Mirror mirror on the wall, which query's fairest of them all?

Georgia Koutrika Alkis Simitsis

HP Labs
Palo Alto, USA
firstname.lastname@hp.com

ABSTRACT

Scientists heavily use database technologies to store and analyze massive amounts of experimental data. However these tasks require (sometimes, advanced) knowledge of a query language, which is not necessarily amongst the normal skills of a scientist. Database experts may help in the initial construction of such queries, which are then stored in a repository. Afterwards, instead of writing queries from scratch, stored queries can be re-used with few or no changes by people in the same or a different research group. However, as much as this is a desired reality, we are actually stepping into dreamland. Searching a pool of database queries for those that fit in a certain analysis is not easy even for experienced database users. One solution could be to provide a description for each query and then, use a keyword search technique to identify queries of interest. But doing this manually is time consuming, and its success relies on the uniformity of the descriptions provided –which is extremely hard when many query writers are involved in this effort as typically happens. In this paper, we envision a system that can understand the problems past queries solve, the computations they involve and their significance, and can help the user compose the queries required for the task at hand using this knowledge from past queries.

1. THE SNOW WHITE

The fairy tale. Nick, an astronomer, analyzes photometric data for stars and galaxies using a database that is accessed by several people and research groups. Here is how Nick could interact with the system.

Nick: "How to compute a list of moving objects consistent with an asteroid?"

System: "Here is a query I have seen that computes that:"

```
SELECT TOP 10 objID, ra, dec,
sqrt( power(rowv,2) + power(colv, 2) ) as velocity
FROM PhotoObj
WHERE (power(rowv,2) + power(colv, 2)) > 50
AND rowv != -9999 and colv != -9999
```


6th Biennial Conference on Innovative Data Systems Research (CIDR '13) January 6-9, 2013, Asilomar, California, USA.

Nick changes the value for rowv and executes the query. The interaction with the system continues below.

Nick: "How to find stars that pass a certain standard of clean photometry?"

System: "I have seen a similar query that selects galaxies based on photometry values. "

Nick: "Let me see and edit this query."

In the interaction above, the system uses its memory of past queries to help Nick with his task. It understands what past queries do but also what individual parts within a single query perform. Given a description of a desired computation or query from Nick, the system is able to figure out the computations required, and by comparing to past queries, it can find those that achieve the same or similar tasks. For example, for the second request from Nick, the system finds a query that involves some photometry computation but on different objects (i.e., galaxies). If no matching queries exist, the system can suggest a group of queries, each query covering some part of the computations required, and a new query could be composed by combining parts of these queries. In addition, the system can provide explanation of the recommended computations or queries to Nick.

The reality. Scientists like Nick heavily use database technologies to store and analyze massive amounts of experimental data. Such data analysis requires (sometimes, advanced) knowledge of a query language, which is not necessarily amongst the normal skills of a scientist. Often, database experts may help in the initial construction of such queries. Instead of writing queries every time from scratch, queries built once can be stored in a repository and can be re-used with few or no changes by people in the same or a different research group.

However, searching a pool of queries for those that fit in a certain analysis is not easy, even for experienced users. Modern DBMSs offer a primitive way to search for stored queries or procedures based only on their definition. For instance, one could search for stored procedures that refer the same column using a SQL query like this one:

```
SELECT OBJECT_NAME(M.object_id), M.* FROM sys.sql_modules M
JOIN sys.procedures P ON M.object_id = P.object_id
WHERE M.definition LIKE '\%thiscolumn\%'
```

Searching based only on the structures the queries touch is not helpful. Ideally, we want to search queries based on their intent, functionality and output. One possibility could be to provide a set of query templates to guide users in writing their own queries. SkyServer is a website that presents data from the Sloan Digital Sky Survey, a project to make

a map of the universe. The website offers a page with "*sample queries designed to serve as templates for writing your own SQL queries. ... (these template queries are) written to solve real scientific problems submitted by astronomers. Those queries are divided by scientific topic.*"[1] One example template query is shown in Figure 1 describing bluer galaxies, and it has three parts: a short title, a description of the query in natural language, and the SQL query itself.

Providing query templates can help users. But it is a manual process and does not automatically scale to any data or queries. Furthermore, users can relatively easily browse a handful of queries; trying to do so with hundreds of queries that may be provided by other scientists is not possible.

The vision. In this paper, we envision a system that can understand the semantics of queries and of the individual computations within a query, and can help a user build the queries for a problem at hand. The system enables users to search for queries based on query purpose, output, and semantics. It digs into the system 'memory' of queries, and it returns queries ranked according to how well they match the user search criteria. The system can also provide a set of queries that partially achieve the user's goal but can be combined or partially re-used to formulate the desired query.

Although a human-like interaction with the system (like the one taking place between Nick and the imaginary system) is nice, here we do not envision an AI system that can perform intelligent dialogues. There are also other ways this interaction could be performed, for example through a collaborative GUI (as in [1]) or based on simple patterns that could build some primitive dialogues. However, this interesting but orthogonal topic is not the focus of this paper.

The challenges here lie in transforming a database system from an efficient but passive query execution engine to an intelligent collaborative system that can help people using search to identify or build queries based on the queries previously seen by the system. Starting from a simple approach, we explain the main challenges, then we outline possible solutions, and research directions.

2. THE SEVEN DWARFS

One possible approach to implementing this system would be an NLP system that starts from a problem description and builds a query that when executed over the underlying database generates the desired results. However, such a system has been the holy grail of NLP for decades [4]. NLP techniques need very fine tuning and even then, ambiguities may still exist. In this paper, we follow a different approach trying to exploit the structure and semantics of queries and thus, to solve the problem by reducing the need for a pure NLP approach. We describe a system that comprises seven components (dwarfs) and we discuss how to approach each component (see Figure 2).

Query Writer. A query interface allows users to write their own SQL queries over the underlying data. Queries issued to the system can be stored and become available to other people. The query interface may allow a query writer to choose the queries that will be stored in the system. This may be desired for a number of reasons. A query may be reformulated several times before getting finalized. Storing all intermediate variations may not be useful. Another reason may be that a person may choose not to share a query.

[1]http://skyserver.sdss3.org/dr9/en/help/docs/realquery.asp

Galaxies with bluer centers, by Michael Strauss.

For all galaxies with r Petro < 18, not saturated, not bright, and not edge,
give me those with centers appreciably bluer than their outer parts,
i.e., define the center color as: u psf - g psf, and
define the outer color as: u model - g model;
give me all objs which have (u model - g model) - (u psf - g psf) < -0.4

```
SELECT TOP 1000
modelMag_u, modelMag_g, objID
FROM Galaxy
WHERE
( Flags & (dbo.fPhotoFlags('SATURATED') +
dbo.fPhotoFlags('BRIGHT') + dbo.fPhotoFlags('EDGE')) ) = 0
and petroRad_r < 18
and ((modelMag_u - modelMag_g) - (psfMag_u - psfMag_g)) < -0.4
```

Figure 1: A sample query from the Sky Survey site

Query Translator. For a new query q to be stored in the system repository, we need to provide a textual description d that explains what the query does. For this purpose, here, we describe the following options.

Query writer as translator. The query writer provides the description d, as happens with the query in Figure 1. In fact, that particular person has provided a title (short description) and a more detailed explanation for the query.

Crowdsourcing. Instead of relying on one user to provide a single description for a query, we can outsource the task to a large group of people in the form of an open call. One advantage of this approach is that we may have more than one translation per query and in this way, we may be able to capture different ways people would follow to describe and, hence, to search for such a query.

Automatic Query Translation. Another option for query translation is to use a system like Logos [2] that provides natural language translations for relational queries expressed in SQL. Logos represents various forms of structured queries as directed graphs and annotates the graph edges with template labels that are used by the translation mechanism.

With the help of the query writer and translator, we can get pairs $<q, d>$ and store them in the system repository. Now each stored query q is represented by its description d. Hence, in order to find q we need to match d. Viewing d as a document, our problem becomes a document search problem that could be solved using an IR approach. A user with a particular analytical task in mind provides a textual description d_s of the desired query or computation. We can retrieve all d's that overlap with d_s and return a list of pairs, $<q, d>$, ranked according to how well d matches d_s. The q with the highest ranked d should be the query that is described by d_s.

Following this approach, the remaining components of the system architecture are as follows.

Indexer. For searching stored queries based on their descriptions, the indexer generates and indexes a representation for each d. First, common stopwords, such as prepositions and frequent words (such as "find", "compute", "define", etc.), are removed. Stemming can be used to create a standard representation for different word forms with similar meaning (e.g., 'search', 'searching'). Finally, the resulting words are used to represent d, for example as a term vector.

System Repository. The system repository stores the queries and their descriptions. Inverted files are used to store how terms are found inside query descriptions and allow for higher efficiency during retrieval. We need two such files: one for mapping terms to descriptions, $t \rightarrow d$, and one for mapping descriptions to queries, $d \rightarrow q$.

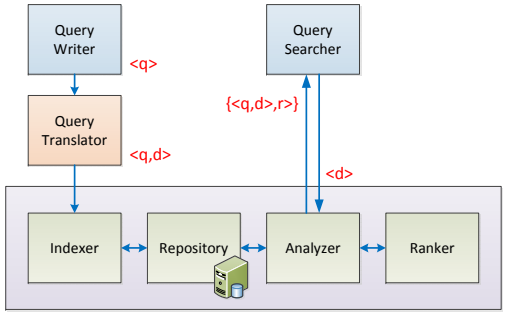

Figure 2: Seven dwarfs to the rescue

Query Searcher. A query interface allows people to search for queries stored in the system. A user provides a description d_s for the desired query. d_s may be a set of keywords or it may be a more free-text description. The system returns a list $\{<q, d>, r\}$ such that d of q matches d_s and r is the matching score. The interface can show the parts of each d that match d_s. The user can select one of the proposed queries and use it as it is or modify it and execute it. Or if not yet satisfied with the returned queries, she could change her desired query description d_s and search again.

The system generates the list $\{<q, d>, r\}$ with the help of two components, the analyzer and the ranker.

Analyzer. The query analyzer is responsible for taking the description d_s and transforming it to a representation that can be matched against the stored information. For example, we could represent d_s as a term vector.

Ranker. The ranker uses the index $t \rightarrow d$ to retrieve all descriptions d that contain common terms with d_s and then computes the similarity between d and d_s using a similarity function, such as the cosine. The result is a list $\{<q, d>, r\}$ ranked according to how well d matches d_s.

This approach to our problem seems reasonable. However, the seven dwarfs can save Snow White if no Evil Queen exists. In what follows, we discuss some of the challenges that complicate the process.

3. THE EVIL QUEEN

Challenges come with the query translation and the interpretation of SQL semantics and query computation.

Different query translations (NL Gap). When humans are involved in providing the explanation of queries, this has the advantage that they will potentially describe the query in a very natural, user-friendly way that other people can understand. However, it does not mean that human-provided query descriptions will be 'informative' enough for the system. Let's take the example query in Figure 1. The writer has provided two different descriptions each one going at a different level of abstraction and hence both being very important. However, the short description alone does not capture the query computations involved while the longer one alone is too detailed and does not easily convey the query intent, that is to find bluer galaxies.

On the other hand, an automated solution guarantees that query translations are machine-useful, but it may not be able to cover all possible queries in a good way. For example, there are queries that in SQL look very complicated (for example, queries with nested parts or several joins) and hence, an automated system may also generate a complicated description, whereas a person may be able to summarize the query intent in a more concise and meaningful way.

q1 Find galaxies and their distances for galaxies with g magnitudes between 18 and 19.

```
SELECT TOP 10 objID, distance FROM Galaxy
WHERE cModelMag_g between 18 and 19
```

q2 Find galaxies and their g magnitudes for galaxies with distances between 0.05 and 0.07.

```
SELECT TOP 10 objID, cModelMag_g FROM Galaxy
WHERE distance between 0.05 and 0.07
```

Figure 3: Similar-looking queries with different semantics from the Sky Survey site

Finally, the query searcher may issue a query description that does not match the description kept in the system.

Missing SQL Semantics. If we follow an IR approach and represent query descriptions as bags or vectors of words, we end up missing the intended semantics of the SQL query. As an example, consider the queries q_1 and q_2 shown in Figure 3. Both of them can be represented with the bag of words {galaxy, distance, g_magnitude}. Then, when we store q_1 and q_2 in the system, we have no way to tell them apart. If the user provides a description d_s targeting any of the two queries, the query analyzer would convert d_s to a term representation, and finally both q_1 and q_2 would be ranked as equally relevant to d_s. Even worse, say the system only knows of q_2 but the user is looking for q_1. Then, the system will provide q_2 as relevant, which is not correct at all.

In our example above, it may be relatively simple for the user to see that the returned query is not what she was looking for and how it can be fixed. However, for more realistic queries, which involve more attributes, joins or nested parts, returning a different query or set of queries that are all ranked the same is not only incorrect, but also confusing.

Understanding Query Computations. Comparing queries using their term representations can tell us how similar their query descriptions are. However, it does not necessarily capture how similar computations the queries perform. The user may be looking for a query involving computations that are partially contained in some stored queries but no single query covers them all. Let's use our example from Figure 1. This query contains two main computations: (a) galaxies not saturated, not bright, and not edge, which is represented by this query construct:

```
( Flags & (dbo.fPhotoFlags('SATURATED')
+ dbo.fPhotoFlags('BRIGHT')
+ dbo.fPhotoFlags('EDGE'))) = 0
```

(b) galaxies with difference between outer color and center color less than -0.4, which is described by this piece of SQL:

```
((modelMag_u-modelMag_g)-(psfMag_u-psfMag_g))< -0.4
```

Say the user is searching for a query that contains one of the above computations and also the computation on distance of query q_2 from Figure 3. Ideally, the system would suggest both queries and highlight in each query the part that performs the desired computation. The user could then compose the two queries into one that performs the desired computations.

4. THE SEVEN DWARVES AGAIN

Such challenges indicate that we need to take into account both the semantics of the queries stored in the query pool and also, the hidden semantics in the description provided by the query searcher. We also need to be able to decompose queries and descriptions into smaller meaningful components that can be searched, matched, and combined.

Having humans in the loop annotating not just a query as a whole but also parts of it cannot help much due to the complexity of this task. Hence, we consider using an automatic query translation method, like Logos [2], as an intrinsic part of the system. The input to the query translator is a query, which can be represented as a query graph (i.e., a DAG). This query targets a database, whose schema can be represented as a database graph (i.e., another DAG). The edges of the database graph are connections among database constructs, for example, tables to tables (e.g., through joins or keys) or tables to attributes (e.g., through projections or selections). These edges can be annotated with labels capturing the semantics of edges in natural language; e.g., a join edge connecting the tables `galaxies` and `stars` may have a label 'stars that belong to galaxies'. For more generic and extensible labels, we can use a template language to write labels. In doing so, we essentially decompose a query to smaller pieces accompanied with a template and a respective description. Thus, as we parse a query graph, we can create sentences by composing the template labels according to the query structure. Then the translation problem can be seen as an effort to maximize the query coverage with the smallest number of templates [3].

Query semantics. Using an automatic query translation method as an intrinsic part of our architecture enables some additional features. When a query q is issued, then the translator decomposes it into a set of query constructs, say $\{q_i\}$, and using the library of predefined templates over the database schema, finds those templates $\{t_i\}$ that match $\{q_i\}$ best. Thus, instead of indexing only the query descriptions, we can also index the query structure at the granularity of the templates composing the query q. In particular, we index both *abstract* templates and their *instantiated* values. For example, assuming that a query q contains a predicate: `dbo.fPhotoFlags('BRIGHT')=0`, an abstract template c covering this would be: `'dbo.fPhotoFlags(' + val + ')=0'`, which can produce the sentence:`'galaxies that are not ' + val`. The latter can be instantiated with `val=BRIGHT` and produce a description d:`'galaxies that are not bright`. We keep both the template c and the description d in our index scheme. The former, as we explain below, helps in finding query patterns that may fit in the description d_s posed by the query searcher based on the computation they capture, and the latter can be used for textual search.

NL semantics. We also need to elaborate on the part of our architecture that analyzes a requested description d_s. Treating d_s as a single keyword query may return a very large hit list depending on the size of the query pool. Instead, we decompose d_s into a set of phrases $\{p_j\}$, which we use to create a set of phrasal queries. We create two kinds of phrases: $\{p_j\}=\{p'_j\}\cup\{p''_j\}$. First, the analyzer using an entity resolution module determines which part of d_s corresponds to what query construct (e.g., predicate, selection) and keeps these results as a set of phrases $\{p'_j\}$. For example, for the query q_1 shown in Figure 3, in the following subset of its description '`find distances for galaxies`', two possible p'_j would be: '`find distances`' and '`for galaxies`'; the former is a projection and the latter identifies the table name. Second, the analyzer performs a join of a sliding window over d_s with the abstract templates stored in the index (these are kept in the inner loop of the join). In doing so, it identifies query patterns that may be hidden in d_s and stores them as a set of pattern phrases $\{p''_j\}$. The same example extract from q_1 matches an abstract template '`find <> for <>`' that gives a query template like '`select <> from <>`' and thus, this template can be used as a pattern phrase p''_j.

Next, we probe the index with $\{p_j\}$. The constraint that p_j's correspond to specific query constructs prunes the search space. The result of this task is to get a set of query constructs $\{q_j\}$ corresponding to the phrases $\{p_j\}$.

Composition. As a next step, the analyzer creates groups of query constructs (e.g., subqueries) that can be combined together for producing larger queries, say $\{q_k\}$. The search space is large, but it can be pruned with heuristics that halt the creation of queries not needed (e.g., by using the query patterns found from d_s). For each query q_k, we also have its translation d_k. Thus, we measure the similarity: $sim(d_s,d_k)$ and use it for ranking the results.

5. HAPPY ENDING (?)

Fairy tales should have a happy ending. We believe that this can be the case with our story too. Here, we described a system that allows users to search into a query pool using textual descriptions for queries previously posed that could be useful for their tasks. If a single query cannot be found, then we help users write a new query by composing parts from queries that only partially fit the user search. But, this is only a first step as several challenges remain unanswered.

For example, for some queries it is hard or even impossible to provide good or meaningful translations [5]. A hard case would be a deeply nested query. On the other hand, a large query may produce a large and convoluted translation; this is not useful either. In such cases, we need a laconic query translation, which can be given by the person that wrote the query or with the help of the crowd (e.g., Mechanical Turk).

In our analysis, we have assumed that the persons using the query writer and the query searcher have access to the same database (e.g., they belong to the same research group or research facility). Open problems exist even in such a setting; for example a database may have evolved over time. Going beyond, we envision a global community of users that share their queries and then, use them in their local environments, which may involve different databases with possibly different schemata. In this setting, additional challenges arise. Schema mapping techniques may help, assuming that the users also share their schemata. But we believe that a more practical approach would be to attack this problem with a generalized indexer in synergy with a domain ontology. For example, if a query maps to a star, we can also connect it to the galaxy to which this query belongs to. Similarly, we can deal with naming issues, for example when two database constructs in two different databases refer to the same concept but have different names.

6. REFERENCES

[1] N. Khoussainova, M. Balazinska, W. Gatterbauer, Y. Kwon, and D. Suciu. A Case for A Collaborative Query Management System. In *CIDR*, 2011.

[2] A. Kokkalis et al. Logos: a system for translating queries into narratives. In *SIGMOD*, 2012.

[3] G. Koutrika, A. Simitsis, and Y. E. Ioannidis. Explaining structured queries in natural language. In *ICDE*, 2010.

[4] M. Minock. C-Phrase: A system for building robust natural language interfaces to databases. *DKE*, 69(3):290–302, 2010.

[5] A. Simitsis and Y. E. Ioannidis. DBMSs Should Talk Back Too. In *CIDR*, 2009.