The CMS Data Management System

M Giffels[1], Y Guo[2], V Kuznetsov[3], N Magini[1] and T Wildish[4]

[1] CERN, CH-1211 Genève 23, Switzerland
[2] Fermi National Accelerator Laboratory, Batavia, Il, USA
[3] Cornell University, Ithaca, NY, USA
[4] Princeton University, Princeton, NJ, USA

E-mail: Nicolo.Magini@cern.ch

Abstract. The data management elements in CMS are scalable, modular, and designed to work together. The main components are PhEDEx, the data transfer and location system; the Dataset Booking System (DBS), a metadata catalogue; and the Data Aggregation Service (DAS), designed to aggregate views and provide them to users and services. Tens of thousands of samples have been cataloged and petabytes of data have been moved since the run began. The modular system has allowed the optimal use of appropriate underlying technologies. In this presentation we will discuss the use of both Oracle and nonSQL databases to implement the data management elements as well as the individual architectures chosen. We will discuss how the data management system functioned during the first run, and what improvements are planned in preparation for 2015.

1. Introduction

The CMS [1] experiment at the LHC has recently successfully concluded it's first run. Between the fall of 2010 and the spring of 2013, over 10 billion raw data events were recorded, and 15 billion monte-carlo events produced. The data were analysed using the distributed resources of CMS, managed through the Worldwide LHC Computing Grid (WLCG) [2] and through the CMS dataflow and workflow management tools.

The operational aspects of the CMS computing system are described separately [3]. In this paper we describe the performance of the CMS data management system during run 1. We describe what worked well and what didn't, and analyse the factors that contributed to the performance. Finally, we describe the improvements planned in the major subsystems for the second run of LHC, starting in 2015.

2. Architecture

2.1. Components and data organisation

The CMS data management system has several important tasks:

- Data bookkeeping catalog - describes the data contents in physics terms
- Data location catalog - maintains knowledge of replicas of the data around CMS
- Data placement and transfer management

The core components which achieve these goals are:

- PhEDEx [4], the data-transfer management system

- DBS [5], the Dataset Bookeeping Service
- DAS [6], the Data Aggregation Service

There are a number of other, more specialised services, such as the RunRegistry, SiteDB, ConditionsDB and ReqMgr, but the core components are the ones the user mostly interacts with. All the components of the data management system are designed and constructed separately, interacting with each other and external users by means of web-services.

CMS data is organised into datasets by physics content. Dataset size varies considerably, usually in the range of 0,1 - 100 TB. Within a dataset, files are organised into *blocks* of 10 - 1000 files, purely as an aid to scalability. The data management tools can manipulate individual blocks of files, instead of entire datasets, which offers a finer granularity of control.

Files are typically about 2.5 GB in size. Small files from various processes are merged to maintain a minimum useful size to help scalability of storage and catalogues.

2.2. Computing infrastructure

The computing infrastructure has evolved since it was first described in the CMS Computing TDR [**?**]. We still have the distinction between tiers, with a Tier-0, several Tier-1s, and a large number of Tier-2s.

The Tier-0 is responsible for custodial storage of a copy of the raw data, for prompt reconstruction and for distributing the raw and reconstructed data to the collaboration. The Tier-1s between them maintain a second custodial copy of the raw data, and distribute data further to the Tier-2s. The Tier-2 sites are where the physicists perform their analyses. Monte carlo data is produced at the Tier-2s and uploaded to the Tier-1s for distribution to other sites.

The original computing model called for a hierarchical organisation of tiers, with each Tier-2 attached to a single Tier-1 and each Tier-1 serving only a handful of Tier-2s. For our system (one Tier-0, 7 Tier-1s, and about 50 Tier-2s) this would mean less than 100 data-transfer links. In practice, we now allow Tier-2s to connect to all Tier-1s and to all other Tier-2s, resulting in a fully-connected mesh of aboutt 2000 data-transfer links.

2.3. Data location and the Trivial File Catalog

CMS does not use a central file catalog to store information about file location at each site. Instead, files are known by their *Logical File Name* (LFN), which maps to a different *Physical File Name* (PFN) at each site. We maintain the knowledge that a file exists at a site by storing the LFN-to-site mapping (in PhEDEx), but the mapping from LFN to PFN is the responsibility of each site.

To achieve this, each site maintains a *Trivial File Catalog* (TFC), which is essentially a regular-expression map from LFNs to PFNs and vice-versa. Anyone wishing to know the PFN for a file at a site simply processes the LFN through the regular expressions in that sites TFC to obtain their answer. This approach has several advantages:

- Batch jobs running at the site do not need to contact central services to perform LFN to PFN conversions, they simply examine the local TFC
- Sites can change their storage layout on the fly, without synchronising updates to a central service. As long as they maintain their TFC in sync with their storage, this is transparent to the rest of the system
- Sites can perform complex matching for special situations. E.g. by matching the first letter of a GUID portion of an LFN files can be load-balanced across several backend servers

3. PhEDEx

3.1. PhEDEx

blah blah

4. DBS

4.1. DBS

blah blah

5. DAS

5.1. Data Aggregation Service

The Data Aggregation Service (DAS) [6] was designed to provide a uniform access to distributed CMS data-services regardless of their security policies, implementation details and data storage solution. DAS provides users ability to query underlying data-services via DAS Query Language (QL). It also aggregates meta-data information from various data-service APIs into common records suitable for end-users. Here we discuss details of DAS architecture and current status of the system.

DAS was build on top of NoSQL document-oriented database (MongoDB [8]). It provides several benefits for DAS use case: schema-less storage, embeded query language and very fast read/write operations.[1] DAS has modular design based on MVC architecture [9] and performs the following workflow upon user query:

- fetch data from underlying data-service via existing APIs
- store unprocessed data into cache database
- process data and perform its transformation to common data representation (we unify differences in naming conventions, units and data-formats, etc.)
- aggregate data on a requested key, e.g. dataset, block, etc.
- store results into merge database
- present results to end-users via and provide set of filters as well as aggregation functions to perform basic operation for slicing and representing the data suitable for user tasks

Such design allowed to developed DAS in data agnostic manner. For instance, data can be fetched from underlying data-services regardless of their data format, e.g. JSON, XML, CSV, etc. The data transformation was done via external set of mapping which was maintaned separately from DAS code development. It also gave us a few benefits which were not foreseen from the design cycle. For example, data aggregation on common entity, e.g. data name, may show discrepancies in underlying services if it exists.

DAS uses in-house Query Language which was based on entity relationship used by physicists, see [6]. It consists of the following structure:

```
<selection keys> <set of conditions> | <filters> | <aggregators>
```

The selection keys were based on well known entities such as dataset, block, file, run. The conditions were formed via key=value pairs. DAS-QL provides limited set of filters such as grep, unique, as well as set of common aggregation function such as sum, min, max. The former supported condition operators and groupping, while later can be extended via custom map-reduce functions to support more complex use cases. Therefore to express a question *Find me all datasets at a given site and show only those who has size more then 50* someone will need to write a DAS query in the following way:

```
dataset site=XYZ | grep dataset.name, dataset.size>50
```

[1] Our benchmark shown that MongoDB can sustain 20k doc/sec for reading and 7k doc/sec for writing operations, respectively.

Turns out that such flexibility were not always clear to certain set of users, mostly those who were unfamiliar with DAS-QL syntax. Therefore a further attempt was made to build native support for keyword queries on top of DAS-QL, see [10].

DAS operates as an intelligent cache in front of CMS data-services. It stores results into two caches upon provided query. The raw-cache is used to store results from data-services *as is*, while the merge-cache stores aggregated records. The lifetime of the records is based on information provided by data-providers via HTTP headers. The records maintenance is done in a lazy fashion, i.e. upon new user query expired records are wiped out from the cache, while new ones come in. DAS server performs many operation in parallel, e.g. it sends concurrent HTTP requests to underlying data-services, processes and stores data using multiple threads, and runs multiple monitoring and pre-fetching daemons. The server runs on a single 8-core node with 24 GB of memory required for efficient MongoDB operations[2].

The discussed modular design, flexible QL and NoSQL storage, allows DAS to aggregate information from distributed data-services without imposing any requirements on them. DAS is able to deal with different security models, various APIs, data-formats and naming conventions. Right now it uses dozens of CMS data-services. Data are aggregated into JSON records based on common entity keys such that it is possible to see information from multiple data-services in a single record, e.g. run information comes from DBS, Condition DB and RunRegistry and represented as single JSON document listed information from three data-services. Daily load on DAS server is constanly growing and has about 10k queries/day with $\sim O(10M)$ records going in and out of DAS cache.

6. Conclusions

References

[1] The CMS Collaboration 2008 The CMS experiment at the CERN LHC *JINST* **3** *S08004*

[2] Knoblock J *et al.* 2005 LHC Computing Grid Technical Design Report *CERN-LHCC-2005-024*

[3] Gutsche O *et.al.* CMS Computing Operations during Run 1 *submitted to CHEP 2013*

[4] Egeland R, Wildish T and Metson S 2008 Data transfer infrastructure for CMS data taking, *XII Advanced Computing and Analysis Techniques in Physics Research (Erice, Italy: Proceedings of Science)*

[5] Giffels M, DBS paper, *submitted to CHEP 2013*

[6] Kuznetsov V, Evans D and Metson S, The CMS Data Aggregation System, *doi:10.1016/j.procs.2010.04.172*

[7] Bonacorsi D 2007 The CMS computing model *Nucl. Phys.* B *(Proc. Suppl.)* **172** 53-56 **is this a good reference or should we use the original CTDR???**

[8] http://www.mongodb.org/

[9] Model-view-controller architecture, see http://en.wikipedia.org/wiki/Model-view-controller

[10] Zemleris V and Kuznetsov V, Keyword Search over Data Service Integration for Accurate Results, *submitted to CHEP 2013*

[2] MongoDB relies on indexes fitted in RAM to provides its superior speed