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.Kuznetsov (Cornell), dr. P.Kreuzer (CERN)

17th April 2013

Outline

1. Introduction
   - Problem statement
   - State-of-the-Art
2. Implementation
3. Conclusions & Future work
4. Backup slides

Outline

1. Introduction
   - Problem statement
   - State-of-the-Art
2. Implementation
3. Conclusions & Future work
4. Backup slides

Preliminaries (1/2)

Virtual data service integration (EII)

- *lightweight **virtual** integration*
  - minimal requirements on services
  - vs. more demanding data-warehousing, publish-subscribe
- queried with structured languages, e.g. SQL, YQL
- growing # of datasources and applications:
  - *e.g.* corporate, governmental, *Yahoo's YQL*, mashups, ...

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)

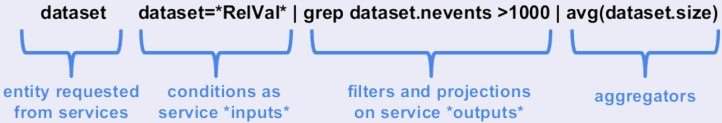
- *lightweight **virtual** integration*
  - minimal requirements on services
  - vs. more demanding data-warehousing, publish-subscribe
- queried with structured languages, e.g. SQL, YQL
- growing # of datasources and applications:
  - *e.g.* corporate, governmental, *Yahoo's YQL*, mashups, ...

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)



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: it is overwhelming to:

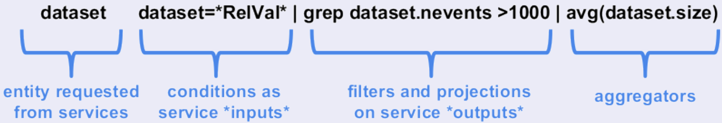
- learn a **query language**
- remember how exactly **data is structured** and **named**

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)



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: it is overwhelming to:

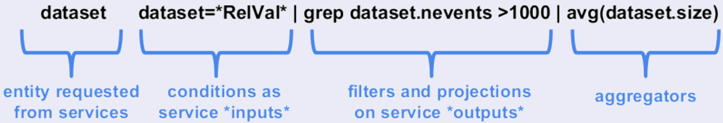
- learn **a query language**
- remember how exactly **data is structured** and **named**

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)



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: 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*

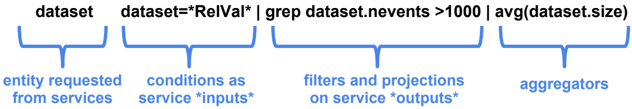
Problem statement

Given:

- schema terms (entity and field names)
- value terms
  - values listing (for some fields)
  - constrains, e.g. regexps, mandatory service inputs
- query: 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
- Less related 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 during 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

1. Introduction
   - Problem statement
   - State-of-the-Art
2. Implementation
3. Conclusions & Future work
4. Backup slides

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

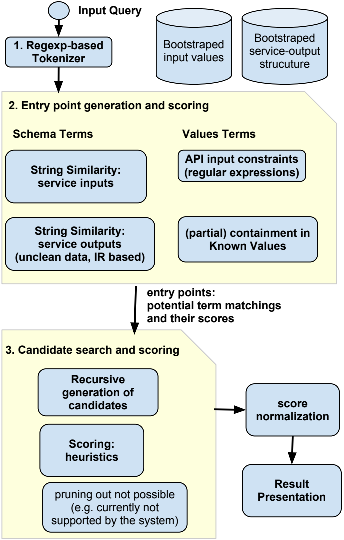

Input Query
1. Regexp-based Tokenizer
Bootstraped input values
Bootstraped service-output strucuture
2. Entry point generation and scoring
Schema Terms
Values Terms
String Similarity: service inputs
API input constraints (regular expressions)
String Similarity: service outputs (unclean data, IR based)
(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

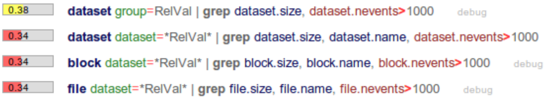
Figure 1: Query processing

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

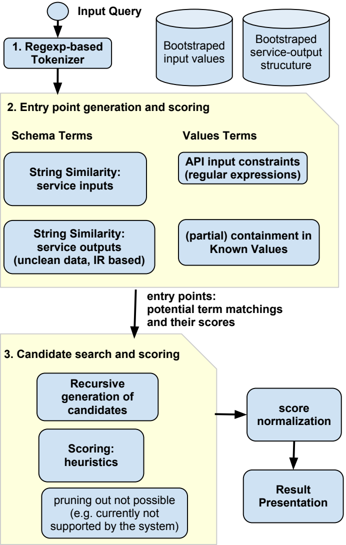

Input Query
1. Regexp-based Tokenizer
Bootstraped input values
Bootstraped service-output strucuture
2. Entry point generation and scoring
Schema Terms
Values Terms
String Similarity: service inputs
API input constraints (regular expressions)
String Similarity: service outputs (unclean data, IR based)
(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 repeated –> IDF needed
- use IR library (*whoosh*) with *BM25F* scoring
- create *virtual documents* each representing "a field of an entity"
  - "technical" field name (e.g. block.replica.creation_time)
    - child: stemmed (e.g. creation time)
    - parents: stemmed (e.g. block; replica)
  - title, if any: stemmed+stopword (e.g. "Creation time")

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

Matching Value terms:

- Regular expression (regexp) can result in false positives:
  - regexp can be too loose
    - to reduce false positives: exclude regexp match if not in known values, and field's values do not change often
  - thus, regexp matches are scored lower than other methods
- 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

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*
- common use-case: retrieve an entity given its "primary key"
- service constraints have to be satisfied
  - if some keyword can be matched as the requested entity, and mapping of other keywords fits the service constraints
  - future work: could be useful to show the interpretations that achieve high rank, even if they do not satisfy some constraints (e.g. a mandatory filter is missing)

Evaluation: Accuracy

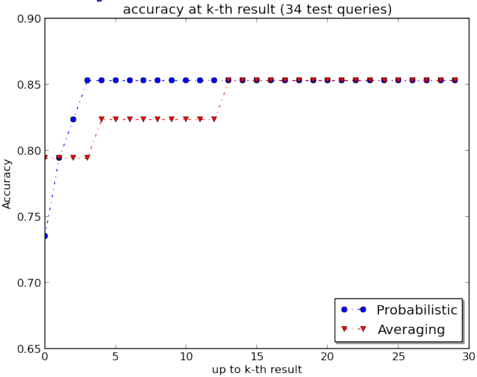


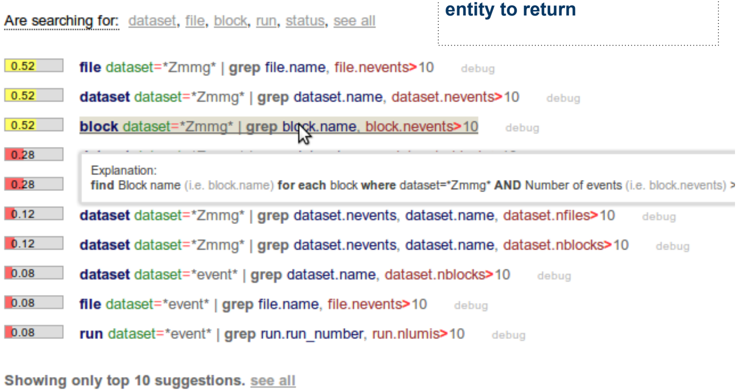
Figure 3: Accuracy comparison of the two scoring methods at kth result

- accuracy of 85% @ 4th suggestion
- “probabilistic” is more accurate
  - but vulnerable to wrong scoring of entry points
  - these scores are just rough estimations
- testing set is limited - need more live feedback

Presenting the results to the user

Query: Zmmg number of events>10

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



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

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

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

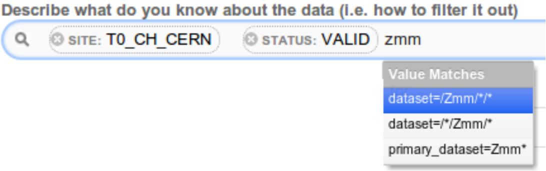
0.08 **dataset** dataset=*event* | **grep** dataset.name, dataset.nblocks>10 debug

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

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

- can be combined with keyword-search
- seamless feedback for improving system components (backup slides)

Outline

1. Introduction
   - Problem statement
   - State-of-the-Art
2. Implementation
3. Conclusions & Future work
4. Backup slides

Conclusions

- popularity of data-service integration grows
  - need accessing data easily
- discussed a real-world case and implementation
  - no assumptions on input but able to use patterns
  - proposed solutions on data-service optimizations (backup slides)
- keyword search proposing ranked queries can be successful
  - for simple schema, no expensive schema ontology needed
  - users liked the idea and prototypes...
- the system will be further supported
  - it will improve efficiency of the physics analysis program by the CMS

Future work

- tuning the accuracy
- explore auto-completion further
- additional query patterns
- explore *machine learning* approaches once more logs gathered?
- non-functional
  - large parts of keyword search can be moved to client-side
  - performance improvements (data providers, keyword search)

Outline

1. Introduction
   - Problem statement
   - State-of-the-Art
2. Implementation
3. Conclusions & Future work
4. Backup slides

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

excluded, due to project constraints:

- question answering & deep language processing
- complex service orchestration
  - not directly supported by the EII system
  - the service performance is not adequate for this[1]
- performance was of lower priority
  - performance dominated by data-services - proposed solutions
  - already covered by the earlier works.

[1] unavailability of basic capabilities such as pagination or sorting of their results; a number of suggestions for the providers have been proposed, but 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-tuned the whole system (by hand)
    - use keyword queries by written users or developer for evaluation

- important variables to be tuned:
    - scoring in matching techniques
        - ⋆ string similarity, regexps, etc
        - ⋆ BM25F “field” and “query” weights
    - likelihood of not taking a keyword
        - ⋆ depending on part-of-speech
    - influence of other heuristics

Using the feedback for self-improvement

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
- investigate impacts to feedback quality:
  1. sequential ML (e.g. HMM in KEYRY) modeled *labelling of keywords* as schema terms, but not into the final result directly
  2. multiple ways to convert the labelling into structured query
     - better implicit feedback from autocompletion: selections are for separate terms, not for the query as whole (but we do not model it)
  3. feedback could depend on the false positives of the earlier mappings[2], and that may potentially impact the machine learning.

[2] 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

Data integration war-stories: Dataservice Performance

- Incremental view maintenance
  - e.g. DBS service (80GB data; 280GB indexes)
    - existing records change rarely
    - joining may large tables (or aggregates)
    - *solutions: materialized refresh fast views with query rewriting;* DBToaster
  - *e.g. 'find files where run in [r1, r2, r3] and dataset=X'*:

```
Dataset (164K) -> Block (2M) -> Files (31M) -> FileRunLumi (902M) <- Runs (65K)
```

- Estimating query run time
  - calculate standard deviations online (4 vars)
  - per different parameter types

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 (entity matching)
    2. custom scoring
    3. specific query language
3. open-source implementation

References

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