Keyword Search over Data Service Integration for Accurate Results

Vidmantas Zemleris
Vilnius University, Lithuania
vidmantas.zemleris@cern.ch
for the benefit of CMS Collaboration

Valentin Kuznetsov
Cornell University, USA
vkuznet@gmail.com

Summary

Virtual data integration aims at providing a coherent interface for querying heterogeneous data sources (e.g. web services, proprietary systems) with minimum upfront effort in integration. Data is usually accessed through structured queries, such as SQL, requiring to learn the 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.

Context: a system for Virtual Data Integration

"CMS Data Aggregation System" (DAS):

- accepts simple structured queries
- integrates heterogeneous services
  - → parse the query
  - → contact services
  - → eliminate inconsistencies in the responses:
    - ∗ entity naming
    - ∗ data formats (XML, JSON)
  - → combine the responses
- requires only minimal service mappings
  - → no predefined schema
  - → minimal effort in defining services

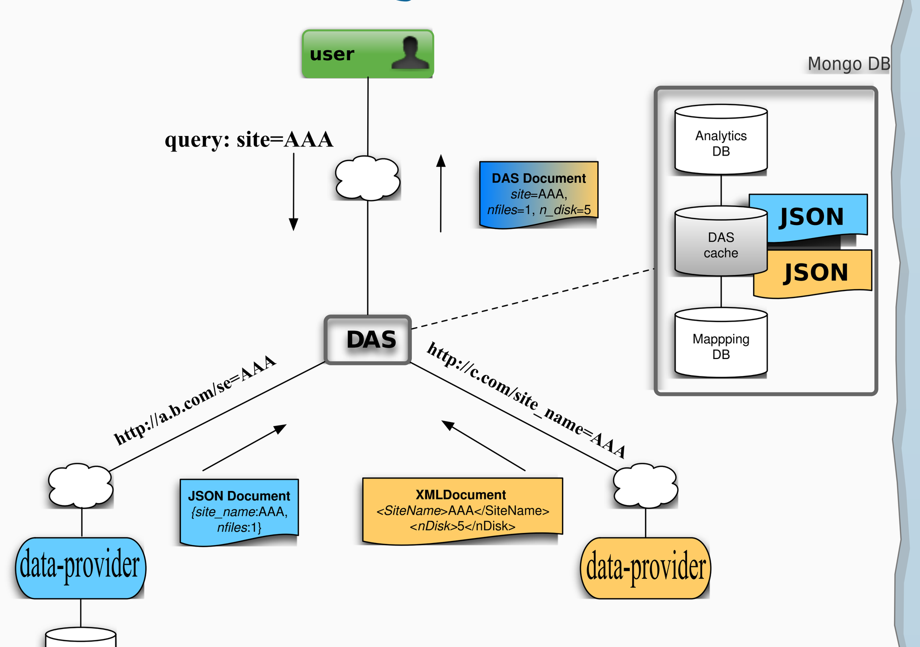


Figure 1: DAS workflow

Queries must specify: entity to be retrieved and filtering criteria. Optionally, the results can be further filtered, sorted or aggregated

dataset dataset=*RelVal* | grep dataset.nevents >1000 | avg(dataset.size), median(dataset.size)

entity requested from services — conditions as service *inputs* — filters and projections on service *outputs* — aggregators

still, it is overwhelming for users to:

- learn the query language
- remember how exactly the data is structured and named

Could keyword queries solve this?

Interpreting Keyword Queries: Problem definition

Input: query, KWQ=$(kw_1, kw_2, .., kw_n)$
ambiguous; nearby keywords are often related

Task: translate it into structured query
made of $tag_j \in$ domain terms: entities and their values, *unknown*, operators

Given: metadata only:

- names of entities and their attributes
  service inputs or their *output* fields
- possible values (only for some inputs)
- *constraints* on data-service *inputs*:
  - → mandatory inputs
  - → regular expressions on values

Example. Consider this query: average size of RelVal datasets with its number of events > 1000

- average RelVal dataset size nevents>1000
- avg(dataset size) RelVal "number of events">1000

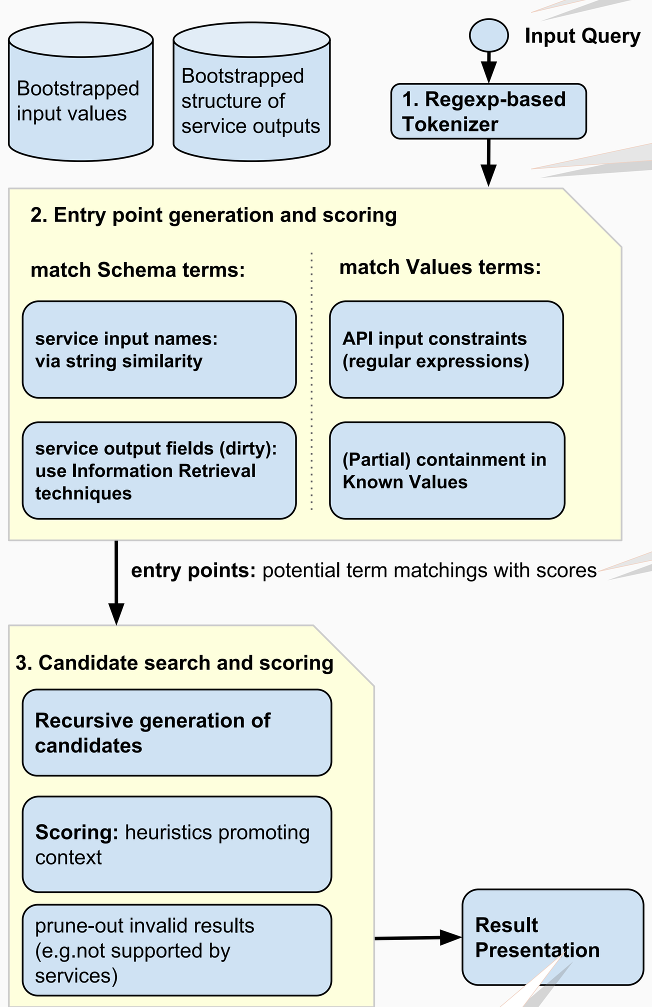
For all, the expected result is:

dataset dataset=*RelVal* | grep dataset.nevents >1000 | avg(dataset.size)

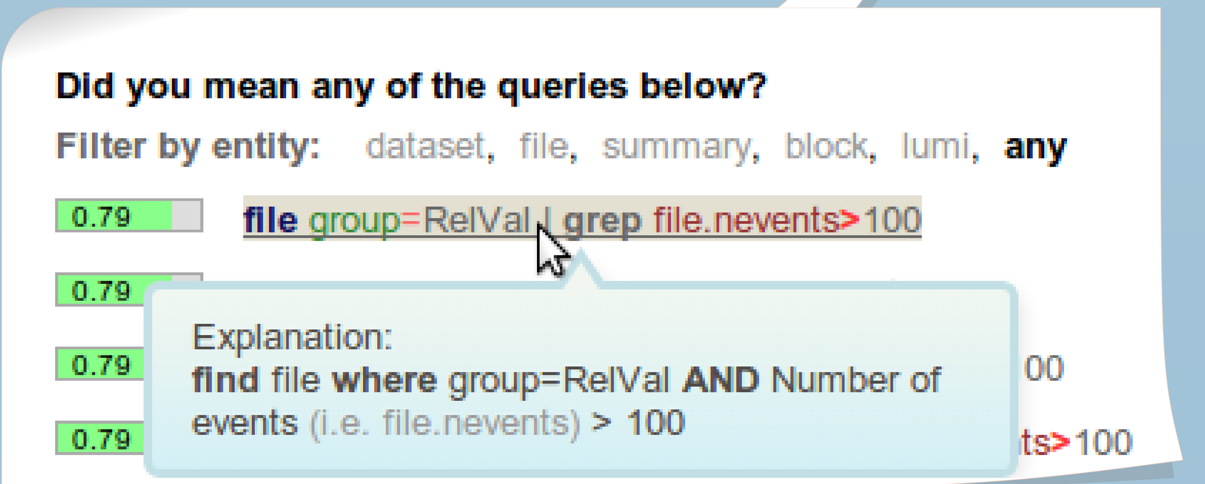
entity requested from services — conditions as service *inputs* — filters and projections on service *outputs* — aggregators

Keyword search overview

1. tokenize the query
   - clean up
   - identify patterns
2. identify and score "*entry points*"
   - score matchings of individual keywords into domain terms with techniques of entity matching and information retrieval
3. combine *entry points* to obtain final score
   - consider various permutations "keyword labellings"
   - promote ones respecting keyword dependencies or other heuristics
   - interpret as structured queries
4. present structured query suggestions ranked accordingly



More info:



Challenges

- keyword queries are ambiguous → return ranked list of structured query suggestions
- querying services is "expensive" → rely on metadata
  - → bootstrap list of allowed values (available only for some fields)
  - → rely on *regexps* with lower confidence (can result in false positives)
- no predefined schema
  - → bootstrap list of fields in service results through queries
  - → some field names are unclean → use IDF (as they come directly from JSON/XML responses)

The ranker

Use exhaustive search:

- allows easily finding optimal solutions, vs. complex methods that'd require post-pruning
- early pruning – filter out many "invalid" candidates e.g. not yet supported by services
- our schema is quite small
  - → *cython*-based implementation is quite fast (bound by *MongoDB* and *Whoosh* IR engines to get entry points)

Scoring function

$$final\ score = \sum_{i=1}^{|KWQ|} \left(\log \left(score_{tag_i|kw_i} \right) + \sum_{h_j \in H} h_j(tag_i|kw_i; tag_{i-1,..,1}) \right)$$

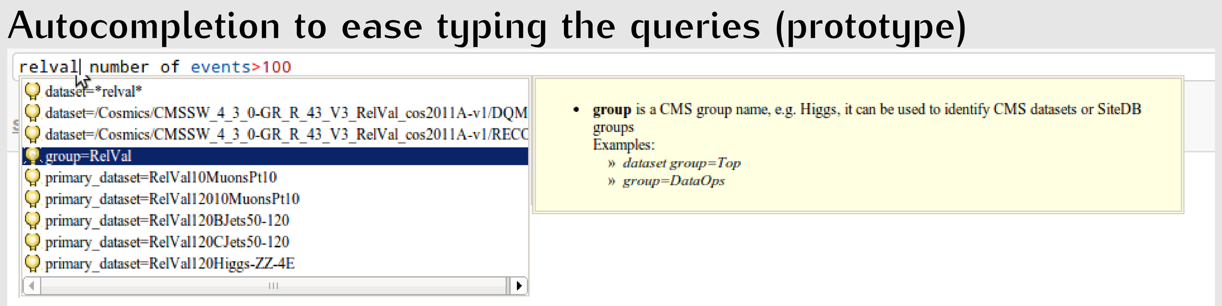
$score_{tag_i|kw_i}$ – likelihood of kw_i to be tag_i (from entry points step)
$h_j(tag_i|kw_i; tag_{i-1,..,1})$ – the score boost returned by contextualization rule h_j given the tag(s) nearby.

Our finding: summing log-likelihoods is better than plain scores (cf. Keymantic)

Related works

- The "Keymantic" – keyword search over databases or data services (the closest work)
  1. score keyword mappings individually (entry points)
  2. solve *"weighted bipartite assignment"* ($kw_i \rightarrow tag_j$) *with contextualizations:*
     - → maximize total sum of weights, selecting each tag only once
     - → uses contextualization rules to account for keyword interdependencies
       - ∗ e.g. <table_name> <its attribute>; <attribute> <its value>;
       - ∗ solves it *approximately* with Munkres algorithm modified to consider *contextualizations:*
         - · contextualize – modify weights of $kw_i \rightarrow tag_j$, if tag_j is "related" to earlier sub-assignments
         - · to get multiple results, repeat recursively forcing/preventing certain sub-assignments
  3. interpret generated mappings as SQL queries
- The "KEYRY" – uses HMM (Hidden Markov Model) to label keywords as schema terms
  - → HMM's initial parameters can be estimated from similar heuristics as above
  - → later machine learning can be used (if logs available)

Autocompletion to ease typing the queries (prototype)



Tokenized query: 'relval', 'number', 'of', 'events>100'

Entry points:

```
RelVal -> 1.0, value:  group=RelVal
RelVal -> 0.7, value:  dataset=*RelVal*
'number of events>1000' -> 0.93, predicate:  dataset.nevents>1000
'number of events>1000' -> 0.93, predicate:  file.nevents>1000
...  and some more with lower scores...
```

Future work

- improve autocompletion prototype
- improve the ranker
- generic ways to improve services' performance, e.g. *materialized views with incremental refresh*

Open problems & ideas

top-k (semi-)optimal assignments with Contextualization?

- could Murty's/Munkres's algs. be adapted for top-k assignments with contextualizations?
  - → this would at least guarantee optimal top-k for with **some** contextualization
  - → out of scope, ask for handouts/chat

Problems with the HMM approach:

- what is modelled is not necessarily same as seen by user
  - → models $kw_i \rightarrow tag_j$, while user sees structured queries
  - → therefore, hard to automatically collect training data