SQL QueRIE Recommendations

Javad Akbarnejad
Computer Engineering Dept.
San Jose State Univ.

Gloria Chatzopoulou*
Computer Science Dept.
Univ. of California, Riverside

Magdalini Eirinaki
Computer Engineering Dept.
San Jose State Univ.

Suju Koshy
Computer Engineering Dept.
San Jose State Univ.

Sarika Mittal
Computer Engineering Dept.
San Jose State Univ.

Duc On
Computer Engineering Dept.
San Jose State Univ.

Neoklis Polyzotis
Computer Science Dept.
Univ. of California, Santa Cruz

Jothi S. Vindhiya Varman
Computer Engineering Dept.
San Jose State Univ.

ABSTRACT

This demonstration presents QueRIE, a recommender system that supports interactive database exploration. This system aims at assisting non-expert users of scientific databases by tracking their querying behavior and generating personalized query recommendations. The system is supported by two recommendation engines and the underlying recommendation algorithms. The first identifies potentially "interesting" parts of the database related to the corresponding data analysis task by locating those database parts that were accessed by similar users in the past. The second identifies structurally similar queries to the ones posted by the current user. Both approaches result in a recommendation set of SQL queries that is provided to the user to modify, or directly post to the database. The demonstrated system will enable users to query and get real-time recommendations from the SkyServer database, using user traces collected from the SkyServer query log.

Keywords: recommender systems, collaborative filtering, relational databases, interactive exploration

1. INTRODUCTION

Relational database systems are becoming increasingly popular in the scientific community. For example the Genome browser[1] provides access to a genomic database, and SkyServer[2] stores large volumes of astronomical measurements. Those databases usually employ a web-based interface that allows a broad user base to submit SQL queries and retrieve the results. Even though such database systems offer the means to run complex queries over large data sets, the discovery of useful information remains a big challenge. As an example, users who are not familiar with the database may overlook queries that retrieve interesting data, or they may not know what parts of the database provide useful information. Moreover, most of the users of such databases do not have the required SQL background that would allow them to run complex queries and retrieve otherwise interesting results. Not being able to fully exploit the capabilities of such systems hinders data exploration, and thus reduces the benefits of using a database system.

To address this important problem of assisting users when exploring a database, we designed the QueRIE (Query Recommendations for Interactive data Exploration) system. Our inspiration draws from the successful application of recommender systems in the exploration of Web data. The premise on which the system is built is simple: If a user A has similar querying behavior to user B, then they are likely interested in retrieving the same data. Hence, the queries of user B can serve as a guide for user A.

Transferring the collaborative filtering paradigm to the database context presents several challenges. First, SQL is a declarative language, thus the same data can be retrieved in more than one way. This complicates the evaluation of similarity among users, since contrary to the web paradigm where the similarity between two users can be expressed as the similarity between the items they visit/rate/purchase, we cannot rely directly on the SQL queries. A second important issue is how to create implicit user profiles that measure the level of importance of the same data for different users. Finally, contrary to the user-based collaborative filtering approach, the recommendations to the users have to be in the form of SQL queries, since those actually describe what the retrieved data represent. Thus, we need to "close the loop" by first decomposing the user queries into lower-level components in order to compute similarities and make predictions, and then re-construct them back to meaningful and intuitive SQL queries in order to recommend them. All those issues make the problem of interactive database exploration very different from its web counterpart.

The first results of this work have been presented in [1]. In this demonstration we will showcase the QueRIE system. The demonstrated system will enable users to query the SkyServer database, obtaining real-time personalized query recommendations that are generated using traces collected from the SkyServer query log. The users will be able to use any of the two recommendation engines, one based on similarities of the content retrieved by the queries, and the other based on similarities between the queries themselves. The users will be provided with various demonstration scenarios, exhibiting different information needs and various querying styles. They will be able to retrace the steps of such scenarios and evaluate the usefulness of the query recommendations by looking at the results. Apart from the graphical user interface of the system we will

*The author worked on this project while affiliated with UC Santa Cruz
[1]http://genome.ucsc.edu/
[2]http://cas.sdss.org/

also demonstrate the underlying functionality of the system.

2. SYSTEM OVERVIEW

The information flow of the QueRIE personalization system is shown in Figure 1. The active user's queries are forwarded through the *database query interface* to both the DBMS and the *recommendation engine*. The DBMS processes each query and returns a set of results. At the same time, the query is stored in the query log. This query log is processed offline in order to create the predictive model. Each time a user accesses the system, the recommendation engine combines her input with the predictive model and generates a set of query recommendations. In what follows, we provide a very brief overview of the QueRIE framework and the underlying algorithms, since the demonstration will follow the same information flow.

QueRIE currently has two distinct recommendation engines, each using a different notion of similarity in order to compute the recommendation set of queries. Our motivation behind the first, "tuple-based" recommendation engine, was to define the similarity between two users in terms of their information needs. In essence, the tuple-based recommendation engine identifies which parts of the database have been "touched" by the current user's queries and retrieves users who have explored overlapping parts of the database in the past. These, along with the current user's queries, define a superset of the database parts covered so far by the user's queries. The final set of recommendations consists of queries that best cover this extended part of the database. However, two queries might be semantically similar but retrieve different results due to some filtering conditions. This was our motivation for the second, "fragment-based" recommendation engine. In essence, we deconstruct each query into fragments and discover other fragments that co-appear with them in sessions of different users (an indication of structural similarity). We use these fragments to identify the most similar queries and generate the final recommendation set.

A detailed technical description of the proposed framework and the tuple-based engine can be found at [1]. The tuple-based recommendation engine component was demonstrated in [7]. In this demonstration, we will present the enhanced QueRIE system, integrating both the tuple-based and the fragment-based recommendation engines. The audience will benefit by comparing the two approaches in terms of performance and accuracy, and will be able to select the engine that fits better to their needs and preferences.

3. RECOMMENDATION ALGORITHMS

The queries of each user touch a subset of the database that is relevant to the analysis the user wants to perform. We assume that this subset is modeled as a *session summary* S_i for user i. We use $\{1, \ldots, h\}$ to denote the set of past users based on which recommendations are generated and 0 to identify the current user. To generate recommendations, our framework extends the summary S_0 of the active user to a "predicted" summary S_0^{pred}. This extended summary captures the predicted degree of interest of the active user with respect to all the parts of the database, including those that the user has not explored yet, and thus serves as the seed for the generation of recommendations.

To summarize, our framework consists of three components: (a) the construction of a session summary S_i for each user i, (b) the computation of a "predicted" summary S_0^{pred} for the active user, based on the active user's and the past users' summaries, and (c) the generation of queries based on S_0^{pred}. Those queries will be presented to the user as recommendations. The details of each step differ for each recommendation engine. We provide a brief overview of both approaches in what follows.

3.1 Tuple-based recommendations

Session summaries. We define the session summary S_i as a vector of tuple weights that covers all the database tuples. The weight of each vector element represents the importance of the respective tuple in the exploration performed by user i. For this purpose we employ two different weighting schemes which are detailed in the accompanying paper [1]. Using the session summaries of the past users, we can define the conceptual *session-tuple matrix* that, as in the case of the user-item matrix in web recommender systems, will be used as input in our collaborative filtering process.

Computing the "predicted" summary. Similarly to session summaries, the extended summary S_0^{pred} is a vector of tuple weights. In order to compute this summary, we assume the existence of a function $sim(S_i, S_j)$ that measures the similarity between two summaries and takes values in $[0, 1]$. Using this function, we compute the extended summary as a weighted sum of the existing summaries: $S_0^{\text{pred}} = \sum_{0 \le i \le h}(sim(S_0, S_i) \times S_i)$. The similarity function sim can be realized with any vector-based metric, such as the cosine similarity measure.

Generating recommendations. The final step is to generate queries that cover the interesting tuples in S_0^{pred}. In order to provide the users with intuitive, easily understandable recommendations, we use the queries of past users. We assign to each past query Q an importance with respect to S_0^{pred} , computed as $rank(Q, S_0^{\text{pred}}) = sim(S_Q, S_0^{\text{pred}})$. Hence, a query has high rank if it covers the important tuples in S_0^{pred}. The top ranked queries are then returned as the recommendation.

Accelerating the online computations. To ensure that the aforementioned approach generates real-time recommendations for the active users of a database, we need to compress the session-tuple matrix and to speed up the computation of similarities. For this reason, we employ the MinHash probabilistic clustering technique that maps each session summary S_i to a "signature" $h(S_i)$ [2]. The Jaccard similarity between vectors is thus reduced to the similarity of their signatures: $JaccardSim(S_i, S_0) = sim(h(S_i), h(S_0))$.

3.2 Fragment-based recommendations

Session summaries. This approach is based on the pair-wise similarity of query fragments (attributes, tables, joins and predicates). We need to identify fragments that co-appear in several queries posed by different users. The session summary vector S_i for a user i consists of all the query fragments ϕ of the user's past queries. Let $\mathcal{Q}_i$ represent the set of queries posed by i and F represent the set of distinct query fragments recorded in the query logs. For a given fragment $\phi \in F$, its importance in session S_i is represented by $S_i[\phi]$ and depends on its importance in the session. We can define $S_Q[\phi]$ as a weighted or binary variable that represents the importance of ϕ in a session's query Q. Then, S_i is defined respectively as a sum ($S_i = \sum_{Q \in \mathcal{Q}_i} S_Q$), or OR-ed ($S_i = \bigvee_{Q \in \mathcal{Q}_i} S_Q$).

Computing the "predicted" summary. Using the session summaries of the past users and a vector similarity metric, we construct the $(|F| \times |F|)$ *fragment-fragment matrix* that contains all similarities $sim(\rho, \phi)$, $\rho, \phi \in F$. The recommendation seed, modeled by S_0^{pred}, represents the estimated importance of each query fragment with regard to the active user's behavior S_0. Similarly to the item-to-item collaborative filtering approach of web recommender systems, we employ the fragment-to-fragment similarities that are computed in the previous step: $S_0^{\text{pred}}[\phi] = \frac{\sum_{\rho \in R} S_0[\rho] * sim(\rho, \phi)}{\sum_{\rho \in R} sim(\rho, \phi)}$,

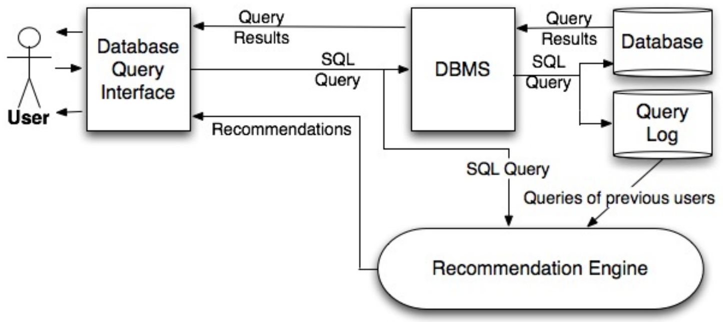


Figure 1: System Architecture

where R represents the set of top-k similar query fragments ($k \leq |F|$). We should note that all pair-wise similarities can be computed and stored off-line. This results in a very efficient execution of the algorithm in terms of computational time.

Generating recommendations. Once the predicted summary S_0^{pred} has been computed, the top-n fragments F_n (i.e. the fragments that have received the higher weight) are selected. Then all past queries Q, $Q \in \bigcup_i \mathcal{Q}_i$ receive a rank QR based on a normalized metric measuring the number of common query fragments of each query Q to the top-n list. Finally, the top-m ranked queries are used as the recommendation set.

4. DEMONSTRATION SCENARIO

As shown in Figure 1, QueRIE consists of two main building blocks, namely the database query interface and the recommendation engine, and uses two information repositories, namely the database itself, as well as its query logs. The database query interface module is built using HTML, JSP and JavaScript. The recommendation engine module is built using Java. The two modules interact through the JNI framework. The demonstrated system interacts real-time with the SkyServer database and uses real user traces as the query logs of past users.

The demonstration shows the functionality of the system and the internal operations of the recommendation engine. In that way, we are able to demonstrate the usefulness of such a system from the user's perspective, and at the same time allow someone to understand and evaluate how the various data inputs are manipulated in order to generate the final recommendations.

Once a user logs in the system, she is able to select one of the two recommendation engines. The user can author and submit an SQL query to SkyServer. QueRIE sends the request to the database, and presents the user with the results. At the same time, the system records the active user's queries, creating an implicit user profile. This user profile is used as input to the algorithm, along with the predictive model to generate real-time, personalized query recommendations. The recommended queries are presented to the user in a list, as shown in Figure 2. For each recommended query, the user is able to examine a sample of the results that will be retrieved, in order to decide whether it addresses his needs, prior to actually submitting it to the DBMS.

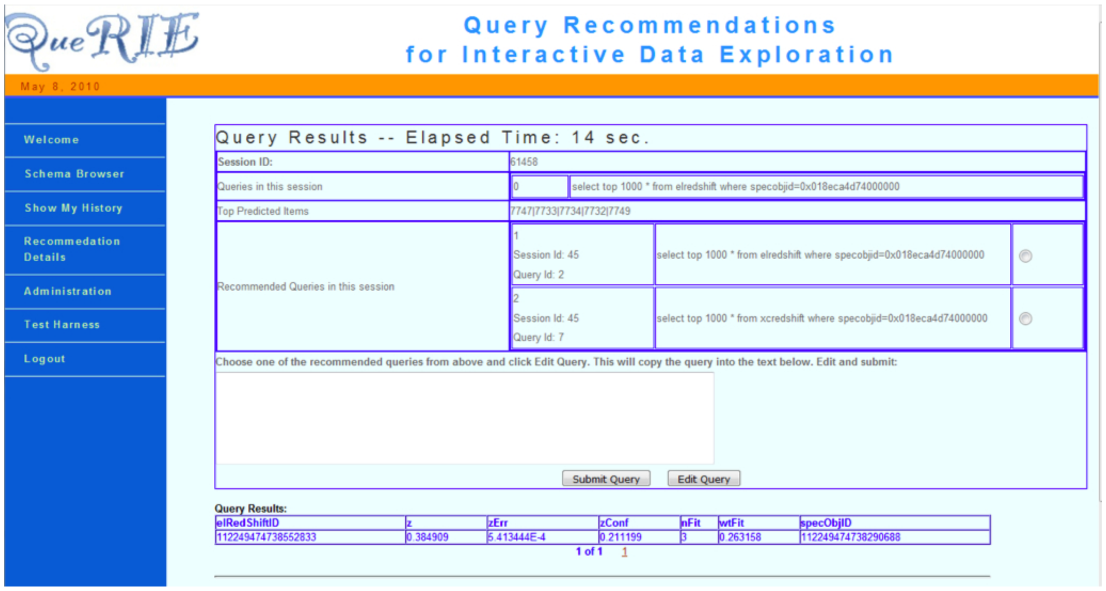
At all times, the active user is able to: (a) formulate a query from scratch, (b) select a recommended query and submit it as it is, or (c) select a recommended query and edit it before submitting it to the database. All the aforementioned options result in an SQL query being submitted to the SkyServer database, which is handled by the system as described before. Moreover, the interface allows the user to browse the database schema in order to find more information on the tables and the respective attributes and formulate the queries. The user is also able to review and re-submit queries that were posed during her recent history. Those options are available through the "Schema Browser" and "Show My History" menu tabs respectively.

Apart from demonstrating the user interface of our system, we also incorporate a "back-end" console that enables users to observe and understand how the underlying algorithms work, as well as evaluate the usefulness of the recommendations in terms of prediction accuracy. This is available through the "Recommendation details" and "Test Harness" consoles for the tuple-based and fragment-based engine respectively. The prototype also allows, through the "Administration" tab, to tune the parameters of the algorithm, such as the number of recommended queries, and the weighted scheme (fragment-based engine).

In order to evaluate the usefulness of the system and the accuracy of the recommendations, we follow the "recommended vs. actual queries" approach. More specifically, we will demonstrate several scenarios, represent different information needs, or querying styles. Each scenario reflects a real user session consisting of $k > 3$ queries, as recorded in the pre-processed query logs of SkyServer. After submitting the first $k - n$ queries to the system, the users will be able to see both QueRIE's recommendations and the actual n last queries of the user side by side, and compare them. In what follows, two different scenarios of accessing the database are given. Their difference lies on the information need of the users, the structure and complexity of the queries involved, and the potential challenges that hinder data exploration in each case. The user sessions shown in both scenarios are real and belong to the query logs of SkyServer.

Scenario 1. The first scenario reflects the case where a user might not be able to locate the information that interests her as a result of the large size of the database. Assume that the user needs to access some data related to a specific object located in four different relations. Such queries are simple SELECT-FROM-WHERE queries so formulating the query is not a challenge even for novice users. If the user is unfamiliar with the database, she will have to first browse the schema, locate the information and then go back and post the queries. This situation inhibits a potential danger of missing some important data:

Query 1:	*SELECT * FROM specline WHERE specobjid=0x014acc9281400000*
Query 2:	*SELECT * FROM speclineindex WHERE specobjid=0x014acc9281400000*
Query 3:	*SELECT * FROM specobjall WHERE specobjid=0x014acc9281400000*
Query 4:	*SELECT * FROM xcredshift WHERE specobjid=0x014acc9281400000*



Query Results -- Elapsed Time: 14 sec.

Session ID:	61458		
Queries in this session	0	select top 1000 * from elredshift where specobjid=0x018eca4d74000000	
Top Predicted Items	7747|7733|7734|7732|7749		
Recommended Queries in this session	1 Session Id: 45 Query Id: 2	select top 1000 * from elredshift where specobjid=0x018eca4d74000000	○
	2 Session Id: 45 Query Id: 7	select top 1000 * from xcredshift where specobjid=0x018eca4d74000000	○

Choose one of the recommended queries from above and click Edit Query. This will copy the query into the text below. Edit and submit:

Query Results:

elRedShiftID	z	zErr	zConf	nFit	wtFit	specObjID
112249474738552833	0.384909	5.413444E-4	0.211199	3	0.263158	112249474738290688

1 of 1 1

Figure 2: QueRIE interface after a query has been submitted (fragment-based engine)

Scenario 2. In this scenario the case where a user who needs to formulate complex SQL queries but does not have the required expertise, is examined. Assume that the user needs to perform some specific analysis which requires the use of aggregations in order to return the desirable results. A novice user might not be able to retrieve the desired information because of lack of experience. Such a user will start by posting simpler queries. If, however, another user in the past has gone through this process, and the respective user session is recorded in the query logs, the system will be able to recommend the correct query to the current user:

Query 1:	*SELECT count(*) FROM region WHERE type like 'tiprimary'*
Query 2:	*SELECT count(distinct id) FROM region WHERE type like 'tiprimary'*
Query 3:	*SELECT id, count(*) FROM region WHERE type like 'tiprimary' GROUP BY id*
Query 4:	*SELECT id, count(*) FROM region WHERE type like 'tiprimary' GROUP BY id HAVING count(*)> 1*

5. RELATED WORK

Contrary to keyword-based query recommendation systems [6], QueRIE is meant to assist users who need to pose complex SQL queries to large relational databases. Moreover, QueRIE does not require from the users to explicitly declare their preferences beforehand in order to generate recommendations. A multidimensional query recommendation system is proposed in [3, 4]. In this work the authors address the related problem of generating recommendations for data warehouses and OLAP systems. The challenges of applying data mining techniques to the database query logs are addressed in [5]. In this work, the authors outline the architecture of a query management system and propose that data mining techniques can be used in order to provide the users with query suggestions. Contrary to our work the authors do not provide any technical details on how such a recommendation system could be implemented. To the best of our knowledge, QueRIE is the first framework to address the problem of generating SQL query recommendations proposing a fully-fledged solution.

6. CONCLUSIONS

We have described the functionalities that we demonstrate for QueRIE, a recommender system that assists users when interacting with large database systems. QueRIE enables users to query a relational database, while generating real-time personalized query recommendations for them. The system incorporates two recommendation engines, a tuple-based one that recommends queries that touch similar parts of the database, and a query fragment-based one that recommends structurally similar queries. Currently QueRIE interacts with the SkyServer database but it is evident that the system can be easily adapted to interact with any relational database given that its query logs are available for analysis. QueRIE does not require an explicit user profile or keyword-based queries. On the contrary, it "closes the loop" by accepting SQL queries as input, decomposing them in order to identify interesting database areas for each user, and re-transforms them in SQL queries that are presented as recommendations.

7. REFERENCES

[1] G. Chatzopoulou, M. Eirinaki, and N. Polyzotis. Collaborative filtering for interactive database exploration. In *SSDBM '09*.

[2] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55:441–453, 1997.

[3] A. Giacometti, P. Marcel, and E. Negre. Recommending Multidimensional Queries. In *DaWaK'09*.

[4] A. Giacometti, P. Marcel, E. Negre, and A. Soulet. Query recommendations for olap discovery driven analysis. In *DOLAP '09*.

[5] N. Khoussainova, M. Balazinska, W. Gatterbauer, Y. Kwon, and D. Suciu. A case for a collaborative query management system. In *CIDR '09*.

[6] G. Koutrika and Y. Ioannidis. Personalized queries under a generalized preference model. In *ICDE '05*.

[7] S. Mittal, J. S. V. Varman, G. Chatzopoulou, M. Eirinaki, and N. Polyzotis. QueRIE: A Recommender System supporting Interactive Database Exploration. In *ICDM '09 (to appear in ICDM '10 proceedings because of editor's error)*.