Keyword Search over Data Service Integration for Accurate Results

Vidmantas Zemleris[1] **and Valentin Y Kuznetsov**[2]

on behalf of the CMS Collaboration

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

[2]Cornell University, USA

E-mail: [1]vidmantas.zemleris@cern.ch

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 require 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 is a lightweight[1] approach to integrate heterogeneous data sources where data physically stays at its origin, and is requested only on demand. Queries are interpreted and sent to relevant services, those responses are consolidated eliminating inconsistencies in data formats and entity namings, and finally combined. However, this forces the users to learn the query language and to get acquainted 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 proposes a ranked list of structured queries with their explanations. This 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 an open-source data integration tool called *Data Aggregation System (DAS)*[1, 2].

2. DAS - a tool for virtual data integration

DAS integrates a dozen of 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 fig.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 nevents>1000*

[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 this 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

Firstly, the query is cleaned up and tokenized identifying any quoted phrase tokens, operators or other structural patterns.

Then, employing a number of entity matching techniques, the "*entry points*" are identified: for each keyword (or their combination), we obtain a list of schema and value terms it may correspond to and a rough estimate of the likelihood.

Lastly, different permutations of *entry points* are evaluated and 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'`. Then, each token may yield some entry points:

```
'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)
...
```

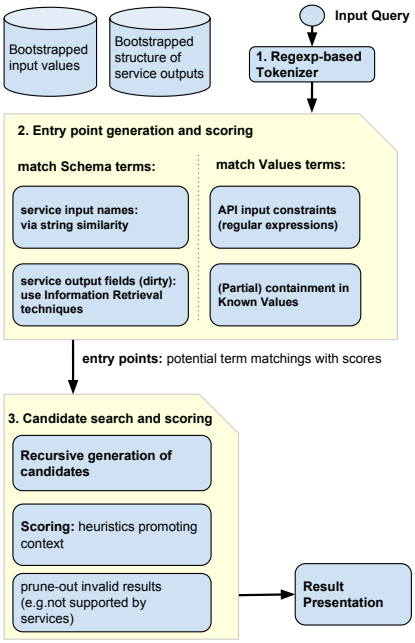


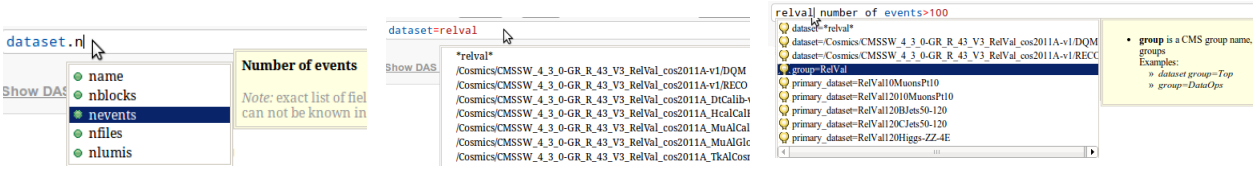
Figure 2. Keyword search stages

It can be seen that both `RelVal` and `'number of events'` are ambiguous. The final results obtained in step 3, where entry point scores are combined, are displayed 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. 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 quite trustful 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 JSON/XML responses. 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.

5.2. 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: a) the average of scores, as used in *Keymantic*[3], and b) 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*

5.2.1. 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*, 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 [4, 3] 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]. To cope with contextualization some internal steps of Munkres algorithm have been modified, this change is discussed in the following section. Finally, the resulting labelings are interpreted as SQL queries and presented to users.

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 [6] 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[7]. The accuracy of this approach was comparable to that of Keymantic[5].

Finally, Guerrisi et al. [8] focused on answering full-sentence open-domain queries over web-services using techniques of natural language processing[3]. We instead focus on closed-domain queries with less resources to start with, not limiting the input to full-sentence queries only.

7. Discussion

....

We also found that using log-likelihoods instead of just summing the scores (as used in Keymantic) gives better ranking, especially if the entry point scores are good approximations of the respective likelihoods.

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 (and data sources in general) 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.

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

[3] Using focus extraction it finds the main entity, splits the query into its constituents, classifies the domain of each constituent, and then tries to combine and resolve these constituents over the data service interfaces (recognizing the intent modifiers such as adjectives as inputs to services)

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] Kuznetsov V, Evans D and Metson S 2010 *Procedia Computer Science* **1** 1535 – 1543 ISSN 1877-0509 iCCS 2010 URL http://www.sciencedirect.com/science/article/pii/S1877050910001730
[2] Ball G, Kuznetsov V, Evans D and Metson S 2011 *Journal of Physics: Conference Series* **331** 042029 URL http://stacks.iop.org/1742-6596/331/i=4/a=042029
[3] Bergamaschi S, Domnori E, Guerra F, Orsini M, Lado R T and Velegrakis Y 2010 *Proc. VLDB Endow.* **3** 1637–1640 ISSN 2150-8097 URL http://dl.acm.org/citation.cfm?id=1920841.1921059
[4] Bergamaschi S, Domnori E, Guerra F, Trillo Lado R and Velegrakis Y 2011 Keyword search over relational databases: a metadata approach *Proceedings of the 2011 international conference on Management of data* (ACM) pp 565–576 URL http://dl.acm.org/citation.cfm?id=1989383
[5] Bergamaschi S, Guerra F, Rota S and Velegrakis Y 2011 *Conceptual Modeling–ER 2011* 411–420
[6] Seshadri N and Sundberg C 1994 *Communications, IEEE Transactions on* **42** 313–323
[7] Rota S, Bergamaschi S and Guerra F 2011 The list viterbi training algorithm and its application to keyword search over databases *Proceedings of the 20th ACM international conference on Information and knowledge management* (ACM) pp 1601–1606
[8] Guerrisi V, La Torre P and Quarteroni S 2012 *Search Computing* 82–97