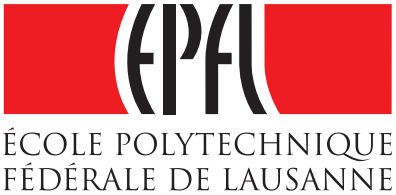
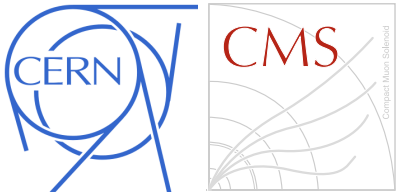



Distributed Information Systems Laboratory
School of Computer and Communications
Swiss Federal Institute of Technology, Lausanne

Computing department
Compact Muon Solenoid Experiment
European Organization for Nuclear Research

**Master Thesis**

# Keyword Search over Data Service Integration for Accurate Results

Vidmantas Zemleris
Section of Computer Science (IC)

15th 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

We consider the case where there is no direct access to the data and only interfaces of limited capabilities are available (e.g. web services, web forms, proprietary systems), while users are in need of a way to query these sources for accurate results (boolean retrieval augmented with additional operators). This renders the traditional methods that return data-tuples inapplicable (e.g. information retrieval or the keyword search over relational databases).

In that instance, the Enterprise Information Integration (EII), provides the means to query the sources in a coherent way. The querying is usually done through structured query languages such as SQL, implying that the user must learn the query language and has to know how the data is structured.

The alternatives to structured queries include: i) forms, that are impractical for complex queries, ii) the keyword search returning query results directly, risking of choosing the wrong interpretation; or iii) the keyword search based on metadata and proposing a list of structured queries for user to choose from.

In this work we explore the third option and present a keyword search system that approaches this problem operating on the available information: the metadata such as the constraints on allowed values, and some portions of the data. It proposes a ranked list of structured queries along with explanations of their meaning. Unlike the earlier works, it is freely available, makes no assumptions on the input query (still being able to leverage the structural patterns in the query, if present).

The system is discussed within context of an EII service at the Compact Muon Solenoid Experiment (CMS), at CERN, where the 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. It has been integrated with the CMS production Data Management system and exposed to the CMS Collaboration.

# Contents

# List of Symbols and Abbreviations

| | |
|---|---|
| CMS | The Compact Moun Selenoid Experiment at the European Organization for Nuclear Research (CERN) |
| DAS | CMS Data Aggregation System - The EII system used at CMS |
| EII | Enterprise Information Integration |
| HMM | HIdden Markov Model |
| IDF | Inverse Document Frequency, used to downrank frequent and therefore likely irrelevant terms |
| IDF | Inverse Document Frequency, used to downrank frequent and therefore likely irrelevant terms |
| KWQ | keyword query |
| NLP | Natural Language Processing |
| schema | by schema we refer to the integration schema (virtual schema based of entities exposed by the services) |
| schema terms | names of entities in integration schema and their attributes (names of either inputs to the services or their output fields) |
| value terms | values of entity attributes (that could be input parameters of data services, or be contained in their results) |

## 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

*Enterprise Information Integration* (EII), which is also called *Virtual Data Integration*, is about "integrating data from multiple sources *without* having to first load data into a central warehouse" [11, page 1]. It allows querying the sources in a coherent way (eliminating the inconsistencies in data formats, naming; combining the results, etc.) and is the most beneficial when 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 *SQL*. However, the latter present a number of user interface issues.

Virtual integration presents an additional challenge, since only limited access to the data instances is available, rendering the traditional methods that return data-tuples inapplicable (e.g. Information Retrieval or Keyword search over relational databases).

The alternatives to structured queries include: 1) interfaces based on regular forms, that are impractical for complex queries over large number of entities/attributes; 2) the keyword search based on metadata returning query results directly, risking that the query interpretation chosen is not the correct one, as the keyword queries are ambiguous; or 3) the keyword search based on metadata and proposing a list of structured queries for user to choose from.

The objective of this work is to investigate the third option, the keyword search proposing the ranked list of queries, as a more intuitive alternative, which, in fact, received little attention in the field of data service integration[10].

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, also touching and the mechanisms for user feedback and some more distant topics such as usability and performance of an EII.

## 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]. Often 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. Still, even the tech-savvy users may have only a vague idea of where exactly to find what they need.

[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] due to the complexity of combining the contributions by a large number institutes in a single project, the software projects result in fair number of proprietary systems[13]

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

## 1.2. State of the Art in the field

In the field of EII, significant experience has been accumulated on its formalisms, source descriptions, query optimization [11], with the recent research focusing on minimizing human efforts on source integration [1, ch.19]. Meanwhile, to the best of our knowledge, the *Boolean Keyword-based search over data service integration*, which is our main focus, received little attention, with only a few attempts [6, 3, 10] to address the problem.

**Nature of keyword queries** Keyword queries are often underspecified, therefore every possible interpretation has to be included in the results [4]. Still, 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 [7, 5, 3].

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 [14, 4]. Most of the existing approaches attempt to profit from these dependencies to ameliorate their candidate answers ranking.

**Keyword querying over EII** Two approaches that are presented below were identified as the closest to our problem.

*Keymantic* [6, 5] answers keyword queries over relational databases with limited access to the data instances or over data integration [5]. First, based on meta-data[4] , individual 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). Then, these scores are combined**,** and refined by heuristics that increase the scores of query interpretations with the nearby keywords having related labels assigned. Finally, these labels are interpreted as SQL queries.

*KEYRY* [3] attempts to incorporate users feedback through training an Hidden Markov Model's (HMM) tagger taking keywords as its input. It uses the List-Viterbi [19] algorithm to produce the top-k most probable tagging sequences (where tags represent the "meaning" of each keyword). This is interpreted as 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 supervised training, while even the keyword queries itself can

[3] e.g. on the popular search engines, information on currency rates, weather and time at given location, etc. is already available through recognition of certain patterns in web queries

[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

serve for unsupervised training [18]. According to [3] the accuracy of the two systems didn't differ much.

## 1.3. Structure of the work and our contributions

Chapter 2 presents the context in which this thesis was conducted - the EII system - and introduces some real-world issues, such as data-service performance and system's usability (the solutions are presented in appendix B).

In chapter 3 the problem of keyword search over EII is formally defined, and 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 novel way.

Next, in section 3.5 we discuss approaches towards incorporating user feedback into the 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).

In chapter 4 the developed system is evaluated quantitatively using test queries and qualitatively through user feedback. Chapter 5 presents the more distant related works, while, in chapter 6 the future work is discussed.

**Deliverables and related tasks** The project also included the following tasks: 1) choosing a precise topic to focus on - because the area is not so actively researched and there is no concise terminology[5], this 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.

For the complete list of project deliverables/artefacts see appendix A.

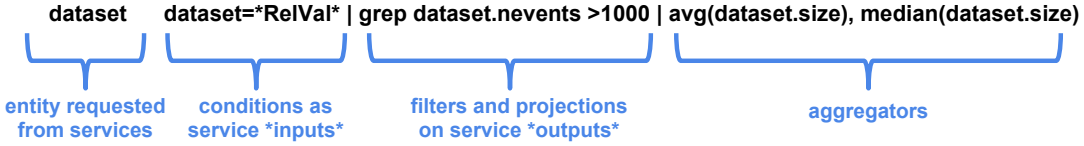
[5] e.g. virtual data integration, enterprise information integration, data virtualization are used as synonyms throughout different time periods; works on keyword search over EII mostly focused on relational databases (with limited access to data instances)

# 2. Project context: EII at CMS

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) [15, 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.

**Query language (DASQL) and its 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, the DASQL 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.

**The Data-Sources** The system integrates approx. 10 data providers (~100 interfaces returning JSON, XML, or other formats), most of them uses Oracle as back-end. Because most of the providers were created initially focusing on data storage without much attention to its retrieval, services are not optimized for querying the fairly large data volumes they store[2]: the data is kept in fully-relational fashion, often requiring complex joins over large tables; the services do not allow result pagination, nor sorting.

**User Feedback, Issues, Solutions** DAS experienced a number of performance and usability issues (users reported either being totally happy with DASQL, or being totally lost). In addition to keyword search, a slight extension to the query language was proposed. A confusing element of DAS was that the fields returned for certain entity were depending on the selection conditions, it was proposed to automatically resolve the additional service orchestration for returning the requested fields (see appendix B.2). Second, analysis of performance bottlenecks indicated that data providers were not ready for data retrieval - appropriate measures were proposed (see appendix B.1).

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

[2] the total size of data growing at rate of 1TB/year. E.g. one of the biggest providers, DBS, is partitioned among ~10 instances in Oracle, with the biggest of 80GB data + 280GB indexes

# 3. Keyword Search over Data Services

## 3.1. Problem statement

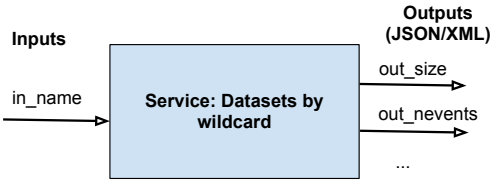



Figure 3.1.1.: a data-service (simplified)

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[1]] in SQL, with selections composed only of conjunctions) as potential results.

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 CMS, the input has to be mapped 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)

[1] the "joins" are currently limited, and can not be explicitly specified by the user

## 3.2. Overview of our approach

In the following sections, we present a fairly simple heuristics-based implementation, that produces a ranked list of best matching structured queries, where we focused on the quality of results with the goal of not enforcing any assumptions on the input queries (it could be plain keywords or a full-sentence, 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.

### The algorithm

Taking inspiration from Keymantic [5], a keyword query is processed as follows (see Fig. 3.2.1).

Firstly, the query is pre-processed by the *tokenizer*: it cleans up the query, identifies any explicit phrase tokens, or basic operators (see section 3.3.1).

Secondly, employing a number of metadata-based entity matching techniques, the "*entry points*" are identified: for each keyword, we obtain a listing of schema[2] and value[3] terms it may correspond to along with a rough estimate of our confidence (see Section 3.3.2). 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 3.3.3.

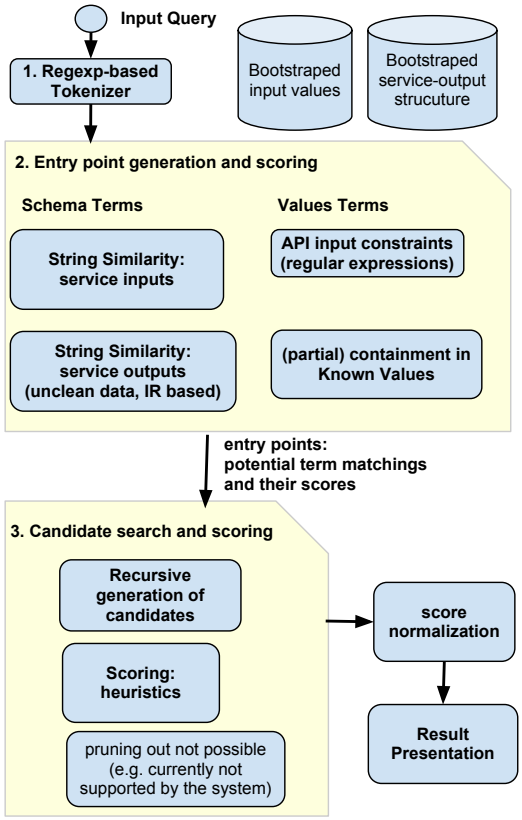



Figure 3.2.1.: Query processing flow

[2]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)

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

**Example.** Consider this query: `dataset sizes RelVal 'number of events > 1000'`
Tokenizer would return these tokens: `dataset; sizes; RelVal; 'number of events>1000'`
Second, Each token result in some entry points:

```
RelVal -> 1.0, value: group=RelVal
RelVal -> 0.7, value: dataset=*RelVal*
datasets -> 0.9, schema: dataset

Chunks:
  'number of events>1000' ->
    0.93, filter: dataset.nevents>1000
    0.93, filter: file.nevents>1000
  'datasets; sizes' -> 0.99, projection: dataset.size
  'sizes'           -> 0.41, projection: dataset.size

... and some more with lower scores...
```

As one can see from the entry points, the 'RelVal' keyword is ambiguous, there is a 'RelVal' group, and also 'RelVal' could be part of dataset name. While, 'number of events' could be attribute of dataset, file or block. Also well there is some ambiguity in chunking, 'dataset sizes' may map to *dataset.size* attribute, while dataset is also the entity type.

After combining the entry point scores, some of the top-four results are:


The first result we see because the score of group=RelVal is higher than the other option for the same keyword.

## 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[4]); 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] 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 B.1); second, these improvements would take a considerable effort to be implemented, pushing this far beyond the scope of this project

## 3.3. Details of the keyword search system

Below each the processing steps is described in more details.

### 3.3.1. 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.

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.

### 3.3.2. 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.

#### 3.3.2.1. 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*.

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.3.3.1). 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).

$$similarity(A,B) = \begin{cases} 1, & if\ A = B \\ 0.9, & if\ lemma(A) = lemma(B) \\ 0.7, & if\ stem(A) = stem(B) \\ 0.6 \cdot dist(stem(A), stem(B)), & otherwise \end{cases} \tag{3.3.1}$$

This improves the matching by incorporating basic linguistic knowledge, and without requiring any domain-specific lexical resources[5]. Further, this is easy to implement using existing libraries (such as *PorterStemmer* and *WordNetLemmatizer* in the *nltk*[6]).

**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).

For simplicity, we used an existing IR library, Whoosh[7], 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

[5] machine-learning based string similarity functions have shown improvements in the accuracy[17], 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

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

[7] 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.

normalized between [0..1]. The scoring could be improved tuning the scoring function, but in our case it works already not so bad.

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.

**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.

#### 3.3.2.2. 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.

### 3.3.3. 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[8] .

#### 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).

$$score\_avg(tags|KWQ) = \frac{\sum_{i=1}^{|KWQ|} \left( score(tag_i|kw_i) + \sum_{h_j \in H} h_j(tag_i|kw_i; tag_{i-1,..,1}) \right)}{N\_non\_stopword} \tag{3.3.2}$$

$$score\_prob(tags|KWQ) = \sum_{i=1}^{|KWQ|} \left( \ln\left(score(tag_i|kw_i)\right) + \sum_{h_j \in H} h_j(tag_i|kw_i; tag_{i-1,..,1}) \right) \tag{3.3.3}$$

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).

#### The heuristics and pruning out the unacceptable candidates

- Relationships between keywords:

[8] 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

### 3.3.4. Tuning the scoring parameters

First the individual system components has been tuned to a "sufficient" level using a number of unit tests, and manual testing. In some cases it required assigning arbitrary estimates likelihood/confidence.

Then using a number of keyword queries either written by the end users of the system, or the developer to address **marginal/specific** use cases, a number of global system parameters had been fine-tuned by hand:

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

### 3.3.5. 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").

### Natural Language Processing and full-sentence search

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[9], didn't show good results for our specific domain, especially then natural language is mixed with technical terms, numbers, and control statements. Still, Enju[10], a wide-coverage probabilistic HPSG[11] 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 the *nltk* language processing toolkit. 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

At the CMS collaboration, implementing additional optimizations is not of a highest priority, as with the current implementation, the queries runs faster than in a second (the time needed to run the queries in average is 0.2s, with the maximum of 1s[12]), 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.

Still a number of methods are available for improving the performance (e.g. bipartite matching problem approached with the top-k results version of Hungarian-Munkres algorithm[5], or dynamic programming with assumption of short maximum length of keyword dependences) in exchange for additional complexity of the implementation or the additional assumptions.

[9] including Stanford parser and others available in nltk

[10] `http://www.nactem.ac.uk/enju/demo.html`

[11] Head-driven phrase structure grammar

[12] measured on an 4 years old laptop with 1.7Ghz processor and 4GB of RAM; server hardware is much more powerful;

## 3.4. Presenting the results to the user

The results presented to the user for the query *Zmmg event number > 10*, are given in fig. 3.4.1. 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 the user knows the entity he is searching for, he may immediately filter only these results.

Hovering on the structured query, the user see its 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).

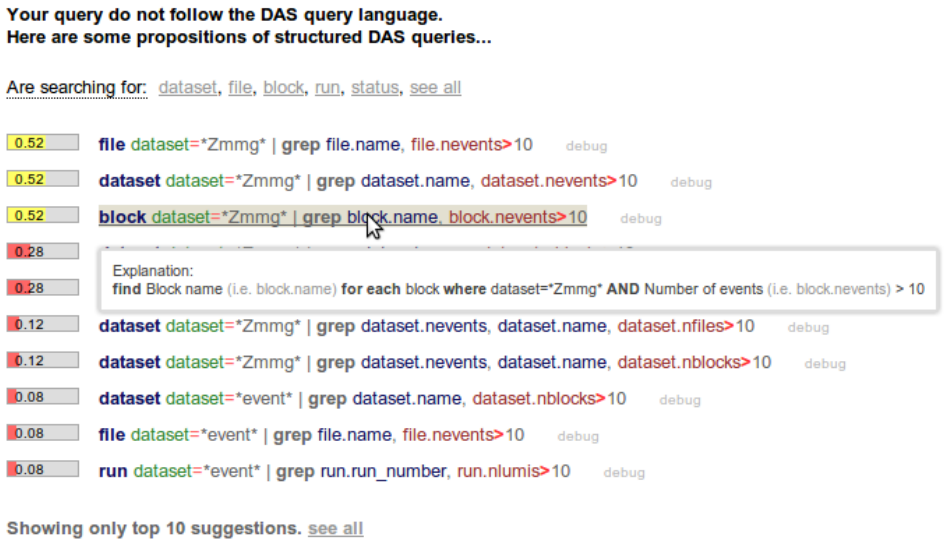



Figure 3.4.1.: Presentation of keyword search results

## 3.5. Incorporating User Feedback

### 3.5.1. Live feedback through auto-completion (prototype)

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. 3.5.1). This, being a structured input, could reduce the ambiguity of the queries ; while the availability of keyword search leaves freedom for expressiveness.

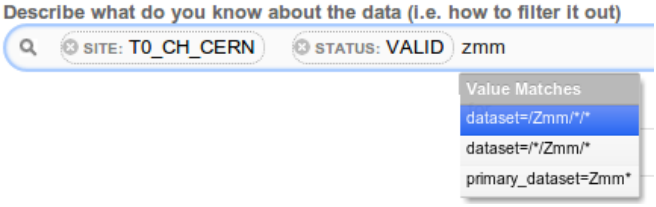



Figure 3.5.1.: prototype of auto-completion based interface

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.

### 3.5.2. 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, 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.*

This may make the semi-explicit user feedback of lower quality, and we propose that this may need additional investigation:

- that is the correspondences between the keywords and their "meanings" as schema or value terms, which still has multiple ways to be converted into structured query.
- the quality of the feedback could also depend on the false positives of the earlier mappings (while working on the system, we have seen that it is possible for a false matching from a keyword to an entity to result in a correct result!), and that may potentially impact the machine learning.

# 4. Evaluation

## 4.1. Accuracy

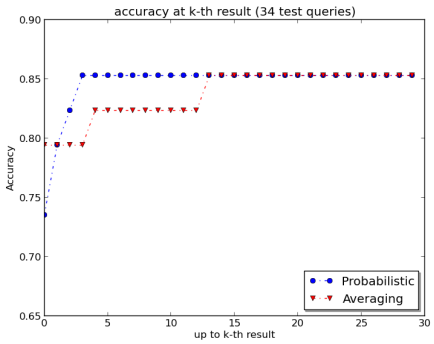



Figure 4.1.1.: Accuracy comparison of the two scoring methods at kth result

At the moment we used 34 test queries written by the end users (around half), or by the developer (to address some specific cases, such as preference of service inputs over filters on their outputs). We use the measure of accuracy at k-th result (i.e. is there expected result among the k top results) for the evaluation.

As seen in figure 4.1.1, with current test data, the difference between "averaging" and the "probabilistic" scoring functions (see section 3.3.3 for details) is not large.

The "averaging method" performing slightly better on the first result , but if we consider more than the first result, the "probabilistic" log-likelihood based function performs better, at the fourth result reaching the maximum accuracy of 0.853**.** From this, it seems that the latter method perform better (as one could have expected), but it requires better score estimates. For instance, currently the multi-term scores that are based almost directly on the BM25F function (i.e. $score/max(goodscore, maxscore)$ ) seem to be not exact enough, nor the IR ranking (which could require adjusting the field weights, the BM25F parameters or slightly modifying it, to account for not clean field titles, repeated irrelevant term, or the fact that same terms could appear in the field-name and it's title).

It is expected to gather more real-world query examples from the users during the upcoming presentations to the CMS Collaboration (they were not possible earlier due to busy schedules at CMS).

## 4.2. Users feedback

Users liked the idea of auto-completion, both because it could autocomplete the schema terms, and some of the value terms. Regarding the keyword search, there there two

groups of users: 1) the advanced users who are used to query languages and how the organization of the data, who did not express big interest in keyword search, and others, 2) others that liked it. Still, both groups liked the fact that sometimes they could find what they didn't known how to query for it, or even didn't know it may exist.

More feedback is expected in the near future, which will help the optimization of the tool and the eventual production deployment for the whole CMS Collaboration.

# 5. Related Work

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 [17], 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, page 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[20]

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 *[16]* 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.

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

# 6. Future work and Conclusions

## 6.1. Conclusions

In comparison to the other fields such as searching the relational databases, the keyword search over data service integration is still lacking attention from the research community.

The availability of public, corporate and governmental services is increasing as well as the popularity of data service repositories and tools[1] for combining them. However, the lack of user-friendly interfaces is becoming a serious issue, not only within the corporate environments.

A implementation of keyword search over Enterprise Information Integration has been presented by discussing the implementation in details, some real-world issues and ways to solving them. The implemented system do not impose any constraints on the input query (unlike some of the earlier works[5]), and it is able to profit from the structure available in the query (phrases, selections through auto-completion).

The concept and a working (prototype) implementation was exposed to the large CMS physics community, with the aim to have it in production soon. The system implemented is going to be further supported and advanced, and this will contribute to improve the overall efficiency of the physics analysis program by the CMS Collaboration.

[1] *such as the YQL* or the *"Google Fusion Tables"*

## 6.2. Future work

Due to the time constraint of this project, a number of potential improvements could not be implemented, however they may be need in the future for various system components. In particular, the accuracy of the current implementation (within top 5 results we get accuracy of over 80%) could be boosted, and the keyword search could be successfully employed as method for learning the structured query language and figuring out how the data is structured through a interactive way. Such improvements should help users to satisfy their information needs even more quickly.

Some of the other possible improvements include:

- better interpretation of patterns in the keyword queries
  - shallow parsing of DAS QL
  - maybe even shallow parsing of natural language, e.g. for question focus extraction (needs to be figured out if useful)
- technical improvements
  - Client-side implementation of keyword search - large parts of keyword search could be moved to client-side, saving the server resources. [2]
  - web sockets could be used for auto-completion
  - one the system's accuracy is finally tuned, evaluate the impact of the methods allowing to further improve the runtime performance (it is already satisfactory)
- then more historical queries and logs are gathered, explore possible improvements to the entity matching methods and the machine learning approaches

The more detailed list of future items is available on the project's repository at the github.

[2] 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/

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

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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. Complete list of Project Deliverables/Artefacts

1. Keyword search engine and related components (as described in chapter 3)
    a) design, implementation, user interface, and tuning of its parameters; including implementation of various entity matching techniques & heuristics
    b) prototype of advanced auto-completion input widget (as described in section 3.5.1)
    c) slight relaxation of DASQL (as described in appendixB.2)
        i. prototype implementation of "simple service orchestration" even then the existing fields are not known in advance
    d) bootstrapping of: 1) allowed values, 2) fields in service results
2. log analysis and data service performance benchmarking at CMS
    a) development of tools for log analysis and data service performance benchmarking (per query types)
    b) presented solutions for data service providers (see appendix B.1)
3. user surveys, presentations and tutorials at the CMS Collaboration
    a) interim report to developers
    b) cooperation with a selected group of ~5+ users on constant feedback on the system developed
    c) tutorial for Computing Operators group on EII at CERN (DAS query language, the constraints, and introduction to keyword search)
    d) Future: at least two more presentations/tutorials to come in the week of 18th of March (CMS Computing & Offline Week for meetings) to get further exposure and feedback from the CMS Collaboration

All of the deliverables, including detailed future work items, are available on the author's copy (fork) of the project's repository, at gihub: https://github.com/vidma/DAS

# B. Data Integration "War stories" at the CMS Experiment

## B.1. 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!).

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

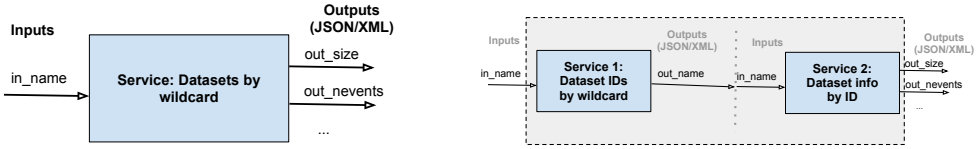
### Estimating query running time

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 [12, 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 .

## B.2. Relaxation of the Query Language




a) how it appears to the user b) actual execution workflow

Figure B.2.1.: Simple service orchestration

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 [2]). This is because by design the DAS system do not store complete service mappings, and do not know in advance all of the fields to be returned. This has a large advantage, as this low coupling, allows seamless extensibility at the services side, and reduces manual efforts needed.

However, this was 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.

A prototype was implemented on the most simple compound queries, so that an entity could always return same list of fields, which is fairly easy to implement and very useful for users:

- it 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)

[2] 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

This is implemented by the fields being returned by the most important services (these could even depend on their input parameters), and if the request didn't return the required fields, and they exist on the results returned by entity's primary key, the results of the first query, are joined to the requests by entity's ID (see figure B.2.1).