Keyword Search over Data Service Integration for Accurate Results

Vidmantas Zemleris[1], Valentin Kuznetsov[2] and Robert Gwadera[3]
on behalf of the CMS Collaboration
[1]DiSCC, Faculty of Mathematics and Informatics,Vilnius University, Lithuania
[2]Cornell University, USA
[3]École Polytechnique Fédérale de Lausanne, Switzerland

Abstract. Virtual data integration provides a coherent interface for querying heterogeneous data sources (e.g., web services, proprietary systems) with minimum upfront effort. Still, this requires its users to learn the query language and to get acquainted with data organization which may pose problems even to proficient users. We present a keyword search system, which proposes a ranked list of structured queries along with their explanations. It operates mainly on the metadata, such as the constraints on inputs accepted by services. It was developed as an integral part of the CMS data discovery service, and is currently available as open source.

1. Introduction

Virtual Data Integration (VDI) is a lightweight[1] approach to integrate heterogeneous data sources where data physically stays at its origin and is requested only on demand. It works as follows: (i) queries are interpreted and sent to relevant services; (ii) the corresponding responses are consolidated eliminating inconsistencies in data formats and entity naming, and (iii) the responses are finally combined. However, this approach forces the users to learn the query language and to get familiar with data organization, which is often not straightforward, especially without a direct access to the data at the services.

In this work, we present a keyword search system which simplifies the interaction with VDI by proposing a ranked list of structured queries. The system operates "off-line" using metadata, such as constraints on inputs accepted by services. It was developed at the *CMS Experiment, CERN,* where it makes a part of a data integration tool presented in the next section.

2. Data Aggregation System: a tool for virtual data integration

The *Data Aggregation System (DAS)*[1, 2] integrates several services, where the largest stores 700 GB of relational data. DAS has no predefined schema, thus only minimal service mappings are needed to describe differences among the services. It uses simple structured queries, formed of an entity to be retrieved, and some selection criteria. Optionally, the results can be further filtered, sorted, or aggregated.

As be seen in Figure 1, DAS queries closely match the physical execution flow demanding users to be aware of it (motivated by large amounts of data the services manage). The keyword search relaxes this need of knowing the internals.

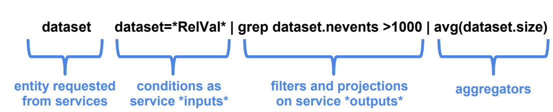


Figure 1. A DAS query: *get an average size of datasets matching *RelVal* with more than 1000 events*

[1] c.f. Publish–subscribe is not applicable to proprietary (reluctant to change) systems; data-warehousing is too complex when large portions of data are volatile or when only limited interfaces are provided by services.

3. Problem definition

Given a keyword query, $kwq = (kw_1, kw_2, .., kw_n)$, we are interested in translating it into a ranked list of the best-matching structured queries. We are given the following metadata:

- *schema terms*: entities and their attributes (*inputs* to the services or their *output* fields)
- *value terms*: for some fields a list of values; but for the most only the *constraints* on data-service inputs (mandatory inputs, regular expressions defining values accepted).

4. Overview of Our Solution

4.1. From keywords to structured query suggestions

In the first step the query is cleaned up and tokenized identifying any quoted tokens or other structural patterns.

In the second step by employing a number of entity matching techniques the "*entry points*" are identified: for each keyword we obtain a list of schema and value terms and a rough estimate of the likelihood.

In the third step different permutations of *entry points* are ranked by combining the scores of individual keywords. In the same step, interpretations that are not compatible with the data integration system are pruned out.

Example. Consider the following keyword query: `RelVal 'number of events'> 100`. Tokenization results in: `'RelVal'; 'number of events>100'`. The entry points include:

```
'RelVal' → (1.0, input-value:  group=RelVal)
'RelVal' → (0.7, input-value:  dataset=*RelVal*)
'number of events>100'→(0.93, filter: dataset.nevents>100)
'number of events>100'→(0.93, filter: file.nevents>100)
```

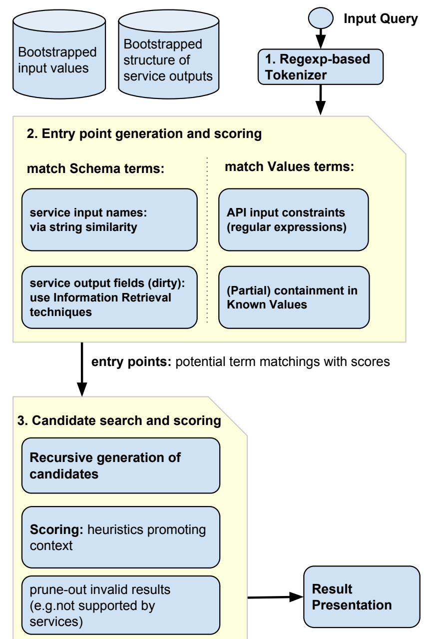


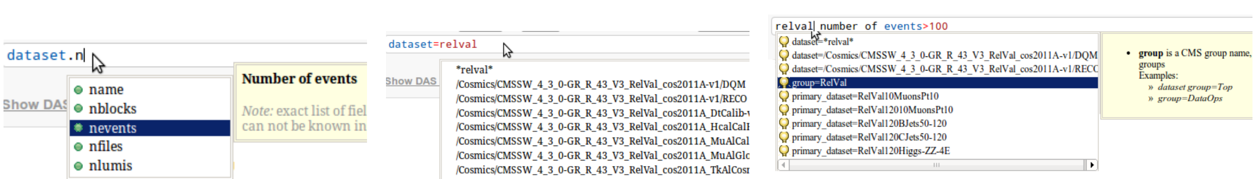
Figure 2. Keyword search stages

Notice that in the token matchings above `RelVal` and `'number of events'` are ambiguous. Lastly, a ranked list of query suggestions is obtained (see Section 5.3) as shown in Figure 3.

Figure 3. Results of a keyword search: structured query suggestions

4.2. Helping users to type queries: autocompletion prototype

To aid users in typing the queries live context-dependent suggestions are shown and a query coloring is provided (see Figure 4). The autocompletion is based on CodeMirror[3], a JavaScript-based versatile text-editor.



(a) list of attributes (b) value matches (c) arbitrary keywords give multiple interpretations (value, entity or attribute name)

Figure 4. Context-dependent autocompletion (prototype)

5. The Components of Our Keyword Search System

5.1. Tokenization (step 1)

At first, the keyword query is standardized (e.g., removing extra spaces, standardizing date formats). Next, the query is tokenized recognizing phrases in quotes and operator expressions (e.g., *nevent > 1, 'number of events'=100, 'number of events>=100'*). This is accomplished by splitting the query using a regular-expression pattern.

5.2. Entry point generation (step 2)

5.2.1. Matching value terms For some fields, a list of values is available. If so, several cases are distinguished: a full match, a partial match, and a match containing wildcards. Otherwise, regular expressions constraining inputs, accepted by services, are used; these are divided into multiple levels of accuracy. Also, we down-rank the unlikely interpretations, where entity names are matched as values (e.g., an entity name *'block'* contained within values of *dataset*).

5.2.2. Matching schema terms We use a combination of these metrics: full, lemma, and stem matches, and a stem match within a small string edit-distance (in order of decreasing weights).

5.2.3. Matching names of fields in service outputs We need to identify multiword keyword chunks corresponding to names of the fields in structured query results: many of these names are unclean, containing technical, irrelevant and common terms as they are obtained directly from service responses in JSON or XML formats. Thus, inspired by [4], we employ *whoosh*[5], an Information Retrieval (IR) library, where for each field in service outputs we create a "multifielded document" which contains field's technical name, its parent, its base-name, and its title if one exists. To find the matches, the IR engine is queried for each chunk of k-nearby keywords (using $k \leq 4$). The IR ranker uses BM25F scoring function, where "document fields" are assigned different weights and phrase matches are scored higher. Finally, the IR score is directly used as a score for the generated match.

5.3. The ranker and scoring functions (step 3)

In this step different combinations of the entry points are explored and ranked. The final score is obtained by combining the scores of individual keywords, $score_{tag_i|kw_i}$, and scores returned by contextualization rules, $h_j(tag_{i,i-1,..,1})$. We experimented with two scoring functions: (i) the average of scores, as used in *Keymantic* [6], and (ii) the sum of log-likelihoods. At first the probabilistic approach seemed to be more sensitive to inaccuracies in entry point scoring, but when this was improved, it became clearly better than the averaging approach.

$$averaging\ score(tags) := \frac{\sum_{kw_i \subset kwq} \left(score_{tag_i|kw_i} + \sum_{h_j \in H} h_j(tag_{i,i-1,..,1}) \right)}{\#\ non\ stopword\ keywords}$$

$$likelihood\ score(tags) := \sum_{kw_i \subset kwq} \left(\log \left(score_{tag_i|kw_i} \right) + \sum_{h_j \in H} h_j(tag_{i,i-1,..,1}) \right)$$

The following contextualization rules were used: (i) promote interpretations where the nearby keywords refer to related schema terms, e.g., entity name and its value, and (ii) promote common use-cases, e.g., retrieving entity by its "primary key".

Currently the ranker is performing an exhaustive search with early pruning of suggestions that are not supported by the services. There exist more complex alternatives, but this one is the simplest that allows early pruning, unlimited contextualization, and listing multiple results.

6. Related work

The Keymantic [7, 6] answers keyword queries over relational databases with limited access to the data instances. First, individual keyword matches are generated using similar techniques as presented in Section 5.2, but focusing on less specific domain than ours. Second, to obtain the global ranking, the assignment problem (assigning tags to keywords) extended with weight contextualization[2], which is discussed in Section 7, is considered. The resulting labels are interpreted as SQL queries and presented to users. We noticed that summing the log-scores instead of plain scores as used in Keymantic, gives a better ranking quality, especially, if these scores are good approximations of the respective likelihoods (see Section 5.3).

The KEYRY [8] took a different approach to the earlier problem allowing to incorporate users feedback. It uses a sequence tagger based on Hidden Markov Model (HMM). The initial HMM parameters can be estimated through heuristic rules (e.g., promoting related tags). To produce the results, the List-Viterbi [9] is used to obtain top-k most probable keyword taggings, which are later interpreted as SQL queries. Once sufficient amount of logs or users' feedback is collected, the HMM can be improved through the supervised or unsupervised training [10]. The accuracy of this method is comparable to that of the Keymantic [8].

Finally, Guerrisi et al. [11] focused on answering full-sentence open-domain queries over the web-services using techniques of natural language processing. Instead, we focus on closed-domain queries without restricting the input to full sentences.

7. Discussion: Assignment Problem with contextualizations

7.1. The standard Assignment Problem

Given n keywords and m tags, $n \leqslant m$, and a n x m matrix of weights, the *assignment problem,* reformulated for keyword search, asks to find a maximum *weighted bipartite matching*, i.e., to maximize the sum of weights such that each keyword is assigned to one tag, and each tag is chosen no more than once. This can be efficiently solved in $\Theta(n^2m)$ by Munkres algorithm. In short, it splits the assignment problem into two easier ones (see [12–14] for details):

(i) Maintain a set of constraints that restrict the currently admissible matches to be "cheap enough"

(ii) Solve N unweighted bipartite assignments: start with an empty matching, find an augmenting path to increase the size of matching. Along the path, the state of edges is flipped matching new ones or deselecting the matched ones; if no augmenting path exists, loosen the constraints on the weights

To efficiently list k best results one can use Murty's algorithm [15] running in in $\Theta(kn^3m)$. To get each additional result, it involves solving $n-1$ smaller assignments with Munkres.

7.2. Supporting the Contextualizations

The contextualization adds additional interdependencies between the assignments, e.g., the tag_j of a keyword kw_i is more likely if its nearby keyword has a related tag_y.

To support this in Keymantic[7, 6] some internal steps of Munkres algorithm were modified. When the size of the matching is increased, the newly matched cells are contextualized, while the unmatched ones are uncontextualized: this triggers weight updates in the dependent cells.

However, the problem with this modification is that unmatching a currently matched cell may lead to weight updates in some other currently matched cells, possibly making them not admissible anymore. Consequently, this may lead to a violation of some of the assumptions of the algorithm, e.g, that each iteration increases the size of matching [12, p. 250]. As a result, we suggest that further investigations are needed.

[2] i.e. conditional increase of the final score if the nearby keywords have related labels assigned

Algorithm 1 top-k assignments with limited contextualization (sketch)

1. Solve once the problem without contextualizations, $\Theta(n^2m)$. The result will be used in the later steps.
2. Enumerate all C contextualization possibilities (in the depth-first order, to reuse matrix modifications)

2.1. Use Murty's algorithm [15] to get top-k results over contextualized cost-matrix.
  - with "Dynamic Munkres" [14], the older solutions can be reused, costing $\Theta(nm)$ per modified matrix line.
  - further, the expected runtime of Murthy's algorithm can be considerably improved [16].

3. Merge all of the top-k solutions found in step 2.1.

7.3. Solution for low number of contextualizations

We are not aware of any method allowing to compute the top-k optimal solutions efficiently to the earlier problem. Fortunately, the problem is simpler *if the number of all contextualization possibilities is low*: one could simply enumerate all of the possibilities and combine the solutions to the standard assignment problem. Still, it is worth observing that, in this special case, there exist large similarities between the contextualized cost matrices, which may be reused. As this is out of the scope of this work, only a brief idea is provided in Algorithm 1.

Assuming that each tag assignment may change at most one line in the weight matrix, we get complexity of $\Theta(n^2m) + C * k * (n-1) * \Theta(nm) = \Theta(Ck * n^2m)$, which is better than simply running Murty over all contextualization possibilities: $C * k * (n-1) * \Theta(n^2m) = \Theta(Ck * n^3m)$.

8. Conclusions

We have presented an implementation of a keyword search over a virtual data-service integration, adapted to the specifics of the *CMS Experiment*. The users' feedback has shown that in data integration, which provides only a limited access to explore the data, an interactive autocompletion can be successful in helping the users to compose semistructured queries.

The public availability of corporate, governmental and other data services is increasing as well as the popularity of repositories and tools for integrating them[3]. Whereas, the availability of user-friendly interfaces is becoming an increasingly important issue. Future challenges may include answering the queries over much larger numbers of data tables and data services, and answering the more complex queries than considered here.

References

[1] V. Kuznetsov, D. Evans, and S. Metson. The CMS data aggregation system. *Procedia Computer Science*, 1(1):1535 – 1543, 2010. ISSN 1877-0509. doi: 10.1016/j.procs.2010.04.172. ICCS 2010.

[2] G. Ball, V. Kuznetsov, D. Evans, and S. Metson. Data Aggregation System - a system for information retrieval on demand over relational and non-relational distributed data sources. *Journal of Physics: Conference Series*, 331(4):042029, 2011.

[3] CodeMirror - a versatile text editor implemented in JavaScript for the browser. http://codemirror.net/.

[4] K. Y. Itakura and C. L. Clarke. A framework for BM25F-based XML retrieval. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '10, pages 843–844, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0153-4. doi: 10.1145/1835449.1835644.

[5] Whoosh Python Search Library. http://bitbucket.org/mchaput/whoosh.

[6] S. Bergamaschi, E. Domnori, F. Guerra, M. Orsini, R. T. Lado, and Y. Velegrakis. Keymantic: semantic keyword-based searching in data integration systems. *Proc. VLDB Endow.*, 3(1-2):1637–1640, Sept. 2010. ISSN 2150-8097.

[7] S. Bergamaschi, E. Domnori, F. Guerra, R. Trillo Lado, and Y. Velegrakis. Keyword search over relational databases: a metadata approach. In *Proceedings of the 2011 international conference on Management of data*, pages 565–576. ACM, 2011.

[8] S. Bergamaschi, F. Guerra, S. Rota, and Y. Velegrakis. A hidden markov model approach to keyword-based search over relational databases. *Conceptual Modeling–ER 2011*, pages 411–420, 2011.

[9] N. Seshadri and C. Sundberg. List Viterbi decoding algorithms with applications. *Communications, IEEE Transactions on*, 42(234): 313–323, 1994.

[10] S. Rota, S. Bergamaschi, and F. Guerra. The list Viterbi training algorithm and its application to keyword search over databases. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1601–1606. ACM, 2011.

[11] V. Guerrisi, P. La Torre, and S. Quarteroni. Natural language interfaces to data services. *Search Computing*, pages 82–97, 2012.

[12] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity.* Courier Dover Publications, 1998.

[13] F. Bourgeois and J.-C. Lassalle. An extension of the Munkres algorithm for the assignment problem to rectangular matrices. *Communications of the ACM*, 14(12):802–804, 1971.

[14] G. A. Mills-Tettey, A. Stentz, and M. B. Dias. The dynamic hungarian algorithm for the assignment problem with changing costs. 2007.

[15] K. G. Murty. Letter to the Editor—An Algorithm for Ranking all the Assignments in Order of Increasing Cost. *Operations Research*, 16 (3):682–687, 1968.

[16] M. L. Miller, H. S. Stone, and I. J. Cox. Optimizing Murty's ranked assignment method. *Aerospace and Electronic Systems, IEEE Transactions on*, 33(3):851–862, 1997.

[3] such as the *YQL,* or the "*Google Fusion Tables*" focusing on regular users instead of developers