Data Aggregation System - a system for information retrieval on demand over relational and non-relational distributed data sources

G Ball[1], V Kuznetsov[2], D Evans[3] and S Metson[4]

[1] Imperial College London, London, UK
[2] Cornell University, Ithaca, New York, USA
[3] Fermilab, Batavia, Illinois, USA
[4] Bristol University, Bristol, UK

E-mail: `gordon.ball@cern.ch, vkuznet@gmail.com`

Abstract. We present the Data Aggregation System, a system for information retrieval and aggregation from heterogenous sources of relational and non-relational data for the Compact Muon Solenoid experiment on the CERN Large Hadron Collider. The experiment currently has a number of organically-developed data sources, including front-ends to a number of different relational databases and non-database data services which do not share common data structures or APIs, and cannot at this stage be readily converged. DAS provides an single interface for querying all these services, a caching layer to speed up access to expensive underlying calls and the ability to merge records from different data services pertaining to a single primary key.

1. Introduction

In 2010, data taking at the CMS experiment began in earnest, with approximately $42pb^{-1}$ of proton-proton collision data recorded at the time of writing. At full data-taking rates, this is expected to produce around 5PiB of raw and reconstructed data per year. Associated with this is around 1TiB per year of metadata, such as data locations, dataset descriptions and machine conditions.

Each of these metadata aspects are stored into a specialised system, using a variety of different technologies. Each provides specialised interfaces to query the specific metadata stored. In the pre-data-taking epoch, this situation was satisfactory as most systems were used in isolation and only then by experts. However, in the data-taking era users regularly need to perform queries across multiple data services (for example, to look up the recorded luminosity, machine conditions and software configuration for a particular dataset) which prove very expensive to serve under the existing model. DAS simplifies data look-up to end users by providing a single point of access and provides a caching layer to data services.

Although designed in the context of the CMS experiment, DAS itself does not make any experiment-specific assumptions about the data services it handles and is generalisable to other similar situations [1].

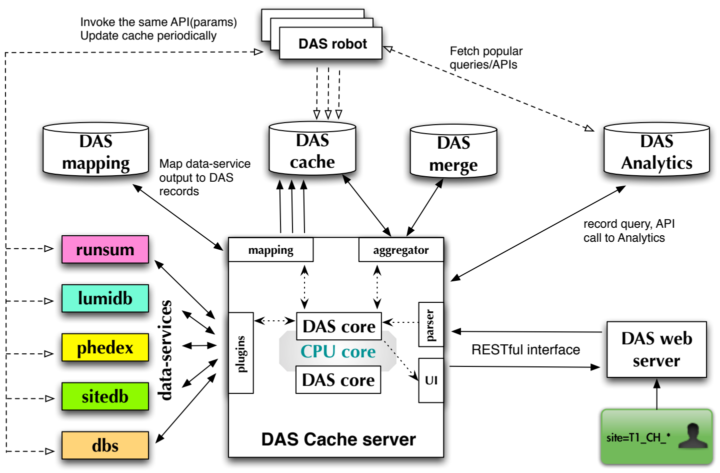

Figure 1. DAS architecture, showing the relationships between *web server*, *cache server*, *analytics*, *data services* and MongoDB (*cache* and *merge*).

2. Architecture & Implementation

DAS is designed as an additional layer on top of a heterogenous ecosystem of existing CMS data services.

The architecture of DAS was designed as a series of independent components, which can scale from all running on a single node to multiple nodes and duplicate components, as necessary (Fig. 1).

The *web server* handles user sessions, whether they originate from a browser or automated scripts. Queries made here are passed onto the *cache server* for processing, during which time the *web server* periodically polls for the current status (displayed to the user by AJAX) until the request is completed. An MD5 hash of the input query is used to identify the correct output (and hence multiple users making identical queries will only result in a single worker thread doing the processing). Methods are provided here for output in machine-readable formats as well as human-readable formatting.

The *cache server* consists of a pool of worker threads which handle the DAS queries received from the web front-end. Each query is handled entirely by a single thread. A detailed description of the workflow is given in [2]. The worker thread first parses a text-based query from the DAS Query Language (Sec. 2.1) into an internal representation. The query is then analysed to identify the set of data service APIs which are relevant. The *cache* is checked for items of data matching the query, or data that would form a superset of the query. If any data cannot be found, the relevant service (Sec. 2.2) is called to fetch and transform the data, and insert it into the raw cache (Sec. 2.3). Once all the data is in the raw cache, records sharing common keys are merged together into a single document, which is stored in the merge cache. If any additional data processing has been specified in the query, it is performed on the merged document before being returned.

The *cache* consists of one or more MongoDB [3] shards. MongoDB is a document store which natively stores the JSON documents DAS uses as its internal representation. This is used both for the primary record stores (the *raw* and *merged* caches) and for the ancillary databases required by DAS, such as storing the server logs, analytics data and mapping between keys. MongoDB also provides GridFS [4], which DAS uses to store otherwise oversize documents, typically the result of merging large numbers of raw documents together.

The *analytics server* provides a facility for scheduling regular tasks. The rest of DAS operates only when triggered by user input, and although some clean-up is performed at run-time (such as discarded expired records) it is also necessary to have a facility for running asynchronous operations. Analytics consists of a task scheduler and a pool of workers which execute the tasks. This is done independently of the *cache server*, so heavy analytics tasks will not interfere with end-user use.

DAS assumes that the data in the cache can be recreated from the original data sources at any time, and thus it can function with a limited amount of space by deleting old data, nor does it require backup of this space. The design assumes the total number of records to be $O(5M)$, and MongoDB is capable of effectively functioning with collections of this size. Furthermore, only a fraction would be expected to be resident in the cache at any one time.

DAS is designed as an entirely read-only system, with no ability to write data back to the underlying services, which simplifies the requirements for authentication and validation that this would otherwise require.

A random subset of results retrieved by DAS are examined to learn which data members are returned by which underlying queries, so that the web interface can suggest which input DAS keys will find the desired result.

The non-MongoDB parts of DAS are entirely written in python. Using python does come with a performance penalty but it allows for a fast development cycle, access to a number of useful libraries and is consistent with other CMS web projects.

2.1. DAS Query Language

DAS queries are made in a custom text-based language. Work originally centred on an extended version of the query language already developed for the CMS Dataset Bookkeeping Service [5] but the syntax was found unsuitable. The DAS syntax broadly resembles using pipes and commands in a UNIX shell, albeit with entirely dissimilar implementation.

Queries are parsed using the PLY [6] parser for tokenising and lexical analysis, with a caching layer for previously parsed queries. The query is internally represented as a python dictionary containing the conditions and any additional operations.

A query is structured:

```
conditions | filters | aggregators OR map-reduce
```

To make a query, a user must know which *DAS key* describes the data they are seeking. DAS keys describe a single logical object, descriptions of which may exist in multiple data services. Examples of DAS keys used for CMS include **dataset**, **run_number** and **person**. In most cases these are the same names used by the data services, but DAS hides small differences in data service terminologies (eg, **run_number** is variously called **runNumber** and **Run** by different services).

The *conditions* consist of one or more DAS keys, optionally followed by an operator and operand. Examples of queries include:

```
city=Geneva
dataset site=institute.ac.uk
run in [123, 456]
```

The conditions determine the total data for the query, but since this may still be a significant volume of data (or contain a significant number of fields of no interest to the user), further operations can be performed on the data.

Filters are commands that either eliminate complete data records or prune data members within a record. At this time only two are provided; *grep* works similarly to the UNIX command of the same name, filtering all but the given data members from the output, and *unique* eliminates duplicate records. This is implemented by converting the filter conditions into MongoDB query format.

Aggregators are functions which run over all the selected (and filtered, if applicable) records, summarising their contents. The basic aggregators available are simple numerical functions like *sum*, *count* and *avg*, which perform the named aggregation over all data members of a given name. Aggregation is implemented using python coroutines.

An example of a “complete” query might be:

```
block dataset=/a/b/c* | grep block.size | sum(block.size), max(block.size)
```

The set of available filters and aggregators is likely to grow in accordance with user needs. For more complex use-cases, it is possible to run a javascript map-reduce function on the MongoDB cache server. DAS QL does not allow functions to be specified, so such functions are written in advance and stored in the cache, and their execution requested with an alias.

2.2. Services

Almost any source of information can be incorporated in DAS, such as an SQL database cursor, a command called in a subshell or an HTML page scraper, but in practice most data services used by DAS are APIs accessed over HTTP and returning data in JSON or XML format.

Each data service is described in a mapping document (supplied in YAML [7] format) which describes the functions available, the mapping from input DAS keys to actual arguments, and finally a mapping from the returned keys back to DAS keys. For a service accessed over HTTP and using a standard format, adding a new service only requires this mapping.

For more complex cases, such as those for which authentication is required, or for different source types a plugin must be written which translates DAS requests into the necessary calls, and then transforms the output into DAS JSON documents.

The mapping document may also specify presentation information which is used to produce more human-friendly output by mapping key names to descriptions.

When a query for which no exact match exists in the cache is received, the set of input DAS keys is considered to determine which services and their respective APIs to invoke. If the query consists only of keys with associated conditions, the relevant APIs are all those for which the accepted DAS keys are either equal or a superset of the input keys. If no API matches the set of input queries, it is decomposed into multiple sub-queries.

Before an API query is made, a check is made for existence of a superset query already in the cache, if wildcarding is supported by the service.

2.3. Caching & Merging

Data in DAS is principally stored in two seperate collections; the raw and merged caches.

Two types of data are stored in the raw cache; the documents returned by the respective APIs and documents describing the current status of queries. The latter serves as the central record for a query, holding the original query, description of the APIs called, status of the processing and a common key with all the results documents.

Once all the data for a query is available in the raw cache, a merged record is created for each primary DAS key. Merged documents may exceed the maximum size possible for MongoDB records, in which case they are stored using GridFS and a reference to them stored in the merged cache instead.

2.4. Analytics

The DAS analytics system is a daemon that schedules and executes small tasks that access the DAS document store. This can be used to perform prosaic maintainance tasks, such as cleaning out expired data or pruning still-valid data if space becomes limited, but the main function is to analyse the queries received by DAS to provide information to developers and to perform automated cache optimisation.

The caching aspect of DAS is meant to reduce the latency users experience when performing complex, cross-service queries, but since the data in question usually has TTL of less than an hour users will relatively often find that the data they request is not in the cache and must be fetched on demand.

The analytics system hence tries to identify the most popular user queries and ensure that they are always resident in the cache and thus available for real-time retrieval, whatever the actual fetch latency may have been.

It is vital to the success of DAS, both in terms of user experience and system load, that the majority of all queries can be served from the cache. To ensure this, we must have a good pre-fetching strategy that maintains itself based on the actions of users.

Tasks that run on the analytics system may be divided into query analysis and cache populators. The former run infrequently, performing expensive (both in terms of processing and MongoDB access) analysis operations. Having identified the set of queries they wish to maintain, they spawn simpler populator tasks that schedule themselves to pre-empt each expiry of the specified data and perform an update, until the next time the analyser runs and the priorities may be altered.

A simple example of an analyser is designated *ValueHotspot*, and is designed to run every few hours, identifying popular arguments to a given DAS key. Upon each invocation, the set of queries issued since the last invocation which reference the specified key is identified, and a count of each argument made. This is then combined with summary documents from previous invocations (containing the same information for those epochs) to calculate a 30-day moving average. The top 15% most-used arguments are selected, which based on past results represent about 50% of all queries are then kept in the cache until the analyser next runs.

There are a large number of possible future analysis strategies (such as the time structure throughout a week of certain types of queries, automatically pre-fetching information about new datasets as they are released or considering a metric based on total preventible latency rather than total numbers of calls), but some experience with actual users is needed to determine what will actually prove useful.

3. Benchmarks

To benchmark DAS performance, we used a 64-bit linux node with 8 cores (each 2.33GhZ) and 16GiB of RAM, which we would expect to be typical of the hardware DAS would use in production. All DAS systems and MongoDB share this node.

Testing was performed with the cache pre-populated with approximately 5×10^7 records (100 times the expected volume), consisting of *block* records from the CMS PhEDEx [8] and DBS systems. We test with up to 500 clients making requests in parallel. Each client requests one random record from the entire result set (which will test the time required for DAS to process the query but largely exclude the time required to serialise a large output set), using queries of the form `block=/[a-z]*`(past experience with the DBS system shows wildcard queries are very common, even when the user knows the complete string in question). Fig. 2 shows the benchmark results.

4. Future Work

We are currently in process of deploying DAS in the production environment for beta testing. The main development focus at this time is to increase the scope of possible queries, both by adding additional services and increasing the number of APIs supported for existing services. Support currently exists to some degree for the CMS services DBS, PhEDEx, SiteDB, Tier-0,

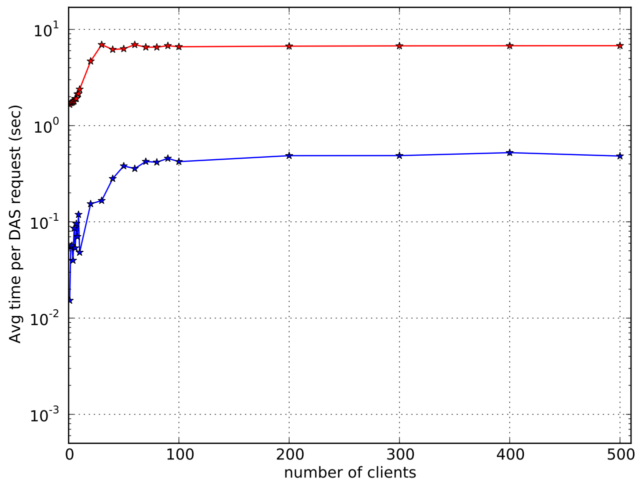

Figure 2. Benchmark showing the average latency for fetching a single randomly-selected document from a wildcard query, with a given number of parallel clients. The cache contains 5×10^7 records. The red line shows the performance of the complete DAS stack, and the blue line the time spent by MongoDB.

LumiDB, RunRegistry, Dashboard and Overview. Most of these however only have a few APIs described by the DAS mapping, and to ensure usefulness to users we need to support as many of the queries supported by their original interfaces as possible.

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

This work was supported by the Science and Technology Facilities Council, UK.

References

[1] DISCOVER RSG *http://drsg.cac.cornell.edu/content/drsg-pilot-projects*
[2] Kuznetsov V, Evans D and Metson S 2010 The CMS data aggregation system *Procedia Comp. Sci.* **1** 1529-37
[3] MongoDB, a scalable, high-performance, open-source, document-orientated database *http://www.mongodb.org/*
[4] MongoDB GridFS *http://www.mongodb.org/display/DOCS/GridFS*
[5] Kuznetsov V, Riley D, Afaq A, Sekhri V, Guo Y and Lueking L 2009 The CMS DBS query language *CHEP*
[6] Python Lex-Yacc *http://www.dabeaz.com/ply/*
[7] YAML ain't markup language *http://www.yaml.org/*
[8] Rehn J, Barrass T, Bonacorsi D, Hernandez J, Semeniouk I, Tuura L and Wu Y 2006 PhEDEx high-throughput data transfer management system *CHEP*