More Like This: Query Recommendations for SQL

Christopher Miles
Department of Computer Science and Engineering
University of Washington, Seattle, WA, USA
cmiles@cs.washington.edu

1. Introduction

The Sloan Digital Sky Survey (SDSS) is a success story for modern data analytics. Exposing over 30 TB of data on approximately 500 million celestial bodies via SQL, SDSS is a widely used resource within the astronomical community [4]. Despite its ubiquity, the learning curve is steep. Empirical data suggests that scientists have little trouble grasping the basic SELECT-FROM-WHERE paradigm of SQL, but have difficulty mastering more advanced concepts, such as nested queries, aggregates, and user-defined functions [3]. Complex database schemas further exacerbate their task. SDSS, for instance, comprises over 90 tables, 200 user-defined functions, and 3,400 columns [4]. In light of these challenges, scientists often compose their SQL queries by locating and iteratively refining queries shared by other users. In some cases, these queries can be modified to achieve the desired result. In others, the user reaches a dead end, unsure of how to proceed. In part, the problem is one of discovery: how can a user go from what is known to what is unknown?

Early work in this area focused on identifying *important* attributes and tables in the database, and suggesting extensions to a user's query based on this information [5]. A fundamental limitation of such importance-based approaches is that they are context insensitive: the importance of a particular attribute or table is independent of the user's intended task. Moreover, it is rarely the case that a user has difficulty identifying the most important attributes and tables in the database. Rather, query assistance is most valuable in the case of infrequently used tables and attributes.

More recent work has sought to address these limitations by producing context-aware suggestions based on partial information entered by the user. Recommendations are drawn from similar past queries authored by other users of the database, thereby leveraging a growing, shared body of experience [1,2].

The intuition behind QueRIE, a collaborative filtering based approach to query recommendation, is that if a pair of users U_A and U_B have similar querying behavior, they likely share an interest in the same data [2]. As a result, the past queries of U_B can serve as a guide for U_A. Users are represented as item vectors, with each element corresponding to the user's preference for a particular tuple. $U_i = \{r_1, r_2, \ldots, r_n\}$. Similarity between a pair of users is computed using a vector similarity metric such as cosine similarity. Given a set of users $U = \{U_1, U_2, \ldots, U_M\}$ and a specific user U_x, a ranked list of the top k most similar users is obtained by computing pairwise similarities between U_x and each element of U. The recommended queries are drawn from the set of queries associated with the top k most similar users.

In contrast to QueRIE, which recommends full SQL queries, SnipSuggest assists users in composing complex queries by recommending a set of context-aware additions to a specific clause in a partially stated SQL query [1]. Queries are modeled as collections of abstract features. A directed, acyclic graph representing the space of queries is constructed by adding a vertex for every possible set of features and an edge between vertices whose feature sets differ by only one element. The user's partial query is mapped onto a particular vertex in the graph.

Descendants of the vertex are referred to as potential goals, as they approximate the user's intended query. The recommendation problem is then to rank the outgoing edges for the vertex that corresponds to the user's partially written query.

SnipSuggest maintains the history of past queries in a set of 3 tables collectively referred to as the Query Repository. The schemas for these tables are shown in Listing 1. The Queries table stores metadata about a query (e.g. timestamp, username), as well as the query's text. The Features table contains metadata about a feature, including the clause from which it was extracted and that feature's abstracted representation. Feature definition, extraction, and representation are discussed in further detail in Sections 2.2-2.4. Finally, the QueryFeatures table associates features with the queries in which they occur.

```
Queries(qid, ts, username, database, query)
Features(fid, clause, expression, snippet)
QueryFeatures(qid, fid)
```

Listing 1. Table definitions for SnipSuggest's Query Repository. Primary keys are underlined.

In this paper, I present an alternative approach, More Like This (MLT), to the problem of full-text query recommendation that builds upon concepts introduced in SnipSuggest. Given an input query Q, MLT searches the Query Repository for the top k most similar queries to Q. Per SnipSuggest, queries are modeled as collections of abstracted features. Similarity between a pair of queries is defined in terms of the term-frequency—inverse document frequency (tf-idf) of their respective feature sets. A subjective, per-clause weight can optionally be applied. A more formal treatment of query similarity can be found in Section 2.5.

The recommendations produced by MLT are quantitatively evaluated on a random sample of 10,000 queries logged by the SDSS server from 2002 to 2009. Syntactically invalid queries and those that contain proprietary SQL features are removed. Since the queries lack metadata, attributing a query to a specific user is impossible. As a result, a simple heuristic based on edit distance was devised for identifying pairs of queries predicted to have been executed consecutively by the same user. Recommendation quality is assessed for such query pairs (Q_i, Q_{i+1}) by determining the number of times the top k recommended queries contain Q_{i+1}, given Q_i as input. Additional details on evaluation can be found in Section 3.

The primary contributions of this paper are:

1. A recursive method for representing arbitrarily complex SQL expressions as abstract feature strings. This representation captures the overall structure of an expression without getting bogged down in unnecessary detail. (Section 2.3)
2. A similarity function based on tf-idf for assessing the similarity between a pair of queries represented as collections of abstract features. The function can optionally be extended with subjective, per-clause weights. (Section 2.4)
3. A formal definition of recommendation diversity and an efficient algorithm for selecting an optimal solution to the query recommendation problem. (Section 2.4.4)
4. An approach for quantitatively evaluating the performance of MLT in the absence of user-query attribution. (Section 3)

2. Methods

2.1 Parsing SQL

The primary responsibility of the parser is to determine the grammatical structure of its input and, if well formed, construct a data structure containing an abstract representation of that input (e.g. parse tree). This is achieved by determining how the input is derived from the start symbol of the formal grammar defining the language. The two basic approaches for accomplishing this task are top-down parsing and bottom-up parsing [9].

SQL is a particularly challenging language to parse. Despite the fact that it was adopted as a standard by the American National Standards Institute (ANSI) in 1986 and International Organization for Standardization (ISO) in 1987, formal grammars that reliably operate across multiple database implementations are rare. One reason for this is that SQL allows a tremendous amount of flexibility at the expression level. Although this likely contributed to SQL's popularity, it complicates the design of parsers significantly. Additionally, both commercial and open source databases support proprietary extensions to the language, further complicating parser construction.

As a result of these challenges, researchers often rely on parsers that support only a subset of the SQL language. Moreover, these parsers are typically selected for pragmatic reasons (e.g. programming language support). Accordingly, I am using the open source Zql parser in MLT [8]. Technically speaking, Zql performs no parsing itself; rather, it delegates responsibility for interpreting the input text to an internal LL(k) parser generated by JavaCC [10]. Zql supports a reasonable subset of the SQL standard, but notably lacks support for table-valued functions and the JOIN, LEFT JOIN, RIGHT JOIN, and FULL JOIN keywords.

Though adequate for the needs of this project, Zql's API suffers from an ill-conceived type hierarchy. In no fewer than 20 places in the MLT source code, the Java instanceof operator is invoked to determine the type of a parsed object at runtime. Program control flow forks depending on the result. The ZStatement class, from which all SQL statements derive, illustrates this point nicely. Listing 2 shows the ZStatement class in its entirety.

```
package org.gibello.zql;

/**
 * ZStatement: a common interface for all SQL statements.
 */
public interface ZStatement extends java.io.Serializable {
};
```

Listing 2. Zql's ZStatement interface.

At issue here is the fact that the interface provides no ability to distinguish between statements representing inserts, deletes, updates, transactions, or queries. Furthermore, in order to make use of a ZStatement instance in *any way whatsoever*, it must be cast to the appropriate type (ZDelete, ZInsert, ZUpdate, ZTransactStmt, ZQuery). An analogous issue is encountered with the ZExp class, from which all SQL expressions derive.

2.2 Feature Definition

We borrow from SnipSuggest the definitions of a feature and a feature set.

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.

Definition 2. The feature set of a query q is defined as: features(q) = {f | f(q) = true}

Features are extracted from the FROM, SELECT, WHERE, ORDER BY, GROUP BY, HAVING, and DISTINCT clauses of a query (or nested query). Features have form:

<clause>-<expression>

For example, if attribute A from relation R were referenced in the SELECT clause, the corresponding feature would be "SELECT-R.A." Features corresponding to complex expressions are abstracted and evaluated using a recursive method described in Section 2.3.

2.3 Abstract Representation

2.3.1 Motivation

To motivate the need for an abstract representation for expressions, consider the following scenario. Assume we are given a database D containing 3 queries– Q_1, Q_2, and Q_3. Each of these queries is associated with a set of extracted features– F_1, F_2, and F_3 respectively.

```
Q1: SELECT R.a, R.b FROM R WHERE R.a=1
F1: [SELECT-R.a, SELECT-R.b, FROM-R, WHERE-R.a,
WHERE-=_R.a,1]

Q2: SELECT R.a, R.b FROM R WHERE R.a=2
F2: [SELECT-R.a, SELECT-R.b, FROM-R, WHERE-R.a,
WHERE-=_R.a,2]

Q3: SELECT (R.d + 6) FROM R, S WHERE R.id=S.id
AND R.a=3
F3: [SELECT-+_R.d,6, FROM-R, FROM-S, WHERE-R.id,
WHERE-S.id, WHERE=_R.id,S.id, WHERE-R.a, WHERE-
=_R.a,3]
```

Assume we are presented with a new query Q for which we wish to identify similar queries in D. Let F_Q be the set of features extracted from Q.

```
Q: SELECT R.a, R.c FROM R WHERE R.a=3
FQ: [SELECT-R.a, SELECT-R.c, FROM-R, WHERE-R.a,
WHERE-=_R.a,3]
```

Observe that F_1, F_2, and F_3 have 3 features in common with F_Q (underlined). For F_1 and F_2, those features are SELECT-R.a, FROM-R, and WHERE-R.a. For F_3, those features are FROM-R, WHERE-R.a, and WHERE=_R.a,3. If we treat all features as having equal weight, we must conclude that all entries in D are equally similar to Q by virtue of the fact that they share the same number of features with F_Q. However, it should be clear that queries Q_1 and Q_2 are structurally more similar to Q, and should thus be ranked ahead of Q_3. The literal representation employed in this example is too fine-grained.

The same example represented in the abstract form used in MLT and described in Section 2.3.2 is shown in Listing 3. In this representation, Q_1 and Q_2 have 4 features in common with F, whereas Q_3 has 3. Treating all features as equally important, Q_1 and Q_2 are correctly ranked ahead of Q_3 in the query recommendation output, again by virtue of the fact that they contain a greater number of features in common with Q.

```
Q1: SELECT R.a, R.b FROM R WHERE R.a=1
F1: [SELECT-R.a, SELECT-R.b, FROM-R, WHERE-R.a,
WHERE-=_R.a,?]

Q2: SELECT R.a, R.b FROM R WHERE R.a=2
F2: [SELECT-R.a, SELECT-R.b, FROM-R, WHERE-R.a,
WHERE-=_R.a,?]

Q3: SELECT (R.d + 6) FROM R, S WHERE R.id=S.id
AND R.a=3
F3: [SELECT-+_R.d,6, FROM-R, FROM-S, WHERE-R.id,
WHERE-S.id, WHERE=_R.id,S.id, WHERE-R.a, WHERE-
=_R.a,?]

Q: SELECT R.a, R.c FROM R WHERE R.a=3
FQ: [SELECT-R.a, SELECT-R.c, FROM-R, WHERE-R.a,
WHERE-=_R.a,?]
```

Listing 3. The example from Section 2.3.1 represented in the abstract format used by MLT.

2.3.2 Recursive Formulation

The process by which an arbitrarily complex SQL expression is transformed into an abstract feature string is shown in Listing 4. Recall that the WHERE clause in the preceding example contained the expression "R.a = 3." The abstract representation of this expression is "=_R.a,?". The constant (3) is replaced by a placeholder (?).

```
def abstract(expr, features):
    if expr is a constant:
        return "?"
    else if expr is a column name:
        return table name + "." + column name
    else if expr is an expression:
        n = num_operands(expr)
        o = operator(expr)

        # handle unary +, -
        if n == 1 and (o == "+" or o == "-"):
            p = expr.operand[0]
            return abstract(p, features)
        else:
            result = o + "_"
            for i = 0 to n:
                p = expr.operand[i]
                result += abstract(p, features)
                if (i+1) < n:
                    result += ","
            return result
    else if expr is a query:
        features.addAll(extractFeatures(expr))
        return "~"
```

Listing 4. Pseudocode describing the process by which an expression is transformed into an abstract feature string

To take a more complex example, Listing 5 illustrates the set of features F extracted from the lengthy but otherwise straightforward query Q.

```
Q: SELECT ra, dec, modelmag_r, modelmagerr_r,
modelmag_i, modelmagerr_i, photoz.z, photoz.zerr
```

```
FROM galaxy, photoz
WHERE (galaxy.objid=photoz.objid)
AND (ra BETWEEN 315.773258 AND 315.779022)
AND (dec BETWEEN -0.968437 AND -0.966228)

F: [
FROM-galaxy,
FROM-photoz,
SELECT-ra
SELECT-dec,
SELECT-modelmag_r,
SELECT-modelmagerr_r,
SELECT-modelmag_i,
SELECT-modelmagerr_i,
SELECT-photoz.z,
SELECT-photoz.zerr,
WHERE-=_galaxy.objid,photoz.objid,
WHERE-BETWEEN_ra,?,?,
WHERE-BETWEEN_dec,?,?,
WHERE-
AND_=_galaxy.objid,photoz.objid,BETWEEN_ra,?,?,BE
TWEEN_dec,?,?,
]
```

Listing 5. A concrete example illustrating the set of features extracted from a more complex query.

2.4 Query Similarity

2.4.1 tf-idf

tf-idf is a statistical measure commonly used in information retrieval to rank a document's relevance to a user's query [11]. The computation of tf-idf and its extension to query similarity in MLT are discussed in detail below.

The intuition behind tf-idf is that the importance of a term t_i to a particular document d_j increases proportionally to its number of occurrences, but is weighted by the general importance of t_i to the larger corpus of documents from which d_j is drawn.

The tf-idf score for term t_i and document d_j is:

$$tf - idf_{i,j} = tf_{i,j} \times idf_i$$

where $tf_{i,j}$ represents the term frequency of term i in document j, and idf_i represents the inverse document frequency of term i in the corpus. A high score is obtained by having a large number of occurrences in a given document and few occurrences in the overall corpus.

Inverse document frequency (idf) measures the general importance of a term t_i. It is computed by taking the logarithm of the result of dividing the number of documents in the corpus |D| by the number of documents in the corpus that contain term t_i.

$$idf_i = \log\left(\frac{|D|}{|\{d \mid t_i \in d\}|}\right)$$

Term frequency (tf) measures the importance of term t_i to document d_j. It is computed by dividing the number of occurrences of t_i in d_j by the size of the document.

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

2.4.2 Application of tf-idf to MLT

tf-idf is readily adapted for use in MLT. A simple ranking function is computed by summing the tf-idf scores for each feature extracted from the input query.

Definition 3. Given a pair of queries Q_i, Q_j and their associated feature sets F_i, F_j, similarity is defined by the following formula.

$$similarity(F_i, F_j) = \sum_{f \in F_i} tf(f, F_j) \times idf(f)$$

$$idf(f) = \log\left(\frac{n_{QR}}{|\{Q \mid f \in features(Q)\}|}\right)$$

$$tf(f, F_j) = \frac{n_{f,F_j}}{|F_j|}$$

2.4.3 Subjective, per-clause Weights

A byproduct of computing query similarity using tf-idf is that relatively rare features in the input query receive large weights, whereas common features receive small weights. This has the effect of biasing the ranking of recommend queries in favor of those containing more selective features.

It may also be desirable to intentionally bias search in order to diversify the set of recommended queries. For example, most users understand the basic SELECT-FROM-WHERE paradigm in SQL. However, they may not understand how aggregates work. Thus, features containing a GROUP BY clause could be given additional weight. This is achieved by introducing a per-clause weight to the tf-idf computation. By default, per-clause weights are uniform (e.g. 1.0).

2.4.4 Diversifying the Results

Recall that the purpose of MLT is to assist non-expert users in formulating queries based on the shared experience of other users of the database. To this end, MLT is only effective if it recommends a diverse set of queries that either introduce new SQL concepts to the user or identify previously unexplored portions of the database.

One of the first things you will notice after implementing tf-idf is that it works *too* well. The top k recommendations are extremely similar to the input query. Consider the top 3 results returned for the query shown in Listing 7.

```
Input query:
SELECT count(*)
FROM photoprimary
WHERE htmid >= 13351851655168
AND htmid <= 13351852703743

Ranked results (decreasing order of
relevance)
1. (1.548096)
select count(*) from photoprimary where (htmid >=
9465428443136 and htmid <= 9465432637439) ;
2. (1.548096 )
select count(*) from photoprimary where (htmid >=
9459903496192 and htmid <= 9459904544767) ;
3. (1.548096 )
select count(*) from photoprimary where (htmid >=
9454279983104 and htmid <= 9454281031679) ;
```

Listing 7. Query recommendations returned by MLT prior to diversification. The top 3 recommendations are shown above, in decreasing order of similarity. Similarity scores are shown in parentheses.

These results are clearly too similar to the input query to be of any value to the user. They differ only in the choice of constants in the WHERE clause.

Definition 4. The diversity of a set of recommendations $R=\{R_1, R_2, \ldots, R_k\}$ for some input query Q is:

$$diversity(R) = \sum_{i<j}^{k} -similarity(R_i, R_j)$$

The desired set of recommendations R^* is maximally diverse. It contains a set of queries that, collectively, are minimally similar to one another. Of course, this criterion could be trivially satisfied by identifying a set of k queries in QR that are pairwise-least similar to one another, regardless of the input query. R^* must be further constrained.

In addition to maximizing the diversity function, R* must also maximize similarity to the input query Q. This is achieved by maximizing the function:

$$\sum_{i=1}^{k} similarity(Q, R_i)$$

When the Query Recommender is constrained by these criteria, a vastly improved set of recommendations is returned.

```
Input query: select count(*) from photoprimary
where (htmid >= 13351851655168 and htmid <=
13351852703743) ;

Ranked results (decreasing order of
relevance)
 1. (1.548096)
select count(*) from photoprimary where (htmid >=
15176972632064 and htmid <= 15176973680639);
 2. (0.428301)
select count(*) from photoobj p, specobj s where
p.objid=s.bestobjid and (s.specclass = 3 or
s.specclass = 4);
 3. (0.408258)
select objid from photoprimary where ra between
104.433294 and 104.730872 and dec between -
56.022222 and -55.855556;
```

Listing 8. Query recommendations returned by MLT after diversification.

3. Evaluation

The recommendations produced by MLT are quantitatively evaluated on a random sample of 10,000 queries logged by the SDSS server from 2002 to 2009. Syntactically invalid queries, duplicates, and those that contain proprietary SQL features are removed. Also, if the Query Repository contains an exact match to a user's query, it is not included in the set of recommendations.

3.1 Database Statistics

After syntactically invalid queries were removed from the data set, as well as duplicates and those containing proprietary SQL features, 1,603 queries remained. 27,967 features were extracted from these queries, 1,465 of which were unique. The average query contained 863 characters, while the longest and shortest queries contained 7,565 and 29 characters, respectively. On average, a query contained 17 features. The minimum number of features in a query was 2, while the maximum was 450.

Clause	# Occurrences
SELECT	21,587
FROM	1,785
WHERE	4,559
DISTINCT	14
GROUP BY	1
HAVING	1
ORDER BY	20

Table 1. Distribution of features extracted from the SDSS data set to SQL clauses. Note the underrepresentation of aggregates and ordering.

3.2 Quantitative Evaluation

3.2.1 Query Attribution

Since the queries in the SDSS data set lack metadata, attributing a query to a specific user is impossible. As a result, a simple heuristic based on edit distance was devised for identifying pairs of queries predicted to have been issued consecutively by the same user. Specifically, given a pair of queries Q_i and Q_j, the edit distance between Q_i and Q_j is calculated using dynamic programming. Edit distance is normalized to the interval [0..1], where 1 represents identity and 0 represents complete dissimilarity. If the normalized edit distance between Q_i and Q_j is between 0.6 and 0.8, the queries are considered to belong to the same session. Empirically, pairs of queries with normalized edit distances in this range produced reasonable results.

3.2.2 High- and Low-quality Matches

Using the query attribution method described above, sets of query pairs (Q_i, Q_{i+1}) hypothesized to have been issued by the same user were collected. Recommendation quality was quantitatively evaluated by tracking the number of times the set of recommendations for Q_i contained Q_{i+1}.

In the evaluation procedure, we distinguish between high- and low-quality matches. A match (Q_{i+1}, R) is considered high-quality if:

$$features(R) \supseteq features(Q_{i+1}) \,\|\, features(Q_{i+1}) \supseteq features(R)$$

The reflexivity in the definition stems from the fact that size of the feature set of a recommendation is not strictly greater than or equal to the size of the feature set of the query. Although this may be a desirable property, particularly in the case of query assistance, it is not currently enforced.

In many cases, features(R) was very similar to features(Q_{i+1}), but not identical. To reward such "near-misses," I introduced the concept of a low-quality match. A match is considered to be low-quality if the intersection of features(Q_{i+1}) and features(R) is sufficiently large. Overlap threshold is defined as:

$$\frac{|\, features(R) \cap features(Q_{i+1}) \,|}{|\, features(Q_{i+1}) \,|} \geq \varepsilon$$

3.2.3 Results

The query evaluation procedure described above was run on the set of predicted user-query pairs for varying values of k. The results are shown in Figure 1.

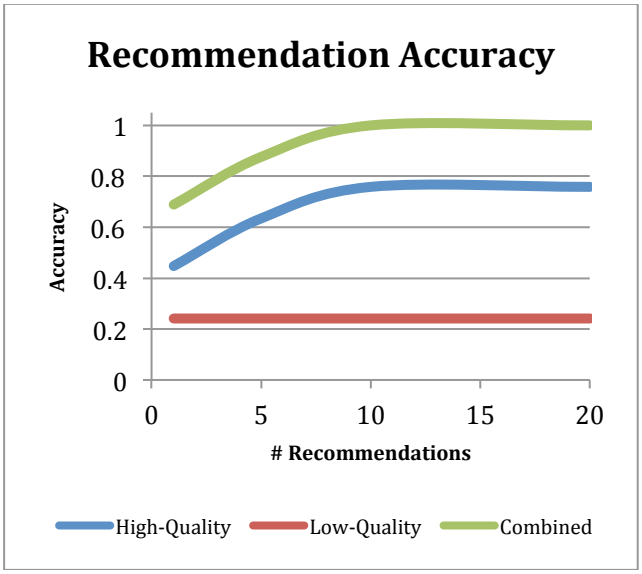

Figure 1. Depicts recommendation accuracy on the predicted user-query pairs as a function of the number of recommendations.

The observed results are consistent with expectations. Namely, the percentage of high-quality matches increases with the number of recommendations. In the extreme case where only a single recommendation is generated, a high-quality match is returned approximately 45% of the time. If we include low-quality matches, recommendation accuracy rises to 68.9%. As the number of recommendations increases to k=10, the percentage of high-quality matches exceeds 75%, with a combined accuracy of 100%.

The number of high-quality matches levels off around k=10 recommendations. This is likely an artifact of the small size of the Query Repository. It is important to remember that k is the *maximum* number of recommendations returned, not the total. If there are m < k queries in the Query Repository whose similarity to the input is non-zero, only m results are returned.

Interestingly, the percentage of low-quality matches was approximately constant for various values of k. This is likely to be an artifact of the user-query attribution heuristic.

These results, although promising, require additional validation. It is possible that the user-query attribution heuristic biases the results in favor of the method. An additional data set with known user-query attribution is currently being collected. When a sufficient number of queries have been catalogued, this experiment will be rerun.

4. Conclusion

In this paper, I presented an alternative approach, More Like This (MLT), to the problem of full-text query recommendation. Queries are modeled as collections of abstracted features. Similarity between a pair of queries is defined in terms of tf-idf of their respective feature sets.

The recommendations produced by MLT are quantitatively evaluated on a random sample of 10,000 queries logged by the SDSS server from 2002 to 2009. Since the queries lack metadata, attributing a query to a specific user is impossible. As a result, a simple heuristic based on edit distance was devised for identifying pairs of queries predicted to have been issued consecutively by the same user. Despite the imprecision of the query attribution method, recommendation quality is promising.

In terms of future work, there is much to be done. First and foremost, I intend on integrating MLT functionality within an actual RDBMS. SQLite appears to be a good candidate. In doing so, additional data sets with known user-query attribution could be created. These "ground-truth" data sets would permit a more thorough analysis of recommendation quality. Should the results turn out well, I would like to qualitatively evaluate the recommended queries by conducting a series of controlled user studies. Despite their cost and complexity to set up, user studies are an excellent mechanism for gaining feedback.

Another aspect of MLT that bears additional scrutiny is the use of subjective, per-clause

weights to bias the set of recommended queries. Empirical data has shown that most scientific users readily grasp SQL's SELECT-FROM-WHERE paradigm. Gaining a better understanding of the interplay between idf scores and subjective, per-clause weighting would allow for the creation of a sophisticated SQL learning tool.

5. References

[1] Khoussainova, N., Kwon, Y., Balazinska, M., Suciu, D. *SnipSuggest: Context-Aware Autocompletion for SQL*. *Proc. VLDB Endow*. 4, 1 (October 2010), 22-33.

[2] Chatzopoulou, G., Eirinaki, M., Polyzotis, N. *Query Recommendations for Interactive Database Exploration*. *SSDBM 2009*, LNCS 5566, pp. 3–18, 2009.

[3] Howe, B., Cole, G., Key, A., Khoussainova, N. *SQL is Dead; Long Live SQL: Smart Services for Ad Hoc Databases*. Microsoft Research Whitepaper. 2010.

[4] Sloan Digital Sky Survey. http://www.sdss.org.

[5] Yang, X., Procopiuc, C.M., and Srivastava, D. *Summarizing relational databases*. Proc. VLDB Endow., 2(1): 634-645, 2009.

[6] Baeza-Yates, R., and Riberio-Neto, B. *Modern Information Retrieval*. Addison-Wesley Longman Publishing, Boston, MA, 1999.

[7] ANTLR. http://www.antlr.org.

[8] Zql. http://zql.sourceforge.net.

[9] Cooper, K.D., Torczon, L. *Engineering a Compiler. Morgan Kaufmann Publishers*, San Francisco, CA, 2008.

[10] JavaCC. http://javacc.java.net.

[11] Sparck-Jones, Karen. *A statistical interpretation of term specificity and its application in retrieval*. Journal of Documentation, 28(1): 11-21, 1972.