SnipSuggest: Context-Aware Autocompletion for SQL

An implementation on mySQL

Ismini Bouliari
Department of Informatics
National University of Athens
i.bouliari@gmail.com

Stavroula Eleftheraki
Department of Informatics
National University of Athens
selefth@yahoo.com

ABSTRACT

In this project we are going to implement the SnipSuggest algorithm introduced in 2010 by Khoussainova et al. (1). To achieve this, we decide to create it as layer on top of mySQL database using Python.

SnipSuggest is a system that helps the user complete an SQL statement based on prior knowledge gained from past queries. Given a partial query, SnipSuggests makes recommendations about possible feature additions in order to help the user take the information she wants from the database.

Our current implementation offers suggestions for the following clauses: select, from, where and group by. For evaluation we use log data from the Sloan Digital Sky Survey database and we compare SnipSuggest with other two (2) methods. The state of the art algorithm, Popularity based and the naïve method Random.

KEYWORDS

SnipSuggest, context-aware recommendations, SQL snippets, SSaccuracy, SScoverage, mySQL, w2vec, recommendation, context-aware recommendations

1. SnipSuggest overview

SnipSuggest, is a SQL autocomplete system that can make recommendations for the non- expert user to make more efficient queries. The basic concept is the following:

When the user poses a query to the database, she can request suggestion for what predicates to add to this query for a specific clause. For example, giving a partial query for the select clause, she can ask which tables, views of table functions to add to the query or in another case, giving the from clause, i.e. she can ask some suggestions for the select clause and so on.

The difference between SnipSuggest and other recommender systems, the base on which the "context-aware" characterization lies, is that it takes into consideration the typed query. As the query becomes more complete, the system has more information on what suggestions to make next. Instead of making suggestions based on popularity or validity, it extracts information from similar past queries retrieved by other users' activity. In this fashion, SnipSuggest can make accurate suggestions, even for rarely used tables or predicates.

SnipSuggest consists of three (3) components:

1. Query Logger: logs the queries posed against the database by the users.
2. Query Repository: features of the queries are stored here by the Query Logger in the form of relations. As features we consider parts of the SQL statement such as the predicates for the SELECT, WHERE and GROUP BY clauses, or the table, view and table function names for the FROM clause. The Query Repository holds five (5) relations:

    Queries: It is populated from standard loggers offered by DBMSs. It contains information about the queries such as ID, timestamp, the statement itself etc.

    Features: It contains each feature (predicate or table name) with its id and the clause it appears in (SELECT, FROM, WHERE, GROUP BY).

    QueryFeatures: It is the link between the two aforementioned tables. It lists which features appear in which queries based on ids.

    MarginalProbs: It holds the ids of all the features associated with a probability which shows how wanted these features are from the users.

    CondProbs: It holds the conditional probability of a feature to be added to a query according to the presence of a second feature. Moreover, it includes the ids of the involved features.

3. Query Eliminator: It periodically eliminates all queries in order to reduce the workload, by keeping only the ones at the end of a query session[1] which -by intuition- is probably of higher quality. The elimination of the query log happens in two phases: segmentation and stitching. First, during the

[1] A query session is a sequence of queries written by the same user as part of a single task.

segmentation phase, the eliminator detects new revision cycles[2] by finding differences between consecutive queries and labels query segments, i.e. the queries that belong to the previous session from the start of the one currently detected. Next, during the stitching phase, query segments that belong to a larger revision cycle – and may come from queries left unfinished at some point and then resumed by the user- are

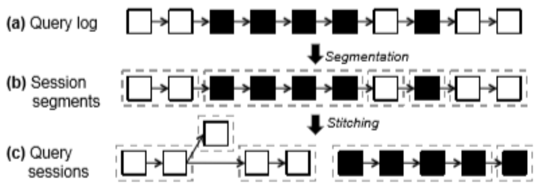


Figure 1 Query Eliminator

concatenated in order to form the whole query session.

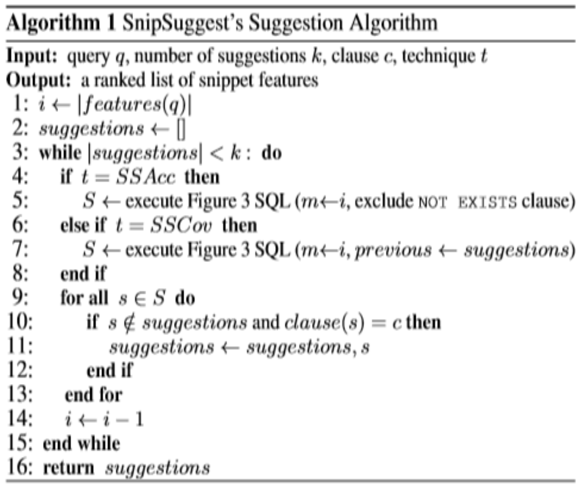
2. Context-Aware Algorithms

Recommendations are based on two (2) techniques, these are SSaccuracy and SScoverage. The first one tries to give relevant suggestions based on the partial query the user gives while the second one tries the same but with the difference that in the same time it tries to give suggestions that are as much diverse as possible with each other.

For the first round of the SnipSuggest Suggestion Algorithm the suggestions will be made using the SSaccuracy technique despite what the user has chosen as technique. That is because SScoverage needs at least one suggestion to propose more.

To optimize the SSaccuracy algorithm the relations MarginalProbs and CondProbs in the query repository are used. On the off chance that the query's features have appeared together in past questions, SSAccuracy can efficiently identify the features with the highest conditional probabilities, posing only one SQL query over the QueryFeatures table. In the case a feature has never appeared in a query before, the algorithm suggests the feature with the highest marginal probability.

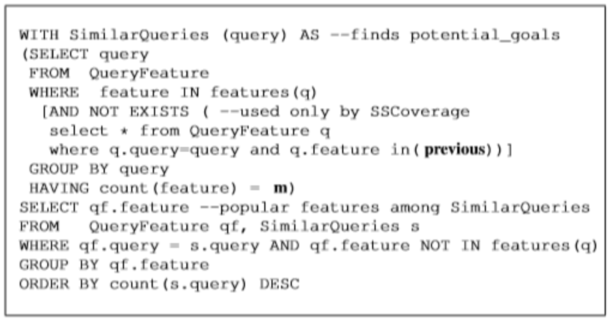
[2] A revision cycle is the iterative process of refining and resubmitting queries until a desired task is complete.



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 SQL ($m \leftarrow i$, exclude NOT EXISTS clause)
6:     else if $t = SSCov$ then
7:         $S \leftarrow$ execute Figure 3 SQL ($m \leftarrow i, previous \leftarrow suggestions$)
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 $suggestions$
```

Figure 2 SnipSuggest algorithm



```
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 SnipSuggest Algorithm SQL Query

3. Evaluation

We evaluate the quality of the SnipSuggest recommendations using average precision. This measure calculates the accuracy after each correct snippet is incorporated in the query, and afterward takes their average. In the event that a correct snippet is excluded from the top-k, its precision is 0. The advantage of average precision is that it rewards the techniques that put the correct snippets near the top of the list.

SnipSuggest returns a ranked list of snippets, Lq, for query q.

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

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

4. Overview of the core classes and implementation

In order to evaluate the performance of SnipSuggest our implementation we used log data[3] from the Sloan Digital Sky Survey (SDSS). SDSS is an effort of mapping the sky, that is making observations through telescopes and storing measures of the positions and properties of the observable celestial objects.

The SDSS database is publicly available and offers the ability to pose SQL queries against it. As a result, millions of log data about these queries are preserved into the database and accessible as well. The SDSS schema is very complex having 88 tables, 51 views, 204 user-defined functions, and 3440 columns.

In order to simplify things and due to computational limitations in our implementation we used log data from SDSS Data Release 14 (DR14) of the period January-June 2019. This data came as a result of the following SQL query submitted to the SkyServer Weblog SQL Search dialog box on the 30th of June 2019:

```
SELECT theTime, clientIP, dbname, statement,
elapsed, rows

FROM SqlLOG

WHERE yy = 2019 and dbname = "BestDR14_SSD"
and rows > 0

ORDER BY clientIP, mm, dd, hh, mi,ss
```

From this, **569.351 rows** were returned (Note: we asked only for logs of queries actually gave at least one result -- rows > 0).

Data cleaning – Query elimination

After gathering our raw data, our first task was to clean them. We first implemented the Query Eliminator (see file eliminator.py).

We first split the queries into 4 parts, the ones separated by the 4 clauses SELECT, FROM, WHERE, GROUP BY. Next, we removed all newline characters from the statements so as to bring all the queries to the same format (cleaner1). We replaced all numeric values that appeared in queries with "#" (cleaner2). Furthermore, we observed that in our data existed queries with features not yet supported by SnipSuggest, such as joins, or features that it cannot process since they do not belong into the database relations (ex.: website links) and so we had to remove them as well (cleaner3).

Entering the main elimination phase, as described in the previous section, we segmented the queries into sessions based on clientIP and theTime columns and used the Word2Vec model for vectorization. Based on the cosine similarity of those vectors we find and delete all the similar intermediate queries coming from the same user during the same session, keeping this way only the final one for each query session (cleaner 4). Lastly, we assigned an id to each query (*qid*).

We also brought the time in 24h format in order to create indexing in mySQL.

After the elimination phase, the remaining, clean data that constitute our workload were **522 rows.**

Query Repository

These 522 rows are the database's main relation, the table ***Queries (qid, theTime, clientIP, dbname, statement, elapsed, rows)***.

Now that the data are clean, we need to construct the rest of the relations for our toy database (see query_repo.py). For ***Features*** we tokenize the clauses of the queries, by extracting the elements that lie between the possible pairs, cleaning them and putting them into lists. For example, for the SELECT clause we take the elements between the SELECT and FROM clauses, remove TOP # or DISTINCT in case they exist in the query. This is because these clauses are always used with SELECT and as a result, they do not give any extra help for making suggestions. Next, we split the extracted elements, strip from redundant spaces and take the proper predicates (function SelectFeatures). We get the features for the rest of the clauses (FROM, WHERE, GROUP BY) by following procedures of the same logic. We keep only the unique values. Lastly, we assign an id (*fid*) to every feature save them into a table of the form***: Features (fid, feature_description, clause)*** i.e., we now know the clause with which each feature has appeared in the queries at least once.

Finally, we construct the ***QueryFeatures (query, feature)*** that connects the previous two tables and holds the information of qid's and fid's, allowing us to know which feature comes from which query (and subsequently all the other information contained in the two tables).

In order to create the **MarginalProbs** and **CondProbs** tables, we use the following equations respectively:

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

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

[3] http://skyserver.sdss.org/log/en/traffic/sql.asp

Database Creation in MySQL framework

Using standard Python commands, we connect to MySQL and create the SDSS toy database. We create the tables described in the Query Repository and load the data from the .csv files we stored them in after their creation.

Next, we downloaded actual SDSS data: we came across nine-teen (19) unique features corresponding to the FROM clause in the Features table we created earlier and downloaded a fraction of the corresponding table or view and more precisely, 500 top rows from each one. All data were retrieved via a simple SQL query against SDSS as the one below, corresponding to the SpecObjAll table:

```
SELECT TOP 500 sp.*

FROM SpecObjAll as sp
```

We imported these table to MySQL 8.0 and we consulted the SDSS Schema in order to create our toy schema (indices, keys, etc). We have to point out that SDSS views were loaded into our database as individual tables and were treated as such, due to time limitations.

Implemetation

In the *main.py* file, we build an interface where we demonstrate SnipSuggest. By running this file, the user can ask from SnipSuggest for suggestions on which features she should add to her query. SnipSuggest makes suggestions based on past experience as it is depicted in the relations which consist our workload.

For example, let's assume we write the partial query:

```
SELECT * FROM Photoprimary
```

Then, SnipSuggest asks for which clause we need suggestions, the desired number of suggestions k and the method we wish to use, SSAccuracy or SSCoverage. Assuming we want 5 suggestions for predicates of the SELECT clause, using, say, the SSAccuracy method, the algorithm locates the feature id (*fid*) of 'Photoprimary' in the Features table to use it as prior knowledge. This *fid* is located in the QueryFeatures table and all query id's (*qid*) that belong to the specific *fid* are retrieved. Back to the Features table, the *qid*'s are located and the *feature_description* is returned for each one of them. Lastly, from all the candidate snippets, the top 5 are returned to the user.

In the *SnipSuggest.py* file we implement algorithm 1 (Figure 2) along with auxiliary functions that help us build it. There, one can find the *SSAccuracy, SSCoverage*.

Furthermore, in the *main.py* we use the functions *Random* and *popularity_based* in order to get the recommendations of the random and popularity-based recommender respectively and compare SnipSuggest recommendations against them. These functions can be found in the *evaluation.py* file where we implement functions that will help us evaluate SnipSuggest in a series of experiments that come next. First of all, we create *average_precision* function that will help us compare the results of the 3 techniques. We also create *remove_features and features_knowledge*, functions that help us set up artificial ground truth in order to test the algorithms' accuracy against it.

5. Experiments (graphs and explanations)

Apart from this interface for the user, we also ran some experiments in order to test the efficiency and accuracy of SnipSuggest. We collected all the log data of the next month for test data in the same way as when creating our workload using the following query:

```
SELECT statement

FROM SqlLog

WHERE yy=2019 and mm=7 and
dbname="BestDR14_SSD" and rows>0
```

submitted on the 20th of July 2019.

Using the *test_data_producer.py*, we create two (2) csv files which contain all the queries that have at least three (3) attributes at the SELECT and FROM clauses. The code to create the graphs can be found in the *experiments.py*. In the following figures we compare SnipSuggest with Popularity based and Random.

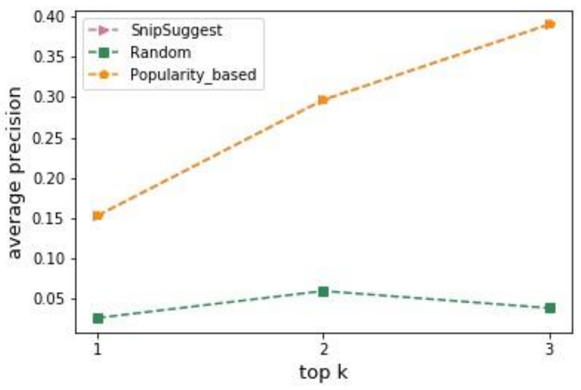
In figures (a) - (c), SnipSuggest offers suggestions for the SELECT clause using the SSaccuracy technique. In the first figure we see how the algorithms react in an empty query from a user which asks recommendations for the FROM clause. As someone can notice the suggestions for both SnipSuggest and Popularity based are the same since they both use the MarginalProbs relation to response.

For figure (b), we removed three (3) attributes in the select clause while in (c) we removed four (4) attributes. In both cases we use the remain predicates as prior knowledge. As a result, SnipSuggest outperforms all the other algorithms.

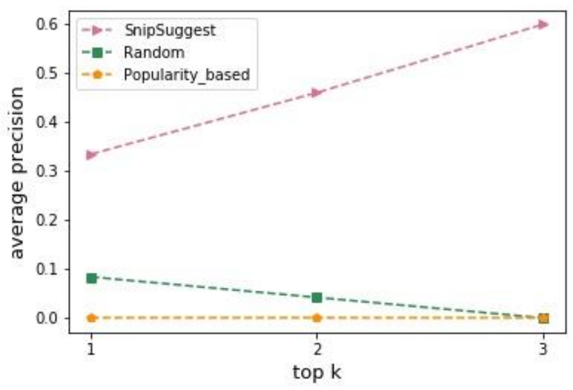
In figures (d) and (e), we have the suggestions for the FROM clause after removing two (2) and three (3) relations respectively. Again, SnipSuggest gives higher accuracy than the other algorithms. We notice here that the popularity based algorithm has the worst performance and that is because of our workload.

The last figure shows the performance of SSaccuracy and SScoverage in the test data. For this plot, we removed all the attributes from the SELECT clause and kept the rest of the queries as knowledge. We cannot notice any difference in the techniques. Here the *average precision* is not used because of the nature of

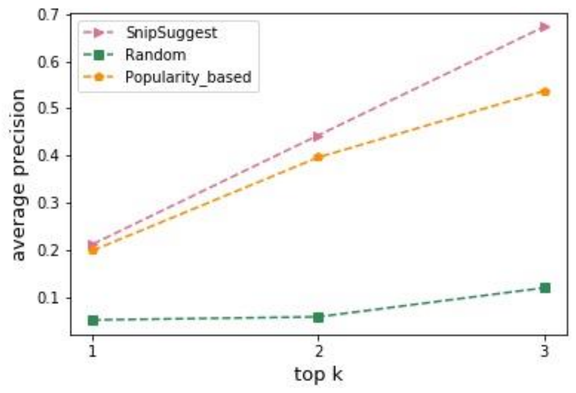
SScoverage that tries to diversify the suggestions. Instead, we use utility which gives 1 if there exists at least one (1) correct suggestion inside the suggestions list and zero (0) otherwise.



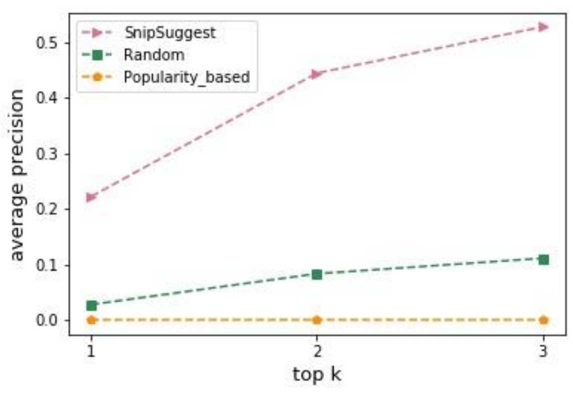
(a)Empty query



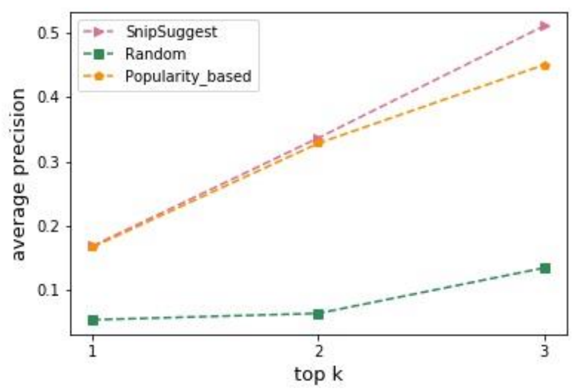
(d) Two relations, From clause



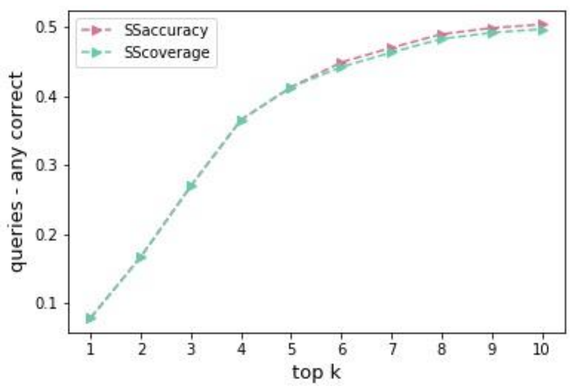
(b)Three attributes, SELECT clause



(e) Three relations, From clause



(c) Four attributes, SELECT clause



(f) SSAvsSSC

6. Conclusion

Our workload and test data were poor from WHERE and GROUP BY clauses and thus we could not make experiments suitable for visualization purposes. Moreover, the choice of the data that consist the workload have to be chosen carefully in order to be used for prediction. A machine learning approach for that could be useful. Despite these, it is clear that SnipSuggest algorithm outperforms all the other approaches.

REFERENCES

[1] Khoussainova, N., Kwon, Y., Balazinska, M. and Suciu, D., 2010. SnipSuggest: Context-aware autocompletion for SQL. Proceedings of the VLDB Endowment, 4(1), pp.22-33.