Keyword Search over Data Service Integration for Accurate Results

Vidmantas Zemleris

CMS Computing dept., CERN / LSIR, IC, EPFL
Supervised by: prof. K.Aberer, dr. R.Gwadera (EPFL);
dr. V.Kuztesov (Cornell), dr. P.Kreuzer (CERN)

16th April 2013

Outline

Outline

Preliminaries (1/2)

Virtual data service integration (EII)

- *lightweight* ***virtual*** *integration (vs. data-warehousing*, publish-subscribe)
- usually queried with structured languages, e.g. SQL, YQL, etc
- growing # of sources and applications: corporate, governmental, mashups...
  - *e.g. Yahoo's YQL, Google Fusion Tables, ...*

How it works?

- process the query & send requests to services
- consolidate the results:
  - namings & dataformats (XML, JSON, ..)
  - apply filters, aggregations, service composition

Preliminaries (1/2)

Virtual data service integration (EII)

- *lightweight **virtual** integration (vs. data-warehousing,* publish-subscribe)
- usually queried with structured languages, e.g. SQL, YQL, etc
- growing # of sources and applications: corporate, governmental, mashups...
    - *e.g. Yahoo's YQL, Google Fusion Tables, ...*

How it works?

- process the query & send requests to services
- consolidate the results:
    - namings & dataformats (XML, JSON, ..)
    - apply filters, aggregations, service composition

Preliminaries (2/2)

Example query (in DASQL used at CMS, CERN)

$$\underbrace{\textbf{dataset}}_{\substack{\text{entity requested}\\ \text{from services}}}\quad \underbrace{\textbf{dataset=*RelVal*}}_{\substack{\text{conditions as}\\ \text{service *inputs*}}}\ \textbf{|}\ \underbrace{\textbf{grep dataset.nevents >1000}}_{\substack{\text{filters and projections}\\ \text{on service *outputs*}}}\ \textbf{|}\ \underbrace{\textbf{avg(dataset.size)}}_{\text{aggregators}}$$

Remark: this is close to boolean retrieval + (aggregation XOR projections).

The problem: for users it is overwhelming to:

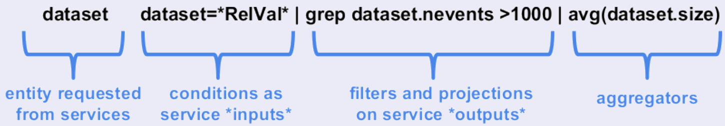
- learn **a query language**
- remember how exactly **data is structured** and **named**

Intuition: Could *Keyword Queries* solve it?

- *list sizes of RelVal datasets where number of events>1000*
- *avg(dataset size) Zmmg 'number of events'>1000*

Preliminaries (2/2)

Example query (in DASQL used at CMS, CERN)


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

Remark: this is close to boolean retrieval + (aggregation XOR projections).

The problem: for users it is overwhelming to:

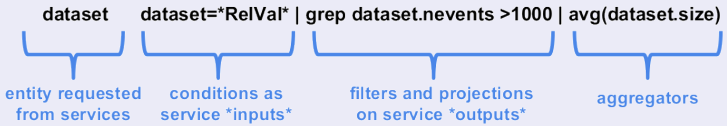
- learn **a query language**
- remember how exactly **data is structured** and **named**

Intuition: Could *Keyword Queries* solve it?

- *list sizes of RelVal datasets where number of events>1000*
- *avg(dataset size) Zmmg 'number of events'>1000*

Preliminaries (2/2)

Example query (in DASQL used at CMS, CERN)



```
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

Remark: this is close to boolean retrieval + (aggregation XOR projections).

The problem: for users it is overwhelming to:

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

Intuition: Could *Keyword Queries* solve it?

- *list sizes of RelVal datasets where number of events>1000*
- *avg(dataset size) Zmmg 'number of events'>1000*

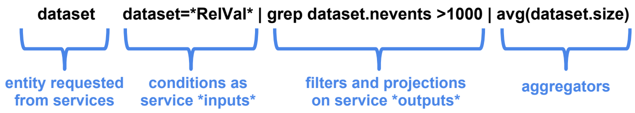
Problem statement

Given:

- schema terms (entity and field names)
- value terms
  - values listing (for some fields)
  - constrains, e.g. regexps, mandatory service inputs
- query: $\text{KWQ} = (kw_1, kw_2, .., kw_n)$

Task: interpret each $kw_i \in KWQ$ as *part of structured query*:

- schema term (result type; projections; or field name in a predicate)
- values term (a value condition in a predicate)
- operator, or *unknown*.



```
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

State of the art

- Nature of Keyword queries:
  - ambiguous: *propose structured queries as results*
  - nearby keywords are often related
- "Keyword Search over EII" received not much attention:
  1. KEYMANTIC - generates SQL suggestions
     - ★ uses heuristics to "cover" keyword interdependencies
  2. KEYRY - same but uses *Hidden Markov Model* (HMM)
     - ★ *List Viterbi* gives "tagging" of keywords, which is interpreted into SQL
     - ★ initially HMM can be estimated from heuristics
     - ★ later supervised and unsupervised learning can be used
- Farther works:
  1. SeCo - Natural Language open-domain queries to compose services
     - ★ focus on closed-domain; both plain keywords and sentences shall work
  2. *Question Answering*, *Natural Language Processing*, *Entity Matching*, *Keyword Search in Structured Databases*

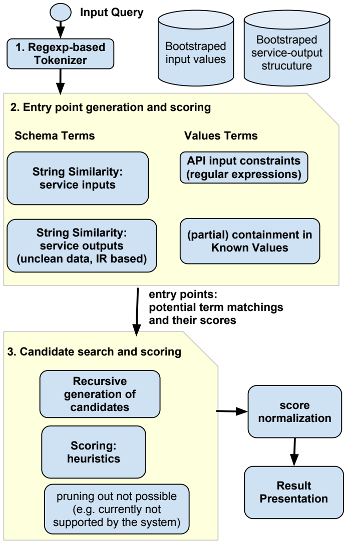
Challenges

- keyword queries are ambiguous
  - solution: ranked list of query suggestions
- no direct access to the data:
  - need bootstrapping values listings (available only for some fields)
  - rely on regexps otherwise –> false positives
- no fully predefined schema
  - bootstrap list of fields through queries and maintain it...
  - some field names are unclean (coming directly from XML, JSON responses)
- unexpected challenges/issues with project realization:
  - lack of concise terminology in the field
  - the area is not so actively researched
  - thus significant effort was needed to choose a *precise topic to focus on*

Outline

Implementation Overview

1. *tokenizer*: clean up; identify patterns
2. identify and score "*entry points*" with
    1. string matching [for entity names]
    2. IR (IDF-based)[unclean fieldnames]
    3. list of known values
    4. regular expressions on allowed values
3. combine *entry points*
    1. consider various *entry point* permutations (keyword labelings)
    2. promote ones respecting keyword dependencies or other heuristics
    3. interpret as structured queries



Input Query
1. Regexp-based Tokenizer
Bootstraped input values
Bootstraped service-output strucuture
2. Entry point generation and scoring
Schema Terms
Values Terms
String Similarity: service inputs
API input constraints (regular expressions)
String Similarity: service outputs (unclean data, IR based)
(partial) containment in Known Values
entry points: potential term matchings and their scores
3. Candidate search and scoring
Recursive generation of candidates
Scoring: heuristics
pruning out not possible (e.g. currently not supported by the system)
score normalization
Result Presentation

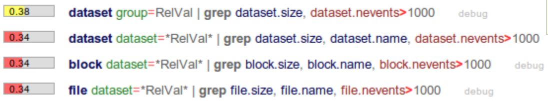
Figure 1: Query processing

Example of query processing

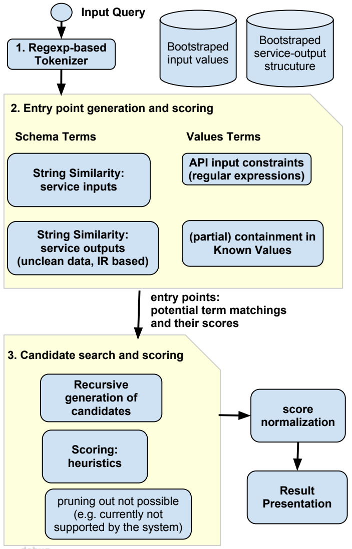
Q:datasets sizes RelVal 'number of events > 1000'

```
Schema terms:
datasets -> 0.9, schema:  dataset
Schema terms (multi-word):
'number of events>1000' ->
     0.93, pred:  dataset.nevents>1000
     0.93, pred:  file.nevents>1000
'dataset sizes'->0.99, project: dataset.size
sizes -> 0.41, project: dataset.size

Value terms:
RelVal -> 1.0, value:  group=RelVal
RelVal -> 0.7, value:  dataset=*RelVal*
...  and some more with lower scores...
```



```
0.38  dataset group=RelVal | grep dataset.size, dataset.nevents>1000  debug
0.34  dataset dataset=*RelVal* | grep dataset.size, dataset.name, dataset.nevents>1000  debug
0.34  block dataset=*RelVal* | grep block.size, block.name, block.nevents>1000  debug
0.34  file dataset=*RelVal* | grep file.size, file.name, file.nevents>1000  debug
```



Input Query
1. Regexp-based Tokenizer
Bootstraped input values
Bootstraped service-output strucuture
2. Entry point generation and scoring
Schema Terms
Values Terms
String Similarity: service inputs
API input constraints (regular expressions)
String Similarity: service outputs (unclean data, IR based)
(partial) containment in Known Values
entry points: potential term matchings and their scores
3. Candidate search and scoring
Recursive generation of candidates
Scoring: heuristics
pruning out not possible (e.g. currently not supported by the system)
score normalization
Result Presentation

Figure 2: Query processing

Step 1: Tokenizer

1. Clean-up
   - remove extra spaces, normalize formatting
   - recognize simple unambiguous expressions
2. Split into tokens on these regular expressions:
   1. [terms] operator value (e.g. "number of events">10, dataset=Zmm)
   2. terms in quotes (e.g. "magnetic field")
   3. individual terms

Step 2: Entry point Generation and Scoring (1/2)

Matching schema terms

- did not work well: string edit-distance, semantic similarity

$$d(w_1, w_2) = \begin{cases} 1, & \textit{if } w_1 = w_2 \\ 0.9, & \textit{if } lemma(w_1) = lemma(w_2) \\ 0.7, & \textit{if } stem(w_1) = stem(w_2) \\ 0.6 \cdot sdist(stem(w_1), stem(w_1)), & \textit{otherwise} \end{cases}$$

$sdist(w_1, w_2) > 0$, iff w_1 and w_2 are within very small string-edit distance (penalize transpositions, and changes in middle)

Matching multi-word unclean schema terms

- some terms are non-informative –> **IDF needed**
- use Information Retrieval library (Whoosh) with BM25F scoring
- create *virtual documents* each representing "a field of an entity"
  - fully-qualified machine readable (e.g. block.replica.creation_time)
    - tokenized+stemmed (e.g. creation_time)
    - context - tokenized+stemmed parent (e.g. block; replica)
  - human readable title, if any (e.g. "Creation time")

Step 2: Entry point Generation and Scoring (2/2)

Matching Value terms:

- Regular expression (regexp) can result in false positives:
  - it do not guarantee that actual match exists as regexp can be loose
  - thus, regexp matches are scored lower than other methods
    - ★ to reduce false positives: exclude regexp match if not in known values, and field's values do not change often
- Known values (strings)
  - these automatically bootstrapped
  - assign decreasing score for: full match, partial match, and keywords with wildcards

Step 3: Answer candidate scoring: formulas

$$score\_avg = \frac{\sum_{i=1}^{|KWQ|} \left(score(tag_i|kw_i) + \sum_{h_j \in H} h_j(tag_i|kw_i; tag_{i-1,..,1}) \right)}{N\_non\_stopword}$$

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

- $score(tag_i|kw_i)$ ~ likelihood of kw_i to be tag_i
- $h_j(tag_i|kw_i; tag_{i-1,..,1})$ - the score boost returned by heuristic h_j given a tagging so far (often all $i-1$ tags are not needed).

Step 3: Answer candidate scoring: heuristics

- Relationships between keywords:
  - nearby keywords refer to related terms (e.g. entity name and it's value)
  - parts of speech of different importances, e.g. stop-words vs. nouns
  - keyword's position (e.g. result type in beginning [focus extraction])
- Qualities of EII system:
  - promote *dataservice inputs* over *filters on their results*
    - it is more efficient, if possible; less false matches
  - common use-case: retrieve an entity given it's "primary key" (or wildcard)
  - service constraints must be satisfied
    - assumption: services directly cover most of requests
    - could useful to show the interpretations that achieve high rank, even if they do not satisfy some constraints (e.g. a mandatory filter is missing)
    - if some keyword can be matched as the requested entity, and mapping of other keywords fits the service constraints

Evaluation: Accuracy

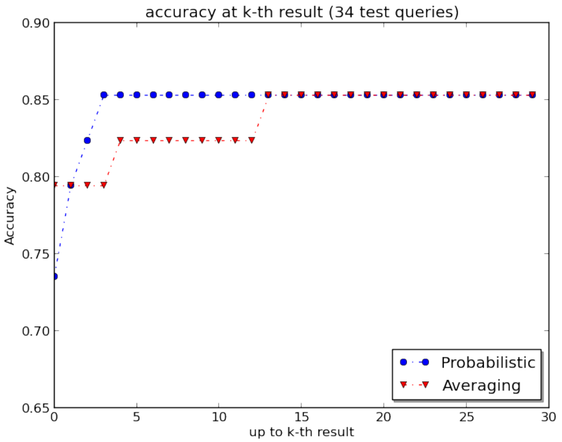
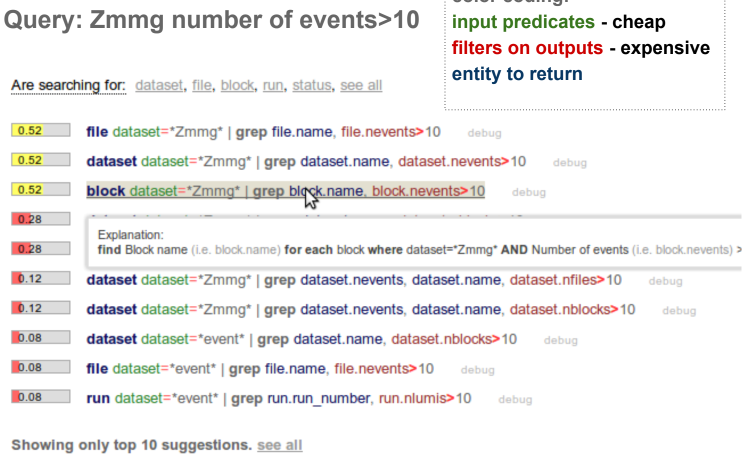


Figure 3: Accuracy comparison of the two scoring methods at kth result

- accuracy of 85% @ 4th suggestion
- testing set is limited - need more live feedback

Presenting the results to the user

color coding:
input predicates - cheap
filters on outputs - expensive
entity to return



Query: Zmmg number of events>10

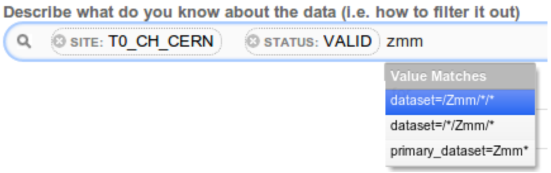
Are searching for: dataset, file, block, run, status, see all

0.52	**file** dataset=*Zmmg* | **grep** file.name, file.nevents>10 debug
0.52	**dataset** dataset=*Zmmg* | **grep** dataset.name, dataset.nevents>10 debug
0.52	**block** dataset=*Zmmg* | **grep** block.name, block.nevents>10 debug
0.28	
0.28	
0.12	**dataset** dataset=*Zmmg* | **grep** dataset.nevents, dataset.name, dataset.nfiles>10 debug
0.12	**dataset** dataset=*Zmmg* | **grep** dataset.nevents, dataset.name, dataset.nblocks>10 debug
0.08	**dataset** dataset=*event* | **grep** dataset.name, dataset.nblocks>10 debug
0.08	**file** dataset=*event* | **grep** file.name, file.nevents>10 debug
0.08	**run** dataset=*event* | **grep** run.run_number, run.nlumis>10 debug

Explanation:
find Block name (i.e. block.name) **for each** block **where** dataset=*Zmmg* **AND** Number of events (i.e. block.nevents) >

Showing only top 10 suggestions. see all

Autocompletion prototype


Describe what do you know about the data (i.e. how to filter it out)

SITE: T0_CH_CERN STATUS: VALID zmm

Value Matches

dataset=/Zmm/*/*

dataset=/*/Zmm/*

primary_dataset=Zmm*

Figure 5: prototype of auto-completion based interface

- autocompletion can be combined with keyword-search to obtain improved results
- it could seamlessly provide feedback for improving system components

Using the feedback for self-improvement (As Backup ???)

Implicit feedback from auto-completion

- improving entity matching (e.g. learned edit-distance)

Users implicit feedback (clicking on the link)

- limited feedback quality - user may click on non-related query
  - ask user to confirm if the final result was the intended one
- sequential ML algorithms, such as HMM, so far modelled not directly the structured query, but *labelling of keywords in query*. we propose to investigate if these could degrade the semi-explicit feedback:
  1. multiple ways to be convert the labelling into structured query: better implicit feedback from autocompletion: we get direct feedback - selections are for separate terms, not for the query as whole, which is not being modelled
  2. feedback could depend on the false positives of the earlier mappings[1], and that may potentially impact the machine learning.

[1] we seen that it is possible for a false matching to result in a correct result! this was more prevalent on ambiguous matching methods, e.g. regexps

Outline

Conclusions

- popularity of data-service integration grows
  - need accessing data easily
- discussed a real-world case and first open-source implementation
  - no assumptions on input but able to use patterns
  - users quite like the idea and the "working prototypes"...
  - main performance issues may be deep in underlying services which were never properly optimized...
- keyword search proposing structured queries can be quite successful
  - for simple schema, no expensive schema ontology needed
- the system will be further supported
  - it will improve efficiency of the physics analysis program by the CMS

Future work

- tuning the accuracy based on users' feedback
- explore the auto-completion further
- explore the Machine Learning approaches once more data is gathered?
- support additional query patterns
- non-functional
  - large parts of keyword search can be moved to client-side
  - performance improvements to the data providers, and keyword search

Outline

Uniqueness of this implementation

1. no assumptions on input query
   - plain keywords vs. full-sentence
   - still can use patterns if present (phrases, predicates/conditions)
2. implements a specific real-world use-case
   1. different selection of entry points / scoring heuristics and entity matching
   2. scoring
   3. specific query language
3. first open-source implementation
   - the code will be further maintained

Project deliverables

1. keyword search engine and related components
    - implementation of entity matching techniques & heuristics
    - code for bootstrapping of: 1) allowed values, 2) fields in service results
    - tuning the system's parameters
    - prototype of advanced auto-completion input widget
    - slight relaxation of DASQL
        - ★ prototype of "simple service orchestration" even then the existing fields are not known in advance
        - ★ gives more power and simplifies the keyword search
2. log analysis and data service performance benchmarking at CMS
    - proposed solutions for data service providers
3. user surveys, presentations and tutorials at the CMS Collaboration
    - constant cooperation with a selected group of ~5+ users for feedback

Project priorities and constraints

Due to the constraints on the project duration, a number of items had to be excluded from the implementation:

- question answering approaches with deep language processing
- complex service orchestration (feeding of outputs into inputs of other services)
  - not directly supported by the EII system
  - the service performance is not adequate for this[2]
- the performance is of lower priority
  - end user's perceived performance is still dominated by services taking minutes to respond
  - performance was already covered by the earlier works.

[2]this due to issues with data service performance and unavailability of basic capabilities such as pagination or sorting of their results; we do not control the data services, so a number of suggestions for the providers have been proposed (see appendix **??**); second, these improvements would take a considerable effort to be implemented, pushing this far beyond the scope of this project

Tuning the scoring parameters

1. tuned individual components to "sufficient" level
   - unit tests and manual testing
2. fine-tune the whole system (by hand)
   - use keyword queries by written users or developer for evaluation
   - important variables to be tuned:
     - weights for regexps, etc
     - likelihood of not taking a keyword
     - BM25F "field" and "query" weights (for IR matching multi-word terms)

Data integration war-stories: Dataservice Performance

References

- prototype online: https://docs-bulk-tool.cern.ch/das/