Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 1 (2010) 1529–1537
Procedia Computer Science
www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2010

The CMS Data Aggregation System

Valentin Kuznetsov[a,*], Dave Evans[b], Simon Metson[c]

[a] *Cornell University, Ithaca, New York, USA*
[b] *Fermilab, Batavia, Illinois, USA*
[c] *Bristol University, Bristol, UK*

Abstract

Meta-data plays a significant role in large modern enterprises, research experiments and digital libraries where it comes from many different sources and is distributed in a variety of digital formats. It is organized and managed by constantly evolving software using both relational and non-relational data sources. Even though we can apply an information retrieval approach to non-relational data sources, we can't do so for relational ones, where information is accessed via a pre-established set of data-services.

Here we discuss a new data aggregation system which consumes, indexes and delivers information from different relational and non-relational data sources to answer cross data-service queries and explore meta-data associated with petabytes of experimental data. We combine the simplicity of keyword-based search with the precision of RDMS under the new system. The aggregated information is collected from various sources, allowing end-users to place dynamic queries, get precise answers and trigger information retrieval on demand. Based on the use cases of the CMS experiment, we have performed a set of detailed, large scale tests the results of which we present in this paper.

Keywords: Meta-data, data aggregation, information discovery, HEP.

1. Introduction

The European Organization for Nuclear Research, known as CERN, plays a leading role in fundamental studies of physics. It is also known as a place where many innovations in the area of computer science were developed, e.g. The World Wide Web. Today, the Large Hadron Collider (LHC) at CERN is marking a new era of High Energy Physics (HEP), promising to deliver a few PB of data each year. At this scale the "information discovery" within a heterogeneous, distributed environment becomes a key ingredient of successful data analysis. The data and associated meta-data are produced in variety of forms and digital formats. They are stored and retrieved from relational and non-relational data-sources, such as RDMS systems, document oriented databases, blogs, twikies, file systems, customized applications, etc. Working in such an environment requires a lot of expertise and demands a straightforward, simple way to look-up the desired information. Well-known solutions, e.g. data-services, are tightly coupled to a specific data sources and end-users are left with the manual tasks of bookkeeping and relating the information from them.

*Corresponding author
Email address: vkuznet@gmail.com (Valentin Kuznetsov)

1877-0509
doi:10.1016/j.procs.2010.04.172

Here we present our work on the Data Aggregation System (DAS) which provides the ability to search and aggregate information across different data-services. While designed for the CMS High-Energy Physics experiment at LHC, the strategies and technology could be used elsewhere.

The rest of this paper is organized as follows. In section 2 we discuss related work in the domain of keyword search over relational data sources. Section 3 breifly describes the Compact Muon Solenoid (CMS) experiment and its data model. In section 4 we present the architecture of the DAS system, including discussion of its various components. Finally, our results are summarized in section 5.

2. Related Work

Even though the idea of keyword queries over the relational database is not new, it is still under discussion in computer science domain. A few alternative solutions have been proposed to address this issue in last several years. In [1] the conjunctive keyword queries, i.e. retrieval of documents that contain all query keywords, has been discussed. It requires that all keywords are matched in a row from a single table or joined tables. Authors of [2] make a step forward and evaluate information around matched values. The results are presented as customized templates to end-users. These and other related works, (see [1, 2] for more references), are based on an approach that builds some kind of graph over the underlying database and generates a new database out of the source database to perform the work. The proximity of the results in those techniques are mostly confined to text-based values stored in a database, since numerical ones are mostly useless without knowledge of the context of the input value. For example, typing a value 100 in the search input field can lead to plenty of inappropriate matches, e.g. row ids. In order to be answered correctly, an additional keyword is required to clarify its meaning, which should be matched with particular entity (table.column) in the underlying database. Therefore logical questions, e.g. *find me the total number of files whose size more then 20 but less then 100* are hard to answer precisely using approaches outlined in [1, 2], while they can be easily accomplished using SQL.

To address these limitations a keyword-based query language was introduced in [3]. It used a set of pre-defined keys mapped to an underlying schema and a shortest path algorithm which employed foreign key constraints to build SQL queries. As a result the question *I'm looking for files which contain data taken on a certain date and located at a particular site* was represented as simply as [3]

```
find file where date 2009-01-02 11:59 CET and site = T2
```

without specifying any relational details. This solution represents a common use case in the HEP community and provides an intuitive mapping between the mental model of the end-users and the underlying database schema.

At the same time, data aggregation is commonly addressed in the context of federated databases, see for example [4]. The data from different RDMS systems are stored into the federated DB where SQL queries can be used to search for desired data. Apart from many internal obstacles, such as schema, domain and naming conflicts, it is external issues that are often a large barrier within a distributed, heterogeneous environment where different security policies are in place. In the end the usage of SQL and/or any other Query Language is required to search for data.

To avoid these limitations we resolved to build a keyword-search based system around existing data-services whose nature and access policies are in place and well understood. The key ingredients are the mapping and analytics services which provide the necessary glue across existing API/notations and user queries. This approach allows us to perform dynamic data aggregation across various data-services, achieve precise answers using keyword-based queries and organize information retrieval on demand.

3. CMS data model

The Compact Muon Solenoid, (CMS) [5] is one of the two general purpose particle physics detectors built on the Large Hadron Collider (LHC) at CERN in Switzerland and France. The experiment is designed to explore the frontiers of physics and provide physicists with the ability to look at the conditions presented in the early stage of our Universe. More then 3000 physicists from 183 institutions representing 38 countries are involved in the design, construction and maintenance of the experiment.

The CMS distributed computing and data model [6] is designed to process and efficiently manage the few PBs of data expected each year during normal operation of the LHC. The computing resources available to the CMS collaboration are geographically distributed, interconnected via high throughput networks and operated by means of various Grid software infrastructure. The model is designed to incorporate a broad variety of both mass data storage and data processing resources into a single coherent system. CMS uses a multi-Tiered model, where specific tasks related to data taking such as processing, archival and distribution, are assigned to each tier based on the CMS data model. For example, the Tier-0 center at CERN is responsible for archiving the data coming out from the detector, prompt first pass reconstruction and data distribution to Tier-1 centers, located around the world. The Tier-1 centers provide archival storage, CPU power for high priority selection and re-reconstructions, and distribution points to the Tier 2 resources. The Tier-2 centers are dedicated for user analysis tasks and production of simulated data.

A broad variety of data-services have been designed and developed to maintain detector and production operations, including detector conditions databases, data bookkeeping services, data transfer and job monitoring tasks. While the majority of these services are located at CERN, they need to be accessed remotely, and combined with other distributed services.

Once such a conglomerate of data-services start operating the obvious question arises: how do we find the desired information across multiple data-services in our distributed environment? Even though the individual data-services were designed to answer specific questions about the data they serve, the ability to search and relate information among them remains a tedious human task. The growing amount of information and desire to make cross-service queries led us to design and develop the new Data Aggregation System.

4. Data Aggregation System

The goal of the Data Aggregation System (DAS) is to fetch and aggregate meta-data on demand from existing CMS data-services under one umbrella by using their APIs, security policies, etc. By contrast, the approaches discussed in [1, 2, 4] enforce a publication step of data into a dedicated database or central repository [7]. Therefore, quite often it was required that data should be converted into common data structure/format in order to enable aggregation, see [8]. In DAS the data are provided by existing data-services and infrastructure, whose implementation, data representation and security policies are left intact. Consequently, DAS does not require data preservation and transaction capabilities. Instead DAS can be treated as a proxy or caching layer where information retrieval and aggregation are triggered dynamically by user queries. Imagine: a user opens a browser and places a query, e.g. *file block=abc*. The DAS system responds with aggregated infromation about files for the provided set of conditions from all CMS data-services. Information is either retrieved from the DAS cache or fetched on-demand from relevant data-providers. Such a system provides uniform access to all meta-data in CMS via simple, intuitive query interface. It neither forces users to learn the structure of CMS data-services nor requires data-providers to agree on common data representations and the requirement to have central meta-data repository.

4.1. DAS architecture

The design of the DAS system was based on previous studies of the CMS Data Bookkeeping System (DBS) [9, 10]. The DBS Data Discovery service [11] was based on the DBS Query Language (DBS-QL) [3]. It provided a syntax similar to SQL, without requiring specific knowledge of the underlying DB schema. Quick adoption and wide usage of DBS-QL in CMS gave us confidence in the chosen approach and provided a basis for building the DAS system.

As was pointed out in [12] the mixed content and mixed meta-data and meta-data consistency of the underlying services needs to be carefully considered in the design of the system if it is to be a successful information discovery service. Starting from here we designed the DAS system as an additional layer on top of the existing data-services within the CMS computing infrastructure by imposing the following set of requirements: support keyword based search queries with the ability to use conditional operators; use existing heterogeneous software environments and the distributed nature of the data-services; preserve the access rules and security policies of individual data-services; retrieve data on demand and aggregate them if necessary across multiple data-services; be transparent to data-services implementation and their data formats.

The choice of an existing RDMS as the DAS back-end(s) was ruled out for several reasons. For instance, we didn't require any transactions and data persistency in DAS; the dynamic typing of stored meta-data objects was one of the

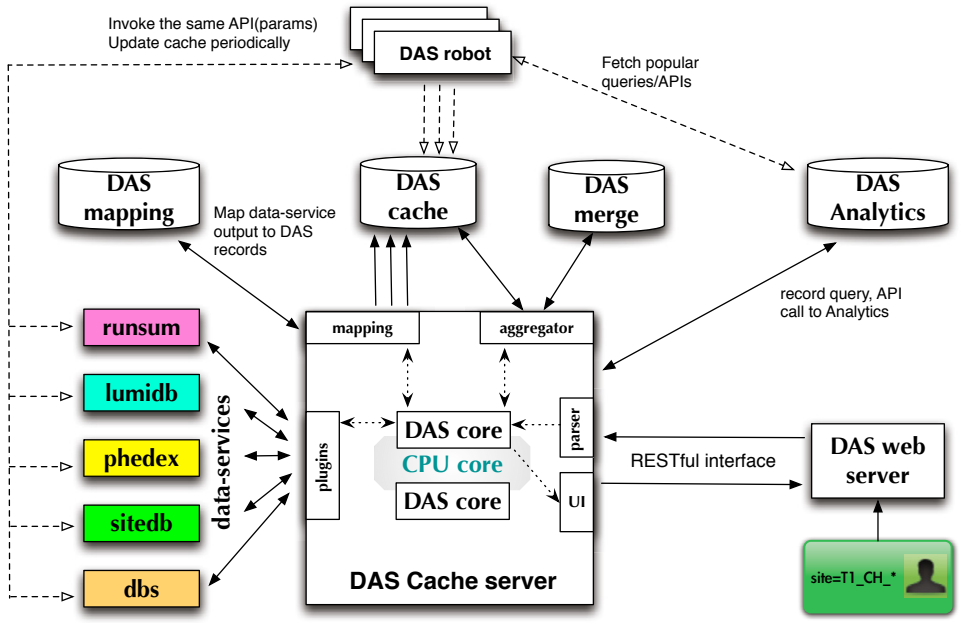

Figure 1: The DAS architecture diagram. It consists of cache server, analytics, mapping and cache databases. The data in a cache can be shared across multiple queries. The information retrieval is triggered on demand basis by invoking appropriate data-service APIs. The optional DAS robots (Unix daemons) can be used to pre-fetch the most popular query requests. The solid, dashed and dotted lines represent outer, inner and optional process communications between different DAS components, respectively.

requirements, etc. Those arguments forced us to look for alternative IT solutions. Through the careful analysis of available options, such as file-based and in memory caches, key-value databases and documented-oriented databases, we made our choice in favor of the last technology. Among them we evaluated CouchDB [14] and MongoDB [15]. Both are "schema-less" databases providing arbitrary document structure storage, replication and fail-over features. We made our decision in favor of MongoDB, due to its support of dynamic queries, full indexes, including inner objects and embedded arrays, and auto-sharding. Our preliminary benchmarks have shown that it can sustain the desired load and size capable of handling our meta-data information. We used the document-oriented database, MongoDB, as a back-end for three main DAS components: Mapping, Analytics and DAS cache databases, which will be discussed next.

The DAS architecture is shown in Fig. 1. It consists of the core library with support of pluggable modules for data retrieval; the caching layer to store and aggregate results into DAS Cache and DAS merge databases, respectively; the request service to handle user queries; the Mapping DB to keep information about data-service APIs, e.g. notations and mapping to DAS keys and the Analytics DB for query analysis.

4.2. DAS core

The DAS core library is responsible for data retrieval on demand. It has the following components: input query validator and parser, query analyzer, workflow builder, execution engine and data-service plug-ins. The communication between client (web server or CLI) and DAS cache server is done via the REST [16] interface. Each input query is recorded into the Analytics DB for further analysis, and is decomposed into a set of conditions and/or selection keys. First the results are looked up in a cache and delivered to the requester if a superset of the input conditions is found in analytics DB.[1] Otherwise, the appropriate workflow, a set of API calls, is constructed for the execution

[1]It is important to note here that we didn't look-up results in a cache, but rather compared the input query condition set to the records in the analytics DB in order to understand if other API calls covered the input request.

engine. It invokes the API calls via the data-service plug-ins and stores the results into the cache. At this step we also performed necessary data aggregation for records with similar key-value pairs. The data-service plug-in implementations are similar and their logic is largely abstracted, apart from a few extensions such as specific security modules for data-services accessible via private network. Finally, the results are retrieved from the cache and delivered to the user.

4.3. DAS Mapping DB

The Mapping DB is used to track services known to DAS and how DAS should interact with them. It collects information about data-service APIs, translates API input/output parameters into DAS keys and contains mapping between DAS records and their UI representation.

On identifying API's that are accessible to DAS we store their names, input parameters and associated DAS keys into the Mapping DB. For example

```
{"system": "phedex", "format": "XML", "urn": "fileReplicas",
 "url": "http://a.b.com/phedex/datasvc/xml/prod/fileReplicas",
 "params": {"node": "*", "block": "required", "se": "*"},
 "das2api": [{"pattern": "", "das_key": "file.block.name",
    "api_param": "block"}, {"pattern": "re.compile('^T[0-3]_')",
    "das_key": "site", "api_param": "node"}],
 "daskeys": [{"map": "file.name", "key": "file", "pattern": ""},
             {"map": "file.block.name", "key": "block", "pattern": ""}]}
```

represents a DAS mapping record. This record can be read as following: the `fileReplicas` API (urn) provided by given url delivers data in XML data format and belongs to the `phedex` system. It accepts `node`, `block` and `se` input parameters, shown in `params` part of the record. Each record has two maps: `daskeys` for mapping input DAS keys into DAS record access keys and `das2api` for mapping DAS access keys into api input parameters.

The Mapping DB identifies a set of pre-defined DAS keys, e.g. `file, block`, etc., to be used by end-users in their queries. Each key is mapped onto set of API calls, whose output is retrieved, cached and aggregated as required.

4.4. DAS Analytics DB

The DAS Analytics DB collects information on user requests against the system. Each request that DAS handles is recorded. The Analytics DB tracks: the users request; how DAS has mapped the request onto data service API's; the time taken by the remote data services to process the API calls.

By recording this information we can plan pre-fetch strategies for common queries, identify issues in remote data services and cross check that DAS resolves requests in a deterministic manner.

4.5. DAS caching system

The DAS caching system is used to dynamically fetch and aggregate data upon user requests. It consists of the DAS cache and the DAS merge independent databases. The former is used to store the raw results coming out from data-services, while the latter contains aggregated records. Each data record in the DAS cache/merge DBs contains expiration timestamps provided either by the data-service itself or based on default values during data-service registration. We use the following algorithm (code snippets are written in Python [17]):

1. Decompose query into set of conditions (`conds`) and selection keys (`skeys`).
2. Check cache for query hash. If found, look-up results from the cache, otherwise proceed to the next step.
3. Identify list of APIs which can serve requested data

   ```
   apilist = [r for r in mapping_db(skeys, conds)]
   ```

   Here `r` represents a triplet `url, api, args`, and `mappind_db` defines the translation between the input query and the data-service APIs which can delivery the data.
4. Iterate over APIs

```
for url, api, args in apilist:
    if  superset_in_analytics(api, args):
        continue
    for row in parser(get_data(url, api, args)):
        yield row
```

Here `superset_in_analytics` checks if the provided `api, args` are a subset of api call(s) already registered in the Analytics DB. The `get_data` and `parser` are define methods to fetch and parse data, respectively.

5. Perform aggregation by iterating over records from the previous step

```
if cache_has(daskey):
   existing_record = get_from_cache(daskey)
   merge(row, existing_record)
   insert_into_cache(row)
else:
   insert_into_cache(row)
```

The `daskey` is a key which identifies the record defined in Mapping DB, e.g. `file.name` for `file` record. Bulk insert operations are used at this step to increase throughput.

6. Get results from the cache

The cache population can be automated by running an independent set of daemons, DAS robots, which monitor the Analytics DB for suitable requests (for example queries that are popular, periodic in nature or expensive) and pre-fetch data into the cache.

4.6. DAS queries

All DAS queries are expressed in a free text-based form, either as a set of keywords or key-value pairs, where a pair can represent condition, e.g. `key > value`. We use a pre-defined set of keys registered in DAS to describe common entities of our data. For example, `dataset`, `file`, `block`, `run`, `event`, etc. The choice of keys is based on the analysis of day to day user operations [10]. Each key represents a DAS record. Every record is stored into the cache in JSON format. Due to the "schema-less" nature of the underlying MongoDB back-end we are able to store DAS records of arbitrary structure, e.g. dictionary, lists, key-value pairs, etc. Therefore every DAS key has a set of attributes describing its JSON structure. For instance a document

```
{"block": {"name": "abc", "size": 1, "replica": [...]}}
```

is associated with a `block` key who has the following attributes `name`, `size`, `replica`. The set of attributes is based on a set of meta-data information stored in a record, allowing for nested structures, e.g. `block.replica.size`. Our users are able to specify the context of their queries by using the appropriate attribute(s). For example, a query for `block` represents a search for all block records rather then the "block" string, while the query `block.name=abc*` instructs the system to look-up appropriate blocks with matching names. Numerical values, conditions and grouping are also easy to express, for example, a query of the form `run > 10 and run < 100` can be used to retrieve all run records for the given range of run numbers. As a complement to the queries, we will soon be adding a dynamic index of DAS records with a keyword-based search within record content.

All input queries are transformed into the MongoDB query syntax which is itself a JSON record. It contains a set of selection keys (`fields`) and conditions (`spec`). For example, the query `dataset, file=abc, run > 10` is represented as

```
{"fields": ["dataset"], "spec": {"file": "abc", "run": {"$gt": 10}}}
```

Therefore users queries are stored into Analytics DB similar to other DAS records and used for further analysis.

4.7. Data-services and aggregation

As discussed in section 4.3 the DAS Mapping DB is the authoritative source of information on the relationships between all the data services known to DAS. Based on the information stored in the Mapping DB we are able to retrieve appropriate data from different data-services, re-map them into DAS notations and aggregate them on demand. For instance, the Data Bookkeeping System (DBS) [9] and data location system (PhEDEx) [18] provide information about CMS file block structure and locations. DBS associates groups of files into blocks, while PhEDEx tracks the blocks location.

When a query for block meta-data is made, we are able to identify the appropriate set of DBS and PhEDEx APIs, retrieve the raw data from the services, format it for DAS use and add the resulting information to the DAS cache. The records are merged (aggregated) based on identical key-value pairs, e.g. based on `block.name` key. Here is an example of such an aggregated record:

```
{"das_id": ["4b195404e2194e21c4000030", "4b1953fee2194e21c4000002"],
 "block": [{"name": "/a/b/c#123", "replica": [{"complete": "y",
            "site": "T1_US", "nfiles": "18", "se": "a.b.com"}, {...}]},
           {"name": "/a/b/c#123", "nevents": "570200", "path":"/a/b/c"}],
 "das": {"expire": 1259952908}}
```

The `das_id` indicates the records stored in the DAS Analytics DB, the `block` part represents the aggregated block meta-data information from DBS and PhEDEx and the `das` part contains expiration time stamps about how long the information is valid. Usage of this information was a valuable contribution in debugging process of data-services themselves, e.g. identification of data-service inconsistencies, latency studies, etc.

5. Results

In this section we discuss preliminary results achieved with the DAS system using different CMS data-services, see [5, 6].

At the moment, the amount of meta-data stored in all services participating in DAS is around 75 GB and 200 GB in tables and index sizes, respectively. We estimate collecting around 500 GB of meta-data each year during normal CMS operations. The average query rate to DAS is expected to be $\leq$10K requests a day.

To test DAS performance we measured the elapsed time and CPU/RAM utilization required to fetch, store and aggregate information from different data-services. Table 1 summarizes the biggest contributors so far, the block meta-data information provided by DBS and PhEDEx CMS data-services.

System	Format	Records	Elapsed time no cached data	Elapsed time w/ cached data
DBS yield	XML	386943	68 sec	0.98 sec
PhEDEx yield	XML	189679	107 sec	0.98 sec
Merge step	JSON	576622	63 sec	0.9 sec
DAS total	JSON	392635	238 sec	2.05/50.7 sec

Table 1: Time required to fetch, parse and aggregate block information from DBS and PhEDEx systems. The last row shows aggregated results and total time spent in all DAS components. The total number of DAS records are calculated as the number of aggregated records plus left overs, the non-matched records from both systems. The look-up time shown in the last column represents the query look-up time (2.05s) and time required to get all records written to disk (50.7s).

The initial DAS implementation is done in the Python language [17]. All tests are performed on a 64 bit Linux platform with 1 CPU for the DAS server and 1 CPU for the MongoDB database server. All tests are performed multiple times to avoid fluctuations. The elapsed time is measured and consists of the following components:

Elapsed time = retrieval time + parsing time + re-mapping time + cache insertion/indexing time + aggregation time + output creation time

Here *retrieval time* is the time required to access data from data-service, *parsing time* is the time required to read and parse the received data from the services, and *re-mapping time* is the time required to convert that information into DAS. *Cache insertion* and *indexing time* represents time spent on DAS back-end caching, *aggregation time* is the time required to merge objects into DAS records based on their common keys (`block.name` in this case) and *output creation time* is the time required to write DAS records to disk.

The individual tests of DAS components (DBS and PhEDEx) show roughly the same performance, although the time taken by the PhEDEx component is longer due to more complex records which leads to higher retrieval and parsing time. As shown in Table 1, the look-up time spent to access individual components (last column) is quite reasonable, roughly 1 second for both systems. The final time to get DAS records on disk is about 50 seconds. This is mostly due to I/O and conversion operations from the binary data format used by MongoDB, BSON, to the DAS data format, JSON.

The individual benchmarks of MongoDB have shown that it can deliver an insert rate of 20K docs/sec and provide a look-up time of 10^{-5} sec per random document. The results of DAS stress tests have demonstrated that we can achieve a throughput of ∼6000 docs/sec for raw cache population, ∼7600 docs/sec for reading and writing DAS records to disk, which is suitable for our needs. Please note, we have a relatively small audience of users who perform complex queries and get precise results, while search engines are designed for a large number of users asking for imprecise, simple data. Since all tests were performed on a single CPU, further improvements from expanding to multiple CPUs are expected. In addition, more performance advantage can be gained by writing dedicated C/C++ python extensions for data intensive operations.

6. Future work

In the near future we plan to put DAS into production and confirm its scalability, transparency and durability for various CMS data-services. Our interests are plug-able interface for participating data-providers, analytics framework and pre-fetch strategies as well as horizontal scaling of the system. Our goal is to sustain 1TB of meta-data per year. In a long term, we would like to use DAS across various data-services in the HEP community and other disciplines. Even though DAS was developed for the CMS experiment, its architecture is transparent to participating data-providers and meta-data content. Therefore the generalization of DAS is feasible. Such work has began under the umbrella of DISCOVER Research Service Group at Cornell University [19] which has expressed their interest to use DAS in other scientific domains.

7. Summary

In this paper we have presented a new data aggregation service (DAS) developed for the CMS High-Energy experiment at LHC, CERN, Geneva, Switzerland. We have developed a prototype for DAS, designed to provide caching and aggregation layers on top of the existing relational and non-relational data-services in a close to real time fashion. The data is retrieved on demand, with pre-fetching of common queries, determined from an extensive analytics database, also possible. The DAS system is currently being commissioned for production use by CMS.

Data taking at CMS began in December 2009, and, alongside simulation data, we expect to accumulate petabytes of data yearly. This will have of the order of a terabyte of associated meta-data that will need to be easily queried by physicists. From the performance studies of DAS presented in this paper we expect that the system will be able to sustain such load and will provide generic data-discovery service for CMS experiment for years to come.

8. Acknowledgments

This work was supported by the National Science Foundation, contract No. PHY-0757894, and Department of Energy of the United States of America.

References

[1] S. Agrawal, S. Chaudhuri, G. Das, "DBXplorer: A System for Keyword-Based Search over Relational Databases", ICDE 2002, pp. 5-16.
[2] G. Koutrika, A. Simitsis, Y. E. Ioannidis, "Précis: The Essence of a Query Answer", ICDE 2006, pp. 69-78.
[3] V. Kuznetsov, D. Riley, A. Afaq, V. Sekhri, Y. Guo, L. Lueking, "The CMS DBS Query Language", CHEP 2009.
[4] L. Haas, E. Lin, "IBM Federated Database Technology", http://www.ibm.com/developerworks/data/library/techarticle/0203haas/0203haas.html
[5] R. Adolphi et al., "The CMS experiment at the CERN LHC", JINST 0803, S08004 (2008).
[6] C. Grandi, D.Stickland, L.Taylor et al., "The CMS Computing Model", CERN-LHCC-2004-035/G-083 (2004).
[7] The Integrated Rule-Oriented Data System, http://www.irods.org/
[8] Open Archives Initiative, http://www.openarchives.org/
[9] A. Afaq, et. al., "The CMS Dataset Bookkeeping Service", J. Phys. Conf. Ser, 119, 072001 (2008).
[10] A. Dolgert, V. Kuznetsov, C. Jones, D. Riley, "A multi-dimensional view on information retrieval of CMS data", J. Phys. Conf. Ser, 119, 072013 (2008).
[11] https://cmsweb.cern.ch/dbs_discovery
[12] C. R. Arms, W. Y. Arms, "Mixed Content and Mixed Metadata Information Discovery in a Messy World", chapter from "Metadata in Practice", ALA Editions, 2004.
[13] http://www.mysql.com/
[14] http://couchdb.apache.org/
[15] http://www.mongodb.org/
[16] Fielding, Roy Thomas "Architectural Styles and the Design of Network-based Software Architectures", Doctoral dissertation, 2000, University of California, Irvine
[17] Python programming language, http://www.python.org
[18] L. Tuura et al., "Scaling CMS data transfer system for LHC start-up", J. Phys. Conf. Ser, 119, 072030 (2008).
[19] http://drsg.cac.cornell.edu/