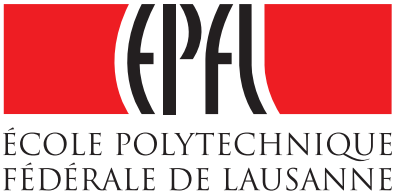
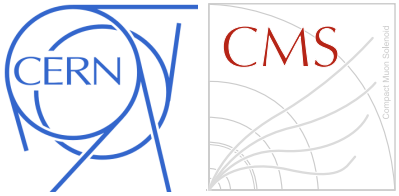




Distributed Information Systems Laboratory
School of Computer and Communications
Swiss Federal Institute of Technology, Lausanne

Computing department
Compact Muon Solenoid Experiment
European Organization for Nuclear Research

Master Thesis

make better & more CS precise title

Keyword Search over Data Service Integration for Precise Results

Vidmantas Zemleris
Section of Computer Science (IC)

14th March 2013

Supervised by:
Dr. Robert Gwadera (LSIR, EPFL)
Prof. Karl Aberer (LSIR, EPFL)
Dr. Valentin Kuznetsov (Cornell Univ., USA)
Dr. Peter Kreuzer (CERN)

Abstract

In the cases when there is no direct access to the data, but only certain interfaces are available (e.g. web services, web forms, proprietary systems, etc), the virtual data integration provides a way to query the sources in a coherent way. Querying is usually done through structured query languages such as the SQL and alike, allowing to obtain the precise results, but implying that the user must learn the query language and has to know how the data is structured. The keyword search is a popular way for finding information, however the **traditional methods** are not applicable, as only a limited access to the data instances is available in this case.

In this work we present a keyword search system that approaches this problem operating on the available: the metadata such as the constraints on allowed values, **analysis of user queries**, and some portions of the data.

Unlike the earlier works, it makes no assumptions on the input query (still being able to leverage the structural patterns in the query, if present) and proposes a ranked list of structured queries along with explanations of their meaning to the end user.

The system is discussed within context of CMS data discovery service where simplicity and capabilities of the search interface places a crucial role for adoption among the end users and their ability to cover their information needs.

Our innovations/distinctiveness from earlier works:

* no assumptions on input
* real-world implementation + war stories
* auto-completion and ideas for further work on incorporating users feedback
* detailed description;
* reference implementation – no open-source exists to our knowledge; still large portions of code needs to be rewritten (as it was initially developed as "research prototypes", and lack of time didn't allow redesigning)

trad of kws @ relational databases (SPARK); also IR, SPARK don't fit as queries as answers needed

Contents

Acknowledgements

This work was financially supported by the Computing group of the Compact Muon Solenoid Experiment at the European Organization for Nuclear Research (CERN). The author of this report is thankful to Dr. Robert Gwadera and prof. Karl Aberer who supervised this Master Thesis project, as well as, to the responsible people at the company, Dr. Peter Kreuzer and Dr. Valentin Kuznetsov, for their support and valuable comments.

1. Introduction

The virtual integration of data-services, also referred to as Enterprise Information Integration (EII), allows querying the sources in a coherent way (eliminating the inconsistencies in data formats such as JSON vs XML, provides naming conventions, combines the results, etc.) and is the most beneficial when the other data integration approaches are not applicable[1]. In EII, data physically stays at its origin, and is requested only on demand, usually, through structured query languages such as the *SQL*, which present a number of user interface issues.

The objective of this work is to research the keyword search as a more intuitive alternative, which, in fact, received little attention in the field of data service integration[10]. Virtual integration presents an additional challenge - only limited access to the data instances is available.

Building on the experience gained while working on an EII system at the *CMS Experiment* at *CERN*, we will focus on the implementation of keyword search and the mechanisms for user feedback, also touching some more distant topics such as usability and performance of an EII.

focus: user interface instead?

1.1. Motivation

At scientific collaborations such as the *CMS Experiment at CERN*, where this work has been conducted, data often resides on a fair number of autonomous systems each serving its own purpose[2]. As data stored on one system may be related to data residing on the others, users are in need of a centralized and easy-to-use solution for locating and combining data from all these multiple sources.

The EII, solves the data integration problem even when data is volatile and systems are heterogeneous and reluctant to change, however the complexity of writing such queries first impacts the simple users, forcing them to learn the "schema" and the query language. However, even the tech-savvy users may have only a vague idea of where exactly to find what they need.

As an another example, the web search engines, which are becoming close to generic question-answering engines[3], could employ the methods presented in this work, for providing the immediate answers to certain types of queries, whose results can not be pre-cached, but are available on the vast quantities of continuously growing public, corporate, or governmental data services. For instance, the query "tnt 123456789"

[1] for instance, publish-subscribe approach is not applicable in the presence of proprietary (and reluctant to change) systems, data-warehousing is too heavy and complex then large portions of data is volatile or when only limited interfaces are provided by proprietary services.

[2] At CERN, due to many reasons (e.g. research-orientedness and need of freedom, politics of institutes involved) software projects usually evolve in independent fashion, resulting in fair number of proprietary systems[14], whereas high turnover makes it harder to extend these systems

[3] for instance, on the Google search engine, information on weather, currency rates, time at given location, etc. is already available through recognition of certain patterns in web queries

can be interpreted as requesting the tracking information for a given TNT shipment tracking code.

1.2. Structure of the work and our contributions

make this part of intro??

First, we present an overview of the state-of-the-art in the field (chapter 2).

The chapter 3, presents the EII system used within this work, **introducing** some real-world issues with EII, such as data-service performance, [users confusion - intricacy of a query language where the fields in the results for same entity differs depending on the query] and discusses ways for solving these.

innovation??? relaxed assumptions? (To check) autocompletion?

Then, in chapter 4, after formally defining the problem of keyword search over EII, a keyword search engine is presented, that given a keyword query, proposes most probable structured queries . We propose a custom string similarity metric, discuss the presentation of results to the end users, and propose combining keyword search with auto-completion in a unique way.

Next, in chapter 4.5, we discuss approaches allowing to incorporate users feedback into keyword search over EII. The earlier mentioned auto-completion would allow getting the feedback of higher quality without overloading the users, that could be used for improving various parts of the system (future work).

Finally, in chapter 5, the developed system is evaluated quantitatively using test queries and qualitatively through user feedback.

add: Related work; conclusions; cms solutions moved to appendix;

Note: The project also included these time-demanding tasks: 1) choosing a precise topic to focus on - because the area is not so actively researched and there is no concise terminology[4], this has took a considerable amount of time, and 2) case analysis at the CMS Experiment included analysing query logs, benchmarking the performance bottlenecks; a users survey, tutorials and presentations.

[4] e.g. virtual data integration, enterprise information integration, data virtualization are used as synonyms throughout different time periods; works on keyword search over EII mostly focused on relational databases (with limited access to data instances)

2. State of the Art in the field

Significant experience has been accumulated in the field of *Enterprise Information Integration*[1] (EII) including: data integration formalisms, query optimization [11], with the recent research focusing on the approaches minimizing human efforts in source integration [1, ch.19]. Meanwhile, to the best of our knowledge, the *Boolean Keyword-based search over data services*, which is our main focus, received little attention from the research community, with only a few attempts [e.g. 6, 3, 10] to address the problem.

Nature of keyword queries

Keyword queries are often underspecified, therefore every possible interpretation shall be included in the results [4]. But for a given keyword query, some interpretations are more likely than the others, therefore, when the users are interested in complete answer sets, the standard approach is to produce a ranked list of most-likely structured queries[2].

other terminology?

Further, it has been noticed that even if keyword queries do not have any clear syntactic structure, keywords referring to related concepts usually come close to each other in the query [15, 4]. Based on this, most of the existing approaches employ the dependencies of nearby keywords for ranking the candidate answers: *Keymantic* [5] includes heuristic rules that score higher whose query interpretations where the nearby keywords have related labels assigned[3], or in machine learning approaches this justifies usage of sequential models such as the Hidden Markov Model in *KEYRY* [3].

TODO: move next to each approach separately

Keyword querying over EII

Two works were identified to be the most closely related to our problem: heuristics-based keyword search (*Keymantic*), and the machine learning approach (*KEYRY*).

Keymantic[6, 5] answers keyword queries over relational databases with limited access to the database instance, which is also the case of data integration [5].
First, based on meta-data,individual keywords are scored as potential matches to *schema terms* [4] (to names of entities and their attributes using some entity matching techniques) or as potential *value* matches (by checking if the keyword matches any available constraints, such as the regular expressions imposed by the database or data-services). Then, based on a number of heuristics the scores are combined, exploiting the earlier mentioned dependencies between the nearby keywords.

it seems it was summing the scores (no log()!) Weighted-bipartite matching as optimization; still exponencial because of first step?

[1] Enterprise Information Integration (EII) is about 'integrating data from multiple sources *without* having to to first load data into a central warehouse'[11, p.1]

[2] e.g. the *SODA*[7] system proposes SQL over a large data-warehouse, or *Keymantic*[5] attempting the same without accessing the data

[3] e.g. schema term's name followed by its value, for instance in query "restaurant Italia", the second term would probably mean name of restaurant

[4] in EII, only limited access to data instances is available, therefore instead of just indexing the all data, the meta-data shall be used

Note: Their assumptions: "keyword can be mapped to only one database term; no two keywords can be mapped into the same database term **[how about multi-keyword-terms?]**. **every keyword plays some role in the query, i.e., there are no unjustified keywords (!)**"

KEYRY [3] attempts to incorporate users feedback through learning an HMM tagger that is given the keywords as its input. It uses the List-Viterbi [20] algorithm to produce the top-k most probable tagging sequences (where tags represent the "meaning" of each keyword). This is converted into a list of SQL queries and presented to the users.
The HMM is first initialized through the supervised training, but even if no training data is available, the initial HMM probability distributions can be estimated through a number of heuristic rules (e.g. promoting related tags). Later, user's feedback can be used for further **supervised** training, while even the keyword queries itself, can serve for unsupervised learning [19].
According to [3] the accuracy of Keymantic and KEYRY systems didn't differ much.

but they mentioned a better evaluation may be needed...

also mention: NL querying over services

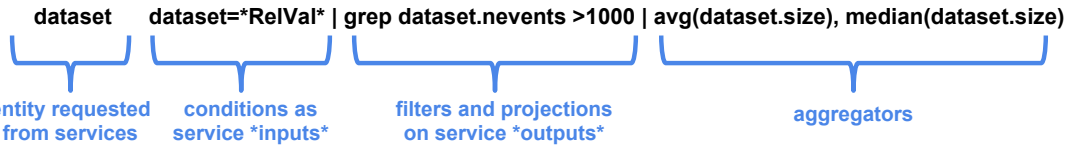
3. Thesis context: EII at CMS, CERN

In this chapter, the context is presented, in which the thesis project was conducted.

The EII system to be extended An *EII* system called *"CMS Data Aggregation System" (DAS) [16, 2]* allows integrated access to a number of proprietary data-sources through simple structured queries (eliminating the inconsistencies in entity naming, data formats, combining the results, etc). DAS uses the *Boolean retrieval model* as users are often interested in retrieving ALL the items matching their query.

augmented with aggregation functions...

Query language and execution The queries are formed specifying the entity the user is interested in (e.g. dataset, file, etc) and providing selection criteria (e.g. attribute=value, attribute BETWEEN [v1, v2]). The results could be later 'piped' for further filtering, sorting or aggregation (min, max, avg, sum, count, median), e.g.:



As seen in the figure above, DAS query language closely corresponds to the physical execution flow over the EII: based on the requested entity and the conditions on service inputs, DAS decides the set of the services to be queried[1]. Then, after retrieving, processing and merging of the results from services, the filters and projections are applied, which are followed by aggregators. The results are cached for subsequent uses. **This close correspondence between the query language and the physical execution has both advantages (users are aware of service constraints) and disadvantages (users have to know exactly how the data structured).**

The Data-Sources The system integrates around 15 data providers. The most of data-sources were created initially focusing on data storage without much attention to its retrieval. The services are not optimized for querying the fairly large data volumes stored[2]: the data is kept in fully-relational fashion, thus queries often require complex joins over large tables; even worse, the interfaces of the services do not allow limiting the number of results to be returned, nor sorting.

(~100 interfaces in JSON, XML, or structured queries)

Issues and User Feedback The EII system suffered from a number of performance and usability problems. System's usability was improved by redesigning the user interface and allowing less restricted keyword search; analysis of the performance bottlenecks

[1] including pre-defined "virtual services", which feed results from one service into inputs of the others

[2] The total sizes of "metadata" that the services refer is growing in order of 1TB/year. Most of this data is stored in relational databases, in Oracle. For instance, the DBS, one of the biggest services, is partitioned among ~10 instances, with the biggest containing 80GB of data + 280GB of indexes in Oracle.

indicated that data providers were not ready for data retrieval - appropriate measures were proposed (see appendix A).

- performance
- the fields that are contained in the results of retrieving some entity depend on the filtering conditions (sometimes it contains all the fields, sometimes only the "primary key" allowing to identify the entity [3]). This is creating additional confusion for the users, as it is not easy to figure out where and how to find what the user is searching for.

Point to appendix: solutions at CMS

[3] e.g. 'dataset dataset=/ZMM/Summer11-DESIGN42_V11_428_SLHC1-v1/GEN-SIM' contains all the possible fields, while 'dataset dataset=/zmm/*/*' only the dataset.name and couple of others

4. Keyword Search over Data Services

4.1. Problem statement

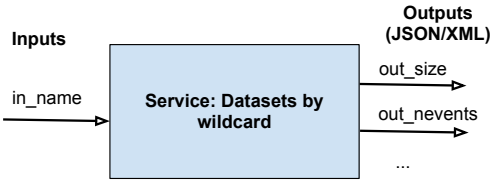


Figure 4.1.1.: a data-service (simplified)

Given an EII system, capable of answering structured queries, we are interested in translating the keyword query into the corresponding structured query. A keyword query, KWQ is an ordered tuple of n keywords $(kw_1, kw_2, .., kw_n)$. Answering a keyword query is *interpreting* it in terms of its semantics over the *integration schema*. We are given the following metadata (*virtual integration schema*):

- *schema terms*: names of entities in *integration schema* and their attributes (names of either *inputs* to the services or their *output* fields)
- information about possible *value terms*:
  - for some fields, we have a list of possible values
  - the *service mappings* define the *constraints* on values allowed as data-service inputs (required fields, regular expressions defining the values accepted)

In this work, we consider as potential results only the *conjunctive queries* augmented with simple aggregation functions without grouping (that correspond to select-project-join in SQL, with selections composed only of conjunctions) as potential results.

aggregations not so important; not yet fully implemented

Consider these queries: "*what is the average size of RelVal datasets where number of events is more than 1000*", "*avg dataset size Zmmg number of events>1000*" and "*avg(dataset size) Zmmg 'number of events'>1000*". For all, the expected result is:

In the particular case of DASQL used at the *CMS Experiment*, we want to map a keyword query into:

- type of result entity (e.g. datasets) and projections of fields in the service outputs
- conditions that will be passed to services as their inputs, e.g. datset=*RelVal*
- post-filters on service outputs: e.g. dataset.nevents > 1000
- basic aggregation functions, applied on service results: e.g. avg(dataset.size)

4.2. Overview of our approach

In the following section, we present a fairly simple heuristics-based implementation, that returns ranked list of best matching queries, where we focused on the quality of results with the goal of not enforcing any assumptions on the input queries (a full-sentence or just keywords; while existence of predefined structural patterns could be used to improve the result quality). The implementation is designed with the goal to be able to employ the user feedback for future improvements to various components of the system, such as initial entry points, or ranking of the results.

solution, **approach, implementation**

Keymantic had assumptions!

The algorithm

Taking inspiration from Keymantic [5], a keyword query is processed as follows (see Fig. 4.2.1).

Firstly, the query is pre-processed by the *tokenizer*: it is cleans up the query, identifies any explicit phrase tokens, or basic operators (see section 4.3.1).

Secondly, employing a number of metadata-based entity matching techniques, the "*entry points*" are identified: for each keyword, we obtain a listing of schema[1] and value[2] terms it may correspond to along with a rough estimate of our confidence (see Section 4.3.2). This includes identifying keyword chunks corresponding to multi-word terms (currently, only fields in service results); many of them are unclean, machine-readable field-names, with irrelevant and frequent terms, motivating the use of IDF-based information retrieval techniques.

Lastly, the *entry points* are combined, evaluating the different permutations of them (called *configurations*) by means of combining the scores of individual keywords and using heuristic rules to boost the scores of *configurations* that "respect" the likely dependencies between the nearby keywords. During the same step, the configurations that are compatible with our data integration system are identified and interpreted as structured queries, where we disambiguate the keyword matchings between the result types, the projections the selections (filters on service inputs or outputs), or simple operators. The ranking is presented in the section 4.3.3.

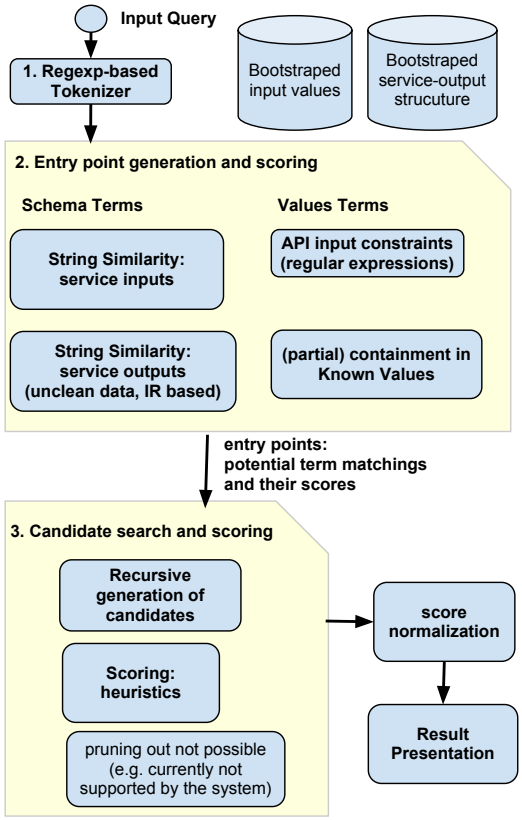


Figure 4.2.1.: Query processing flow

Example. Consider this query: `avg dataset sizes RelVal "number of events > 1000"`

Tokenizer would return these tokens: `avg; datasets; sizes; RelVal; "number of events>1000"`

Each tokens result in some entry points:

[min, avg, unique etc] **not fully implemented, lower priority**

make a nice table!

```
Zmmg -> 0.7, value: dataset.name=*RelVal*
```

[1] schema term is the name of an entity or it's attribute in the integration schema (e.g. the "result type", input parameter or output field for some data service)

[2] value term - is the name of schema term which could gain value of given keyword

```
datasets -> 0.9, schema: dataset

Chunks:
number of events>1000 ->
    1.0, filter: dataset.nevents>1000
    0.8, filter: file.nevents>1000
"datasets; sizes" -> projection: dataset.size
"avg; datasets; sizes" -> aggregation:avg(dataset.size)
```

Below each the processing steps is described in more details.

Project constraints and priorities

subsection: constraints, limitations, considerations?

Knowing 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[3]); 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.

4.3. Details of the keyword search system

detailed algorithm?

4.3.1. Step 1: Tokenizer

The tokenizer do not try to parse the natural language, however attempts to cover as many of unambiguous cases as possible.

With the goal to simplify subsequent processing, first the keyword query is cleaned-up, standardizing its notation (e.g. removing extra spaces, normalizing date formats from YYYY-MM-DD into YYYYMMDD accepted by EII system, also recognizing some expressions in natural language, such as simple operators [X equals Y, X more than Y, etc]). This is accomplished using a number of regular expression replacement patterns.

but NL support is limited, as there are many ambiguous ways of expressing the same: e.g. more than 123 events

Then, the keyword query is tokenized into tokens of:

- strings of "terms operator value" (e.g. `nevent > 1, “number of events”=100, “number of events>=100”`), if any
- phrases with compound query terms in brackets (e.g. `"number of events"`), if any
- individual query terms, otherwise

The second task is accomplished by splitting the input string on a regular-expression matching pattern which match the three cases above (n proper order), but exclude white-spaces outside of the brackets.

[3] 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 section A.2); second, these improvements would take a considerable effort to be implemented, pushing this far beyond the scope of this project

4.3.2. Step 2: Identifying entry points

The second step of query processing is identifying the starting points through applying the techniques below. To lower false positives, only the matches that score above some predefined cut-off threshold are included.

4.3.2.1. Matching the schema terms

Custom string similarity function Our experience is that basic string-edit distance metrics, such as the standard Levenshtein edit-distance (where inserts, edits, and mutations are equal) or *Jaro-Winker's* being designed for general matching tasks (e.g. matching people names, correcting typing errors, etc.), do not perform well in the task of matching keywords into specific entity names, either introducing too many false-positives (e.g. 'file' for 'site'), or not recognizing lexically farther word combinations that still make sense, such as *config* vs *configuration*.

there was a paper on this

Therefore, to minimize the false positives (which have direct effect on ranking), we propose a simple combination of more trustful metrics in the order of decreasing score: full match, lemma match (indicating also the meaning match), stem-match (meaning is further), and finally a stem match within a very small edit distance (see eq.4.3.1). Below *dist* is some string distance metric with tight limitations with score $\in [0, 1]$ (e.g. max 1-3 characters differing with beginning or end preferred, max 1 mutation/transposion).

TODO: lemmatization depend on POS tags (saw[v]->see, [n]->saw)

$$similarity(A, B) = \begin{cases} 1, & if\ A = B \\ 0.9, & if\ lemma(A) = lemma(B) \\ 0.7, & if\ stem(A) = stem(B) \\ 0.6 \cdot dist(stem(A), stem(B)), & otherwise \end{cases} \tag{4.3.1}$$

This improves the matching by incorporating basic linguistic knowledge, and without requiring any domain-specific lexical resources[4]. Further, this is easy to implement using existing libraries (such as *PorterStemmer* and *WordNetLemmatizer* in the *nltk*[5]).

Matching multi-word schema terms This component identifies keyword chunks corresponding to *multi-word schema-terms* representing field-names or titles of service results fields. The fields in service results are quite specific: some may have only the machine-readable field-names and no human readable description, and may contain some frequent and not so informative terms. All this motivates the use of IDF-based information retrieval techniques. In addition, the field names in service outputs (which are processed artefacts of JSON or XML responses) have some parent-child structure which may provide useful contextual information (e.g. block.replica.creation_time vs block.creation_time).

TODO: mention paper from XML retrieval

TODO: values within small edit distance a paper has proposed: dynamic thresholds for Levenstein distance

no fuzzy matching except stemming is currently used; one more heuristic - if terms on the ends are not used by search engine, they are useless – eliminate such results

any cases when this is misleading? - context itself is NOT sufficient to identify a

[4] machine-learning based string similarity functions have shown improvements in the accuracy[18], however they require domain-specific training data, that is often not available or costly to obtain, especially in the beginning of a project when no post logs can be used

[5] an open-source natural language processing toolkit for Python http://nltk.org/

For simplicity, we used an existing IR library, Whoosh[6], where we store "documents" each representing "a field in service outputs" (that is a field of an entity in integration schema). Each such *document* consists of multiple *fields*, with different weights of importance assigned to each:

- fully-qualified machine readable field-name (e.g. block.replica.creation_time)
  - a field containing a tokenized+stemmed version of machine readable field-name (e.g. creation_time)
  - context - a field containing a tokenized+stemmed version of machine readable field-name's parent (block.replica)
- human readable field title, if any (e.g. "Creation time of block's replica", but often this do not include the context: "Creation time", or it does not exist)

To find the matches, we query the IR library, both for phrase and single term matches of up to the k-nearby keywords (we use maximum of 4, which both provides sufficient context, and is short enough to be computationally inexpensive), phrase matches given larger weight.

The ranking is done using **BM25F** scoring function. After filtering out the worst results these will become the entry points for mapping keywords into the fields of data service outputs. Currently we directly use the score returned by the IR librarymanually normalized between [0..1]. The scoring could be improved, but in our case it works already not so bad.

normalization: threshold based on weights signifying a good score + (maybe smoothing function)

Finally, the same functionality could be also achieved through retrieving a list of matching fields for each keyword separately and then combining them through the scoring function, however the earlier approach allows pruning out the worst scoring token pairs and supporting the phrase search more easily. In either case, the problem of how to incorporate the IDFs and context information remains.

TECHNICAL NOTES FOR MYSELF: CAVEATS AND TODOs TO CONSIDER:

- full field match [all tokens] is much better even than phrase match: "number of events" vs "events; number; block" [idf shall penalize this, but this seems not enough] - (how about modified document length in BM25F?)
  - how about to force BM25F to count only distinct tokens once? e.g. tf(d, t)c = max(w_c), regardless of field c
- multiple matches of same term result in unfair scores
  - e.g. number -> run **number,** where number is repeated run.run_number, tokenized: 'run; number', title: 'run number'
  - idf shall be stronger than physical term count
  - as mentioned in a paper on "Semantic Search with BM25F" which criticized Lucene ranking, naive inclusion of multiple fields could lead to false high-matches. *~approx:~Matching terms on different fields would be scored higher than matches of the same keywords in the same field, which is semantically better defined~~.*

[6]Even if Apache *Lucene* is assumed as the most mature of the open-source libraries, it requires Java and has large footprint. Even if that may impact the results slightly, we use *whoosh,* a python library which has no dependencies.

- Far future: using context information without enforcing that to belong only to that field (we already use result type = result entity type)

Semantic similarity (not used) While semantic matching based on freely available open-domain ontologies, such as the *WordNet*, could work well for open-domain, it do not work well in a very specific domain, out-of-the box. As the EII system at CMS currently has no specific linguistic ontology, no semantic similarity is used - if enabled, it just worsens the results by introducing false positives.

separate not used section?

4.3.2.2. Matching the value terms

Regular expressions For the most of service interfaces there exist regular expressions that constraint the *input values* accepted by services. A regular expression (regexp) match do not guarantee that a certain value exists, but also it could result in incorrect keyword interpretations, as as a regexp could be loosely defined. Thus, in the general case, the regexp matches are scored lower than matches of other matching methods. Still, some of the regular expressions are sufficiently restrictive (e.g. email), which we selectively score higher.

Known values For some schema terms, we have a list of possible values, that we obtained bootstrapping them through respective data service interfaces. For matching we have a number of cases, with the decreasing score: full match, partial match, and matches of keywords containing wildcards. If keyword's value matches a regular expression, but is not contained in the known values list and the accepted values of the given field are considered to be static (not changing often), we exclude this very likely false match that reduces the false positives.

there are some specific cases with wildcard matches

4.3.3. Step 3: Candidate-answer generation and ranking

As the last step, different combinations of the entry points (where each permutation of keyword "meanings" is called a *configuration*, defining a tagging of input keywords as schema or value terms) and ranked combining the scores of individual keywords in some way (e.g. summing of log likelihoods, averaging, described below) with addition of a couple of heuristics that boost the scores of *configurations* that "respect" the likely dependencies between the nearby keywords[7] .

The scoring functions

We experimented with two scoring functions, the first one basically averaging the scores (as it was used by Keymantic[5]), and the other of more probabilistic nature - summing the log likelihoods. There, the $score(tag_i|kw_i)$ signifies the score assigned for scoring an individual keyword kw_i as tag_i (an entry point); $h_j(tag_i|kw_i; tag_{i-1,..,1})$ denotes the score boost returned by heuristic h_j given a concrete tagging so far (in most cases all tags are not needed).

$$score\_avg(tags|KWQ) = \frac{\sum_{i=1}^{|KWQ|} \left(score(tag_i|kw_i) + \sum_{h_j \in H} h_j(tag_i|kw_i; tag_{i-1,..,1}) \right)}{N\_non\_stopword} \tag{4.3.2}$$

$$score\_prob(tags|KWQ) = \sum_{i=1}^{|KWQ|} \left(\ln\left(score(tag_i|kw_i)\right) + \sum_{h_j \in H} h_j(tag_i|kw_i; tag_{i-1,..,1}) \right) \tag{4.3.3}$$

Note that in the ***probabilistic*** approach we introduce a set of "fake" tags where keywords are not mapped to any known entity or operator (e.g. a keyword is unrecognized, or it is a stop word) and we assign some predefined score to it depending on it's category (e.g. no score for stopword).

The two methods **seemed to perform almost equally well**, with the probabilistic approach being more sensitive to variations in scoring quality of the entry points (the scores are just estimates of our confidence, not real probabilities; the results improved with improvements to accuracy of string matching functions), however **it seems** the probabilistic approach is more exact in ranking the results when the entry point scores are quite exact.

The final scores of the probabilistic approach are also more complex to interpret (we would like to present the user with color coding which identifies the our confidence, and the scores if exponented vary much more than in averaging).

sort of useful stopwords: where, who, when?

do we need better test data?

, still one could estimate the high and low thresholds based on initial scores

The heuristics and pruning out the unacceptable candidates

- Relationships between keywords:

[7] the nearby keywords are expected to be related[6], e.g. a configuration is promoted if the tags of nearby keywords refer to the same entity

  - promoting such combinations where nearby keywords refer to related schema terms (e.g. entity name and it's value)
  - balance between taking the keyword or leaving it out (the one that we are unsure about)
  - boost important keywords (different parts of speech are of different importance, e.g. stop-words are less useful than nouns)

- Qualities of Data Integration System:
  - promote data service inputs over filters on their results: 1) it is more efficient, especially when this is possible; 2) there are much more of possible entities to filter, so more false matches are expected there, while the service inputs shall cover large part of cases
  - if some keyword can be matched as the requested entity, and mapping of other keywords fits the service constraints
  - if requested entity and a filter condition is the same (a small increase, a common use-case is retrieving an entity given it's "primary key" identifier or a wild-card)
  - for being able to execute the query, the service constraints must be satisfied; still it could useful to the interpretations that achieve high rank, even if they do not satisfy some constraint (e.g. a mandatory filter is missing) informing the user

Keymantic assumed all keyword have interpretation!

only stop-words are distinguished now; '(tell|show|display|fi me?' filtered out by tokenizer

shall we allow both as filter name and result type? no?

not yet implemented

4.3.3.1. Tuning the scoring parameters

TODO

- weights for regexps, etc
- not taking a keyword
- multi-word matching

4.3.4. Miscellanea

Automatically identifying the qualities of data services

The integration schema mappings that are used in the EII system are minimal - they only describe services, their input parameters, and mappings between inconsistently named output fields. Any other information, such as the complete listing of fields in the service outputs, or their types are identified by processing results of historical queries. To get satisfactory coverage immediately, a list of bootstrapping queries is used to initialize the most important field listings (of the services that retrieve entities by their "primary key").

Note: it was a sub-task of this project to make this work based on existing broken prototype.

Natural Language Processing and full-sentence search

describes what was NOT used. future work?

We first looked into parsing as this is a prerequisite for most other natural language processing methods such as Keyword extraction or Relation Extraction. However it didn't look worth the investment given our time constrains and our specific domain.

None of the existing out-of-the-box parsers we looked at (TODO:list), didn't show good results for our specific domain, especially then natural language is mixed with technical terms, numbers, and control statements. Still, Enju[8], a wide-coverage probabilistic HPSG[9] rule-based parser, seemed the most robust for our specific domain, even without any additional training, giving much better results than the standard packages available in NLTK. As the project scope was limited, we didn't want additional dependences on third-party code, and natural language is still complex to interpret well, this was excluded.

Performance

describes what was NOT used. future work?

A number of methods are available for improving the performance (e.g. Munkres/Hungarian bipartite matching algorithm, or dynamic programming with assumption of maximum length of dependences) [most probably] in exchange for additional assumptions.

Actually, at the CMS collaboration, implementing additional optimizations is not of highest priority: queries with the length of up to **n=8** keywords runs faster than in a second, while for most of the queries the EII system requires tens of seconds to minutes to retrieve the actual query results from the data services .

[8] `http://www.nactem.ac.uk/enju/demo.html`
[9] Head-driven phrase structure grammar

4.4. Presenting the results to the user

The results presented to the user for the query *Zmmg event number > 10*, are given in fig. 4.4.1. First, it can be seen that the query is ambiguous, as the first three suggestions are equally feasible correct results - the three entities (file, block, dataset) contain the same field ("nevents"). However, if user knows the entity he is searching for, he may immediately filter only these results.

Hovering on the structured query, user sees it's explanation (an interactive way to learn the semantics of the query language). Different elements of the query are presented visually in different colors (green is used to indicate *conditions applied as service inputs*; red is for *post_filters applied only on service outputs*, which are more expensive in terms of performance).

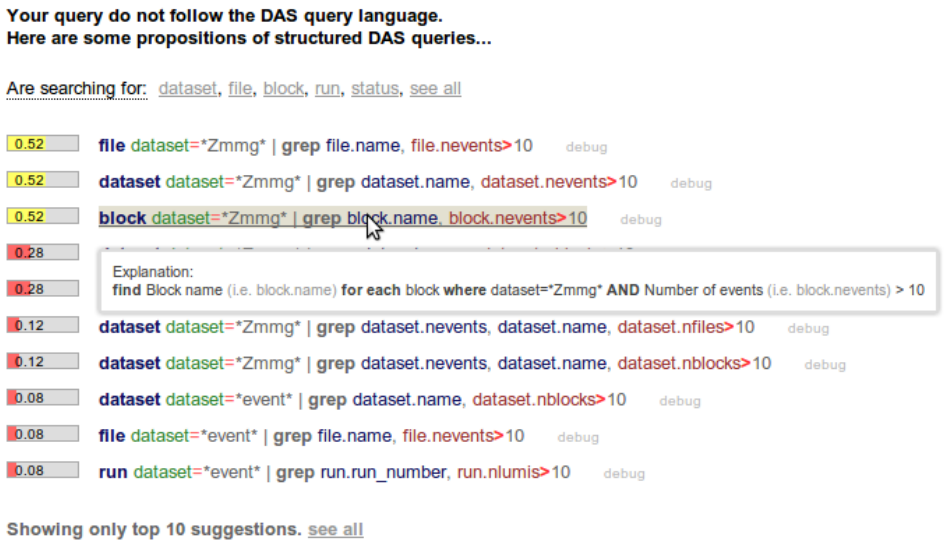


Figure 4.4.1.: Presentation of keyword search results

4.4.1. Entry points

TODO: showing entry points to the user, so used would understand why he sees results like; and could give live-feedback?

4.5. Incorporating User Feedback

4.5.1. Live feedback through auto-completion (prototype)

here or separate chapter?

While structured queries are hard to compose, keyword and natural language queries are complex for a machine to interpret because of their ambiguity or inherent complexity. Form-based interfaces has been around since many years, however with many structural items being candidates for the input, they are not very practical, while static predefined forms are limiting the user's expressiveness even more.

We are argue that a simple user-interface could combine the advantages of both: properly-implemented variation of forms (which for instance could be implemented as an input widget accepting multiple tokens and providing suggestions and

disambiguations in real time, see fig. 4.5.1). This, being a structured input, could reduce the ambiguity of the queries ; while the availability of keyword search leaves freedom for expressiveness.

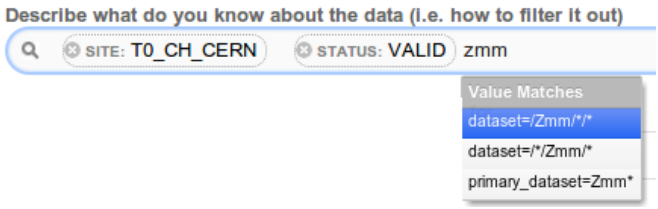


Figure 4.5.1.: prototype of auto-completion based interface

Further, such user interface can be closely integrated with keyword search - some of the suggestions would include multiple interpretations of the last term being typed, allow to get immediate user's feedback and resulting in better-defined input to the system,. This, in turn, would result in more exact query suggestions. Second, this could be potentially useful for evaluating and improving the quality of the entity matching and keyword search. We will elaborate on this is the subsequent sections.

4.5.2. Customization and user-preferences

shall be per user or per user-group! SODA didn't use any clustering?

4.5.3. Using the feedback for self-improvement

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, this could serve as training data for machine learning-based algorithms, however

There are many ways to employ the users feedback, from .

- live feedback
  - more: if unsure, ask user to specify the interpretation of his query (like SODA but with options) or even while waiting for results – calculating entry points is cheap. evaluating all interpretations is more expensive even with performance optimizations (keymantic was up to 6s)
- influence keyword to schema term matching
  - similarity metrics and their weights
  - allow users to add new: [this is the explicit feedback, that is more valuable than implicit]
    - values for schema entities
    - synonyms for schema terms
- weights on particular entities or notes in schema graph: SODA [12]

- promote/demote query suggestions - machine learning

5. Evaluation

5.1. Accuracy

success @ K-th result
1
3
5
10
20
-
query types:
value: known, regexp
entity + known val
entity + unknown value
multiple entities values
result attribute filter
(meaningless – false match)
aggregator / sorting??

String matching

5.2. Users feedback

Usability
Usefulness of KWS vs structured query language

6. Related Work

TODO: introduce; mention the closest ones: Keymantic/KEYRY; compare with our work (?); works on user feedback? where?!

In addition to the few most closely related works namely Keymantic[6, 5] and KEYRY [3] presented earlier, significant amounts of research exists in more distant fields presented below.

Enterprise Information Integration

During the last 15 years, significant experience has been accumulated in the field of *Enterprise Information Integration*[1] (EII) including: data integration formalisms, ways of describing heterogeneous data sources and their abilities (e.g. database vs web form), query optimization (combining sources efficiently, source overlap, data quality, etc) [11]. Recent research in Enterprise Information Integration mostly focused on approaches minimizing human efforts on source integration, e.g. on probabilistic self-improving EII systems [1, ch.19].

String and Entity Matching provides the possible interpretations of individual keywords, as entry points for further processing. From the fields of information retrieval, entity and string matching, vast amounts of works exist, including various methods for calculating string, word and phrase similarities: string-edit distances, learned string distances [18], and frameworks for semantic similarity.

Natural Language Processing (NLP) could be useful in gaining the better understanding of the meaning of a question or a query. Large amount of works exist on Question Answering and NLP including: question focus extraction identifying the requested entit(-ies), parsing into predicate argument structure providing more generic representation of a sentence (e.g. removing differences between passive and active voices) simplifies further analysis, the relation extraction allows grasping more exact relationships between constituents of a clause[2], word sense disambiguation and other semantic techniques allow choosing more semantically correct interpretations.

It is worth mentioning, that the current state-of-the art methods such as the *IBM Watson* [9], a complex open-domain question answering system, do not even try gaining the complete understanding of the question (which is still a very challenging task), but focuses instead on scoring and analysing the alternative interpretations of questions and result candidates.

NL interfaces to Data Services: [10] attempts to process multi-domain full-sentence natural language queries over web-services. It uses focus extraction to find the focus entity, splits the query into constituents (sub-questions), classifies the domain of each constituent, and then tries to combine and resolve these constituents

[1] Enterprise Information Integration (EII) is about 'integrating data from multiple sources *without* having to to first load data into a central warehouse'[11, p.1]

[2] especially good for a small predefined set of important relations, but requires lots of manual work; less common relations can be covered through machine-learning based relation extraction, but that requires large corpus[21]

over the data service interfaces (tries recognizing the intent modifiers [e.g. adjectives] as parameters to services). (too ambitious/not-mature; open domain, real natural language questions - a bit farther from our focus, our domain is very specific.).

Searching structured DBs The problem of keyword search over relational and other structured databases received a significant attention within the last decade. It was explored from a number of perspectives: returning top-k ranked data-tuples *[17]* vs suggesting structured queries as SQL [7], performance optimization, user feedback mechanisms, keyword searching over distributed sources, up to lightweight exploratory[3] probabilistic data integration based on users-feedback that minimize the upfront human effort required [1, ch.16]. On the other extreme, the *SODA [7]* system has proved that if enough meta-data is in place, even quite complex queries given in business terms could be answered over a large and complex warehouse.

what in addition to schema mappings for item-based ranking?

[3]because of probabilistic nature of schema mappings, it do not provide 100% result exactness

7. Conclusions and Future work

Keyword search over data services is still lacking attention from research community.

In addition to structured query languages, this could help in learning and getting results more quickly.

We presented ...

Future work...

—-

Future work (technical notes):

- Client-side implementation of keyword search - large parts of keyword search could be moved to client-side, saving the server resources. A couple of issues exist: making sure that the load is not to high and synchronizing the logs. The first one could be solved by using so called worker threads. http://www.w3.org/TR/workers/ http://www.sitepoint.com/javascript-execution-browser-limits/

- web sockets for auto-completion
- better interpretation of patterns in keyword queries
  - NL - focus
  - semi-structured? - (weak|shallow|permissive) parsing of DAS QL

Bibliography

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

[2] 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. Available from: `http://stacks.iop.org/1742-6596/331/i=4/a=042029`.

[3] 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.

[4] Sonia Bergamaschi, Elton Domnori, Francesco Guerra, Raquel Trillo Lado, and Yannis 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.

[5] Sonia Bergamaschi, Elton Domnori, Francesco Guerra, Mirko Orsini, Raquel Trillo Lado, and Yannis Velegrakis. Keymantic: semantic keyword-based searching in data integration systems. *Proc. VLDB Endow.*, 3(1-2):1637–1640, September 2010. Available from: `http://dl.acm.org/citation.cfm?id=1920841.1921059`.

[6] Sonia Bergamaschi, Elton Domnori, Francesco Guerra, Raquel Trillo Lado, and Yannis 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. Available from: `http://dl.acm.org/citation.cfm?id=1989383`.

[7] Lukas Blunschi, Claudio Jossen, Donald Kossmann, Magdalini Mori, and Kurt Stockinger. Soda: generating sql for business users. *Proc. VLDB Endow.*, 5(10):932–943, June 2012. Available from: `http://dl.acm.org/citation.cfm?id=2336664.2336667`.

[8] P Lane et al. Oracle database data warehousing guide, 11g release 2 (11.2). chapter 9: Basic materialized views, September 2011. Available from: `http://docs.oracle.com/cd/E11882_01/server.112/e25554/basicmv.htm#i1007299`.

[9] DA Ferrucci. Introduction to "this is watson". *IBM Journal of Research and Development*, 56(3.4):1–1, 2012.

[10] Vincenzo Guerrisi, Pietro La Torre, and Silvia Quarteroni. Natural language interfaces to data services. *Search Computing*, pages 82–97, 2012.

[11] Alon Y. Halevy, Naveen Ashish, Dina Bitton, Michael Carey, Denise Draper, Jeff Pollock, Arnon Rosenthal, and Vishal Sikka. Enterprise information integration:

successes, challenges and controversies. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 778–787, New York, NY, USA, 2005. ACM. Available from: http://doi.acm.org/10.1145/1066157.1066246, doi:10.1145/1066157.1066246.

[12] M. Klausmann. User feedback integration-incremental improvement. Master's thesis, Thesis Nr. 31, ETH Zürich, 2011, 2011.

[13] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Semi numerical Algorithms*. Number 9788177583359. Addison-Wesley Longman, Inc, 1998.

[14] Christoph Koch, Paolo Petta, Jean-Marie Le Goff, and Richard McCatchey. On information integration in large scientific collaborations, 2000.

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

[16] Valentin Kuznetsov, Dave Evans, and Simon Metson. The cms data aggregation system. *Procedia Computer Science*, 1(1):1535 – 1543, 2010. ICCS 2010. Available from: http://www.sciencedirect.com/science/article/pii/S1877050910001730, doi:10.1016/j.procs.2010.04.172.

[17] Yi Luo, Wei Wang, Xuemin Lin, Xiaofang Zhou, Jianmin Wang, and Kequi Li. Spark2: Top-k keyword query in relational databases. *Knowledge and Data Engineering, IEEE Transactions on*, 23(12):1763–1780, 2011.

[18] Andrew McCallum, Kedar Bellare, and Fernando Pereira. A conditional random field for discriminatively-trained finite-state string edit distance. *arXiv preprint arXiv:1207.1406*, 2012.

[19] Silvia Rota, Sonia Bergamaschi, and Francesco 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.

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

[21] Chang Wang, Aditya Kalyanpur, J Fan, Branimir K Boguraev, and DC Gondek. Relation extraction and scoring in deepqa. *IBM Journal of Research and Development*, 56(3.4):9–1, 2012.

A. Data Integration "War stories" at the CMS Experiment, CERN

or just: Problem solutions

A.1. Relaxation of the Query Language

shall we put the solutions here or somewhere lower in the paper after describing Keyword Search?

Initially the queries had to operate exactly on the fields returned by data services...
implementation of simple compound queries so that an entity could always return same list of fields, easy to implement and very useful for users

- makes the system much more clear
- that even makes keyword searching easier to implement, as we could assume that all the fields are almost "stable" for each entity being returned (except a couple of exceptional cases, which could be check afterwards)

A.2. Solutions to the Performance Issues

Analysing query logs and benchmarking the most popular queries, it was found out that most of the performance issues were due to large data amounts of data the providers are processing, including some unnecessarily work being repeated or requested without need (due to current limitations of services, which are not under our direct control).

Incremental view maintenance

In the cases when new records are coming, but the existing ones are not changing much, the incremental view maintenance that computes only differences from earlier results could be a fairly easy solution for greatly improving the performance of queries containing heavy joins and/or aggregations.

This is exactly the case with the most popular expensive query over the DBS system (80GB of data + 280GB of indexes): *'find files where run in [r1, r2, r3] and dataset=X'* that requires joining most of the biggest tables in the database (number of tuples in parentesis, arrow indicate join direction):

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

Having a materialized view with all these tables joined together would allow answering such queries much quicker. Given low change rates (in comparison to data already present), maintaining the view incrementally should be comparatively cheap with the only expense of just couple of times of storage space (storage is bound by the size of the largest table anyway).

In Oracle, which is the standard back-end, the *materialized refresh fast views with query rewriting* provide a completely transparent operation not requiring any changes to the proprietary system. Still, it has a couple of limitations on the queries and the ways of refreshing the view[8]. Alternatively, some another continuous view maintenance

tool (e.g. DBToaster[1]) could be used, however this is not be as transparent as the earlier solution.

Pagination and Sorting of results

As many of the queries on the web interface are exploratory and request only the first page of results, supporting pagination is one of the major factors towards performance improvements . As the DAS system is combining records from multiple systems, pagination also requires retrieving results from the data providers in an ordering that is common among the services (in many cases that can be the "Primary key" of the entity that is being requested; however, some cases are more complex from the side of data provider: an ordering not supported by database indexes could induce full table scan!).

Estimating query running time

not yet implemented

Tracking of the execution time of each data-service, was proposed to be implemented, that would inform user of long lasting queries, starting them only with his confirmation

It has been chosen to track the mean of execution time, and its standard deviation. Knuth has shown that the standard deviation can be efficiently computed in an online fashion without need to store each individual value, nor recomputing everything from scratch [13, p. 232].

Because the input parameters passed to the service may heavily impact the service performance, we differentiate between these parameter types: 1) some specific value, 2) a value with wild-card (presumably returning more results than specific value as it may match multiple values), 3) not provided (matches all values). So we store only four values per each different combination of data-service input parameter's .

[1] http://www.dbtoaster.org that is being developed at EPFL