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; [†]Member 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 a new 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 [1]. It works as follows: (i) queries are interpreted and sent to relevant services; (ii) the corresponding responses are consolidated with the removal of 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 CMS, the Compact Muon Solenoid experiment at CERN, where it makes a part of an open source VDI tool presented next.

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

At the CMS experiment at CERN, the *Data Aggregation System* (DAS)[2–4] 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 which consist 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 proposed keyword search approach relaxes this need of knowing the internals.

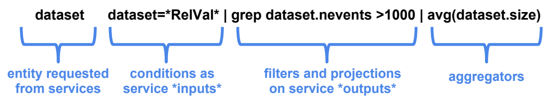

dataset
dataset=*RelVal* | grep dataset.nevents >1000 | avg(dataset.size)
entity requested from services
conditions as service *inputs*
filters and projections on service *outputs*
aggregators

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

[1] cf. publish–subscribe (hard to apply to proprietary systems reluctant to change) and data-warehousing (problematic when large portions of data are volatile or when only limited interfaces are provided by services).

3. Problem definition and solution overview

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 composed of:

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

For example, in Figure 3, `‘dataset’` can be both an entity or input name, `‘dataset.nevents’` is an output field, whereas `‘RelVal’` can be an input value of `dataset`, `primary_dataset`, and `group` fields.

The keyword search works as follows. In the first step the keyword query is cleaned up and tokenized identifying any quoted tokens or other structural patterns.

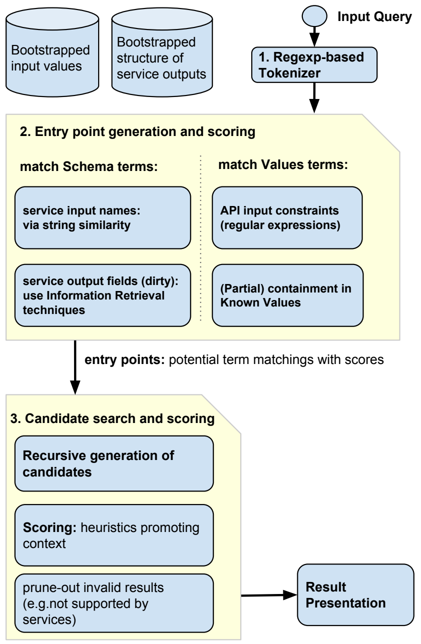
In the second step each token is assigned with a list of its interpretations and a rough estimate of each interpretation's likelihood (called *entry point*). By using a mixture of entity matching techniques each token is considered individually and interpreted as either *schema* or *value term*.

In the third step the permutations of *entry points*, representing a matching of keywords into their interpreted meanings, are enumerated and ranked by combining the entry point scores.

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

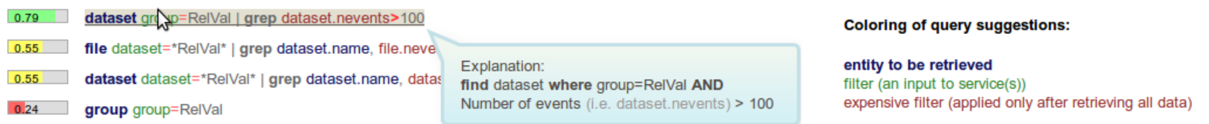
```
‘RelVal’ → (1.0, input-value:  group=RelVal)
‘RelVal’ → (0.7, input-value:  dataset=*RelVal*)
‘number of events>100’→(0.93, filter: dataset.nevents>100)
‘number of events>100’→(0.93, filter: file.nevents>100)
```

Finally, execution of the third step yields a ranked list of query suggestions as shown in Figure 3. For more details on the internals of the processing steps see Section 4.



Input Query
Bootstrapped input values
Bootstrapped structure of service outputs
1. Regexp-based Tokenizer
2. Entry point generation and scoring
match Schema terms:
match Values terms:
service input names: via string similarity
API input constraints (regular expressions)
service output fields (dirty): use Information Retrieval techniques
(Partial) containment in Known Values
entry points: potential term matchings with scores
3. Candidate search and scoring
Recursive generation of candidates
Scoring: heuristics promoting context
prune-out invalid results (e.g.not supported by services)
Result Presentation

Figure 2. Keyword search stages



0.79 dataset group=RelVal | grep dataset.nevents>100
0.55 file dataset=*RelVal* | grep dataset.name, file.neve
0.55 dataset dataset=*RelVal* | grep dataset.name, datas
0.24 group group=RelVal

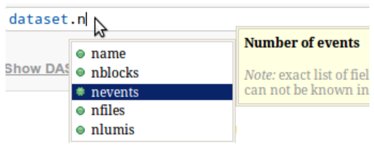
Explanation:
find dataset where group=RelVal AND
Number of events (i.e. dataset.nevents) > 100

Coloring of query suggestions:

entity to be retrieved
filter (an input to service(s))
expensive filter (applied only after retrieving all data)

Figure 3. Results of a keyword search for *`RelVal ‘number of events’> 100`*

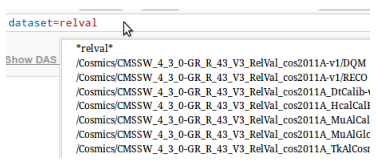
To aid users in typing the queries, live context-dependent suggestions are shown and the different parts of a query are colored by their semantic role (see Figure 4). The autocompletion is based on CodeMirror[5], a versatile JavaScript-based text-editor.



dataset.n
Show DAS
name
nblocks
nevents
nfiles
nlumis

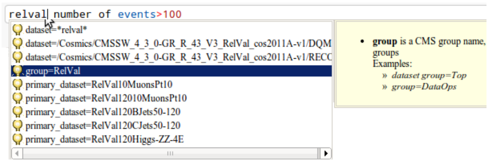
Number of events
Note: exact list of fiel
can not be known in

(a) list of attributes



dataset=relval
Show DAS
relval
/Cosmics/CMSSW_4_3_0-GR_R_43_V3_RelVal_cos2011A-v1/DQM
/Cosmics/CMSSW_4_3_0-GR_R_43_V3_RelVal_cos2011A-v1/RECO
/Cosmics/CMSSW_4_3_0-GR_R_43_V3_RelVal_cos2011A_DtCalib-v
/Cosmics/CMSSW_4_3_0-GR_R_43_V3_RelVal_cos2011A_HcalCalI
/Cosmics/CMSSW_4_3_0-GR_R_43_V3_RelVal_cos2011A_MuAlCal
/Cosmics/CMSSW_4_3_0-GR_R_43_V3_RelVal_cos2011A_MuAlGlc
/Cosmics/CMSSW_4_3_0-GR_R_43_V3_RelVal_cos2011A_TkAlCosr

(b) value matches



relval number of events>100
dataset=*relval*
dataset=/Cosmics/CMSSW_4_3_0-GR_R_43_V3_RelVal_cos2011A-v1/DQM
dataset=/Cosmics/CMSSW_4_3_0-GR_R_43_V3_RelVal_cos2011A-v1/RECC
group=RelVal
primary_dataset=RelVal10MuonsPt10
primary_dataset=RelVal12010MuonsPt10
primary_dataset=RelVal120BJets50-120
primary_dataset=RelVal120CJets50-120
primary_dataset=RelVal120Higgs-ZZ-4E

- **group** is a CMS group name,
  groups
  Examples:
  » *dataset group=Top*
  » *group=DataOps*

(c) arbitrary keywords give multiple interpretations (value, entity or attribute name)

Figure 4. Context-dependent autocompletion (prototype)

4. Description of the Keyword Search System

4.1. Tokenization (step 1)

At first, the 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'`). Stop words (e.g. a, which, when) are identified and given less importance in the later processing steps.

4.2. Entry point generation (step 2)

This step matches each token into a *schema* or a *value term*. For each token, a list of its interpretations is obtained, including a *score* providing a rough estimation of interpretation's likelihood when considered individually ignoring influence of other nearby tokens.

For matching value terms, a listing of known values is used when available (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. Also, the unlikely interpretations are penalized, e.g. entity names and numbers are questionable wildcard value matches (the values of *dataset* are string typed, but contain numbers and an entity name *'block'*).

The schema terms are matched by checking each keyword for: full, lemma, and stem matches, and a stem match within a small string edit-distance (in the order of decreasing scores).

Finally, matching fields in service outputs involves identifying multiword keyword chunks corresponding to the field names (e.g. `number of events in a file` $\rightarrow$ `file.nevents`). Many of these names are directly extracted from service responses in JSON or XML formats, which were not directly intended to be used by humans: names contain irrelevant or common terms. Thus, inspired by [6], we employ *whoosh*[7], an Information Retrieval (IR) library, where for each field in service outputs we create a "multifielded document" which contains the field 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 the BM25F scoring function [6], 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 entry point match.

4.3. Ranking the query suggestions (step 3)

In this step various permutations of the entry points, forming *mappings* between keywords kw_i into their interpretations tag_i, are evaluated and ranked. The score of a mapping is obtained by combining the entry point scores, $score_{tag_i|kw_i}$, and scores returned by contextualization rules, $h_j \in H$. The latter aim to account for keyword interdependencies by promoting permutations where (i) the nearby keywords refer to related schema terms, e.g., entity name and its value or (ii) matching frequent use-cases, e.g. retrieving entity by its "primary key".

We experimented with two ranking functions: the average of scores (1), as used in Keymantic [8, 9] (see Section 5), and the sum of log-likelihoods (2). At first the probabilistic approach (2) seemed more sensitive to inaccuracies in entry point scoring, but after improvements to the accuracy of entry point generation it became clearly better than the averaging approach (1).

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

$$likelihood\ score(mapping) := \sum_{tag_i \subset mapping} \left(\log\left(score_{tag_i|kw_i}\right) + \sum_{h_j \in H} h_j(tag_{i,i-1,..,1}) \right) \tag{2}$$

4.4. Implementation details

The keyword search as well as the DAS data integration system is implemented in Python. The ranker simply enumerates all the possible mappings 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. The ranker is implemented in *cython* [10] which, by compiling the critical parts into a bare C code, allow to easily obtain sufficient performance even if exhaustive search is performed.

5. Related work

Keymantic [8, 9] 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 4.2, but focusing on a less specific domain than ours. Second, to obtain the global ranking, the assignment problem (associating each keyword with a tag/interpretation) extended with weight contextualization[2] is considered (see Section 6). 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 4.3).

The KEYRY [11] 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 algorithm [12] 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 method is comparable to that of Keymantic [11].

Finally, Guerrisi et al. [14] focused on answering full-sentence open-domain queries over the web-services by using techniques of natural language processing. Instead, we focus on closed-domain queries without restricting the input to full sentences.

6. Discussion: Contextualized Weighted Assignment Problem

At a larger scale, more efficient ranking algorithms are need than discussed in Section 4.3 (and they would add additional assumptions or complexity). In Section 5, the contextualized weighted assignment problem was introduced as one of approaches to the keyword search over VDI. In this section, stepping away from our current implementation, we discuss the theoretical basis of this approach and the related works. We raise some unresolved issues and propose an efficient algorithm for a special case when the number of contextualizations is very low.

6.1. Introduction: Weighted Assignment Problem

Given n keywords and m tags, $n \leqslant m$, and a $n \times m$ matrix of scores (called *weights*), the assignment problem, reformulated for the keyword search, asks to find a maximum weighted bipartite matching - maximize the sum of weights such that each keyword is assigned to one tag, and a tag is chosen no more than once. This can be efficiently solved in $\Theta(n^2 m)$ by Munkres algorithm. In short, it splits the assignment problem into two easier ones (see [15–17] 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 this path, flip the state of edges: match new ones or deselect 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 [18] running in in $\Theta(kn^3 m)$. To get each additional result, it involves solving $n-1$ smaller assignments with Munkres. We've seen that solving the basic Weighted Assignment Problem can efficiently generate the assignments, however this still would not account for interdependencies between the keywords interpretations.

[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 the problem without contextualizations once, $\Theta(n^2m)$. 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 [18] to get top-k results over contextualized cost-matrix.
    - with "Dynamic Munkres" [17], 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 [20].
3. Merge all of the top-k solutions found in step 2.1.
```

6.2. Supporting the Contextualizations and Issues with that

Even if keyword queries have no clear structure such as the natural language, it was noticed that the nearby keywords are often related [19]. To account for these likely dependencies, the contextualization adjusts the scores depending on the context, e.g., say keyword kw_i was assigned tag_i, then the nearby keywords become more likely to get the tags related to tag_i, and thus their scores get increased.

To support this in Keymantic[8, 9] 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 [15, p. 250]. As a result, we suggest that further investigations are needed.

6.3. Solution for a low number of contextualizations

We are not aware of any method allowing to efficiently compute the top-k optimal solutions to the earlier problem (Keymantic can efficiently list only the approximately best results). Fortunately, the problem is simpler *if the number of all contextualization possibilities is low*: one could simply enumerate all of contextualization possibilities and combine solutions to the basic assignment problem. It is worth observing that, in this special case, there exist large similarities between the contextualized cost matrices, thus parts of sub-solutions can 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: $C * k * (n-1) * \Theta(n^2m) = \Theta(Ck * n^3m)$.

7. 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 queries more complex than the ones considered in this work.

References

[1] Z. I. Anhai Doan, Alon Halevy. *Principles of data integration.* Number 9780124160446. Morgan Kaufmann, 2012. 497p.

[3] e.g. *YQL[21]* for developers, or "*Google Fusion Tables*" [22] and "*Yahoo Pipes*" [23] that focus on regular users

[2] 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.

[3] 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.

[4] DAS - an opensource tool for virtual data integration. `https://github.com/dmwm/DAS`.

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

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

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

[16] 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.

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

[18] 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.

[19] R. Kumar and A. Tomkins. A characterization of online search behavior. *IEEE Data Engineering Bulletin*, 32(2):3–11, 2009.

[20] 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.

[21] Yahoo! Query Language. `http://developer.yahoo.com/yql`.

[22] H. Gonzalez, A. Halevy, C. S. Jensen, A. Langen, J. Madhavan, R. Shapley, and W. Shen. Google fusion tables: data management, integration and collaboration in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 175–180. ACM, 2010.

[23] Yahoo! Pipes. `http://pipes.yahoo.com/pipes`.