Keyword Search over Data Service Integration for Accurate Results

Vidmantas Zemleris[1] **and Valentin Kuznetsov**[2]
on behalf of the CMS Collaboration

[1]DiSCC, Faculty of Mathematics and Informatics,Vilnius University, Lithuania
[2]Cornell University, USA

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 namings, and (iii) the responses are finally combined. However, this approach forces the users to learn the query language and to get familiarized with data organization, which is often not straight-forward, especially without 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 "offline" using metadata such as constraints on inputs accepted by services. It was developed at the *CMS Experiment, CERN,* where it makes part of a data integration tool called *Data Aggregation System (DAS)*[1, 8].

2. DAS - a tool for virtual data integration

DAS integrates several services, where the largest stores 700GB 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 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). Keyword search relaxes this need of knowing the internals.

Figure 1. a DAS query: *get 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 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 most, only *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, employing a number of entity matching techniques, the "*entry points*" are identified: for each keyword, we obtain a list of schema and value terms it may correspond to 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, the *interpretations* 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)
```

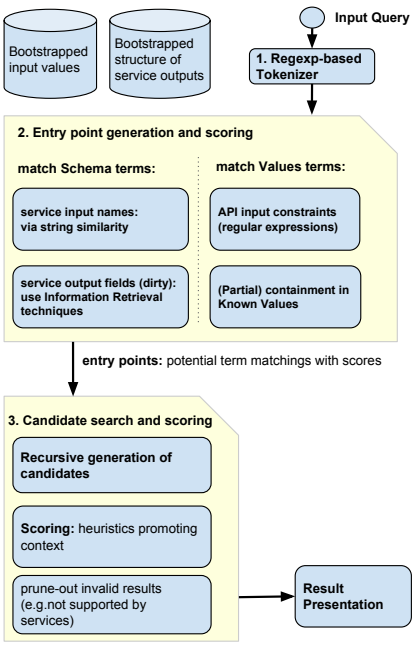


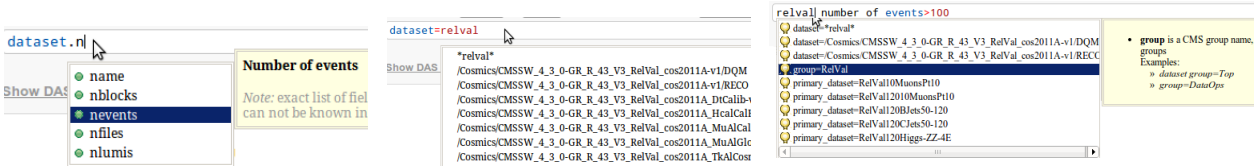
Figure 2. Keyword search stages

Clearly, from the above token matchings, `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 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 query coloring is provided (see Figure 4). This is implemented on top of CodeMirror's javascript-based "source-code editor" library implementing a custom parser and autocompletion routines.



(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 (stage 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 on a regular-expression pattern defining all such cases.

5.2. Scoring individual keywords (stage 2)

Matching the value terms For some fields, a list of possible values is available. If so, different cases are distinguished: full match, partial match, and matches containing wildcards. Otherwise, regular expressions describing inputs accepted by services are used distinguishing multiple levels of accuracy among them. Also, we down-rank the unlikely interpretations, where entity names are matched as values (e.g., entity name 'block' is contained within values of dataset).

Matching the schema terms We use a combination of these metrics: full, lemma, stem matches, and a stem match within a small string edit-distance with a lower weight.

$$sim(A,B) = \begin{cases} 1, & if\ A = B \\ 0.9, & if\ lemma(A) = lemma(B) \\ 0.7, & if\ stem(A) = stem(B) \\ 0.6 \cdot edit\_dist(stem(A), stem(B)), & otherwise \end{cases}$$

Matching names of fields in service outputs We also need to identify multi-word keyword chunks corresponding to names of fields in query results: many of these field names are unclean, technical names, with irrelevant and common terms, as they are obtained directly from service responses in JSON/XML. Thus, we employ *whoosh*, an Information Retrieval (IR) library, where for each field in service outputs we create a "multi-fielded document" containing: field's technical name, its parent, its base-name, and human readable form if exists. To find the matches, we query the IR library 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 full phrase matches are scored higher. After normalizing the IR score, it is used directly as entry point score.

5.3. The ranker and scoring functions (stage 3)

At this stage, different combinations of the entry points are explored and ranked. The final score is obtained 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*[3], and (ii) the sum of log-likelihoods (scores are rough estimations of likelihood). At first the two methods seemed to perform almost equally well**,** with the probabilistic approach being more sensitive to inaccuracies in entry point scoring, but it became clearly better when the accuracy of entry point generation was improved.

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

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

Contextualization rules used: a) promote interpretations where the nearby keywords refer to related schema terms. e.g. entity name and it's value b) promote common use-cases, e.g. retrieving entity by it's "primary key", e.g. dataset dataset=*Zmm*

5.3.1. Implementation details Currently the ranker is implemented as exhaustive search with early pruning to remove the suggestions not supported by the services. Being implemented in *cython*[2], this already gives sufficient performance, often dominated by the entry points stage.

There exist more complex alternatives, but this one is the simplest one that allows early pruning, unlimited contextualization and listing multiple optimal results.

6. Related works

6.1. Keyword search over data-services

Keymantic [3, 4] answers keyword queries over relational databases with limited access to the data instances. First, individual keyword matches are scored as entry points using similar techniques as we did, but focusing on less concrete domain than ours. Then, to obtain the global ranking**,** they consider the problem of "weighted bipartite matching" (of keywords into their tags) extended with weight contextualization[2]. Finally, the resulting labelings are interpreted as SQL queries and presented to users. We noticed that using log-likelihoods instead of just summing the scores (as used in Keymantic) gives better ranking quality, especially if the entry point scores are good approximations of the respective likelihoods (see section 5.3).

KEYRY [5] took a different approach to the problem of Keymantic, with goal to incorporate users feedback. It uses a sequence tagger based on Hidden Markov Model (HMM). At first, the HMM parameters can be estimated through heuristic rules (e.g. promoting related tags). To produce the results, the List-Viterbi [14] 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 supervised or unsupervised training[13]. The accuracy of this approach was comparable to that of Keymantic[5].

Finally, Guerrisi et al. [7] focused on answering full-sentence open-domain queries over web-services using techniques of natural language processing. We instead focus on closed-domain queries with less resources to start with, not limiting the input to full-sentence queries only.

7. Discussion: Assignment Problem extended 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 asks to find a maximum *weighted bipartite matching*, i.e. to maximize the sum of chosen 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 well-known Munkres algorithm. In short, it splits the assignment problem into two easier ones (see [6, 10, 12] for details):

(i) maintain a set of constraints that restrict the currently admissible matches (i.e. edges or cells in matrix) to be cheap enough

(ii) solve N unweighted bipartite assignments: starting with an empty matching, find an augmenting path to increase the size of matching - new edges are selected or existing deselected along the path; if no augmenting path exist, loosen the constraints on the weights

To list k best results efficiently, one can use Murty's[11] algorithm running in in $\Theta(kn^3m)$: for each additional result, which involves solving $n-1$ smaller assignments with Munkres.

7.2. Supporting the Contextualizations

In Keymantic [3, 4], to support the contextualizations, some internal steps of Munkres algorithm have been modified. When the size of the matching is increased (i.e. an augmenting path is found), the newly matched cells are contextualized, while the newly unmatched are uncontextualized. The (un)contextualization updates weights of the dependent cells.

However, we worry that this change may impact either optimality of results or even correctness of the algorithm. The problem is that unmatching a currently matched cell, may lead to uncontextualization of some other currently matched cells, possibly making them not admissible anymore (because of uncontextualization, their weight has decreased). Consequently,

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

this may lead to violation of some of algorithm's assumptions, such as (i) once the cell becomes admissible it stays so, or (ii) that each iteration increases the size of matching by at least one. Thus, we suggest that more investigation is needed.

We are not aware of any method allowing to efficiently compute optimal top-k solutions to this extended problem. But, the problem is easier *if the number of all contextualization possibilities is low* - one could simply enumerate over 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, and the sub-solutions to the shared parts can be reused. As this is out of scope of this work, only a brief idea is provided:

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

2.1. use Murty's[11] algorithm to get top-k results over contextualized cost-matrix.
- with "Dynamic Munkres"[10], the older solutions can be reused, costing $\Theta(nm)$ per modified matrix line.
- further, the expected run time of Murthy's algorithm can be greatly improved[9].

3. Merge all of top-k solutions found in step 2.1. Complexity: $\Theta(n^2m)+C*k*(n-1)*\Theta(nm) = \Theta(Ck*n^2m)$

cf. $C*k*(n-1)*\Theta(n^2m) = \Theta(Ck*n^3m)$ if simply running Murty's over each possible contextualization.

8. Conclusions

We have presented an implementation of keyword search over virtual data-service integration, adapted to particularities of our specific domain. The early users feedback has shown that, in data integration that provide only limited access to explore the data, the interactive auto-completion can be a successful ingredient in helping the users to compose the semi-structured queries. Also we have opened up a couple of issues for further discussion.

The public availability of corporate, governmental and other data services is increasing as well as the popularity of data service repositories and tools for integrating them (such as the *YQL,* or the "*Google Fusion Tables*" focusing on regular users instead of developers). Whereas, availability of user-friendly interfaces is becoming an increasingly important issue. Future challenges may include answering the queries over much larger number of data tables or data services and answering the more complex queries than considered by us.

Acknowledgments

Part of this work was conducted as a master thesis project between *École Polytechnique Fédérale de Lausanne* and *CMS Experiment, CERN*. The first author is thankful to Robert Gwadera who supervised the project for his valuable advice.

References

[1] 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.

[2] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

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

[11] 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.

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

[13] 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.

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