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)

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)**

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

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: $\mathrm{KWQ} = (kw_1, kw_2, .., kw_n)$

Task: interpret each $kw_i \in KWQ$ as:

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

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

State of the art

- Nature of Keyword queries:
  - ambiguous: *structured queries as results*
  - nearby keywords are often related
- "Keyword Search over EII" received not much attention:
  1. KEYMANTIC - generates SQL suggestions from keyword query
     - uses heuristics to "cover" keyword interdependencies
  2. KEYRY - also generates SQL suggestions
     - uses Hidden Markov Model (HMM) as first step
- Farther works:
  1. SeCo - Natural Language to compose services
  2. SODA - SQL suggestions over large data-warehouse; rely on metadata ontology

Challenges

- keyword queries are ambiguous
  - solution: ranked list of query suggestions
- no direct access to the data:
  - need bootstrapping values lists (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)

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

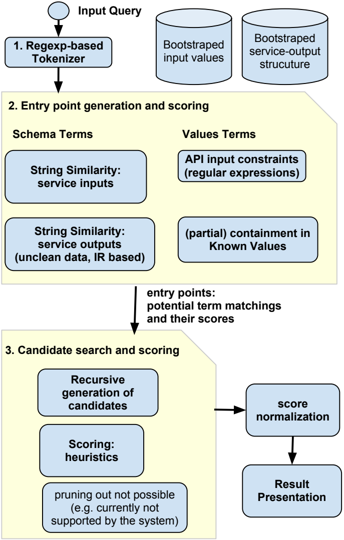

Figure 1: Query processing

Example 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

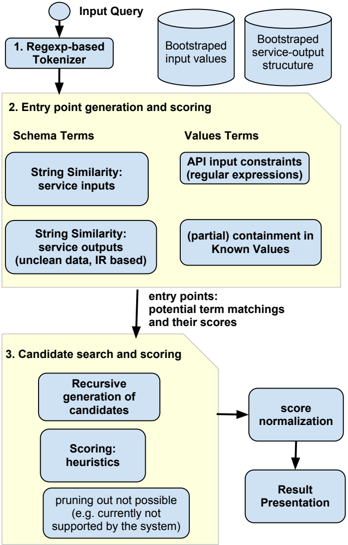

Figure 2: Query processing

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:
  - promote if nearby keywords refer to related schema terms (e.g. entity name and it's value)
  - boost important keywords
    - parts of speech of different importances, e.g. stop-words vs. nouns
    - keyword's position (e.g. expect result type in beginning [focus extraction])
- Qualities of Data Integration System:
  - promote *dataservice inputs* over *filters on their results*
    - it is more efficient, if possible
    - more choices, so more false matches
  - if some keyword can be matched as the requested entity, and mapping of other keywords fits the service constraints
  - common use-case: retrieve an entity given it's "primary key" or wildcard on it
  - to execute the query, service constraints must be satisfied
    - 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)

Evaluation: Accuracy

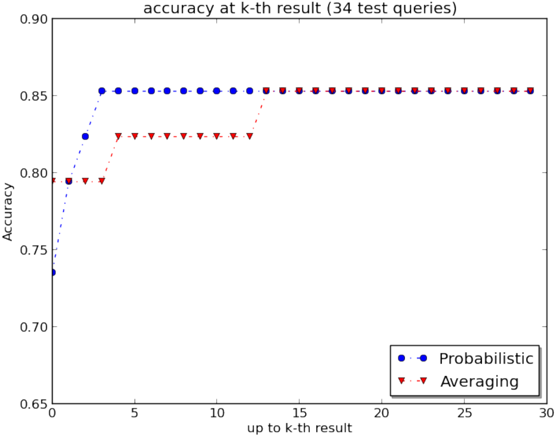

Figure 3: Accuracy comparison of the two scoring methods at kth result

- 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	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

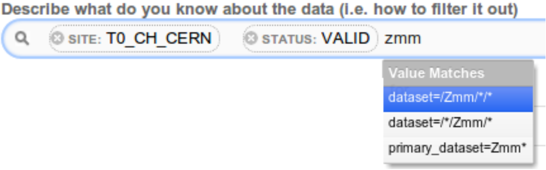

Figure 5: prototype of auto-completion based interface

- 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 ???)

First, the implicit feedback from auto-completion could be useful for evaluating and improving the quality of the entity matching (learned edit-distance metrics, updating the weights of different matching metrics).

Second, the user's selections in auto-completion fields could serve as training data for machine learning-based algorithms (and it is of better quality because user is selecting autocompleted values for separate terms), however it is important to gather sufficiently large sets of high quality feedback, to avoid over-fitting the machine learning models.

Also users implicit feedback (clicking on the link), could be useful, however it is of limited quality (the user may click on it just for figuring out what the query returns even if it was not directly related with his information needs expressed as the the input keywords). An alternative to this could be asking users if the result was what they were asking for, when they see the results of actual structure query (this could be a little bit overwhelming, but looks quite optimal).

Additional problem is that what was so far modelled by the sequential machine learning algorithms, such as HMM, was not directly the structured query, however the *query configurations*

Outline

Conclusions

- growing popularity of data-service integration systems, increase need accessing data easily
  - one successful approach can be keyword search proposing structured queries
  - keyword search in EII (i.e. then no complete access is available) received fairly little attention...
- discussed a real-world case and first open-source implementation
  - able to use patterns
  - users quite like the idea..
  - most of performance problems may be deep in underlying services...
  - this will contribute to improve the overall efficiency of the physics analysis program by the CMS Collaboration
    - the system is going to be further supported

Future work

- better interpretation of patterns in queries
- tuning the accuracy based on users' feedback
- exploring the auto-completion further
- exploring the Machine Learning approaches once more data is gathered?
- technical
  - large parts of keyword search can be moved to client-side
  - performance improvements

Outline

Closely related works

1. **(I do not need that much detail:)**
2. KEYMANTIC
   1. based on metadata, keywords are scored as potential matches to *schema terms* (entity names and their attributes, using some entity matching techniques) or as potential *value* matches (by checking any available constraints, such as the regular expressions imposed by the database or data-services).
   2. Then, these scores are combined**,** and refined by heuristics that increase the scores of query interpretations with the nearby keywords having related labels assigned.
   3. Finally, these labels are interpreted as SQL queries.
3. KEYRY incorporate users feedback through training an Hidden Markov Model's (HMM) tagger taking keywords as its input.
   1. It uses the List-Viterbi [**?**] algorithm to produce the top-k most probable tagging sequences (where tags represent the "meaning" of each keyword).
   2. This is interpreted as SQL queries and presented to the users.
   3. The HMM
      1. is first initialized through the supervised training, but even if no

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

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

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, which is anyway not directly supported by the EII system and the service performance is not adequate for this[1]); and lastly the performance is of lower priority, as the end user's perceived performance is still dominated by services taking minutes to respond, and the performance was already covered by the earlier works.

[1] 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

List of 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

References

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