their domain, they learn quickly the most important tables. Where they truly need help are precisely the "advanced" features, the rarely-used tables and attributes, the complex domain-specific predicates, etc. Some new systems, such as QueRIE [8], recommend entire queries authored by other users with similar query patterns. These, however, are designed for users who have already written multiple queries, and wish to see past queries that touch a similar set of tuples.

In this paper, we take a radically different approach to the problem. We introduce SnipSuggest, a new SQL autocomplete system, which works as follows. As a user types a query, she can ask SnipSuggest for recommendations of what to add to a specific clause of her query. In response, SnipSuggest recommends small *SQL snippets*, such as a list of k relevant predicates for the WHERE clause, k table names for the FROM clause, etc. The key contribution is in computing these recommendations. Instead of simply recommending valid or generally popular tables/attributes, SnipSuggest produces *context-aware suggestions*. That is, SnipSuggest considers the partial query that the user has typed so far, when generating its recommendations. Our key hypothesis is that, as a user articulates an increasingly larger fragment of a query, SnipSuggest has more information on what to recommend. SnipSuggest draws its recommendations from similar *past queries authored by other users*, thus leveraging a growing, shared, body of experience.

This simple idea has dramatic impact in practice. By narrowing down the scope of the recommendation, SnipSuggest is able to suggest rarely-used tables, attributes, user-defined functions, or predicates, which make sense only in the current context of the partially formulated query. In our experimental section, we show an increase in average precision of up to 144% over the state-of-the-art (Figure 5(c)), which is recommendation based on popularity.

More specifically, our paper makes the following contributions:

1. We introduce *query snippets* and the *Workload DAG*, two new abstractions that enable us to formalize the context-aware SQL autocomplete problem. Using these abstractions, we define two metrics for assessing the quality of recommendations: *accuracy* and *coverage* (Section 3.2).

2. We describe two algorithms *SSAccuracy* and *SSCoverage* for recommending query snippets based on a query log, which maximize either accuracy or coverage (Sections 3.2.4, 3.2.5).

3. We devise an approach that effectively distinguishes between potentially high-quality and low-quality queries in a log. We use this technique to trim the query log, which drastically reduces the recommendation time while maintaining and often increasing recommendation quality (Section 3.3).

4. We implement the above ideas in a SnipSuggest prototype and evaluate them on two real datasets (Section 4). We find that SnipSuggest makes recommendations with up to 93.7% average precision, and at interactive speeds, achieving a mean response time of 14ms per query.

2. MOTIVATING EXAMPLE

In this section, we present an overview of the SnipSuggest system through a motivating scenario based on the SDSS query log.

Astronomer Joe wants to write a SQL query to find all the stars of a certain brightness in the r-band within 2 arc minutes (i.e., $\frac{1}{30}$th of 1°) of a known star. The star's coordinates are 145.622 (ra), 0.0346249 (dec). He wants to group the resulting stars by their right ascensions (i.e., longitudes). This is a *real query* that we found in the SDSS query log. Joe is familiar with the domain (i.e., astronomy), but is not familiar with the SDSS schema. He knows a bit of SQL, and is able to write simple select-from-where queries.

It is well known that `PhotoPrimary` is the core SkyServer view holding all the primary survey objects. So, Joe starts off as follows: `SELECT * FROM PhotoPrimary`.

Joe is interested in only those objects that are near his coordinates. Browsing through the 454 attributes of the PhotoPrimary table's schema fails to reveal any useful attributes over which to specify this condition. Joe suspects that he needs to join PhotoPrimary with some other table, but he does not know which one. Joe thus turns to SnipSuggest for help and asks for a recommendation of a table to add to his FROM clause.

SnipSuggest suggests the five most-relevant snippets for this clause: `SpecObj`, `Field`, `fGetNearbyObjEq(?,?,?)`, `fGetObjFromRect(?, ?, ?, ?)`, and `RunQA`. (All the suggestions in this section are real suggestions from SnipSuggest.)

In this example, `fGetNearbyObjEq` is what Joe needs. There are several challenges with showing such a recommendation. First, the recommended snippet is not a table, it is a user-defined function. Second, the desired tables, views, or UDFs are not necessarily popular by themselves. They are just frequently used in the context of the query that the user wrote so far. Finally, all recommendations must be done at interactive speed for the user to remain focused.

Additionally, upon seeing a recommendation, a user can be confused as to how to use the recommended snippet. To address this challenge, SnipSuggest can show, upon request, either documentation related to the suggested snippet, or real queries that use it.

After this first step, Joe's query thus looks as follows:

```
SELECT *
FROM PhotoPrimary P,fGetNearbyObjEq(145.622,0.0346249,2) n
WHERE
```

Joe would now like to restrict the objects to include only those with a certain redness value. Encouraged by his early success with SnipSuggest, instead of browsing through documentation again, he directly asks SnipSuggest for recommendations for his `WHERE` clause. First is the missing foreign-key join `p.objId = n.objId`. Once added, SnipSuggest's next recommendations become: `p.dec<#`, `p.ra>#`, `p.dec>#`, `p.ra<#`, and `p.r≤#`. These predicates are the most popular predicates appearing in similar past queries. After a quick glance at `p.r`'s documentation, Joe picks the last option, and adds the following predicate to his query: `p.r < 18 AND p.r > 15`. This example demonstrates the need for the query log. With the exception of foreign-key joins, it is not possible to determine useful predicates for the `WHERE` clause using the database schema alone. Past queries enable SnipSuggest to select the relevant predicates among the large space of all possible predicates.

Now, his query looks as follows:

```
SELECT *
FROM PhotoPrimary P,fGetNearbyObjEq(145.622,0.0346249,2) n
WHERE p.objID = n.objID AND p.r < 18 AND p.r > 15
```

In a similar fashion, SnipSuggest can help Joe to add a second predicate (i.e., keep only objects that are stars: `p.type = 6`), and to write the `SELECT` and `GROUP BY` clauses.

In summary, the challenges for SnipSuggest are to (1) recommend relevant features without knowledge of the user's intended query, (2) leverage the current query context to improve the recommendation quality, and (3) produce recommendations efficiently.

3. SnipSuggest

SnipSuggest is a middleware-layer on top of a standard relational DBMS as shown in Figure 1. While users submit queries against the database, SnipSuggest's Query Logger component logs these queries in a Query Repository. Upon request, SnipSuggest's Snippet Recommender uses this query repository to produce SQL-autocomplete recommendations. Finally, SnipSuggest's Query Eliminator periodically prunes the query log to improve recommendation performance by shrinking the Query Repository. We now present these three components and SnipSuggest's algorithms.

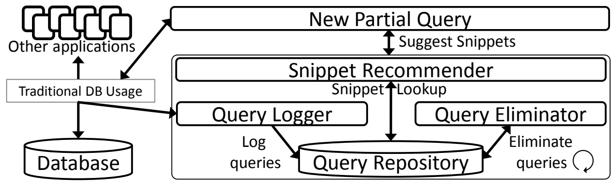


Figure 1: SnipSuggest System Architecture

3.1 Query Logger and Repository

When the Query Logger logs queries, it extracts various *features* from these queries. Informally, a feature is a specific fragment of SQL such as a table name in the `FROM` clause (or view name or table-valued function name), or a predicate in the `WHERE` clause.

The Query Repository comprises three relations: `Queries`, `Features`, and `QueryFeatures`. `Queries` contains metadata about the query (e.g., id, timestamp, query text) and is populated using the existing infrastructure for query logging offered by most DBMSs. The `Features` table lists each feature (i.e., SQL fragment) and the clause from which it was extracted. The `QueryFeatures` table lists which features appear in which queries. Appendix B outlines these tables' schemas, and the features that are currently supported.

3.2 Snippet Recommendation

While a user composes a query, she can, at any time, select a clause, and ask SnipSuggest for recommendations in this clause. At this point, SnipSuggest's goal is to recommend k features that are most likely to appear in that clause in *the user's intended query*.

To produce its recommendations, SnipSuggest views the space of queries as a directed acyclic graph (DAG) [1] such as that shown in Figure 2 (which we return to later). For this, it models each query as a set of features and every possible set of features becomes a vertex in the DAG. When a user asks for a recommendation, SnipSuggest, similarly, transforms the user's partially written query into a set of features, which maps onto a node in the DAG. Each edge in the DAG represents the addition of a feature (i.e., it links together sets of features that differ by only one element). The recommendation problem translates to that of ranking the outgoing edges for the vertex that corresponds to the user's partially written query, since this corresponds to ranking the addition of different features.

The query that the user intends to write is somewhere below the current vertex in the DAG, but SnipSuggest does not know which query it is. It approximates the intended query with the set of all queries in the Query Repository that are descendants of the current vertex in the DAG. We refer to such queries as the $potential\ goals$ for the partial query. For now, we assume that the set is not empty (and discuss the alternative at the end of Section 3.2.4). Given this set of $potential\ goals$, there are several ways to rank the features that could possibly be added to the user query. We investigate two of them in this paper. The first approach is simply to recommend the most popular features among all those queries. The problem with this approach is that it can easily lead to k recommendations all leading to a single, extremely popular query. An alternate approach is thus to recommend k features that cover a maximal number of queries in the $potential\ goals$ set.

We now describe the problem and our approach more formally.

3.2.1 Definitions

We begin with the definition of features.

DEFINITION 1. *A* **feature** f *is a function that takes a query as input, and returns true or false depending on whether a certain property holds on that query.*

[1]Note that the DAG is purely a conceptual model underlying SnipSuggest. The user never interacts with it directly.

Some examples are $f^{\texttt{FROM}}_{\texttt{PhotoPrimary}}$, representing if the `PhotoPrimary` table appears in the query's `FROM` clause, $f^{\texttt{WHERE}}_{\texttt{PhotoPrimary.objID=Neighbors.objID}}$, representing whether the predicate `PhotoPrimary.objID = Neighbors.objID` appears in the `WHERE` clause, or $f_{\texttt{distinct}}$ representing whether the `distinct` keyword appears anywhere in the query. A feature can have a clause associated with it, denoted $clause(f)$. For example, $clause(f^{\texttt{FROM}}_{\texttt{PhotoPrimary}}) = \texttt{FROM}$. Through this paper, we use the notation f^c_s to denote the feature that string s appears in clause c.

DEFINITION 2. *The* **feature set of a query** q*, is defined as:*

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

When SnipSuggest 'recommends a snippet', it is recommending that the user modify the query so that the snippet evaluates to true for the query. For example, when it recommends $f^{\texttt{FROM}}_{\texttt{PhotoPrimary}}$, it is recommending that the user add `PhotoPrimary` to the `FROM` clause.

DEFINITION 3. *The* **dependencies of a feature** f*,* $dependencies(f)$*, is the set of features that must be in the query so that no syntactic error is raised when one adds* f*.*

e.g., $dependencies(f^{\texttt{WHERE}}_{\texttt{PhotoPrimary.objID=Neighbors.objID}})=\{f^{\texttt{FROM}}_{\texttt{PhotoPrimary}}, f^{\texttt{FROM}}_{\texttt{Neighbors}}\}$. SnipSuggest only suggests a feature f for a partial query q if $dependencies(f) \subseteq features(q)$. In the workload DAG, feature sets have parent-child relationships defined as follows:

DEFINITION 4. *A feature set* F_2 *is a* **successor** *of a feature set* F_1*, if* $\exists\ f$ *where* $F_2{=}F_1{\cup}\{f\}$ *and* $dependencies(f){\subseteq}F_1$*.*

A successor of a feature set F_1 is thus a feature set F_2 that can be reached by adding a single, valid feature.

Additionally, recommendations are based on feature popularity that is captured by either marginal or conditional probabilities.

DEFINITION 5. *Within a workload* W*, the* **marginal probability of a set of features** F *is defined as*

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

i.e. the fraction of queries which are supersets of F*. As shorthand, we use* $P(Q)$ *for* $P(features(Q))$*, and* $P(f)$ *for* $P(\{f\})$*.*

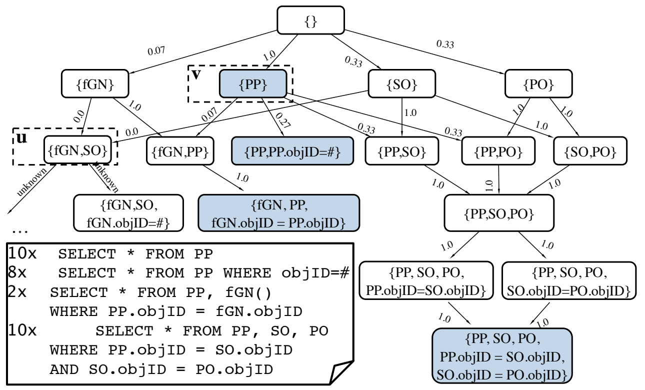
DEFINITION 6. *The* **conditional probability of a feature** f **given a feature set** F *is defined as*

$$P(f|F) = \frac{P(\{f\} \cup F)}{P(F)}$$

We are now ready to define the workload DAG. Let F be the set of all features (including those that do not appear in workload).

DEFINITION 7. *The* **workload DAG** $T = (V, E, w, \chi)$ **for a query workload** W *is constructed as follows:*

1. *Add to* V*, a vertex for every syntactically-valid subset of* F*. We refer to each vertex by the subset that it represents.*
2. *Add an edge* (F_1, F_2) *to* E*, if* F_2 *is a successor of* F_1*. Denote the additional feature of* F_2 *by* $\mathbf{addlFeature}((\mathbf{F_1}, \mathbf{F_2})) = f$*, where* $F_2 = F_1 \cup \{f\}$*.*
3. $w : E \rightarrow [0, 1]$ *is the weight of each edge. The weights are set as:* $w((X, Y)) = P(addlFeature((X, Y))|X)$*. If* $P(X) = 0$*, then set to* `unknown`*.*
4. $\chi : V \rightarrow \{blue, white\}$ *is the color of each vertex. The colors are set as:* $\chi(q) = blue$ *if* $q \in W$*, otherwise white.*



```
10x  SELECT * FROM PP
8x   SELECT * FROM PP WHERE objID=#
2x  SELECT * FROM PP, fGN()
    WHERE PP.objID = fGN.objID
10x      SELECT * FROM PP, SO, PO
    WHERE PP.objID = SO.objID
    AND SO.objID = PO.objID
```

Figure 2: Example of a workload DAG.

Figure 2 shows an example workload DAG for 30 queries. The queries correspond to the blue nodes, and are summarized at the bottom. Ten are of the form `SELECT * FROM PhotoPrimary`, eight are `SELECT * FROM PhotoPrimary WHERE objID = #`, etc. For simplicity, we exclude, from the figure, features in the `SELECT` clause, and nodes that are not reachable from the root along edges of weight > 0, with the exception of node u. We use the acronyms `fGN`, `PP`, `SO`, and `PO` to represent $f^{\text{FROM}}_{\texttt{fGetNearbyObjEq}}$, $f^{\text{FROM}}_{\texttt{PhotoPrimary}}$, $f^{\text{FROM}}_{\texttt{SpecObjAll}}$, and $f^{\text{FROM}}_{\texttt{PhotoObjAll}}$, respectively. The edge $(\{PP\}, \{PP, SO\})$ indicates that if a query contains `PhotoPrimary`, there is 33% chance that it also contains `SpecObjAll`.

Every syntactically-correct partial query appears in the workload DAG since there is a vertex for every valid subset of F. Consider a partial query, and its corresponding vertex q. Given q, SnipSuggest's goal is to lead the user towards their intended query $q*$ (also a vertex in the DAG), one snippet at a time. We assume that there is a path from q to $q*$, i.e., that the user can reach $q*$ by adding snippets to their query. Since SnipSuggest suggests one snippet at a time, the recommendation problem becomes that of ranking the outgoing edges of q. Note that recommending an edge e corresponds to recommending $addlFeature(e)$.

For example, suppose Anna has written: `SELECT * FROM PhotoPrimary`. This puts her at vertex v in Figure 2. Then, she requests snippets to add to the `FROM` clause. At v, we see that she can add `fGetNearbyObjEq()`, `SpecObjAll`, or `PhotoObjAll`. Remember, Figure 2 is not showing the whole DAG. In fact, the full workload DAG contains an outgoing edge from v for each of the 342 tables, views, and table-valued functions in the SDSS schema. The job of SnipSuggest is to recommend the edges that are most likely to lead Anna towards her intended query.

3.2.2 Naïve Algorithms

We present three techniques that we compare against SnipSuggest in Section 4. The most naïve, the **Random** recommender, ranks the outgoing edges randomly. The second approach, **Foreign-key-based**, used only for suggesting snippets in the `WHERE` clause, exploits the schema information to rank the features. It suggests predicates for foreign-key joins before other predicates. The third approach, the **Popularity-based** technique actually leverages the past workload. It considers $f = addlFeature(e)$ for each outgoing edge from q, and ranks them by $P(f)$, the marginal probability of f. Even this simple technique, outperforms the above two algorithms by up to 449%.

The problem with these approaches is that they do not exploit the rich information available in the workload DAG; the weighted edges can tell us which features are likely to appear in the intended query, given the current partial query. SnipSuggest's algorithms aim to better recommend snippets by leveraging such information.

3.2.3 Context-Aware Algorithms

In this section, we introduce the two algorithms that SnipSuggest uses to recommend suggestions based on the current context (i.e., the current partial query). First, we need one more notion, and a precise definition of the problem that each algorithm aims to solve.

DEFINITION 8. *Given a workload DAG and a vertex q, define:* $potential\_goals(q) = \{v | v \text{ is blue and reachable from } q\}$.

The $potential\_goals$ of q is the set of queries that could potentially be the user's intended query, if it appears in the workload. Sometimes, $potential\_goals(q)$ is the empty set.

We consider two variations of the Snippet Suggestion Problem. Given a workload DAG G, and a partial query q, recommend a set of k outgoing edges, $e_1, \ldots, e_k$, from q that:

1. **Max-Accuracy Problem:** maximizes
$$\sum_{i=1}^{k} P(addlFeature(e_i)|q)$$
2. **Max-Query-Coverage Problem:** maximizes
$$P(addlFeature(e_1) \vee \ldots \vee addlFeature(e_k)|q)$$

Max-Accuracy aims to maximize the number of features in the top-k that are helpful (i.e., appear in the intended query), whereas Max-Query-Coverage aims to maximize the probability that at least one feature in the top-k is helpful.

Consider the earlier example. Suppose SnipSuggest recommends the top-2 snippets to add to the `FROM` clause. If the goal is Max-Accuracy, then it suggests `SO` and `PO`. This corresponds to the two outgoing edges from q, with the highest conditional probabilities. If the aim is Max-Query-Coverage, then it suggests `SO` and `fGN`. The reasoning is as follows: if Anna's intention is the rightmost blue query, then suggesting `SO` covers this case. If her intention is not that query, then rather than `PO`, it is better to suggest `fGN` because it increases the number of $potential\_goals$ covered.

It is infeasible to build the workload DAG as it can have up to 2^n vertices, where $n = |F|$. Thus, SnipSuggest implements two algorithms, which simulate traversing parts of the DAG, without ever constructing it: $SSAccuracy$ and $SSCoverage$.

3.2.4 $SSAccuracy$

Given a partial query q and a query workload W, the goal of the $SSAccuracy$ algorithm is to suggest the k features with the highest conditional probabilities given q. If q's features have appeared together in past queries, $SSAccuracy$ is able to efficiently identify the features with the highest conditional probabilities, with a single SQL query over the $QueryFeatures$ table, as shown in Figure 3. By setting m to $|features(q)|$, the first half of the query finds $potential\_goals(q)$, i.e. the queries which have all the features of q. It then orders all the features which appear in these `SimilarQueries`, by their frequencies within this set of queries. Note that each `qf.feature` f corresponds to one outgoing edge from q (i.e. the edge e where $addlFeature(e) = f$). Additionally, if we divided `count(s.query)` by $|potential\_goals(q)|$, we would find $P(f|q)$. Thus, this query returns a list of edges, ordered by weight (i.e., the conditional probability of the feature given q).

What if the partial query q does *not* appear in the workload? Every partial query q appears in the DAG, but it can happen that all incoming edges have weight 0, and all outgoing edges are `unknown`. This happens when $potential\_goals(q) = \emptyset$. e.g., if Bob has written q = `SELECT * FROM SpecObjAll, fGetNearbyObjEq(143.6,0.021,3)`, and requests suggestions in the `FROM` clause (which is represented by vertex u in Figure 2).

In this case, SnipSuggest traverses up the DAG from q until it reaches the vertices whose marginal probability is not zero

```
WITH SimilarQueries (query) AS --finds potential_goals
(SELECT query
 FROM  QueryFeature
 WHERE  feature IN features(q)
  [AND NOT EXISTS ( --used only by SSCoverage
   select * from QueryFeature q
   where q.query=query and q.feature in( previous))]
 GROUP BY query
 HAVING count(feature) =  m)
SELECT qf.feature --popular features among SimilarQueries
FROM   QueryFeature qf, SimilarQueries s
WHERE qf.query = s.query AND qf.feature NOT IN features(q)
GROUP BY qf.feature
ORDER BY count(s.query) DESC
```

Figure 3: Finds the most popular features among queries that share m features with partial query q. `NOT EXISTS` **clause is included for SSCoverage, but omitted for SSAccuracy.**

Algorithm 1 SnipSuggest's Suggestion Algorithm

```
Input: query $q$, number of suggestions $k$, clause $c$, technique $t$
Output: a ranked list of snippet features
1: $i \leftarrow |features(q)|$
2: $suggestions \leftarrow []$
3: while $|suggestions| < k$ : do
4:     if $t = SSAcc$ then
5:         $S \leftarrow$ execute Figure 3 query ($m \leftarrow i$, exclude NOT EXISTS clause)
6:     else if $t = SSCov$ then
7:         $S \leftarrow$ execute Figure 3 query ($m \leftarrow i, previous \leftarrow suggestions$)
8:     end if
9:     for all $s \in S$ do
10:        if $s \notin suggestions$ and $clause(s) = c$ then
11:            $suggestions \leftarrow suggestions, s$
12:        end if
13:    end for
14:    $i \leftarrow i - 1$
15: end while
16: return $suggestions$
```

(i.e., there exists an incoming edge with weight > 0). This corresponds to finding the largest subsets of $features(q)$ that appear in the workload. In the above example, SnipSuggest traverses up to the vertices {SO} and {fGN}. Then, it suggests the most popular features among the queries under these vertices. This can be achieved by executing the SQL query shown in Figure 3. First, SnipSuggest sets m to $|features(q)|$, thus looking at $potential\_goals(q)$. If fewer than k features are returned, then it sets m to $|features(q)| - 1$, thus considering queries that share $|features(q)| - 1$ features with q. It repeatedly decrements m until k features are returned. Note that SnipSuggest executes the query in Figure 3 at most $|features(q)|$ times. In other words, SnipSuggest will not iterate through every subset of $features(q)$. Instead, it considers all subsets of size m all at once, and it does this for $m = n, n-1, n-2, \ldots, 0$, where $n = |features(q)|$.

This process can cause some ambiguity of how to rank features. For example, if $P(f_1|\{SO, fGN\}) = 0.8$ and $P(f_2|\{SO\}) = 0.9$, it is not clear whether f_1 or f_2 should be ranked first. Heuristically, SnipSuggest picks f_1, because it always ranks recommendations based on more similar queries first. Algorithm 1 outlines the full SSAccuracy algorithm (if we pass it $t = SSAcc$). In Section 4, we show that $SSAccuracy$ achieves high average precision, and in Appendix C, we describe two simple optimizations.

3.2.5 SSCoverage

The Max-Query-Coverage problem is to suggest the features $f_1, \ldots, f_k$ that maximize the probability that at least one suggestion is helpful. The goal is to diversify the suggestions, to avoid making suggestions that all lead toward the same query. It turns out that the problem is NP-hard. Instead of an exact solution, we propose an approximation algorithm, SSCoverage, which is a greedy, approximation algorithm for Max-Query-Coverage. We prove in Appendix D that Max-Query-Coverage is NP-hard and that SSCoverage is the best possible approximation for it. We show this by proving that Max-Query-Coverage is equivalent to the well-known Maximum Coverage problem [12]. Given a set of elements U and a set of sets $S = S_1, \ldots, S_n$, where each $S_i \in U$, the Maximum Coverage problem is to find a subset of S of size k (fixed) such that a maximal number of elements in U are 'covered'. Our equivalence proof is sufficient because it is known that the Maximum Coverage problem is NP-hard, and that the best possible approximation is the greedy algorithm [12], achieving an approximation factor of $1 - \frac{1}{e}$.

The $SSCoverage$ algorithm proceeds as follows. To compute the first recommendation f_1, it executes the SQL query in Figure 3, with the `NOT EXISTS` clause and `previous` $\leftarrow \emptyset$. This is equivalent to $SSAccuracy$'s first recommendation because the Max-Accuracy and Max-Query-Coverage formulas are equivalent when $k = 1$. For its second suggestion, $SSCoverage$ executes the Figure 3 query again, but with `previous` $\leftarrow \{f_1\}$. This effectively removes all the queries covered by f_1 (i.e., $potential\_goals(q \cup \{f_1\})$), and finds the feature with the highest coverage (i.e., conditional probability) in the remaining set. In terms of the workload DAG, this step discards the whole subgraph rooted at f_1, and then finds the best feature among the remaining DAG. It repeats this process k times in order to collect k features. Algorithm 1 describes SSCoverage in detail (if we pass it $t = SSCov$).

3.3 Query Elimination

All queries, correct or incorrect, are logged by the Query Logger. This is problematic for SnipSuggest because a large workload can deteriorate its response time. The Query Eliminator addresses this problem; it periodically analyzes the most recent queries and drops some of them. The goal is to reduce the workload size, and the recommendation time, while maintaining recommendation quality.

For the Query Eliminator, we introduce the notion of a query session. A *query session* is a sequence of queries written by the same user as part of a single task. The Query Eliminator eliminates all queries, except those that appear at the end of a session. Intuitively, queries that appear near the end of the session are of higher quality, since the user has been working on them for longer. Since many users write a handful of queries before reaching their intended one, this technique eliminates a large fraction of the workload. We show in Section 4.3 that the Query Eliminator reduces the response time, while maintaining, and often improving, the average precision.

SnipSuggest extracts query sessions in two phases (as shown in Figure 4). First, SnipSuggest segments the incoming query log. It does so by monitoring changes between consecutive queries in order to detect starts of new revision cycles. A *revision cycle* is the iterative process of refining and resubmitting queries until a desired task is complete. When it detects a new cycle, SnipSuggest labels, as query segment, the set of all queries since the beginning of the previous cycle. We call this phase *segmentation*. Second, SnipSuggest stitches multiple segments together, if they are part of a single, larger revision cycle. For example, when a user is stuck on a difficult task A, they often move to a different task B and later return to A. In this scenario, A will produce multiple query segments, because the queries for task B will separate the later queries of A from the earlier ones. Via *stitching*, the segments are concatenated together to create a large session for A.

Both phases require an expert to provide some training data, in the form of a query log with labeled sessions. SnipSuggest leverages machine learning techniques to learn the appropriate thresholds for the segmentation and stitching. We present a more detailed

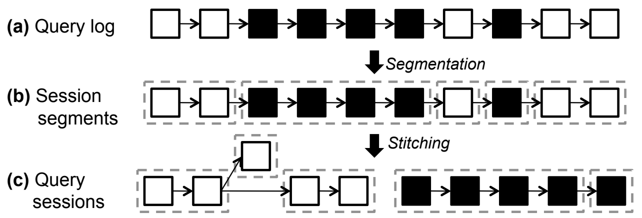


Figure 4: Extracting query sessions from the query log.

description of the two algorithms in Appendix E. We also show that our technique is able to more accurately segment a query log into sessions, in comparison to a time-interval-based technique.

4. EVALUATION

We evaluate SnipSuggest over two datasets. The first consists of queries from the Sloan Digital Sky Survey. The SDSS database logs all queries submitted through public interfaces including web pages, query forms, and a web services API. Thus, query authors vary from bots, to the general public, and real scientists. We downloaded 106 million queries from 2002 to 2009. Removing queries from bots, ill-formed queries, and queries with procedural SQL or proprietary features, left us with approximately 93 million queries. For our evaluation, we use a random sample of 10,000 queries, in which there are 49 features for tables, views and table-valued functions, 1835 columns and aggregates, and 395 predicates.

The second dataset consists of SQL queries written by students in an undergraduate database class, which were automatically logged as they worked on nine different problems for one assignment. All queries are over a local copy of the Internet Movie Database (IMDB), consisting of five tables and 21 columns. To evaluate, we use a sample consisting of ten students' queries, which results in a total of 1679 queries. For each student, we manually label each query with the assignment problem number that it was written for, which gives us ground truth information about query sessions and thus serves as training data for the Query Eliminator.

We aim to answer four questions: Is SnipSuggest able to effectively recommend relevant snippets? Does its recommendation quality improve as the user adds more information to a query? Can it make suggestions at interactive speeds? Is the Query Eliminator effective at reducing response times, while maintaining recommendation quality? We evaluate the first three questions on both datasets. The fourth is answered on only the IMDB dataset because we do not have the ground truth for the SDSS query sessions.

4.1 Evaluation Technique

Neither query log includes "partial queries"; they only include full SQL queries that were submitted for execution. Therefore, in order to evaluate SnipSuggest on a given query, we remove some portion of the query, and the task is to recommend snippets to add to this new partial query. We denote by $fullQuery(q)$, the full SQL query from which the partial query q was generated. For this setting, we define correctness for a partial query q, and feature f.

DEFINITION 9. *For a given partial query q, a suggested snippet feature f is* correct *if and only if $f \notin features(q)$ and $f \in features(fullQuery(q))$.*

To measure the recommendation quality of SnipSuggest, we use a measure called average precision [3]. It is a widely-used measure in Information Retrieval for evaluating ranking techniques. SnipSuggest returns a ranked list of snippets, L_q, for query q.

DEFINITION 10. *The average precision at k for the suggestions L_q is*

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

where $P(q, L_q, k)$ is the precision of the top-k recommendations in L_q and $rel(q, L_q[i]) = 1$ $if L_q[i]$ is correct, and 0 otherwise. Precision is defined as $P(q, L_q, k) = \frac{\sum_{i=1}^{k} rel(q, L_q[i])}{k}$

This measure looks at the precision after each correct snippet is included, and then takes their average. If a correct snippet is not included in the top-k, it contributes a precision of zero. The benefit of using this approach, instead of recall or precision, is that it rewards the techniques that put the correct snippets near the top of the list. For an example, we refer the reader to Appendix G.

4.2 SDSS Dataset

We first evaluate SnipSuggest on the real SDSS dataset.

4.2.1 Quality of Recommendations

We evaluate several aspects of SnipSuggest: its ability to recommend relations, views and tables-valued functions in the `FROM` clause, predicates in the `WHERE` clause, columns and aggregates in the `SELECT` clause, and columns in the `GROUP BY` clause. We evaluate only the SSAccuracy algorithm here because it is able to achieve an interactive response time, and it is the algorithm that aims to solve the Max-Accuracy problem, which corresponds to achieving high average precision. We compare the SSAccuracy and SSCoverage algorithms in Section 4.2.3.

We do a 10-fold cross-validation, with the queries ordered randomly; we use 90% of the queries as the past query workload and test on the remaining 10%. We repeat the experiment 10 times, each time selecting a different set of test queries, so that all queries are part of the test set exactly once. We measure the mean average precision for each of the ten experiments at top-1 through top-10.

Figures 5(a) - (c) show the results for predicting snippets in the `FROM` clause. For this experiment, we consider queries with at least three tables, views or table-valued functions in the `FROM` clause. Figure 5(a) shows how accurately SnipSuggest recommends snippets, if an empty query is presented. It achieves the same average precision as the Popularity-based algorithm (and thus, SnipSuggest's points are not visible in the graph). This is expected, because with no information, SnipSuggest recommends the most popular snippets across all queries. As soon as the user adds one table to the query (out of three), SnipSuggest's $AP_{@5}$ jumps from 0.49 to 0.77 (nearly a 60% increase). In contrast, the other two techniques' average precisions degrade, because once we add one table to the query, the number of correct snippets has decreased from 3 snippets per query, to only 2 snippets per query. The Popularity approach's average precision, for example, drops from 0.49 to 0.46. This trend continues when we add two tables to the `FROM` clause. SnipSuggest's average precision jumps to 0.90 (an 84% increase from 0 tables), and Popularity drops to 0.42 (a 16% decrease from 0 tables). In brief, *Figures 5(a)-(c) show that SnipSuggest's average precision improves greatly as the user makes progress in writing the SQL query, whereas the other two techniques degrade.*

Figures 5(d) - (e) show how accurately SnipSuggest recommends predicates in the `WHERE` clause, for queries with at least one predicate (d), and with at least two predicates (e). In our sample, 75% of the queries have exactly one predicate, and 23% have more. We see that the Popularity approach performs well ($AP_{@5} = 0.81$). This is because all the techniques recommend only valid snippets, and so the Popularity approach restricts its recommended predicates to only those which reference tables that are already in the partial query, and then suggests them in popularity order. Many predicates are join predicates, but the ForeignKey approach still lags because many of the join predicates involve table-valued functions, and thus are not across foreign-key connections. SnipSuggest remains the top approach, achieving an average precision ($AP_{@5}$)

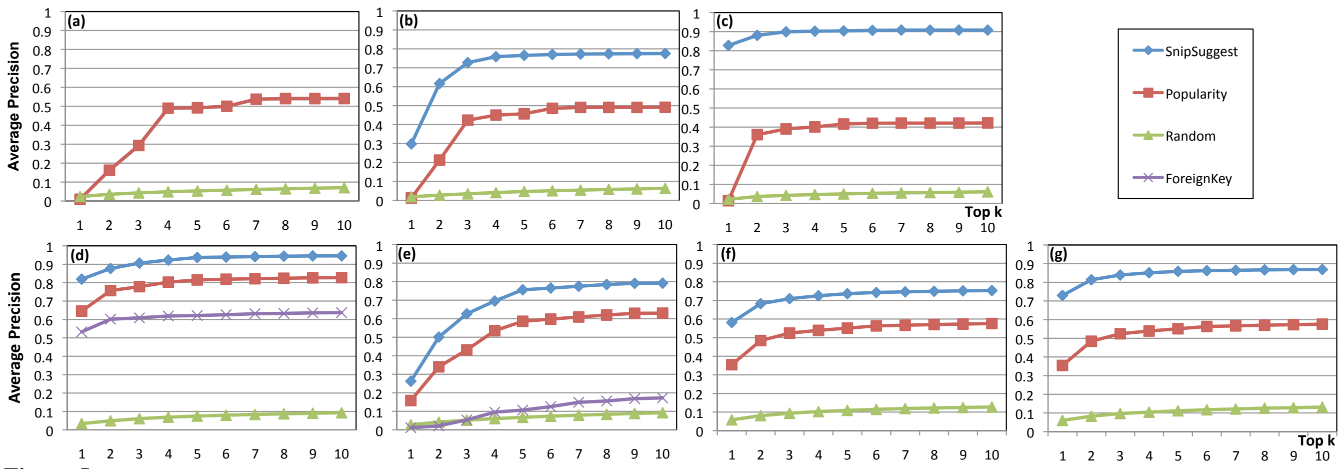


Figure 5: Average Precision for recommending tables/views/table-valued functions in `FROM` clause given an empty query, 1 table, or 2 tables (a-c), recommending predicates in `WHERE` clause, for queries with > 0 predicates, or > 1 predicates (d-e), recommending columns in `GROUP BY` clause, given the `FROM` clause, or given both the `FROM` and `WHERE` clauses (f-g). In (a), SnipSuggest's average precision is equal to Popularity's.

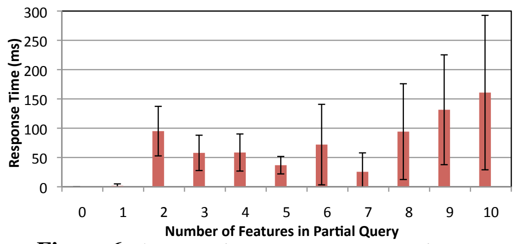


Figure 6: Average times to recommend snippets.

of 0.94. Once we consider the less common case, when the query has multiple predicates, Figure 5(e) shows a larger difference between the techniques. The ForeignKey technique's performance degrades drastically because we are now looking at mostly non-join predicates. The Popularity approach and SnipSuggest's average precisions also drop (because the queries contain rarer predicates), but now there is a significant discrepancy between the two. In summary, *Figures 5(d)-(e) show that both the Popularity and SnipSuggest approaches recommend predicates with high average precision. However, if we consider only queries with multiple predicates, SnipSuggest outperforms the Popularity approach by 29%.*

Figures 5(f) - (g) show SnipSuggest's performance for recommending columns in the `GROUP BY` clause, given the `FROM` clause (f) and given both `FROM` and `WHERE` clauses (g). We see a similar trend to recommending snippets in the `FROM` and `WHERE` clauses. SnipSuggest, once again, outperforms the Popularity approach, with $AP_{@5} = 0.86$ versus 0.55. We also see that SnipSuggest's $AP_{@5}$ increases from 0.74 to 0.86 between Figures 5(f) and (g). *From Figures 5(f)-(g), we learn that, for suggesting snippets in the `GROUP BY` clause, SnipSuggest's average precision increases by 16% when the `WHERE` clause is provided in addition to the `FROM` clause, and that SnipSuggest outperforms, by 56%, the Popularity approach.*

For suggesting columns in the `SELECT` clause (not shown), we learn that the Popularity and SnipSuggest approaches perform similarly because most queries select many columns (the average number of columns selected is 12.5), and that the benefit of leveraging the `WHERE` clause, in addition to the `FROM` clause, is small.

4.2.2 *Efficiency*

Past research shows that a response time of up to 100ms is considered interactive [6]. In the experiments above, SnipSuggest achieves a mean response time of 14ms, and an interactive response time for 94.21% of the partial queries. Figure 6 shows the mean response time for different numbers of features in the partial query. The response time can not be determined by the number of features alone; the popularity of the features plays a large role. If the features are popular, there are more relevant queries, and thus more data to process. Therefore, there is no clear trend in the results. We see, however, that the response time increases when we reach 9-10 features. This is because there are often no queries in the workload which contain *all* 9-10 features, and thus SnipSuggest needs to run the SQL in Figure 3 multiple times. We exclude partial queries with over 10 features, because there are fewer than 25 such queries.

4.2.3 *SSAccuracy versus SSCoverage*

Now, we compare the SSAccuracy and SSCoverage algorithms. We use a small dataset of only 2000 queries because the SSCoverage algorithm is slow. Since the aim of SSCoverage is different, we use a different measure to evaluate recommendation quality. We define the *utility* of a ranked list of suggestions to be 1 if there is any correct suggestion in the top-k, and 0 otherwise. We report the mean utility across the queries. This is equal to the percentage of queries for which there is a correct suggestion in the top-k.

Figure 7 shows the results for predicting columns in the `SELECT` clause, given the `FROM` clause. The difference in the percentage between i and i+1 represents the average additional coverage provided by the i+1th suggestion. We see that this difference is monotonically decreasing in Figure 7, which indicates that SSCoverage suggests the features with the most additional coverage earlier in its ranking. Figure 7 shows that *the mean utility of SSCoverage is 15.13% higher than SSAccuracy at the top-5.*

The trend continues for the `FROM` and `WHERE` clauses, though not to the same extent. For the `FROM` clause, given an empty query, SSCoverage achieves a 2% improvement for top-5. Most queries have only one or two tables in the `FROM` clause, so SSAccuracy is guaranteed to suggest features from different queries in the top-5, thus already achieving high coverage. For the `WHERE` clause, SSCoverage outperforms SSAccuracy by 3% in the top-3 (75% of queries contain only one predicate). For queries with multiple predicates, the difference increases to 4%. Although these differences appear small, SSAccuracy already achieves 87% utility at top-3 for the `FROM` clause, and 96% for the `WHERE` clause. Given the little room for improvement, these increases are significant.

4.3 IMDB Dataset

We utilize the IMDB dataset for three tasks. First, we study the Query Eliminator's effect on the response time and average precision. Second, we evaluate SnipSuggest over a second dataset.

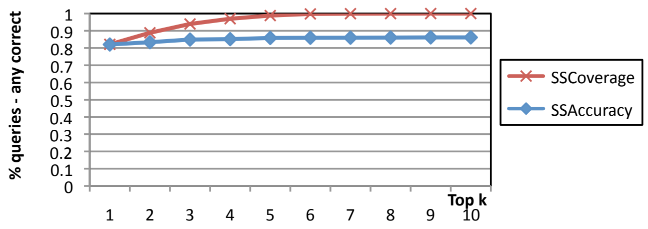


Figure 7: SSAccuracy vs. SSCoverage.

Task	Decrease in Time	Increase in $AP_{@3}$
`FROM` → `WHERE`	74.49%	8.12%
∅ → `FROM`	9.17%	4.67%
`1 table in FROM` → `FROM`	79.76%	0.74%
`2 tables in FROM` → `FROM`	79.47%	-0.71%
`FROM` → `SELECT`	88.87%	15.80%
`FROM, WHERE` → `SELECT`	79.37%	2.72%
`FROM` → `GROUP BY`	77.04%	7.11%
`FROM, WHERE` → `SELECT`	60.91%	5.37%

Table 1: The benefits and drawbacks of the Query Eliminator.

Third, we measure the Query Eliminator's ability to correctly detect end-of-session queries. We present the results of the first task here, and the remaining two are discussed in Appendix F.2.

We summarize the benefits and drawbacks of the Query Eliminator in Table 1. From 1679 queries, the Query Eliminator maintains only 7%, or a total of 117 queries. The goal here is to decrease the response time, while maintaining a similar average precision. Table 1 shows that the technique decreases the response time by up to 89%. For this dataset, it also increases the average precision ($AP_{@3}$) for all tasks but one. Even in the worst case, the average precision decreases by less than 1%!

5. RELATED WORK

There are various efforts toward making databases easier to use. Jagadish *et al.* [16] present six challenges in using databases today.

Nandi *et al.* [21] present a technique for phrase autocompletion for keyword search over structured databases. Another similar tool [20] allows users to construct search queries without knowledge of the underlying schema. As the user types in the search box, the tool starts suggesting elements of the schema, followed by fragments of text from the database content. This tool supports standard conjunctive attribute-value queries. Although both papers are related to SnipSuggest, they focus on keyword and key-value queries, which make the problems different from suggesting relevant snippets for a structured SQL query.

QueRIE [8] analyzes a user's query log, finds other users who have executed queries over similar parts of the database, and recommends new queries to retrieve relevant data. SnipSuggest differs in several aspects. It provides assistance based on the user's partial query, does not require the user to have written previous queries, and only recommends small snippets at a time.

Annotating a query with a description can increase its understandability, but users can not be expected to annotate every query they write. Koutrika *et al.* [17] present methods for automatically translating SQL into natural language. This work is complementary to ours, and we hope to extend SnipSuggest with this capability.

Several commercial tools aim to ease the SQL composition process, through autocomplete for tables and columns [2], and visual querying [4, 7, 15, 19]. Similarly, SnipSuggest also supports autocomplete. However, it supports autocomplete for snippets (versus only table and column names [2]), and provides suggestions that are context-aware. We view the visual query building work as complementary to ours. Extending SnipSuggest with these visual techniques would enrich its usability. To enable such features, SnipSuggest would need the capability to map a SQL query to a state of visual query composition.

Many projects mine past query logs. Most focus on keyword search logs [5, 10, 11, 18], for goals ranging from predicting the next user action to designing a taxonomy of searches. One paper mines SQL query logs [9], in order to rank the result tuples of a query. Though these projects are similar to SnipSuggest since they leverage query workloads, they do so for a different goal.

6. CONCLUSION

In this paper, we presented SnipSuggest, a context-aware, SQL-autocomplete system. SnipSuggest is motivated by the growing population of non-expert database users, who need to perform complex analysis on their large-scale datasets, but have difficulty with SQL. SnipSuggest aims to ease query composition by suggesting relevant SQL snippets, based on what the user has typed so far. We have shown that SnipSuggest is able to make helpful suggestions, at interactive speeds for two different datasets. We view SnipSuggest as an important step toward making databases more usable.

7. ACKNOWLEDGMENTS

This work is partially supported by NSF Grants IIS-0627585, IIS-0713576, NSF CAREER award IIS-0845397, NSF CRI grant CNS-0454425, gifts from Microsoft Research, as well as Balazinska's Microsoft Research New Faculty Fellowship. The authors would also like to thank the reviewers for their helpful feedback.

8. REFERENCES

[1] SkyServer. `http://cas.sdss.org/dr6/en/`.
[2] Aqua Data Studio. `http://www.aquafold.com/`.
[3] R. Baeza-Yates and B. Riberio-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing, Boston, MA, 1999.
[4] BaseNow. Database front-end applications. About SQL Query Builder. `http://www.basenow.com/help/About_SQL_Query_Builder.asp`.
[5] A. Broder. A taxonomy of web search. *SIGIR Forum*, 36(2):3–10, 2002.
[6] S. K. Card, G. G. Robertson, and J. D. Mackinlay. The information visualizer, an information workspace. In *Proc. SIGCHI*, 1991.
[7] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: A survey. *J. of Visual Languages & Computing*, 8(2):215–260, 1997.
[8] G. Chatzopoulou, M. Eirinaki, and N. Polyzotis. Query recommendations for interactive database exploration. In *SSDBM 2009*, pages 3–18.
[9] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. In *Proc. VLDB*, pages 888–899, 2004.
[10] D. Downey, S. Dumais, D. Liebling, and E. Horvitz. Understanding the relationship between searchers' queries and information goals. In *CIKM*, pages 449–458, 2008.
[11] D. Downey, S. T. Dumais, and E. Horvitz. Models of searching and browsing: Languages, studies, and application. In *IJCAI*, 2007.
[12] U. Feige. A threshold of ln n for approximating set cover. *Journal of the ACM*, 45:314–318, 1998.
[13] UCSC Genome Browser. `http://genome.ucsc.edu/`.
[14] B. Howe and G. Cole. SQL Is Dead; Long Live SQL: Lightweight Query Services for Ad Hoc Research Data. In *4th Microsoft eScience Workshop*, 2010.
[15] IBM. Cognos software. `http://cognos.com/`.
[16] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *Proc. SIGMOD*, pages 13–24, 2007.
[17] G. Koutrika, A. Simitsis, and Y. Ioannidis. Explaining structured queries in natural language. In *Proc. of the 26th ICDE Conf.*
[18] T. Lau and E. Horvitz. Patterns of search: analyzing and modeling web query refinement. In *Proc. UM*, pages 119–128, 1999.
[19] MicroStrategy 9. Microstrategy. `http://www.microstrategy.com)`.
[20] A. Nandi and H. V. Jagadish. Assisted querying using instant-response interfaces. In *Proc. SIGMOD*, pages 1156–1158.
[21] A. Nandi and H. V. Jagadish. Effective phrase prediction. In *Proc. VLDB*, pages 219–230, 2007.
[22] Sloan Digital Sky Survey. `http://www.sdss.org/`.
[23] V. Singh, J. Gray, and A. T. A. S. Szalay. Skyserver traffic report - the first five years, 2006.
[24] A. Szalay and J. Gray. Science in an exponential world. *Nature*, March 23 2006.
[25] University of Washington eScience Institute. `http://escience.washington.edu/`.
[26] X. Yang, C. M. Procopiuc, and D. Srivastava. Summarizing relational databases. *Proc. VLDB Endow.*, 2(1):634–645, 2009.

APPENDIX

A. SMALL-SCALE SURVEY

To better understand how scientists use DBMSs today and, in particular, how they query these databases, we carried out a small-scale, informal, online survey. Our survey included 37 questions, mostly multiple-choice ones and took about 20 minutes to complete. We paid respondents $10 for their time. Seven scientists from three domains responded to our survey (four graduate students, one postdoc, and two research scientists); these scientists have worked with either astronomical, biological, or clinical databases.

Of interest to this paper, through this survey, we learned the following facts. All respondents had at least one year experience using DBMSs, while some had more than three years. Three scientists took a database course, whereas the other four were self-taught DBMS users. Four participants have been working with the same dataset for over a year, while three of them acquired new datasets in the past six months. The data sets ranged in size from less than one gigabyte (one user), to somewhere between one gigabyte and one terabyte (five users), to over a terabyte (one user). The reported database schemas include 3, 5, 7, 10, 30, and 100 tables. One respondent did not report the number of tables in his/her database. All respondents reported using a relational DBMS (some used other alternatives in addition).

More interestingly, three participants reported writing SQL queries longer than 10 lines, with one user reporting queries of over 100 lines! All users reported experiencing difficulties in authoring SQL queries. Two participants even reported often not knowing which tables held their desired data.

Five respondents reported asking others for assistance in composing SQL queries. All but one reported looking at other users' queries. Five participants reported looking "often" or even "always" at others' queries, whereas all participants either "often" or "always" look for sample queries online. Three participants mentioned sharing their queries on a weekly to monthly basis. Finally, all but one user save their own queries/scripts/programs primarily in text files and reuse them again to write new queries or analyze different data.

The findings of this informal survey thus indicate that many scientists could potentially benefit from tools to share and reuse past queries.

B. QUERY REPOSITORY DETAILS

B.1 Repository Schema

The Query Repository component stores the details of all the queries logged by the Query Logger, along with the features which appear in each query. The Query Repository contains the following three relations:

1. `Queries(id, timestamp, user, database name, query text, running time, output size)`
2. `Features (id, feature description, clause)`
3. `QueryFeatures(query, feature)`

The first relation, named `Queries`, stores the details of each logged query. The second relation, `Features`, consists of all the features that have been extracted from these queries. Note that feature descriptions are parameterized if there is some constant involved (e.g., the predicate `PhotoPrimary.objID = 55` is translated into the parameterized predicate `PhotoPrimary.objID = #`). Finally, the third table, `QueryFeatures`, maintains the information about which feature appears in which query.

In addition to these two tables, for the optimization of SSAccuracy, SnipSuggest's Query Logger also maintains the following tables:

1. `MarginalProbs(featureID, probability)`
2. `CondProbs(feature1, feature2, probability)`

The first table stores the marginal probability for each feature across the whole workload. The second contains the conditional probability of `feature1` given `feature2`, for every pair of features that have ever appeared together. We discuss how these tables are used below, in Appendix C.

B.2 Features Supported

The current implementation of SnipSuggest supports the following classes of features:

1. F_T^{from} for every table, view and table-valued function T in the database, representing whether T appears in the `FROM` clause of the query.
2. F_C^{select}, F_C^{where}, and $F_C^{groupby}$, for every column C in the database, representing whether this column appears in the `SELECT`, `WHERE`, or `GROUP BY` clause of the query, respectively.
3. $F_{aggr(C_1,\ldots C_n)}^{select}$, for every aggregate function and list of columns, representing whether this aggregate and list of columns appear in the `SELECT` clause.
4. $F_{C_1\ op\ C_2}^{where}$ for every pair of columns C_1, C_2, and every operator which appears in the database, representing whether this predicate appears in the `WHERE` clause of the query.
5. $F_{C\ op}^{where}$ for every column C in the database, and for every operator, representing whether there is a predicate of the form $C\ op\ constant$ in the `WHERE` clause of the query.
6. $F_{ALL}^{subquery}$, $F_{ANY}^{subquery}$, $F_{SOME}^{subquery}$, $F_{IN}^{subquery}$, and $F_{EXISTS}^{subquery}$ representing whether there is a subquery in the `WHERE` clause, of the form `ALL(subquery)`, `ANY(subquery)`, `SOME(subquery)`, `IN(subquery)`, `EXISTS(subquery)`, respectively.

C. SSACCURACY OPTIMIZATIONS

SnipSuggest materializes two relations to improve the SSAccuracy algorithm's recommendation time.

The first is $MarginalProbs$, which contains $P(f)$ for every feature f. When the user's partial query q is the empty query (i.e., $features(q) = \emptyset$), or if q consists of only features that have never before appeared in the workload, SnipSuggest can execute an `order by` query over $MarginalProbs$, instead of the more complex SQL in Figure 3. (When q contains only unseen features, SnipSuggest traverses up to the root vertex since it is the largest subset of q that appears in the workload. So, SnipSuggest makes its suggestions for $\emptyset$, and thus exploits $MarginalProbs$.)

The second is $CondProbs$, which contains the conditional probability $P(f_1|f_2)$ for every pair of features f_1, f_2. It is indexed on the f_2 column. It is leveraged when the user's partial query q contains just one feature f, or it contains multiple features, but only one feature f has appeared in the workload. In these cases, SnipSuggest can execute a simple query over $CondProbs$ with filter $f_2 = f$, and order by the conditional probability, instead of executing the slower SQL in Figure 3.

D. Max-Query-Coverage PROBLEM

In this section, we show that the Max-Query-Coverage problem is NP-hard, and that the SSCoverage algorithm is the best possible approximation algorithm for it (up to lower order terms).

Remember, the Max-Query-Coverage problem, as presented in Section 3.2.3, is defined as follows. Given a workload DAG G and a partial query q, recommend a set of k outgoing edges, $e_1, \ldots, e_k$, from q that maximizes

$$P(addlFeature(e_1) \vee \ldots \vee addlFeature(e_k)|q)$$

For shorthand, denote by f_i the feature $addlFeature(e_i)$. To make our next step easier, we want to show that the features $f_1, \ldots, f_k$ that maximize the formula above are the ones that maximize the number of queries covered, under the q vertex.

Consider $P(f_1 \vee \ldots \vee f_k|q)$. If we select some random query whose feature set is a superset of $features(q)$ (i.e. any query in $potential\_goals(q)$), then this is the probability that it also has feature $f_1, f_2, \ldots$, or f_k. So, $P(f_1 \vee \ldots \vee f_k|q)$ can be written as:

$$\sum_{u \in potential\_goals(q)} Pr(f_1 \vee \ldots \vee f_k|q \wedge u) \cdot Pr(u|q)$$

$Pr(f_1 \vee \ldots \vee f_k|q \wedge u)$ is 1 if u contains $f_1, f_2, \ldots$, or f_k, and 0 otherwise (because q does not contain any of $f_1, \ldots, f_k$ either).

Hence this is equal to:

$$\sum_{u \in potential\_goals(q):(f_1 \in u \vee \ldots \vee f_k \in u)} Pr(u|q)$$

Therefore, we can now rewrite the Max-Query-Coverage problem to maximize:

$$\sum_{v \in U} P(v|q),\ where\ U = \bigcup_{i=1}^{k} potential\_goals(q \cup \{f_i\})$$

Next, we define the Maximum Coverage Problem, which is known to be NP-hard [12].

DEFINITION 11. *Given a set of elements U, a number k and a set of sets $S = S_1, \ldots S_n$, where each $S_i \subseteq U$, the* **maximum coverage problem** *is to find a subset of sets $S' \subseteq S$ such that $|S'| \leq k$ and the following is maximized:*

$$|\bigcup_{S_i \in S'} S_i|$$

i.e., the number of elements covered is maximized.

In his paper [12], Feige proves the following theorem.

THEOREM D.1. *The maximum coverage problem is NP-hard to approximate within a factor of $1 - \frac{1}{e} + \epsilon$, for any $\epsilon > 0$.*

Moreover, the greedy algorithm achieves an approximation factor of $1 - \frac{1}{e}$. The analysis of the greedy algorithm was well-known, and is reproved in Feige's paper [12].

We prove our results by showing Max-Query-Coverage's equivalence to the Maximum Coverage Problem.

THEOREM D.2. *The Max-Query-Coverage is equivalent to the Maximum Coverage problem.*

In particular, there are polynomial time reductions between the two problems, which preserve the values of all solutions. With this theorem, and the known results described above, we can conclude that Max-Query-Coverage is NP-hard to approximate within any factor substantively better than $1-\frac{1}{e}$, and that SnipSuggest's greedy algorithm, SSCoverage, achieves this bound.

PROOF OF THEOREM D.2. We show two mappings. First, we show that an instance of the Maximum Coverage Problem can be mapped to an instance of the Max-Query-Coverage problem, and that there is a bijection between the set of solutions to each. Second, we show the reverse.

Consider an instance I_1 of the Maximum Coverage Problem, where U is the set of elements and $S = S_1, \ldots, S_m$ is the collection of sets. We translate I_1 into an instance I_2 of the Max-Query-Coverage problem as follows. U is the set of queries, S is the set of features. Denote by f_T the feature that corresponds to the set T. T represents the set of queries which contain the feature f_T. For a given element/query $q \in U$, $features(q) = \{f_T : q \in T\}$. The input partial query is the empty query, and so the problem is to suggest the top-k features given the empty query. Next, we show that there is a bijection between the solutions for I_1 and I_2.

Given any solution $S' = \{S'_1, \ldots, S'_k\}$ to I_1 (not necessarily an optimal solution), the equivalent solution in I_2 is to suggest the features $F' = \{f_{S'_1}, \ldots, f_{S'_k}\}$. Of course, conversely, given a solution $F' = \{f'_1, \ldots, f'_k\}$ to I_2, the equivalent in I_1 is $S' = \{S'_i : f'_{S_i} \in F'\}$. Clearly, the number of elements of U covered by S' is the same as the number of queries covered by F' (which is the mass covered by F' multiplied by $|W|$).

Now, consider an instance I_1 of the Max-Query-Coverage problem, where the query workload is W, the set of all features is F, and the partial query is q. We translate this into the Maximum Coverage Problem as follows. Scan through W to find $potential\_goals(q)$ (i.e., all the blue vertices under q). For each potential goal q', if it is a leaf node, add $P(q') \times |W|$ elements to U. If it is not a leaf, add $(P(q') - \sum_{f \in F} P(q' \cup \{f\})) \times |W|$ elements. This represents the number of queries in the workload that have exactly this set of features. Note that the number of elements added to U is at most the number of queries in W. Add to S, one set per feature $f \in F$, consisting of all queries that contain f. Let's denote this by $queries(f)$.

Given any solution $F' = \{f_1, \ldots, f_k\}$ to I_1 (not necessarily optimal), the equivalent solution in I_2 is to select the sets $S' = \{queries(f_1), \ldots, queries(f_k)\}$. Conversely, a solution $S' = \{S'_1, \ldots, S'_k\}$ to I_2 can be translated to the following solution for I_1: $F' = \{f'_i : queries(f'_i) = S'_i\}$. If we consider the number of queries covered by F', this is equal to the number of elements of U covered by S'.

We have shown that we can translate an instance of the Maximum Coverage Problem into an instance of the Max-Query-Coverage problem, and the reverse, in time polynomial in $|W|$. We've also shown that given an instance of one problem, and its corresponding instance in the other, there is a bijection between the solutions for the two instances. □

E. QUERY ELIMINATION: DETAILS

In this section, we describe the segmentation and stitching algorithms in more detail. The two algorithms extract query sessions from the query log, then the Query Eliminator eliminates all queries except those that appear at the end of a session. Using this technique, the Eliminator aims to dispose of a large fraction of the queries, while maintaining the high-quality ones.

The goal of the **segmentation phase** is to take a pair of *consecutive* queries P, Q, and decide whether the two queries belong to the same query session or not. The algorithm proceeds in three steps. First, it constructs the Abstract Syntax Tree (AST) for each query and transforms it into canonical form, which includes, for example, removing any constants, and alphabetically ordering the list of tables in the FROM clause (this process is also used by the Query Logger). Second, it extracts a set of segmentation features from P and Q as well as extra information such as timestamps of queries and the query output of the preceding query P. Unlike SnipSuggest features, the segmentation features capture the difference between two queries. Some examples include the time interval between the queries, the cosine similarities between their different clauses, and the relationship between ASTs. In the third step, using these segmentation features, our technique uses a perceptron-based classification algorithm to decide whether the queries belong in the same segment. For this final step, as training data, SnipSuggest requires some labeled data in the form of queries labeled with the identifier of the task that the queries are intended for.

The most significant segmentation feature is *AST inclusion type*. This feature represents whether the relationship between the two queries' ASTs is the **Same**, **Add**, **Delete**, **Merge**, **Extract** or **None**. This feature captures the following intuition. Within a query segment, the user incrementally adds or removes terms from the query after seeing the query result of the previous query. Such incremental edits are captured by the values **Same**, **Add**, or **Delete**. Occasionally, the user may introduce a subquery written in the past or copied from some sample query to compose a more complex query. The user may also pull-out a subquery to debug or analyze unexpected results. In both these cases, the user may start to work towards a different purpose when the change involves subqueries; this signals a new query segment. **Merge** and **Extract** captures such changes involving subqueries. Table 2 summarizes these five AST inclusion types.

Although the *AST inclusion type* may be a strong indicator of continuing or breaking a query segment, it does not capture the amount of change. Thus, the other segmentation features that we described above, which capture the degree of change, are necessary for better accuracy.

Label	Description	Expect a new segment?
Same	Q is canonically the same as Q	No
Add	Q is based on P and has more terms	No
Delete	Q is based on P and has fewer terms	No
Merge	P is a subquery of Q	Yes
Extract	Q is a subquery of P	Yes
None	All other types of changes	Unknown

Table 2: Summary of AST inclusion type. Each feature value has an expected decision on segmentation discussed in the text. P and Q denote preceding query and following query, respectively.

Query segments are useful because of the coherency among the queries in the same segment. However, in order to extract more complete query sessions, SnipSuggest often needs to stitch together multiple segments.

The **stitching phase** tests whether two segments create a single revision cycle, smooth transitions via small changes, when they are concatenated in time order. To perform stitching, SnipSuggest iterates over each segment s in the input and all its time-wise successor segments. It runs the core of the segmentation algorithm between the last query of s and the first query of a time-wise successor segment t, with a modification. Since SnipSuggest is now considering segments that are separated by multiple segments, the time interval between the queries becomes less meaningful. Thus, the algorithm treats the time interval as a missing attribute. Then, if the segmentation algorithm outputs that the last query of s and the first query of t belong in the same segment and this does not contradict to value of *AST inclusion type*, SnipSuggest concatenates the two segments.

In Appendix F.2, we show that the Query Eliminator is able to accurately perform segmentation and stitching for a workload of queries written over the movie database.

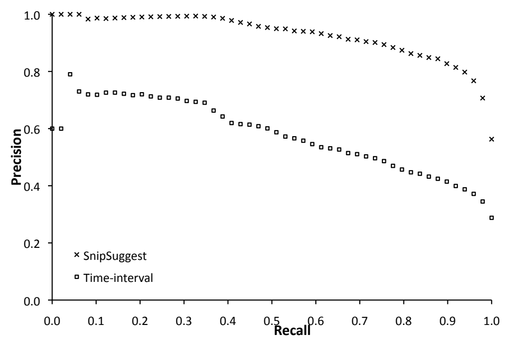


Figure 8: Average precision-recall for the Query Eliminator algorithm versus the time interval based algorithm.

F. EVALUATION OVER IMDB DATASET

F.1 Recommendation Quality

First, for SnipSuggest's recommendation quality over this dataset, we saw similar patterns to the SDSS dataset. Namely, the SnipSuggest approach outperforms the other approaches (e.g., by 5 to 20% in average precision in the top-3, in comparison to the Popularity approach), and the recommendation quality improves as the user gives more information (e.g., if there are no tables in the `FROM` clause, SnipSuggest is able to recommend a correct table with 0.62 average precision in the top 2, which increases to 0.91 after a table is added). Although, the general trends still hold, the benefit of the SnipSuggest approach is less significant in the IMDB dataset simply due to the magnitude difference in the schema size (and thus the number of possible features). This schema consists of only five tables and 21 columns.

F.2 Query Eliminator Accuracy

We study the session extraction accuracy for only the IMDB dataset. We were able to manually label the session information for this dataset because we have the ground truth in the form of problem numbers from the course assignment. In other words, we can determine which problems (from the assignment) that queries are written for.

F.2.0.1 Segmentation.

To quantify the segmentation algorithm's performance, we measure the precision and recall with which it identifies segment boundaries. We compare its performance against using only the time interval between queries. The time interval technique is a common method for extracting sessions from web search logs [5, 10, 11, 18]. The procedure is to set a threshold for the time interval, say 30 minutes, and consider a query to be part of a new session if the time interval between it and the last query is more than the threshold.

Figure 8 shows the average precision-recall for the two different segmentation algorithms. We show the average results across 10-fold cross validation. Overall, for SnipSuggest's approach, the area under the curve is 0.930. It is only 0.580 for the time-interval based technique. Our approach significantly outperforms the time-based segmentation technique.

F.2.0.2 Stitching.

As mentioned above, the most significant segmentation feature for session extraction, is what we call the *AST inclusion type*. This feature represents whether the relationship between the two queries'

Threshold	μ	$\mu+\sigma$	$\mu+2\sigma$	∞
Precision	0.707	0.656	0.628	0.578
Recall	0.801	0.893	0.934	1.000
F-measure	0.751	0.756	0.751	0.733
# of Edges	714	896	993	1165

Table 3: Precision-recall of stitched edges for different thresholds.

ASTs is the **Same**, **Add**, **Delete**, **Merge**, **Extract** or **None** (as defined in Table 2). We said that two queries are in the same session if this feature has a value of **Same**, **Add**, or **Delete**, with some threshold on the amount of change. We examine how this threshold can affect the performance of the stitching algorithm. Table 3 shows the results. μ is the mean difference amount in the training data, among those queries that lie on a session boundary, while σ is the standard deviation. We see that, as expected, smaller thresholds yield better precision while larger thresholds yield better recall. The F-measure, however, remains approximately constant. The mean threshold already recalls 80% of session boundaries, while achieving a precision of 70.7%.

G. EXAMPLE OF AVERAGE PRECISION

In this section, we present a toy example of the Average Precision measure in action. The aim is to give the reader a better understanding of Average Precision.

Suppose that Carol has an empty query, and has requested the top-5 suggestions for the `FROM` clause. Her intended query includes two features in the `FROM` clause: `PhotoPrimary` and `fGetNearbyObjEq()`. Suppose that SnipSuggest has suggested the following snippets, in this order: `PhotoPrimary`, `SpecObjAll`, `fGetNearbyObjEq()`, `PhotoObjAll`, `Columns`.

Consider the following table, which will help us calculate the Average Precision of these suggestions.

Rank	Suggestion	Correct?	Precision
1	`PhotoPrimary`	true	1.0
2	`SpecObjAll`	false	0.5
3	`fGetNearbyObjEq()`	true	0.67
4	`PhotoObjAll`	false	0.5
5	`Columns`	false	0.4

The $Precision$ column at rank i indicates the precision of the first i suggestions. With this table, we collect the ranks where the suggestion is correct. In this example, these are ranks 1 and 3. Then, we take the sum of the precisions at these ranks and divide by two (i.e., the number of ground truth features). For this example, the Average Precision is $\frac{(1.0+0.67)}{2} = 0.83$. Note how this measure rewards correct suggestions that appear early in the ranking more than those that appear later. For example, if `fGetNearbyObjEq()` had appeared second in the ranking, then the average precision would be $\frac{(1.0+1.0)}{2} = 1.0$. Whereas if `fGetNearbyObjEq()` appeared fourth in the ranking, the Average Precision would be $\frac{(1.0+0.5)}{2} = 0.75$.

Now, suppose Carol's intended query also includes the `RunQA` table, but that SnipSuggest still suggests the recommendations listed above. In this case, the Average Precision drops down to $\frac{(1.0+0.67)}{3} = 0.56$. We divide by 3 (instead of 2) because the number of ground truth features is now 3.