FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Specification and implementation of a data warehousing system for the ATLAS' distributed data management system

Pedro Emanuel de Castro Faria Salgado

Report of Dissertation

Master in Informatics and Computing Engineering

Supervisor: Markus Elsing

Supervisor: José Luís Cabral Moura Borges

2008, July

Specification and implementation of a data warehousing system for the ATLAS' distributed data management system

Pedro Emanuel de Castro Faria Salgado

Report of Dissertation

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: Ademar Manuel Teixeira de Aguiar Doctor

External Examiner: Miguel Leitão Bignolas Mira da Silva Doctor

Internal Examiner: José Luís Cabral Moura Borges Doctor

15^{st} July, 2008

Abstract

The ATLAS collaboration has built a detector in the CERN's Large Hadron Collider (LHC). Each year, several petabytes of data will be retrieved from the LHC, by this detector, to be distributed over several countries and analyzed by more than 1700 physicists, from 159 institutes, spread over 37 countries.

DQ2 is the name of the distributed data management system for ATLAS which was implemented to overcome this tremendous challenge. Its goal is to be capable of transferring gigabytes of information per second, manage the petabytes of valuable information produced each year by the LHC, provide tools for the thousands of physicists to easily access this data and hide the inherent complexities of interacting with several computing grids and storage elements.

Started in 2004, DQ2 has already proven its capacity of transferring ATLAS data, at the required 1.2 gigabytes per second, and reliably manage all ATLAS' data.

The central catalogs is one of the most important components of DQ2 since its where the information needed to manage and transfer ATLAS' data is stored. The quantity of users interacting with DQ2, together with the high capacity of the computing centers at ATLAS' disposal, requirements to manage petabytes of data and sustain data transfer in the order of terabytes per day, makes the central catalogs play a vital role to effectively accomplish these goals.

On this report, we present some of the work done on the DQ2's central catalogs component, almost from its inception until today. We described some of the available solutions for reliable and scalable databases, discuss many of the problems, solutions and ideas we implemented to overcome memory problems, security issues, client-server interaction, performance, reliability and scalability.

We also present some research in the subject of knowledge discovery and data warehousing, as well as the steps accomplished in the implementation of data warehouse namely, the dimensional modeling process, important considerations taken in design of the relational model and the data staging process. This work is the base to integrate a reporting tool, in the central catalogs, to better understand the usage of system. This tool will be a valuable asset since it will help the development team identify priorities, possible misuse and security problems, predict the load of the system during certain ATLAS' activities, future hardware needs, among others.

Acknowledgements

We would like to thank Markus Elsing, the group leader of the ATLAS Distributed Computing group, for his support and encouragement regarding the pursuit of this master thesis; Kors Bos for his comments and diagrams, on the section regarding the ATLAS experiment data flows; professor José Luís Borges for his time, invaluable comments and support; the DQ2 team for their help during this project.

Pedro Emanuel de Castro Faria Salgado

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Abbreviations

API	Application Programming Interface
ATLAS	A Toroidal LHC ApparatuS
CERN	European Organization for Nuclear Research
DAO	Data Access Objects
DBA	DataBase Administrator
DDM	Distributed Data Management
DQ2	Don Quijote 2
DUID	Data set Unique IDentifier
ETL	Extract, Transform and Load
FTS	File Transfer Service
GSI	Grid Security Infrastructure
GUID	Global Unique IDentifier
HTTP	Hyper Text Transfer Protocol
HWM	High Water Mark
IOT	Index-Organized Table
IT	Information Technology
JSON	JavaScript Object Notation
LCG	LHC Computing Grid
LFC	Local File Catalog
LFN	Logical File Name
LHC	Large Hadron Collider
NDB	Network DataBase
OLAP	OnLine Analytical Processing
RAC	Oracle Real Application Clusters
RAM	Random Access Memory
SQL	Structured Query Language
SSL	Secure Socket Layer
TDR	Technical Design Report
UML	Unified Modeling Language
VUID	Version Unique IDentifier
XDR	External Data Representation
XML	eXtensible Markup Language
YAML	YAML Ain't a Markup Language

Chapter 1

Introduction

In this chapter, we present a brief introduction of the European Organization for Nuclear Research (CERN), the Large Hadron Collider (LHC), LHC Computing, the ATLAS experiment, the ATLAS Distributed Data Management group and the Don Quijote 2 (DQ2) system.

The work described in this report is focused on one of DQ2 components, called central catalogs, which is described in more detail on section 1.1.4.

1.1 Context

1.1.1 Quest to reproduce the Big Bang

The European Organization for Nuclear Research, also known as CERN, was founded in 1954 and it's one of the world's largest and respected scientific research centers [4, 5]. CERN main research area is fundamental physics: finding out what the Universe is made of (fundamental particles) and how it works (laws of Nature). In order to achieve this, the world's largest and complex scientific instruments, such as particle accelerators and detectors, were built.

Particle accelerators are used to boost beams of particles to high energies after which they collide against another beam of particles or with stationary targets. The detectors track and record the results of these collisions. The Large Hadron Collider (LHC) is one of CERN's particle accelerators. It was built inside a circular tunnel of 27km in circumference, buried around 50 to 175m underground, as shown on Figure 1.1, and it's located near the Swiss and French border near Geneva, Switzerland [6, 7].

This accelerator will help physicists study the fundamental particles by recreating the conditions just after the Big Bang, by colliding two beams at very high energy. These two beams of subatomic particles, also know as "hadrons", will be composed of either protons

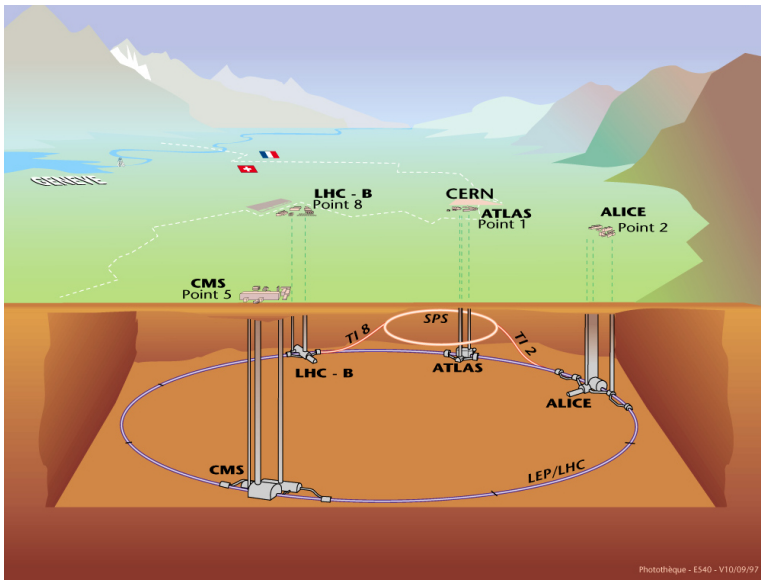

Figure 1.1: The LHC experiments (copyright CERN) [1].

or lead ions, which will move inside the accelerator in opposite directions, gaining energy with every lap, until they collide against each other.

A number of experiments have placed special detectors in the LHC to observe and record the outcome of these collisions which will be used by teams of physicists around the world for their research. ATLAS is one of the experiments which placed a special detector at the LHC [8, 9].

The detector is 44 meters long, by 25 meters high and 25 meters wide as it can be seen on Figure 1.2. It weights 7,000 tons and it's the largest volume particle detector ever built. Its enormous doughnut-shaped magnet system consists of eight 25m long superconducting magnet coils, arranged to form a cylinder around the beam pipe through the center of the detector. While in operation, the magnetic field will be contained within the central cylindrical space defined by the coils.

The goal of the experiment is to investigate a wide range of physics like extra dimensions, particles that could make up dark matter and the search for the Higgs boson[1]. In order to achieve this, the detector will record the particles' paths, energies and identities before, during and after the start of the collisions.

A group of 1700 scientists from 159 institutes in 37 countries work on the ATLAS experiment.

1.1.2 Computing and data storage infrastructure

Thousands of scientists around the world need to be able to access and analyze the 15 petabytes of data which will be produced in the LHC per year. In order to fulfill this chal-

[1]The Higgs boson, is a hypothetical elementary particle predicted by the Standard Model of particle physics.

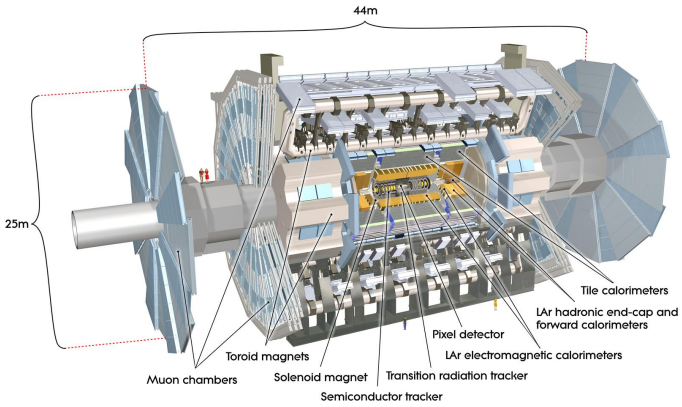

Figure 1.2: The ATLAS detector (copyright CERN) [2].

lenge, a large distributed computing and data storage infrastructure, called LHC Computing Grid (LCG), was put in place [10].

All data from the LHC experiments will be stored on tape at CERN as a primary backup. Afterwards, a large fraction of this data will be distributed around the world to a series of large computer centers.

For the ATLAS experiment and according to the ATLAS Computing Technical Design Report (TDR), the first event processing will also be done at CERN at a facility called Tier-0 [11, 3]. This data will be copied along with the raw data to other facilities, called Tier-1s, which will be spread around the world (Figure 1.3).

At a Tier-1, this second copy of the raw data will again be subject to other processing steps, as in the Tier-0, as shown in Figure 1.4.

The products of these new steps will be stored at the site and a portion of this reprocessed data and, in some cases, some raw data, will be moved to several other facilities, called Tier-2.

Here the data will be used for physics analysis research and for further calibrations, in order to tune the detector.

For example, in the search for the Higgs boson, 10 million events per day will be produced at the detector, amount which is already a filtered subset of the total number of events. From this huge number of events, only one event per week has the characteristics of the Higgs boson. Due to the imperfections of the detector itself, a lot of events may seem to have the properties of the Higgs event, therefore a lot of simulation is needed to produce these possibles scenarios, according to the characteristics and properties of the machine, at the time, when the results were produced. This will help physicists validate the results they will obtain on their analysis: is this a Higgs event or not?

According to Figure 1.5, in the Tier-2s, simulated data will be produced locally (PRODDISK) which, afterwards, will be sent to their corresponding Tier-1. This CPU intensive

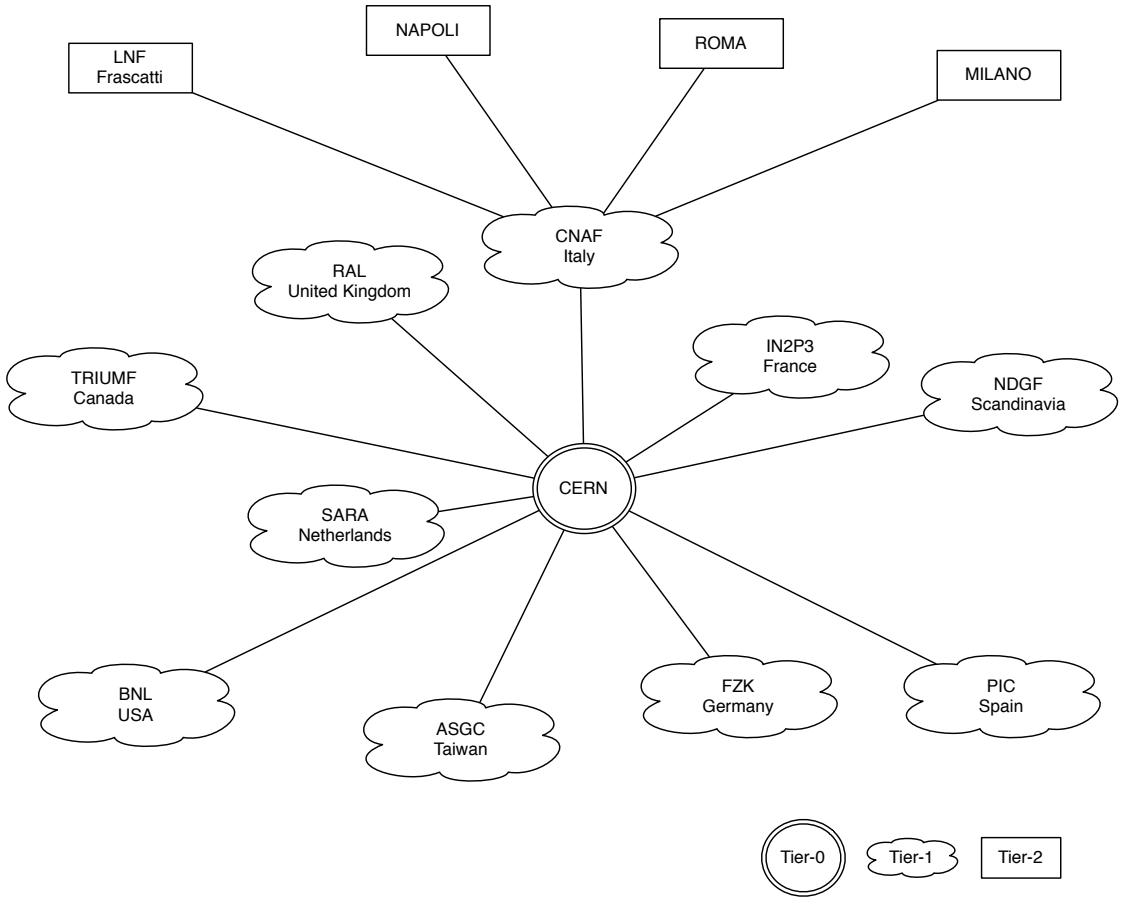

Figure 1.3: Example of how the tiers of ATLAS are organized.

step will be done in the Tier-2 facility and, due to an ATLAS requirement, it will not take more than 24 hours to finish. After being shipped to the Tier-1 (MCDISK and MCTAPE), this simulated data together with the processed real data will then be used by physics groups for analysis and will be stored in a well-defined storage space area (GROUPDISK). In the end, part of this data will be shipped to the Tier-2 for local users analysis (USERDISK).

As you can see, each Tier-1 facility not only needs a great computing power to process raw data and to cope with scheduled analysis by the various physics analysis groups but also needs to have a very large storage capacity to store raw, processed and physics group data, as well as simulated data sent by the Tier-2s. Tier-2 facilities will mainly have computing power and only enough storage capacity to have a disk buffer to send data to the Tier-1 and for group and user analysis.

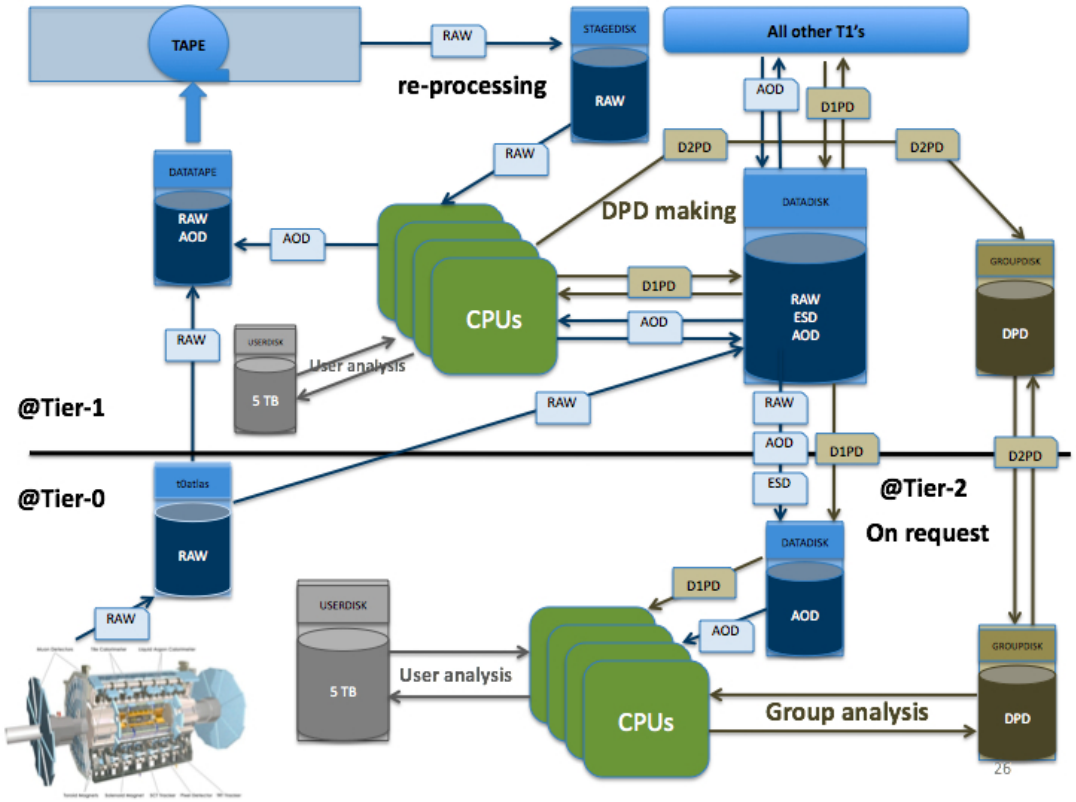

Figure 1.4: Flows of raw data [3].

1.1.3 Data management system

As described previously, the quantity and the size of data being generated, processed, stored and transfered is at a the petabyte-scale and of distributed nature which makes the task of managing such data a complex problem. The ATLAS Distributed Data Management group (DDM) [12] was created to build a scalable and reliable system as described on the ATLAS Computing TDR, [11, section 4.6 Distributed Data Management].

The DDM group was set to implement a system to manage ATLAS' data, that is distributed in several sites all around the world, to schedule data transfers, to schedule data deletion and to enable interactive user access [13]. Such data is stored in files and contain relevant information for research, in the area of particle physics. Among these files we may find certain sets which are intended to be used together, due to the properties of the data they contain. In order to effectively manage ATLAS' data, the system needs to identify and manage these groups of files or, what we call, data sets (appendix A).

In 2004, the DDM group deployed the first version of this system. Its name was Don Quijote 2 (DQ2) [14, 15, 12, 13, 11] and it was composed by five major components:

- a data set based bookkeeping or central catalogs.

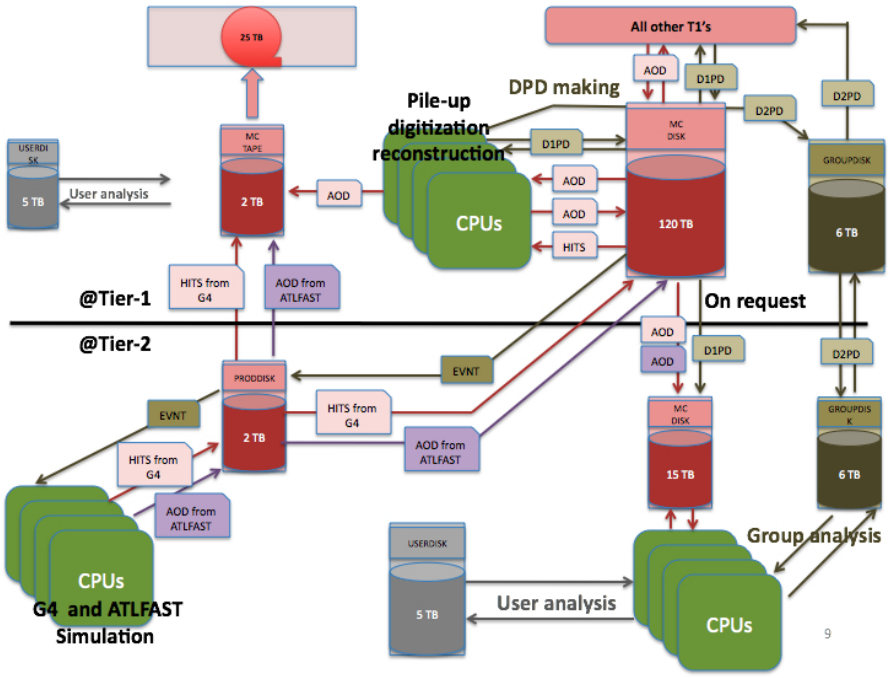

Figure 1.5: Data flows for simulation [3].

- site services to handle data transfers.
- consistency services to handle data consistency.
- end-user tools to give access to the data itself.
- a monitoring service to capture the state of the many data transfers.

Each of the components is described in the following sections 1.1.4, 1.1.5, 1.1.6, 1.1.6, 1.1.7, 1.1.8 and 1.1.9.

1.1.4 Central catalogs

The central catalogs [16, 17] consists on a set of web services with a database back-end. These web services provide, among others, the possibility for a user to:

- define a data set.
- register files in a data set.
- register versions on a data set.
- register replicas for a data set.
- request the transfer of data set into a site.

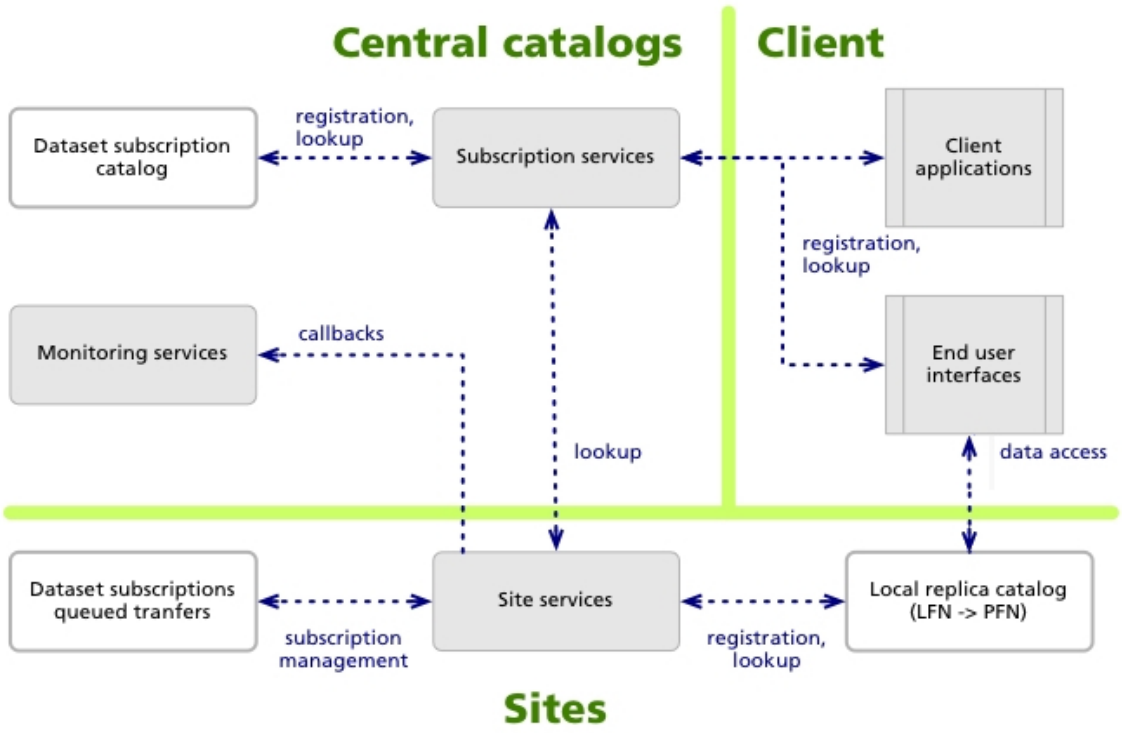

Figure 1.6: DQ2 components.

Data set is a key concept of the system, as specified in [11], therefore this is one of the most important components of the data management system since it stores the mappings between data sets and files.

Several iterations have been made in the architecture and implementation of this part of the system which are described, in some detail, on sections 3.1, 3.2, 3.3 and 3.4.

1.1.5 Site services

This component is responsible for the data movement and data registration. It's composed by several agents which behave according to a set of pre-defined rules, in an autonomous way, on behalf of a user and a site, to fulfill a data transfer request (subscription) that is designed in the central catalog.

This service is hosted on several machines and each machine has a set of agents which serve subscriptions for a configurable set of sites.

To have an overview of the site services' behaviour we will describe a simplified and optimal situation where a user wants to have a particular data set on a certain site, in order to do some analysis over it.

The user would make a transfer request by registering a subscription for a data set for a destination site into the central catalogs. The agent which serves the destination site, in a certain point in time, will poll the central catalogs for new subscriptions and pick up the

user request. According to some pre-defined workflow, rules and user given subscription options, the site services would:

- resolve the data set into files by querying the central catalogs.
- interact with a site external component called Local File Catalog (LFC) [18] to determine if the files are at the site.
- find the best sites to transfer this data set from.
- interact with other site's LFC to check if they have the missing files and what are their paths.
- group all missing files for which their source paths are know (for this and other requests) according to the existing network channels to maximize the throughput.
- register a transfer request into an external component of the destination site (FTS - File Transfer Service) [19] providing the source and destination paths.
- from time to time, FTS will be polled to verify which files have been transfered.
- register transfered files into the destination site's LFC to make the file available to all users and systems.

For more information, please consult the DDM review documentation [17].

1.1.6 Consistency services

These services provides the means by which regular data set consistency checks are done at a site, in order to verify the completeness of a data set or to discover the loss of files. This service also provides the possibility to delete data sets centrally, in all ATLAS sites.

1.1.7 Client

This component provides an application programming interface (API) to interact with the central catalogs.

They also provide some basic command-line tools to manipulate data sets, data set versions, files, data set replicas and data set subscriptions.

1.1.8 End-user tools

These tools are designed to help a physicist access and modify the data needed for his analysis. They use the DQ2 client tools but they extend their functionality by interacting with the storage elements to download, upload or search for data. This way physicists (the end-users):

- can use a single tool to interact with different storage elements.
- have access to local resident data.
- search existing data sets.
- upload files and register new data sets.
- access remote data by making a transfer request to DQ2 or by using Grid tools to copy the data to their local machine.

For more information, please consult the DDM review documentation [17].

1.1.9 Monitoring service

This component's goal is to provide to the users information related to their subscriptions, namely:

- present the status of ongoing transfers.
- determine the status of each site.
- store information regarding past transfers.

The monitoring information is sent using a set of callbacks associated with every subscription. These callbacks are triggered at certain points of the data transfer and registration process which update the current status of the subscription and provide the user feedback about his subscription.

The monitoring service has a web interface and command-line tools to query the status of a site, of any ongoing or past transfer, number of data sets transfered, amount of data transfered, among others. The user can also have a view by cloud, site or time period (Figure 1.7).

More information can be found in [17, 20].

1.2 Motivation and goals

Frequently, users and other systems change the way they interact with the DQ2 central catalogs, raising new unforeseen problems in terms of performance, memory usage, reliability and scalability. On this section, we cover goals set to overcome some of these problems.

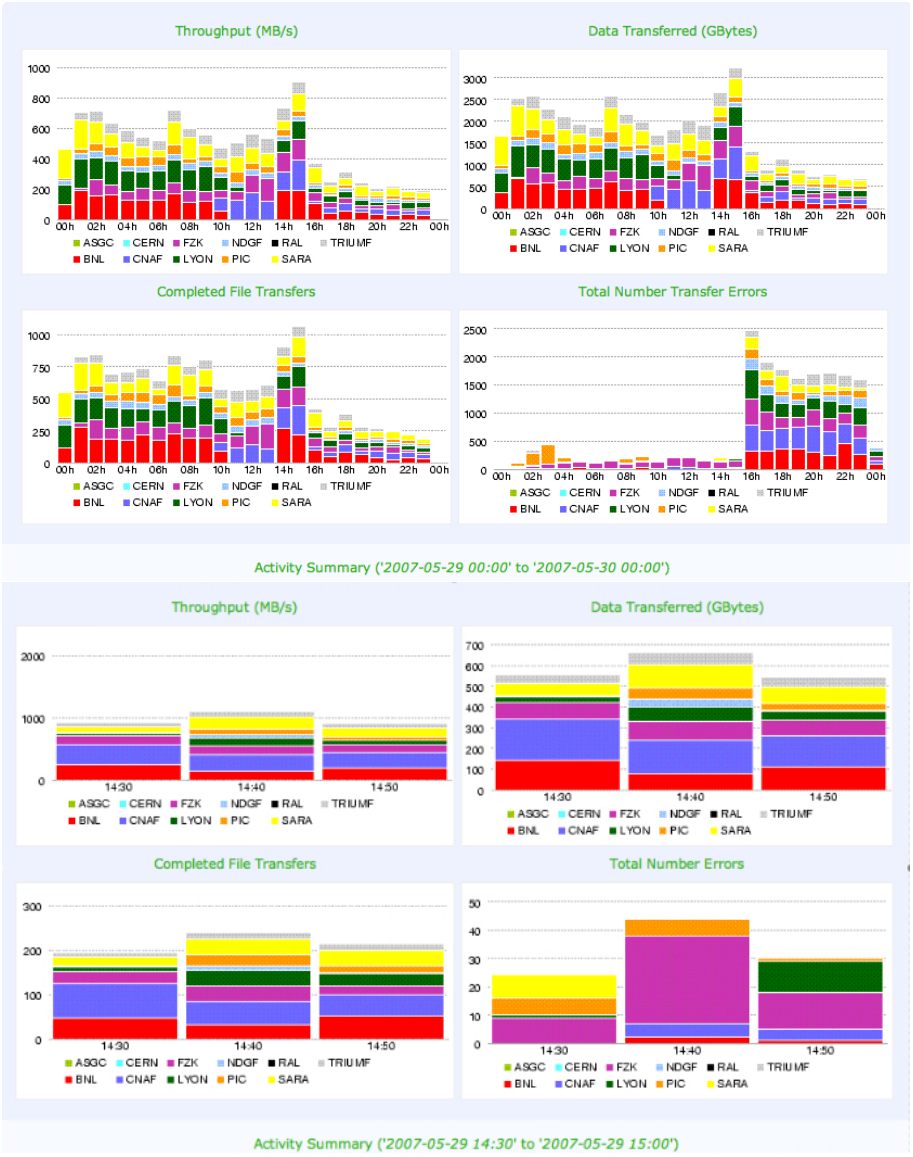

Figure 1.7: Screenshot of the ATLAS DDM monitoring, 29 May 2007.

1.2.1 Reliability

Before the project (section 3.1), one of our major problems, was maintaining a good quality level of service of our services.

Our database, from time to time, would go down and our service stop responding which always originated a large number of complaints.

An ATLAS MySQL expert, from outside our team, installed a script that ran on the database server machine to check the server status and restart it, if needed. Obviously, this could not be the final solution.

The database server technology we were using, was clearly not enough for the current or future requirements. We needed a new database software which could solve our current

stability problems and make our services more reliable and scalable, even when handling a large amount of requests.

The steps taken to accomplish this goal were:

1. make an overview of the currently available database solutions in the market.
2. evaluate these solutions and make a decision.
3. if needed, implement the central catalogs for this database.

The actions taken are described, in more detail, in section 3.2.1.

1.2.2 Containers

Right before the deployment of the DQ2 0.3, with the brand new database back-end, our services stopped working, very often, for no apparent reason. The database and web servers were running but there was no activity. After installing some brand-new database monitoring tools, we were able to conclude that there was a user blocking all database activity. A careful analysis of the queries lead to some data sets which had tens of thousands of files[2].

This made us realize that certain users were using regular data sets to be a copy of the contents of several other data sets and that they had use cases that supported this usage. In the ATLAS Computing TDR [11], this requirement had been expressed but, due to other priorities and lack of definition of how users should use containers, it wasn't implemented. Therefore, the users themselves scanned our catalogs with certain data set name patterns, retrieved their contents, retrieved the content of the jumbo data set and updated its files, accordingly.

These kind of queries slowed down the database server, usually taking 30 minutes to finish. This meant that the client had already timed out and had already issued the same query again making the situation even worse.

With the move to Oracle, this problem disappeared but there was a need to find a solution for this.

Some action were taken regarding this subject:

1. reach an agreement of what a container should be, contain and how it should be handled.
2. make a written specification of the use cases.
3. implement this requirement in our system.

Details about the specification and implementation, can be found in section 3.3.2.

[2]These data sets are also called jumbo data sets, see appendix B.1 for more details.

1.2.3 Memory problems

Related to the problem mentioned before about jumbo data sets, in the beginning of June 2008, it was identified a problem when several parallel requests would reach the central services to request a large amount of information from the database but this time it was a problem with our application.

This activity lead to a very rapid increase of the memory used by the web server and more and more frequent swaps of pages, done by the operating system, between memory and disk. The performance of the services on that machine would decrease and, as a consequence, requests would start being queued, waiting for their turn to be served, up to a point that, very often, clients would timeout and other ATLAS' activities started being affected, due the malfunction of our services.

To overcome this problem, a solution was implemented (section 3.4), in order to make the central catalogs, more reliable and robust.

1.2.4 Reporting system

As described previously on section 1.1.4, data set is a key concept in the DDM system, making the central catalogs one of the most critical components of them all. Without this service running, no new data sets can be defined nor no new transfers can be scheduled.

From past and present experience (sections 1.2.3, 1.2.2, 1.2.1), the team has often needed to quickly react to unexpected downtimes or performance degradation of our services. Some of the reasons were:

- deployment of a new site services version together with a high demand of data transfers, made the system go into a halt for all users.
- consistency services were checking the contents of multiple very large (jumbo) data sets which, due to a bad configuration of these services, didn't share the load between machines and made one of the central services machines to almost stop responding.
- invalid workflows from user scripts or other systems often cause multiple invalid requests which decrease performance and consume resources.
- a very wide, worldwide distributed community of users, not always proficient in software development or knowledgeable in DQ2, together with other systems, all of them using powerful computing facilities makes the system more prone to peaks of high level of requests and risk of unexpected downtimes.
- a certain combination of software libraries often made secure requests break and stop being fulfilled.

We also know that, with more powerful hardware being installed on the database servers and central services machines, potential problems or bad usage of the system and performance issues will pass unnoticed more easily.

Several steps have already been made in order to improve the robustness and reliability of the system but, for the time being, there isn't a way to confirm what our next challenges will be unless we have a better idea of who is using the system and how, what's the evolution of its usage and the impact of certain activities in our services. As real data taking is approaching, more and more users will need to start using the system to run their analysis, more tasks will be requested to run in several computing clusters more load will arrive into the system and more important will become the answers to these questions.

With this in mind, we thought of adding a reporting system, to the central catalogs, which could give answers to the following questions:

- who is using the system? from where?
- who is the most active user?
- what are the most requested operations? from where? from whom?
- do we have performance bottlenecks? on which calls?
- what are the most time consuming calls?
- which data sets were used on these calls?
- what is the impact of testing activities in terms of number of requests? and performance on the overall system? and what about real-data taking activities? and user activities?
- what is the contribution of functional tests in the number of requests? do you have performance degradation during these activities?
- what is the contribution of real data taking activities in the number of requests? do you have performance degradation during these activities?
- which of the physics groups is being more active?
- which site is producing more data sets?

The work done in this tool is described in more depth, on section 3.5.

1.3 Structure of the report

Besides this introduction, this report contains 3 chapters more.

In chapter 2, it's presented some topics regarding knowledge discovery and data warehousing, as well as, solutions for reliable and scalable databases and a distributed memory cache solution.

Chapter 3, describes the work done on the implementation of the central catalogs and on the new reporting system. The last chapter 4, presents the results and future work.

Chapter 2

Research

On this chapter, we present a brief study on the subjects of knowledge discovery and data warehousing, which are the base of a proposal for a system to analyze the central catalogs usage (section 3.5).

We also make an overview of the some databases products, like Oracle Real Application Clusters (section 2.3.1) and MySQL cluster (section 2.3.2), to implement reliable and scalable database services, which were taken into consideration during the development of the DQ2 central catalogs (section 3.2). A distributed cache solution (section 2.4) was also taken into account in order to reduce the database load of the central catalogs (section 4.2.4).

2.1 Knowledge discovery

On any organization, one of the most important assets is its information. To produce the best possible results, it needs to have the capacity to analyze this information so that interesting knowledge can be found which, as a consequence, may lead to the best possible decisions or opportunities.

In many cases, analyzing data can be a challenging task due to its huge volume, distributed nature, heterogenous sources or even lack of analysis tools. This often leads to the "data rich-information poor" situation where the amount and numerous data repositories exceed the human ability for comprehension without powerful tools and important decisions are made by the decision maker through intuition.

For this purpose, data mining tools and techniques were conceived, in order to enable the decision maker to perform data analysis, discover important patterns and make a significant contribution into the organization's business strategy, knowledge base or research.

According to [21], data mining is the process of discovering interesting knowledge from large amounts of data or, in other words, the process of transforming data into "golden nuggets" of knowledge.

This process is composed of an iterative sequence of steps which are [21]:

1. data cleaning (removal of noise and inconsistencies).
2. data integration (combination of multiple data sources).
3. data selection (retrieval of relevant data for analysis).
4. data transformation (or consolidation of data into a more adequate form for mining).
5. pattern evaluation (identification of interesting patterns).
6. data presentation (visualization and representation techniques used to present the mined knowledge to the user).

The implementation of a typical data mining system usually involves the following components [21]:

- information repositories: set of databases, data warehouse or other kinds of information repositories where the data to be mined is stored.
- data mining repository: database or data warehouse, usually located in one place, where the relevant data is fetched into, based on the user's data mining request.
- domain knowledge: information used to evaluate the interestingness of resulting patterns and guide the search.
- data mining engine: set of functional modules to characterize and analyze data.
- pattern evaluation: module responsible for the application of interestingness measures over the data mining engine to focus the search towards interesting patterns.
- user interface: module by which the user is able to interact with the data mining system to specify a data mining query, change interestingness rules to focus the search, browse the data mining repository schema or any of its data structures, evaluate mined patterns and visualize the patterns in multiple forms.

In the scope of the work done at CERN, only the data warehouse itself will be described more throughly, on the next section.

2.2 Data warehousing

As mentioned previously, a data mining system, typically, has a centralized repository where the data is fetched to be mined. One of the possible implementations of this repository is the usage of a data repository architecture called, data warehouse. The definition given by [21] states that a data warehouse is a repository of information built from multiple heterogeneous data sources, typically, stored in a single site, with a unique schema, for knowledge discovery purposes. The construction of any data warehouse implies a process of data cleaning, data integration, data transformation, data loading and periodical refreshing of the information [21].

Data in a data warehouse is typically summarized, organized around major subjects and is stored with the purpose to provide historical information. To model such a repository, usually, a multidimensional database structure is used, where a dimension corresponds to one or more attributes in the schema and each cell stores some kind of aggregation measure such as counts or sums.

As part of the project this report refers to, a data warehouse was specified, as described in section 3.5.

2.3 Reliable and scalable databases

In this section, we describe some solutions, provided by Oracle[1] and MySQL[2], that were taken in consideration for reliable and scalable databases (section 3.2.1).

2.3.1 Oracle real application clusters

Oracle's Real Application Clusters (RAC) [22, 23] provides the support for deploying a single database across a cluster of servers (cluster database). This database has a shared cache architecture [24] that, according to Oracle RAC datasheet [23], overcomes the limitations of the traditional shared-nothing and shared-disk approaches. This solution, has the ability to be fault-tolerant from hardware failure or planned outages and it's Oracle's best solution, in terms of availability and scalability.

When a node in the cluster fails or is shut down for maintenance, the database continues to run on the remaining nodes. In fact, the user applications are not affected since the single points of failure are removed.

In case there is the need for more processing power, a new server can simply be added to the cluster without requiring a shutdown. Also, a cluster can be built from standardized commodity-priced machines, storage and network components which can help keep the costs low.

[1] http://www.oracle.com/
[2] http://www.mysql.com/

2.3.2 MySQL cluster

MySQL Cluster [25, 26] is a product from MySQL AB designed to provide a fault-tolerant and reliable database.

This solution is based on a distributed architecture which can spawn through multiple machines or regions, to ensure continuous availability in case of node or network failure. It has an in-memory clustered storage engine called NDB, which integrates with the standard MySQL server.

The MySQL Cluster [25] product consists on three types of nodes (Figure 2.1):

1. storage or data nodes where all data is stored and replicated between these nodes.

2. management server nodes whose goal is to manage the configuration of the cluster.

3. MySQL Server nodes provide access to the clustered data nodes through a standard SQL interface.

Typically, on this architecture there is only one management node which is only used at startup and cluster reconfiguration. After being started, the storage nodes and the MySQL servers can operate without any management nodes.

A MySQL Server, in this type of solution, is connected to all storage nodes and multiple MySQL servers can be used in the same cluster, as shown on Figure 2.1. A transaction executed by a MySQL server is handled by all storage nodes which means that the changes will immediately be visible to all other MySQL servers, data is synchronously replicated into multiple nodes therefore, in case of failure, there is always another node storing the same information, leading to very low fail-over times.

This product is based on a shared-nothing architecture, where each storage node has its own disk and memory storage, although the option to share disk and memory exist when running several storage nodes on the same computer. Since there is no single point of failure, if any of the nodes goes down there won't be any loss of data nor any downtime of the applications using the database.

The usage of the MySQL servers to connect to the cluster gives, the developer and database administrators, a standard SQL interface making it easy to achieve high availability without requiring any low level programming. Also, the NDB storage engine gives the ability, to the developer, to abstract the way the data is stored physically, how data is replicated or partitioned and how automatic failover is achieved. This makes it possible, in case of failures, to dynamically reconfigure the cluster without any intervention on the application program.

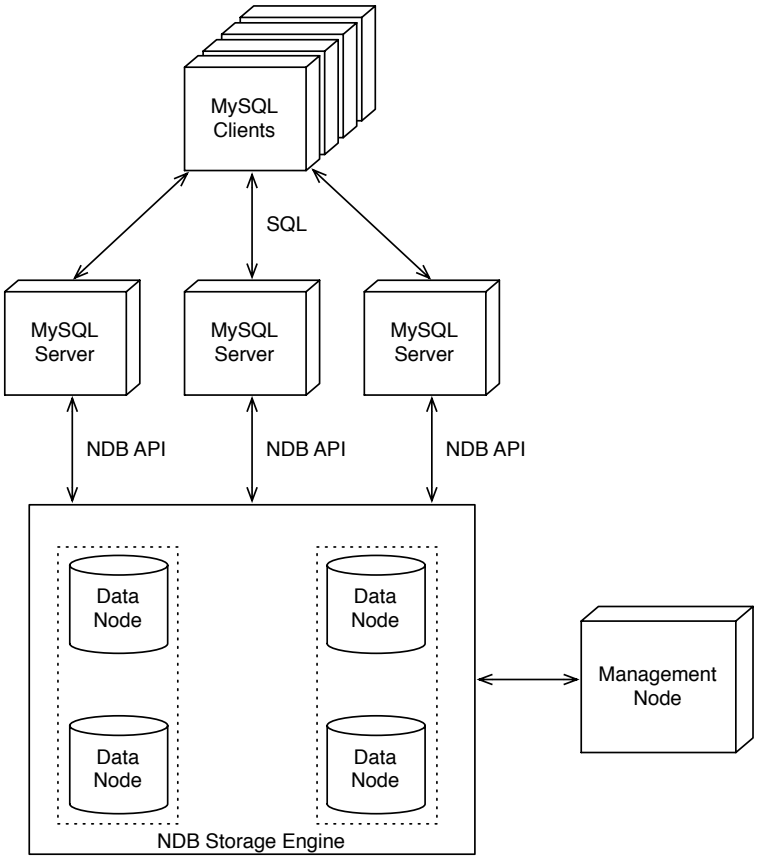

Figure 2.1: Typical architecture with MySQL Cluster.

2.4 Distributed memory object caching system

memcached [27] is a generic distributed memory object caching system, with very high-performance whose initial goal was to speed up dynamic web applications by reducing the load and resource consumption at the database level.

This software in particularly, is very popular among the Web 2.0 developer community and is being used in very popular web sites like Facebook[3], LiveJournal, WikiPedia and Slashdot.

The cache itself implements a least-recently used algorithm to remove less used elements from the cache when memory is needed which means that old information will be recycled without the need of any programming effort. A timeout can also be specified to automatically recycle the information.

If a machine hosting a cache has a failure it will only lead to more database accesses in the beginning but as other available machines fill-up their cache, the database load will reduce again.

[3]Facebook has a short description and a statement regarding the performance impact of memcached in http://developers.facebook.com/opensource.php.

2.5 Summary

On this chapter, we presented a brief study regarding knowledge discovery and data warehousing. We have given an overview of some database products to implement reliable and scalable database services.

We have also described a distributed cache software, its possible usage to reduce the load on databases.

Chapter 3

Implementation

In this section, we will describe the work done on the central catalogs component (sections 3.1, 3.2, 3.3 and 3.4) and a reporting tool system and data set usage, in particular, the specification and early steps of the development of a data warehouse (section 3.5).

3.1 DQ2 0.2

In this section, we discuss the work done on the DQ2 system at its very early steps of development.

3.1.1 Improving the client-server communication

One of the first important decisions that had to be made was to have a good client-server protocol.

In the DQ2 application, the user has a well-defined interface over which several clients are used to interact with the server through HTTP, as it can be seen on Figure 3.1.

To contact the server, the client can use insecure calls for read requests but if a request is made to change or write something in our system, then the client has to use the Grid Security Infrastructure (GSI), in order for his request to be authenticated and authorized by the central catalogs.

The basic client-server interaction through HTTP and GSI had already been implemented, by another person within the team. Still, the protocol wasn't meeting our expectations. First because it didn't benefit from the HTTP protocol to differentiate between errors responses from good ones and a different parsing of the responses had to be done for each different type of request we had in our system.

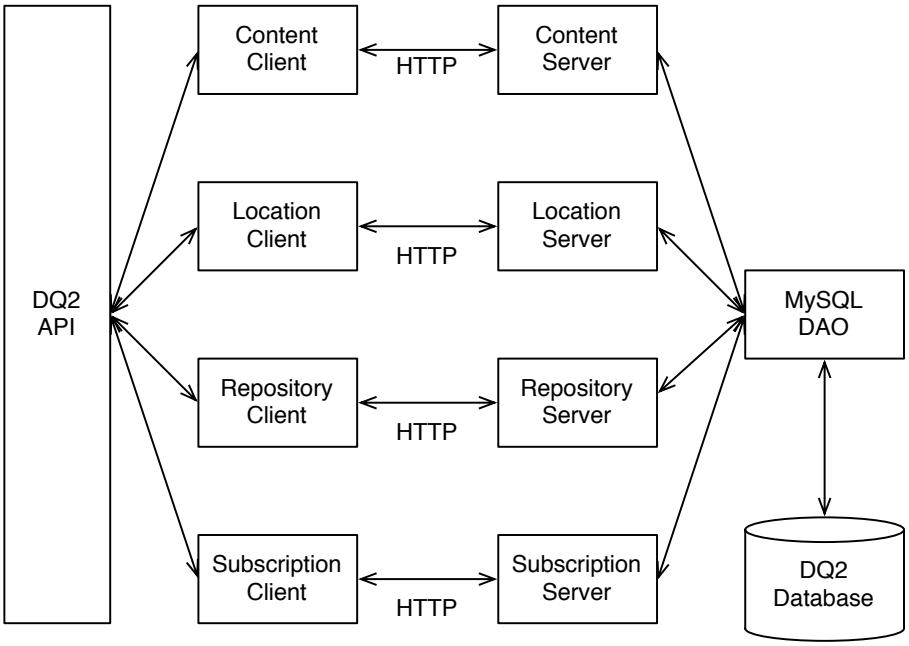

Figure 3.1: Don Quijote 2 architecture (v0.2).

After a careful analysis of the situation, the following measures were proposed:

- errors would be handled through the usage of *Python* exceptions.
- the output of a client would always be a *Python* structure.
- the HTTP client and server would have a well-established communication protocol.
- parsing, discovery of type of message and construction of the *Python* structure would be moved completely into the HTTP client.
- error messages would be serialized into *Python* exception objects, using *Python* 's pickle [28] library to be raised by the HTTP client.
- usage of the HTTP status code to determine the type of message sent by the server.
- refactoring of some aspects related to HTTP handling from the clients into a base client class (DQClient).

This change on the exception handling, the clear separation of roles and encapsulation of tasks in very well-defined components made it easier to:

- build test cases and effectively add a validation step before releasing, which had a big impact on the reliability of the application itself.
- maintain the code and more quickly spot the source of problems.

- implement new remote procedure calls since the new architecture promoted the reuse of components.

At a later stage, the client-server protocol was changed so that all data sent by the server started to be formatted according to the *Python* 's "official" string representation (__repr__ [29]). As a consequence, all server messages stopped being parsed and through the use of *Python* 's eval built-in function [29] we managed to rebuild the data structure as soon as the response arrived from the server.

This architecture and protocol has been in use, with success, for more than two years.

Some ideas have already risen to improve the client-server interaction and they can be seen on section 4.2.2, page 41.

3.2 DQ2 0.3

In this section, we talk about the evaluation done on several database products, implementation, migration and tuning of the final solution using the new database back-end. We also provide a brief comparison between this version and the previous one.

3.2.1 Database evaluation

One of our established goals for this work was to improve the stability of the central catalogs (section 1.2.1). This task required, as a first step, an evaluation of the available solutions. It was clear we should only consider products where, CERN or the ATLAS experiment, would already have some expertise, so that, in order to maintain the services, the DQ2 development team could, at least, share responsibility. Since CERN and Oracle collaborate very closely in many areas and within the ATLAS experiment there were a few MySQL experts, only the products from these companies were taken into account.

Our database used a MySQL server product, therefore the first approach was to see if they had other products that could better fit our needs.

At the time, there was a product called MySQL Cluster [25] which, although it wasn't still a mature product, seemed to provide what we needed: high availability, performance and scalability (see section 2.3.2 for details).

With the help of MySQL experts, we did a setup of a MySQL cluster using two data nodes, installed our schema and used our development environment to use this database. After some initial problems, the setup was ready and working and, in no time, we managed to use this setup without changing anything in our software. Still, at the time, this product had an important limitation which we only realized later: the whole tables needed to fit in memory. The machines only had 4 gigabytes of random-acess memory (RAM) but even if it was larger this could never be the final solution. We expected to store much more

data in our databases. We didn't want an intermediate solution, therefore we had to drop this option.

The only alternative left was Oracle. After some internal discussion, we realized that several other ATLAS projects were already using an ATLAS' dedicated Oracle Real Application Clusters (RAC) [22] hosted and supported by CERN IT department and had another cluster just for development. Plus, ATLAS had two full-time Oracle database administrators. Oracle has a very high reputation still some research was done, which can be seen on section 2.3.1, and showed that this product would fit our needs and give us even more guarantees than the MySQL Cluster solution. The MySQL Cluster was still a beta version and had the RAM memory limit, while Oracle's solution for network-clustered database was already present prior to Oracle 9i and had a substantial improvement, after this version. Plus, we could drop our support to the database services and gain more time for development.

The next step was to find a good Python package to interact with Oracle ([30, cx_Oracle]) and direct our development efforts for this database.

3.2.2 Implementation

In January 2007, we started developing for Oracle but we could not stop maintaining the MySQL version so we added a factory class [31] per catalog to abstract the database implementation. This made it possible to use the test cases used for the MySQL flavour to be reused by this new implementation. This speeded up the development, test and validation effort plus it guaranteed backward compatibility.

3.2.3 Migration

Middle May 2007, all of our application had been successfully developed but one important step was still missing: a full migration of the database.

On this new version of the DQ2 central catalogs, a new requirement was implemented. A file produced in ATLAS with a certain logical name (LFN) could only map to one single global unique identifier (GUID). This caused several problems and after the scripts were ready and doing incremental migration, they still needed several iterations to cope and try to fix these errors.

In the end, not all the cases were solved. A list of problematic data sets was published and users had some time to react. The migration proceeded without anyone replying back. Many months after, some problematic and badly registered data sets were needed and their files were recovered and their information corrected when needed.

3.2.4 Comparison

On some benchmarks we did, due to reasons we only discovered afterwards (section 3.2.5), we could not see any performance improvements from the move between MySQL and Oracle except for the case of jumbo data sets, where Oracle not only would continue to serve other requests without any problems, but it was actually 33% faster than MySQL.

The reliability problems of the system were successfully overcome (section 1.2.1). The central catalogs didn't slowed down or stop while listing jumbo data sets and its response was faster, we the DQ2 development team stopped doing database support and our database services started being maintained by the CERN IT department.

3.2.5 Tune the services

After the deployment of this version into production, it became clear that there was a big gap between the development and production setup that we had which had not be taken into account. Also, the performance was still slower and this needed to be improved.

After a careful analysis, we realized that establishing a connection in Oracle was much more expensive than in MySQL, therefore a pool of connections was mandatory.

Since the architecture of the system was based on data access object (DAO), as shown in Figure 3.2, we managed to quickly add the connection pool without changing anything on our main application.

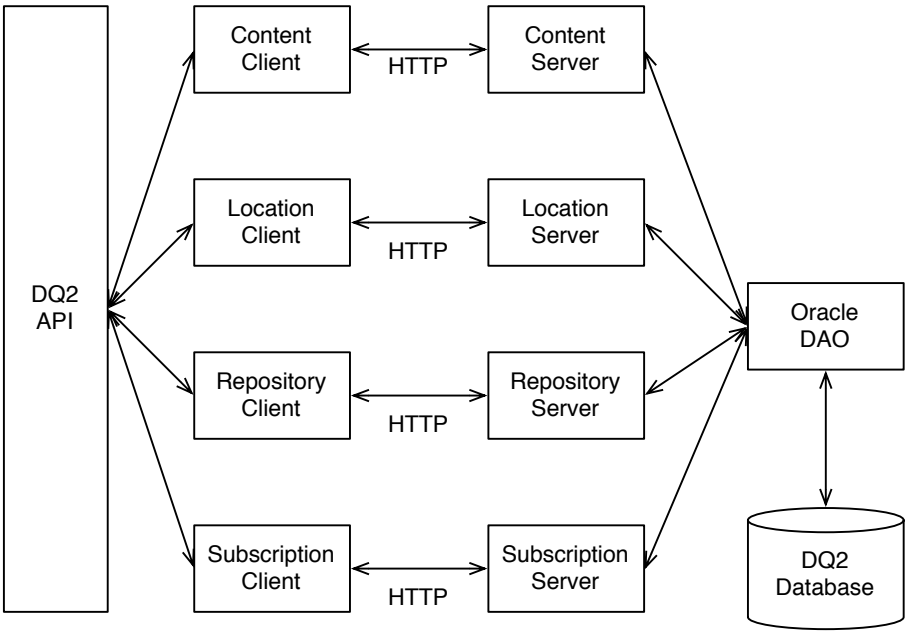

Figure 3.2: Don Quijote 2 architecture (v0.3).

The final tuning step had to do with thousands of short connections being done by our application, which was reported by CERN IT with a request for action on our side. After

a careful analysis of the apache server status, we realized that a new process was being created every time the number of requests increased above a certain value and would be killed by the web server when this value decreased. In order to solve this, we had to do some research and have a better understanding of how the apache web server handled the processes and threads creation, plus the configuration parameters associated to them. In the end, we changed the configuration parameters and supposedly fixed the situation.

Later on, we spotted that the short connections problem was not as serious as it was reported and that the CERN's reporting tool, actually, had a bug which passed unnoticed to all database administrators and other ATLAS application owners, up to the moment we firmly stated, backing up on our acquired knowledge and experience with apache, that the numbers were not being correctly calculated.

3.3 DQ2 1.0

In this version of the central catalogs component, we introduced the container concept, as referred in section 1.2.2, together with some schema changes improvements and a new architecture.

Sections 3.3.1 and 3.3.3 describe the central catalogs database schema and architecture, since some of the future work (section 4.2) will be focused on these topics.

3.3.1 Schema changes

On this iteration of the central catalogs schema, we took advantage of the increased knowledge and experience with Oracle, acquired during the previous year, with DQ2 0.3 (section 3.2).

The new database schema is presented on Figure 3.3, and the main motivations for these changes were:

- usage of raw field types in the data set unique identifier which would reduce the size of the database in terms of disk space (and also have an impact on the data warehouse size, see section 3.5.3)
- store the complete set of files per version (version 1 has files A and B; version 2 has files A and C), instead of the storing only the file changes between versions (version 1 has files A and B; version 2 file B was deleted and file C was added).
- primary key change on the data set versions table (t_versions) from a version unique identifier (vuid) to a data set unique identifier and version number (duid+version).
- index organized tables (IOT) on the data set versions table to make versions concerning the same data set to be stored on the same or a close data block.

- file table was changed to IOT with a primary key on the logical file name, so that similar logical file names would be in on the same or nearby data blocks.

3.3.2 Containers

As referred in section 1.2.2, containers were already defined in the ATLAS Computing TDR but this feature was missing from our system. Therefore, users had implemented their own tools to overcome this problem which caused a lot of problems into our system.

For the development of this concept, we took a more formal approach, trying to first write the requirements and reach an agreement with the users of what a container was, what type of data sets it could contain and how they should behave, before actually starting to implement anything.

From October 2007 to February 2008, a lot of work was put to retrieve, document and reach a consensus about the requirements for data set containers.

Since ATLAS is a very wide collaboration and to make it easier for anyone to make his own contributions, this specification was done using wiki[1] pages. The final work can be found in [32].

This process, although more time-consuming since clarifying concepts, constraints and define attributes among different people is not always an easy task, made a big difference in the implementation process. The advantage of having the use cases before hand, was that we could develop the tests more easily and always have a clear idea of what was still missing, how much we had progressed and what was still to do. More, once the tests we implemented, we didn't have to change them anymore, which was not the case before.

The container catalog was deployed as part of the DQ2 1.0 release, in the end of April 2008.

3.3.3 Architecture changes

In the beginning of 2008, it was decided that the container, content, repository catalog endpoints would be merged.

A new architecture for the system was conceived which allowed for each catalog to start sharing the same database schema and some portions of the code. The new architecture can be seen on Figure 3.4. The previous architecture can be seen on Figure 3.2, page 25.

Since the container catalog was still under development, it was already built under this new architecture. Still, we could not phase-out the previous clients because this would originate a service disruption. Many users and systems have their own clients installed

[1] Brief information about wikis and the usage we give them, can be found in appendixes C.1 and C.2.

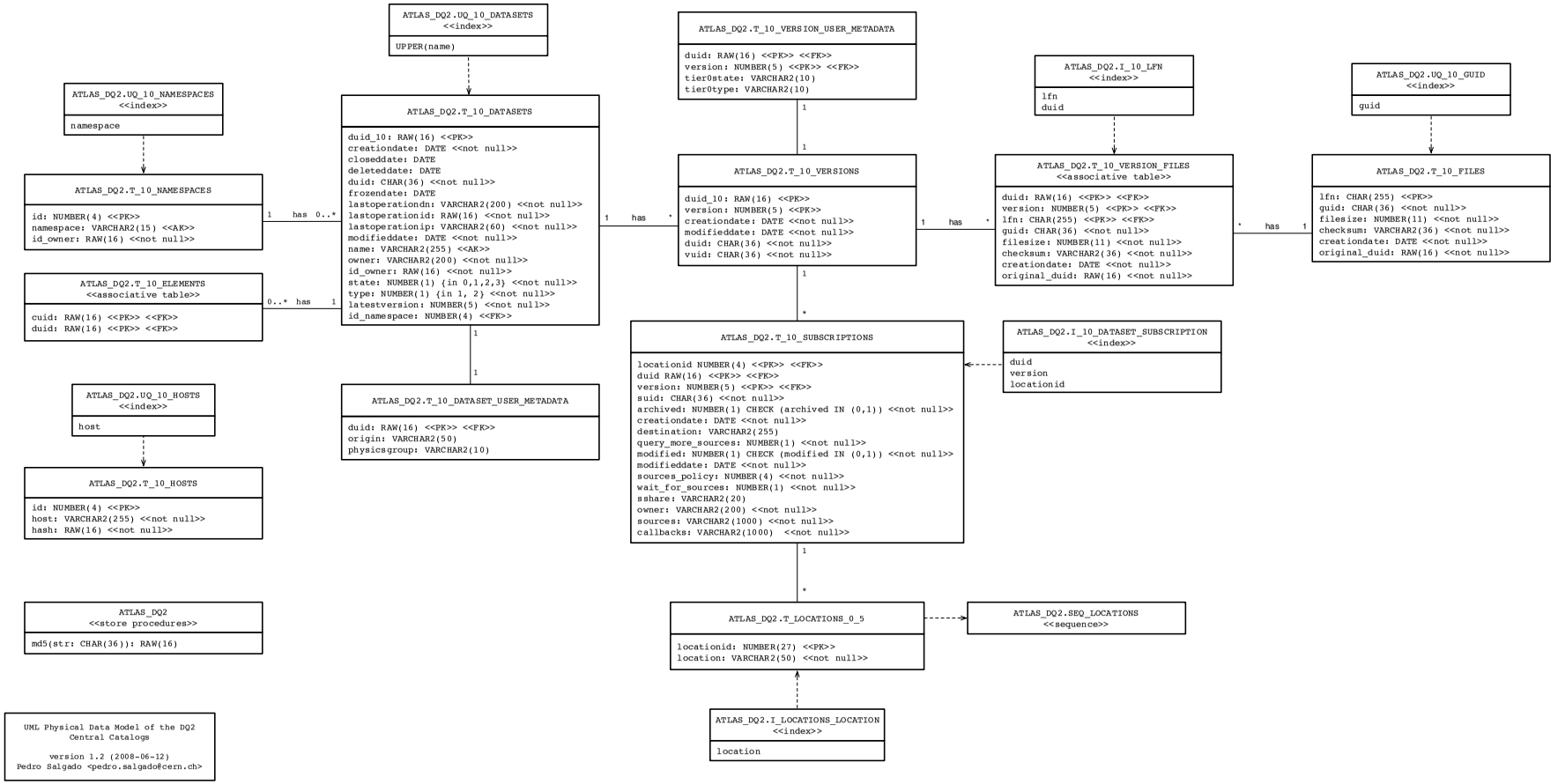

Figure 3.3: UML Physical Data Model of the DQ2 central catalogs (v1.2, June 2008).

locally which means a transition period needed to be foreseen where both the 0.3 and the 1.0 clients could be used together.

As mentioned in 3.3.1, we wanted to introduce several database schema changes, which obliged a new implementation of the old endpoints. This meant a double implementation, validation and deployment of the system which, due to the time constraints that we had, was not feasible.

So, we took an intermediate alternative:

- the container catalog was implemented on the new endpoint, as scheduled.
- the whole data access object (DAO) [31] layer would be implemented as foreseen.
- usage of two session façades [31]; one to be backward compatible with 0.3 clients and another for the next version of the catalogs (section 4.2.4).
- change the old endpoints to start using the data access layer.
- delay the deployment of the new clients.

This work was finished and successfully deployed on the 26th April and the final architecture can be seen on Figure 3.4.

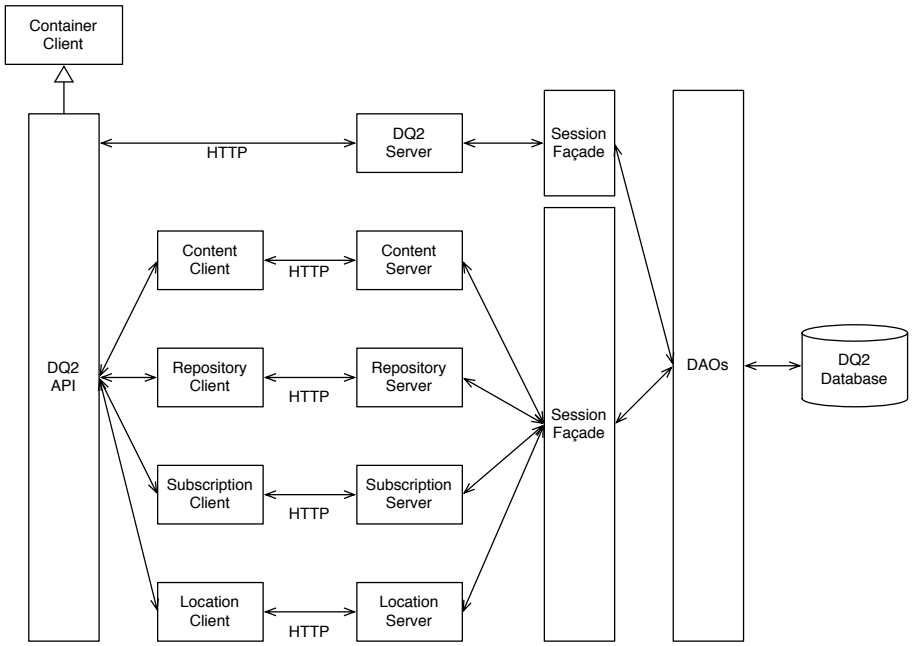

Figure 3.4: Don Quijote 2 architecture (v1.0).

3.4 DQ2 1.1

As described in section 1.2.3, when several parallel requests reached the central services, requesting a large amount of information from the database, the memory used by our application would increase, up to a point where the operating system would start *swapping*[2] and the performance of the services would start to degrade.

To better understand the origin of the problem, the architecture of the web services was based on a 3-tier model (section 3.3.3). A layer would be responsible for interaction to the database, another one would implement the business logic and finally, the top layer, would be responsible for the presentation. To fulfill any request, all of the information was retrieved from the database into memory, processed and only then shipped into the client. In this process, there were two main reasons for the memory problem:

1. when retrieving very large amounts of information, to be processed by simultaneous requests, could easily lead to memory shortage.

2. right before shipping the result to the client, the web service transformed the object one last time to comply with the established client-server protocol which would produce a second in-memory copy of the output.

To solve the first problem, certain objects that interact with the database were changed, in order to return iterators that would load the information directly from the database.

For the second one, on the operations where we needed to list large amounts of information to the user we merged the presentation and business logic layer. This way we could send the data has soon as it arrived from the database, process it and ship it to the client immediately.

In June, a new version of the central catalogs was deployed which not only fixed the mentioned problem, but also added new functionality to the existing system, namely, new data set meta data attributes and minor schema improvements.

3.5 System and data set usage

As described on section 1.2, in order to have a better overview of the usage of the central catalogs, our goal was to specify and implement a reporting tool, which could provide us information regarding the usage of this component and an overview of its performance.

We had considered using web server log analyzers but a brief look through some of the most popular software available for this, showed that they were, usually, web page centered and didn't take into account HTTP headers or parameters sent in the request. Since we only use one endpoint for all requests, it would clearly not be enough for our

[2] Swapping is a technique, used by the operation system, to replace pages or segments of data in memory into disk, and vice-versa, in order to manipulate data which is larger than the available memory.

needs. Moreover, usually the statistics retrieved from the web server log analyzers only present static information, in the sense that the user cannot look deeper into a certain set of data to see the distribution of other attributes for the specified set.

Due to its features of multi-dimensional perspectives over the same data, the usage of a data warehouse (section 2.2) was chosen, to store the information for this tool.

3.5.1 Design of the dimensional model

The implementation of a data warehouse obliges the definition of a dimensional model. According to [33], a dimensional design process should have the following steps:

- select the business process.
- declare the grain.
- choose the dimensions.
- identify the facts.

In our case, we needed to better understand what requests were being made in the central catalogs, how many and how much time they took to be fulfilled; therefore the process we needed to model was the central catalogs' usage. The next step was to define the level of granularity that we needed. [33] states that a data warehouse always demands data to be expressed at the lowest possible grain of each dimension, in order for the model to be as extensible as possible and minimize the impact of the changes if a new dimension is to be added.

Based on this, our grain will be a central catalog request. After having stated the grain, we can easily deduct the possible dimensions:

- date (when the request was finished).
- time (when the request was finished).
- user (who made the request).
- host (where the request came from and which machine it arrived into).
- operation (requested by the user).
- status (the HTTP status of the message, to distinguish good responses from errors).

The fact we need to measure is the time taken to fulfill the request. Since we cannot guarantee that the time resolution of the log file will be enough to avoid having duplicate entries in the fact table, we added a new field to store the number of requests, as well.

3.5.2 Dimension attributes

[33] describes the dimension table attributes as having an important role in the data warehouse because they are the source of almost all interesting constraints and report labels. Therefore, they are the key to make the data warehouse usable and understandable. All effort done in filling up the attributes, insuring the quality of their values and making sure they express business terminology, the better and more powerful the data warehouse will be.

One of the suggestions made by [33] is to use the dimension attributes with meaningful values (like yes/no) instead of cryptic ones (like y/n). This way the same values can be displayed consistently regardless of the users' reporting environment. Taking this into consideration, we have added attributes into the dimensions, as shown on Table 3.1.

Table 3.1: Dimension attributes.

Date dimension	
Date Key	surrogate values, 1-N
Full Date Description	Friday, 27 June 2008
Day of Week	monday, tuesday, ..., sunday
Day Number in Calendar Month	1, 2, 3... 28, 29, 30, 31
Month in Calendar	january, ..., december
Month Number in Calendar	1, 2, 3... 12
Year in Calendar	2008, 2009, ...
SQL Date Stamp	oracle representation of the date

Time dimension	
Time Key	surrogate values, 1-N
Hour	0, ..., 11
Minutes	0, ..., 59
Seconds	0, ..., 59

Operation dimension	
Operation Key	surrogate values, 1-N
Name	list files in a data set, ..., list data sets
HTTP Method	GET or POST
HTTP Type	secure or insecure

continued on next page

continued from previous page	
User dimension	
User Key	surrogate values, 1-N
Hash of Distinguished Name	hash of the user's grid certificate distinguished name.

Host dimension	
Host Key	surrogate values, 1-N
Host Name	the name of the machine

Status dimension	
Status Key	surrogate values, 1-N
Status	success or error
HTTP Status Code	200, ..., 500

Data set dimension	
Data Set Key	surrogate values, 1-N
Data Set Identifier	data set identifier in binary format
Creation Date	date when the data set was created
Closed Date	date when the data set was closed
Deleted Date	date when the data set was deleted
Frozen Date	date when the data set was frozen
Modified Date	date when the data set was modified
Name	the name of the data set
Owner	name of the owner of the data set
State	open, closed, frozen or archived
Type	data set, container or transient data set
Last Operation User	name of the user who last changed the data set
Last Operation Host	name of machine from where the last change was requested

3.5.3 Relational model

While translating the dimension into the relational database schema several points were taken in consideration [33]:

- avoid null keys in the facts table.
- dimension tables should remain as flat tables.

- joins between dimension and fact tables should be based on meaningless surrogate keys[3].
- size of the surrogate keys.
- what information we had available for each dimension.

Some numbers were taken into account in order to choose the type of the primary keys of the dimension, since the size of them has an impact on the size of the fact table and on the overall disk space usage of the data warehouse. The number of entries in our web server access log files showed that we had about 5.5 million requests per day in all machines. This means that in a year we would get 1.825 thousand million requests. If we saved 1 byte in any primary key, of any dimension, we would be saving 1.825 gigabytes of disk space per year.

For the date dimension, we had to take into consideration that the experiment will last a minimum of five years. We will need to have, at least, two years of historical information on the data warehouse, in order to compare the evolution of the number of requests as sites get more and more machines, storage space and the detector starts sending more and more data. This would mean a minimum of 730 days. The types we analyzed were: date, number and raw (or binary). The date type field, in Oracle, always occupies 7 bytes and we could store all the dates we wanted. For the number type, we have first to choose the number of digits. In our case we needed three digits ($999 > 730$ days). Since Oracle always stores the shortest byte representation of the number some tests were needed to compare the size of this type with the size of the date field. The result was, more or less, 2.8 bytes per key (the code used for this analysis can be seen in the appendix C.3). The last possibility was the raw data field which must be defined in the order of bytes. One byte would not be enough since it only gave 256 ($2^8 = 256$) days therefore, we calculated the possibilities with 2 bytes ($2^{2\times 8} = 65,536$) and with this option we could use up to 179 years and use less disk space.

To populate the dimension we would need to generate the primary keys ourselves. The option was made to use a sequence but since this gives a number, a small overhead would be needed to transform it into a hexadecimal number and then into a byte[4] to effectively use the raw type we chose. Since populating the date dimension will be a one time operation, it was not a problem.

For the time dimension, a similar exercise was done. We needed 86,400 ($24 \times 60 \times 60 = hours \times minutes \times seconds$) different keys and after calculating the size of using number, timestamp or raw types, the choice ended up in the raw field with 3 bytes.

Afterwards, we determined the cases were null values could appear on the fact table:

[3] Surrogate keys are sequential numbers which are generated, as needed, to populate a dimension [33].
[4] The function to use would be: HEXTORAW(TRIM(TO_CHAR(integer,'XXXXXXXX'))).

1. user cannot be determined when the request is not secure.

2. search for data sets operation may map to several data sets.

For the first case, we added a user called "*unknown*". For the list data sets operation problem, we added a data set called "*many datasets*", which does not follow the data set naming convention, therefore it cannot be created by users.

In the user dimension, we took care not to store the user name or anything which could be easily related to him since, in some european countries, it's illegal to use the user's grid certificate distinguished name (DN) to determine a usage profile. Since we could not confirm this information and ATLAS is a very wide international collaboration, for precaution, this information is not stored in our data warehouse but a hash of the DN is used to map requests to the same user.

The final relational schema for this model can be seen on Figure 3.5.

3.5.4 Extract, transform and load

On Oracle's Data Warehousing Guide [34], it is stated that uniqueness can be enforced as part of the extract, transform and load (ETL) processing. This means that we aren't obliged to use unique indexes, if the ETL process guarantees the unicity of the table fields in question.

For the date and time dimensions, we had to generate the values of these dimensions. In order to have a more optimized schema, the best thing would be to have closer dates and times, in closer or adjacent blocks. Oracle has a type of table, called index-organized table (IOT) [35], which could make this possible. In this case, the data is stored, in sorted order, in the leaves of the B-tree index. Therefore, in these two dimensions, we took some caution populating them to match the database sequences order so that closer dates or time would be on the same or closer blocks. Also, according to [34], the usage of index-organized tables gives the possibility of doing parallel fast full scans.

On the fact table, we took the same approach but we added compression over the date and time keys, meaning that we would avoid repetitions of the time key for the same date key, therefore increasing performance and reducing storage space [35]. Since most of the dimensional queries will be constrained on the date dimension, the date foreign key is the leading term of the primary key.

Another optimization was done for the fact table and some dimension tables regarding the space Oracle reserves on the data blocks for updates. For example, in the fact table we always append information and possibly delete very old information but not update existing one. The date and time dimensions, once generated they will not be updated. The same for operation, user and status dimensions. Therefore, on these cases, we have

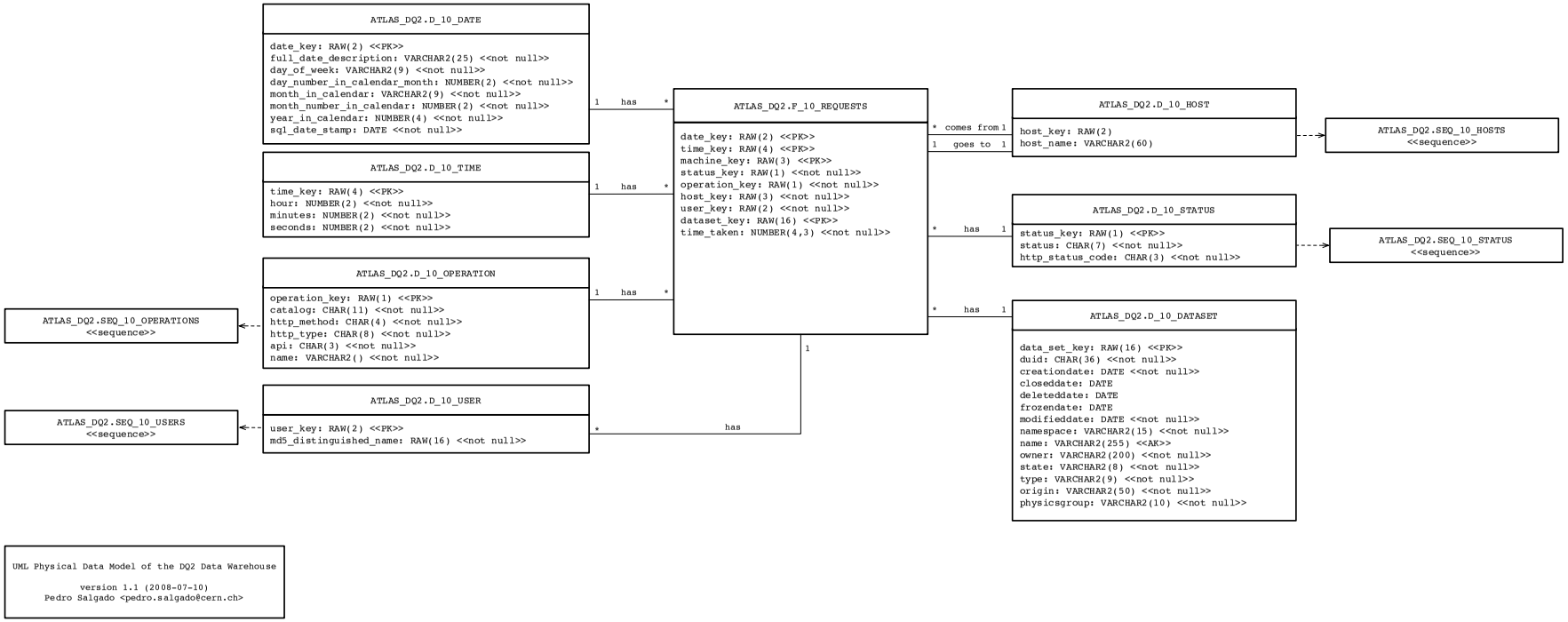

Figure 3.5: UML Physical Data Model of the DQ2 data warehouse (v1.0, June 2008).

set the PCTFREE[5] parameter to zero. This reduces fragmentation at the data block level and reduced the consequent waste of disk space.

We use partitions [34], on the fact table, to improve the manageability and performance of the data warehouse. For example, we could assign different partitions into different table spaces and easily stay below the 250 gigabyte limit per table space, imposed by CERN IT department. In terms of performance, the usage of partitioned index-organized tables gives the possibility of doing parallel fast full scans, as the non-partitioned IOT, and parallel index range scans [34].

As advised in [34], since we are gathering historical data, we do range partitioning on the facts table over the date dimension key. From the feedback we got from the database administrators at CERN, we should not have many partitions therefore we started doing partitioning per month and, if needed, we will revise this strategy later on and make the necessary changes. More, since we know that most of the queries will be constrained by date, most of the queries will perform better since, typically, they only need to access the partition required to resolve the query [33]. Also, loading data will run more quickly since we will only need to rebuild the index for a partition, instead of the entire table [33].

3.6 Summary

In this section, we have made a brief description of the development of the DQ2 central catalogs component and the steps made to implement a new reporting system to determine the usage of the system.

[5]The PCTFREE [36] parameter sets the minimum percentage of free space for possible updates to rows that already exist in a data block.

Chapter 4

Conclusions and Future Work

On this chapter, we talk about the results, future directions and planned work for the central catalogs and its reporting tool.

4.1 Results

One of our first goals was to improve the stability of the central catalogs component (section 1.2.1). For this objective, all of the steps have been accomplished. An overview of the MySQL cluster and Oracle RAC solutions has been provided and is described in sections 2.3.1 and 2.3.2, the evaluation of these solutions have been done and can be seen on section 3.2.1 and the central catalogs have been implemented accordingly (section 3.2).

Another of the objectives we had set was to make a specification of the use cases for containers and implement them in the DQ2 system. The specification can be found in [32] and in the appendix B.3. A new version of the central catalogs that supports containers has been deployed, in April 2008 (section 3.3). Currently, users are slowly changing their tools and scripts to take advantage of these new features.

In the path towards making the central catalogs more robust, we had also set as a priority the memory problems mentioned in section 1.2.3. This has been implemented and deployed in DQ2 1.1 version, as described in section 3.4

Regarding the reporting tool mentioned in section 1.2.4, some research has been made and its presented in 2.2. A data warehouse has been chosen to store the information for this tool, the dimensional model has been specified and implemented, as described in section 3.5. Also, in section 3.5.4, some study and techniques are presented to implement the extraction, transformation, load process. These techniques have been retrieved from Oracle's Data Warehousing Guide [34] and from the experience acquired in the Oracle 10g: Administration Workshop I and II and their respective guides [37, 38]. Some of the

dimensions (time, date, status, operation and status), which don't require external sources of information already have the final data inserted. Still, at the time of this writing, the data staging process is still not ready for the other dimensions, which means we cannot present any results of its usage. This part of the work is planned and is described in section 4.3.2. We also plan the introduction of a new dimension (section 4.3.3) and a final decision of what tools will be used to do provide the presentation layer over the data warehouse (section 4.3.4).

Although not stated as one of the goals for the system, the DQ2 system architecture and implementation has been in the publication of several articles, namely on the 2007 International Conference in High Energy Physics [15, 20] (Victoria, Canada) and the Proceedings of the 2007 Europhysics Conference on High Energy Physics [14], organized by the European Physical Society.

4.2 DQ2 2.0

For the next version of the central catalogs, we intend to add minor database changes, test and take a final decision regarding a new asynchronous client-server protocol and finalize the implementation of the data warehouse reporting tool.

4.2.1 Database schema changes

In May 2008, we had almost 2 million data sets registered from which almost more than 75% percent had been marked as deleted (archived), as seen on Table 4.1.

state	# data sets
OPEN	148,228
CLOSED	43,412
FROZEN	176,507
ARCHIVED	1,631,215

Table 4.1: Number of data sets per state.

On the data set version-file table, we had more than 156 million entries. From these, more than 23 million entries were from archived data sets (Table 4.2). With the introduction of containers (section 3.3.2), this number is expected to increase substantially since large data sets will start being replaced by references to other data sets.

Since DQ2 never deletes the data set definition nor any of its contents, this data accumulates and leads to fragmentation[1] and lower performance. Not to mention that this

[1] Fragmentation, in this case, it's not due to the fact that the data block remains with many empty rows but because the rows will not be used any longer, therefore they act as having been deleted.

state	# files
OPEN	61,632,720
CLOSED	27,769,258
FROZEN	43,837,955
ARCHIVED	23,176,922

Table 4.2: Number of files per data set state.

table had to be denormalized for performance reasons, plus we copy the files from one version to the next, meaning that we also waste a lot of disk space.

To solve part of this problem, we are planning the usage of a second table, so that we can:

- separate live data set contents from archived information.
- reduce the speed on which the main table space is being filled.
- reduce the total disk space occupied by the database since the table to be used for garbage collection would be according to the third normal form; therefore it will have less columns.

Due to the backup technology used by CERN IT, they have a requirement that table spaces should be below 250 gigabytes, therefore we will also place this table in a different table space from the one holding the original table.

Still, one last topic is missing. The deletion of a large quantity rows on the table does not necessarily mean that a large disk space will be freed, due to the fact that the data is stored in blocks which can only be made free if all of the rows on them are all deleted. This means that even if the rows of archived data sets were deleted from the table, their space may not be retrieved and, regarding to fragmentation and disk space recovery, this may not improve as much as we would hope for. In order to effectively recover space and reduce the index size, we will need to regularly, what in Oracle is called, shrink the table [37].

4.2.2 Memory usage and asynchronous behaviour

Any machine, using our clients, executing several parallel calls into the central catalogs could start having the same effect, as described in section 1.2.3, but on this case, on the client itself and not on the server.

Based on the solution to reduce memory footprint on the server-side, described on section 3.4, we could also reduce our clients memory footprint and, at the same time, make them more robust.

If the server response hasn't completely arrived, the client has a buffer which contains the part of the response already sent, which can be parsed and sent to the user for further treatment. In practice, on some read calls where the server response is very large, we could enhance our clients so that an almost immediate feedback can be given to the user, giving the impression of a faster response, even though the total time of the operation would almost be the same.

For example, imagine a user who wanted to make a certain operation over a large number of data sets. By using this solution, as soon as the first data set information arrives, it can be returned by the client and processed by the user. Behind the scenes, the rest of the data sets will still be arriving and being inserted into the client buffer, waiting to be retrieved.

The *Python* package our client is currently using, and several other *Python* alternatives that were already tried, only return the server response when all the output has arrived. This was expected when the protocol was established in the beginning (section 3.1.1), therefore no asynchronous behaviour can be achieved with the current packages we use. More, when making the conversion of the server response into a *Python* readable structure, another copy of the same information will exist, which significantly increases the maximum amount of memory the client uses, compared with the strategy we are proposing.

The first part of the solution, will be to solve this constraint on the client side. For this, we will need to implement a new protocol which gives the possibility of identifying parts of the output that could be resolved as soon as they arrive.

We know that no *Python* package that works with HTTP, at a higher level, has the ability to show the content of the response while being retrieved. Still, a more lower-level package (asyncore [39]), that works at the socket level, has proven to work as required and it will, most probably, be our final choice.

Working at a lower-level, though, would mean implementing our own package to handle HTTP interaction, Secure Socket Layer (SSL) connections together with Grid Security Infrastructure (GSI) connections. Due to man power and time constraints, we can only implement a HTTP client, therefore the final solution will be:

- for requests which are secure or have a small output, we should use the current protocol with its synchronous behaviour.
- for the cases where the server response can raise memory issues or profit from asynchronous behaviour, we will use the new protocol.

4.2.3 Data serialization

On our current client-server protocol, one of the most important issues we have is related to the usage of *Python* 's eval [29] and pickle [28] libraries to interpret the server

response. This can lead to the execution of arbitrary code making it a very high security vulnerability.

Some research and investigation of the currently available solutions and has shown some interesting alternatives like the JavaScript Object Notation (JSON) [40] and YAML [41]. The simplest solution would be to move the client and the server into a model which would use XML (eXtended Markup Language) with remote procedure calls (RPC). Python already has the libraries to easily implement this (xmlrpclib [42]). Still, performance is always one of our concerns, so the overhead of marshaling and unmarshaling will need to be taken into account.

Our next step is to make an evaluation of each of the alternatives and, only then, start using the solution which may better fit our needs.

Another possible solution is to use some data serialization package, to transfer data from client to server and vice-versa. For the *Python* programming language several alternatives were found: array [43], marshal [44], struct [45], YAML [41], Javascript Object Notation (JSON) [40], xdrlib [46] library, which supports Sun Microsystems' External Data Representation Standard [47] [48].

4.2.4 Architecture

The central catalogs are currently deployed into four machines which are serving requests through an web server (apache [49]) configured to use a single process which maintains 64 threads to serve requests. This means a total of 256 simultaneous requests could be served immediately. If this value is passed, the web server will queue the extra requests and start serving them, as soon as possible.

On our database instance we have a limit, set by CERN IT department, of 150 connections for reading and another 150 connections for writing. With the current configuration, if the services were used at their fullest, the load on the machines would be at a reasonable level but certain threads could be locked, for some time, waiting for a free connection from the pool, since the maximum number of threads (256) is greater than our connection pool limit (150).

Due to CERN IT's machine replacement policy, at this very moment, much more powerful hardware is already available for our central catalogs. Some rough estimates have shown us that with just one of these new machines, we can server up to eight times more requests than with the current setup and the load on the machine would still be at a very low rate. But this is not all. With our continuous efforts to reduce the memory footprint of the central catalogs (section 3.4 and 4.2.2) together with the high value of random-access memory (RAM), the memory usage of these machines is low[2], and, as a consequence, we have a large quantity of unused RAM which we could benefit from.

[2]While doing a stress test we could only use a little bit more than two thirds (6 gigabytes) of the total amount of memory (16 gigabytes). This was before, we introduced the memory usage improvement referred in section 3.4.

In order to reduce the usage of our database resources and better use the available hardware, we will be using, in the near future, a distributed memory object caching system called memcached [27], which has been presented in section 2.4. This solution will be placed next to the application's session façade and data access object (DAO) [31] layer, as seen on Figure 4.1. The DAO layer will first access the cache to fetch the data and if the data is present, it will send it back to the application; otherwise it will issue a query, get the rows, store them into the cache and send the response back to the user. The same will happen with the session façade. It will query the cache for a previous processed result and, if this cannot be found, it will use the data access layer.

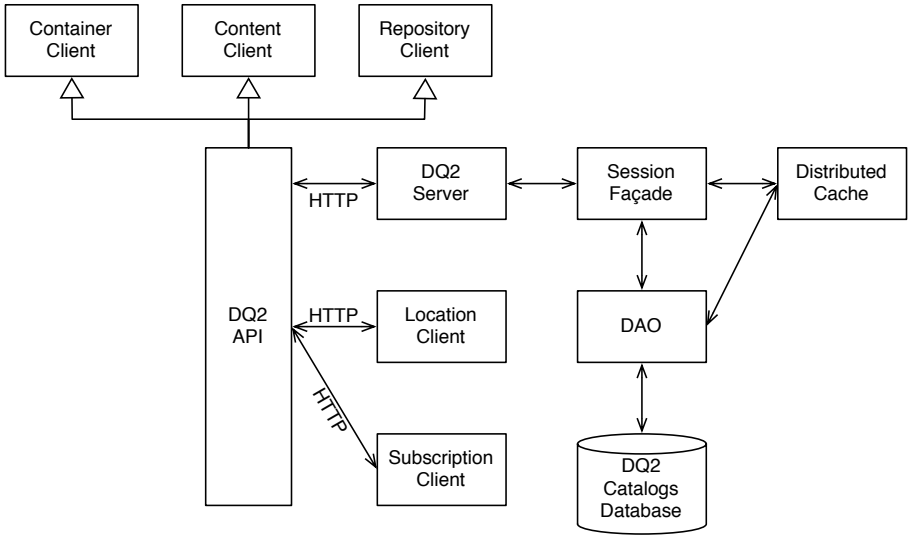

Figure 4.1: Don Quijote 2 architecture (v2.0).

For example, file registrations on a data set happen very frequently, in a rather short period of time. Every time a request arrives on our servers to add a file to a data set, the first thing our application does is to retrieve the data set owner, which is static information. After that, the information retrieved from the database is compared with the user's grid certificate information, in order to allow the change or not. This is a typical case where a cache would be of tremendous help. If the cache had this information, we could use it for many simultaneous requests and the database would only handle the insert statements.

Since this cache stores all information in RAM it will behave just like having a second, much larger, database cache and since its size is only dependable on the number of machines being used for caching, it makes it a scalable solution. If the load justifies it, we can easily add more machines, which means more RAM and more cache and less load on the database.

There is also another benefit. The more and more requests we serve using memcached, the more space Oracle's database buffer cache will have to deal with inserts and updates,

meaning that with the same database setup, we can effectively manage more simultaneous inserts and updates since less reads will be using blocks in the database buffer cache.

4.3 Data warehouse

In this part of the report, we describe the work planned for the system and data set usage reporting tool, mentioned in section 3.5.

4.3.1 Populating the fact table

The information needed to populate the fact table is stored in the log files of the DQ2 system. Since the application is deployed in several machines, the log files are spread among them. Each log file will be copied into a specific directory, in a network accessible area so that we have a centralized place to hold this data, we don't use any of the central machines' processor or memory and have a backup of the log files assured by CERN's IT department.

The next step will be to develop a software package to read and parse the log files, transform their data and populate the fact table. The usage of a network accessible area, will make it possible to develop a solution using CERN's batch system therefore, there will be no need to have dedicated hardware for this purpose.

4.3.2 Populating the dimension tables

For the dimension tables, since all of their data comes from our central catalogs database, we will load the data directly from the source.

In order to achieve this we have to take into account two cases. The first time we will need to load a large amount of data. According to Oracle's Data Warehousing Guide [34], one of the possibilities is to give a hint on the insert query which will ensure that the database will insert data blocks above the high-water-mark[3] (HWM), bypassing the database buffer cache and writing directly the information into the data file. During this loading process, the table will not be accessible.

After this step, we will only need to do regular inserts of a small portion of new data and synchronize the previous loaded data. For this case, we will be using the merge statement which gives the possibility of updating or inserting data according to certain conditions specified in the query.

[3] High-water-mark is a term to specify the boundary between used and free space in a table segment.

4.3.3 More dimensions

A new dimension (activity) will be added to our data warehouse, so that the analysis of specific periods of time where certain activities, like functional tests or real data taking, happen can be chosen in a straightforward way by the users. The dimension's attributes are show in Table 4.3.

Activity dimension	
Activity Key	surrogate values, 1-N
Name	M6 - real data taking, FDR2 - functional tests
Begin Date	the date when the activity started
End Date	the date when the activity finished

Table 4.3: Description of the activity dimension attributes.

4.3.4 Analyzing

Populating the data warehouse won't be the final step. Afterwards, we need to analyze it. From the Oracle's documentation [50], a way to analyze data contained in a data warehouse is through its OLAP API, using a Java application.

The OLAP API contains objects for measures, dimensions, hierarchies, levels, and attributes. It also has a object-oriented model called multidimensional metadata (MDM) which, after being mapped into the relational data in the data warehouse, can be used to access the information in the data warehouse.

There are also, other alternatives like Oracle Data Miner, Oracle Spreadsheet Add-In for Predictive Analytics using Microsoft Excel and Oracle Data Mining JDeveloper with SQL Developer Extensions.

Within the team, there is no experience with any of these tools, therefore some time will also be needed to learn how these tools work, before making a final decision.

References

[1] Jean-Luc Caron. AC collection. legacy of AC (pictures from 1992 to 2002). published on http://cdsweb.cern.ch/record/841555, May 2008.

[2] Joao Pequenao. Computer generated image of the whole ATLAS detector. published on http://cdsweb.cern.ch/record/1095924, March 2008.

[3] Kors Bos. Tier-0&1&2 storage classes for 2008. published on http://www.nikhef.nl/~bosk/presentations/p2008.html, June 2008.

[4] CERN European Organization for Nuclear Research. CERN - European Organization for Nuclear Research. published on http://cern.ch/, 2008.

[5] CERN European Organization for Nuclear Research. CERN in a nutshell. published on http://public.web.cern.ch/public/en/About/About-en.html, 2008.

[6] CERN European Organization for Nuclear Research. The large hadron collider. published on http://public.web.cern.ch/public/en/LHC/LHC-en.html, 2008.

[7] CERN European Organization for Nuclear Research. LHC machine outreach. published on http://lhc-machine-outreach.web.cern.ch/lhc%2Dmachine%2Doutreach/.

[8] CERN European Organization for Nuclear Research. The ATLAS experiment. published on http://atlas.ch/, 2008.

[9] CERN European Organization for Nuclear Research. ATLAS - A Toroidal LHC ApparatuS. published on http://public.web.cern.ch/public/en/LHC/ATLAS-en.html, 2008.

[10] CERN European Organization for Nuclear Research. LHC computing grid goes online. published on http://press.web.cern.ch/press/PressReleases/Releases2003/PR13.03ELCG-1.html, September 2003.

[11] ATLAS Computing Group. ATLAS computing technical design report. published on http://atlas-proj-computing-tdr.web.cern.ch/atlas-proj-computing-tdr/PDF/Computing-TDR-final-July04.pdf, July 2005.

[12] Miguel Branco, David Cameron, Pedro Salgado, Mario Lassnig, and Vincent Garonne. Distributed data management (DDM). published on https://twiki.

cern.ch/twiki/bin/view/Atlas/DistributedDataManagement, 2007.

[13] Miguel Branco. DDM review wiki page, 2006.

[14] Armin Nairz, Luc Goossens, Miguel Branco, David Cameron, Pedro Salgado, Kors Bos Dario Barberis, and Gilbert Poulard. ATLAS computing system commissioning: real-time data processing and distribution tests. In Institute of Physics Publishing, editor, *The 2007 Europhysics Conference on High Energy Physics*, volume 110. European Physical Society, 2008.

[15] Mario Lassnig, Miguel Branco, David Cameron, Benjamin Gaidioz, Vincent Garonne, Birger Koblitz, Massimo Lamanna, Ricardo Rocha, and Pedro Salgado. Managing ATLAS data on a petabyte-scale with DQ2. In *Journal of Physics: Conference Series*, Bristol, England. Institute of Physics Publishing.

[16] Pedro Salgado. DQ2 central catalogs wiki page. published on https://twiki.cern.ch/twiki/bin/view/Atlas/DonQuijote2CentralCatalogs, 2007.

[17] Miguel Branco, David Cameron, and Pedro Salgado. *DDM Design and Implementation*. CERN - European Organization for Nuclear Research, Geneva 23, CH-1211, Switzerland, November 2006.

[18] Jean-Philippe Baud and Sophie Lemaitre. The LCG File Catalog (LFC). published on http://hepix.fzk.de/upload/lectures/LCG-File-Catalog-HEPIX-2005-1.pdf, May 2005.

[19] Gavin McCance. Grid file transfer service. published on https://twiki.cern.ch/twiki/bin/view/EGEE/FTS, June 2008.

[20] Ricardo Rocha, Miguel Branco, David Cameron, Benjamin Gaidioz, Vincent Garonne, Birger Koblitz, Massimo Lamanna, Mario Lassnig, Dietrich Liko, and Pedro Salgado. Monitoring the ATLAS distributed data management system. In *Journal of Physics: Conference Series*, Bristol, England, September 2007. Institute of Physics Publishing.

[21] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, second edition edition, 2006.

[22] Oracle Corporation. Oracle real application clusters. published on http://www.oracle.com/technology/products/database/clustering/index.html, June 2008.

[23] Oracle Corporation. Oracle real application clusters datasheet. published on http://www.oracle.com/technology/products/database/clustering/pdf/ds_rac11g.pdf, June 2008.

[24] Oracle Corporation. Cache fusion: Extending shared-disk clusters with shared caches. published on http://www.dia.uniroma3.it/~vldbproc/086_683.pdf, June 2008.

[25] MYSQL AB. MYSQL cluster. published on http://www.mysql.com/products/database/cluster/, June 2008.

[26] MYSQL AB. MYSQL cluster evaluation guide. published on http://www.mysql.com/why-mysql/white-papers/cluster-technical.php, June 2008.

[27] Brad Fitzpatrick. memcached. published on http://www.danga.com/memcached/, June 2008.

[28] Python Software Foundation. Python library reference: pickle – python object serialization. published on http://docs.python.org/lib/module-pickle.html, February 2008.

[29] Python Software Foundation. Python library reference: 2.1 built-in functions. published on http://docs.python.org/lib/built-in-funcs.html, February 2008.

[30] Sourceforge. cx_oracle. published on http://sourceforge.net/projects/cx-oracle/, June 2008.

[31] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Sun Microsystems, 1 edition, 2001.

[32] Pedro Salgado. DQ2 dataset container catalog. published on https://twiki.cern.ch/twiki/bin/view/Atlas/DonQuijote2ContainerCatalog, October 2007.

[33] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit*. Wiley Computer Publishing, 2nd edition edition, 2002.

[34] Oracle Corporation. Oracle database data warehousing guide. published on http://download.oracle.com/docs/cd/B19306_01/server.102/b14223.pdf, December 2005.

[35] Oracle Corporation. Oracle9i index-organized tables. published on http://www.oracle.com/technology/products/oracle9i/datasheets/iots/iot_ds.html, June 2008.

[36] Oracle Corporation. Oracle database concepts: 2 data blocks, extents, and segments. published on http://download.oracle.com/docs/cd/B19306_01/server.102/b14220/logical.htm, June 2008.

[37] Oracle University. *Oracle Database 10g: Administration Workshop I Student Guide*, volume 1-3. 3rd edition, November 2005.

[38] Oracle University. *Oracle Database 10g: Administration Workshop II Student Guide*, volume 1-2. 3rd edition, January 2006.

[39] Python Software Foundation. Python library reference: asyncore – asynchronous socket handler. published on http://docs.python.org/lib/module-asyncore.html, February 2008.

[40] Douglas Crockford. RFC4627 - The application/json Media Type for Javascript Object Notation (JSON). published on http://www.ietf.org/rfc/rfc4627.txt, August 1995.

[41] yaml.org. The official yaml web site. published on http://www.yaml.org/, August 1995.

[42] Python Software Foundation. Python library reference: xmlrpclib – xml-rpc client access. published on 'urlhttp://docs.python.org/lib/module-xmlrpclib.html, February 2008.

[43] Python Software Foundation. Python library reference: array – efficient arrays of numeric values. published on http://docs.python.org/lib/module-array.html, February 2008.

[44] Python Software Foundation. Python library reference: pickle – python object serialization. published on http://docs.python.org/lib/module-pickle.html, February 2008.

[45] Python Software Foundation. Python library reference: struct – interpret strings as packed binary data. published on http://docs.python.org/lib/module-struct.html, February 2008.

[46] Python Software Foundation. Python library reference: xdrlib – encode and decode xdr data. published on http://www.python.org/doc/1.5.2/lib/module-xdrlib.html, February 2008.

[47] Inc. Sun Microsystems. RFC1014 - XDR: External Data Representation Standard. published on http://rfc.net/rfc1014.html, June 1987.

[48] Inc. Sun Microsystems. RFC1832 - XDR: External Data Representation Standard. published on http://rfc.net/rfc1832.html, August 1995.

[49] The Apache Software Foundation. Apache HTTP Server. published on http://httpd.apache.org/, June 2008.

[50] Oracle Corporation. Oracle olap developer's guide to the olap api. published on http://download.oracle.com/docs/cd/B19306_01/server.102/b14223.pdf, July 2006.

[51] Wikipedia: wiki. published on http://en.wikipedia.org/wiki/Wiki, August 1995.

Appendix A

Data set

A.1 What is a data set?

Data set is defined as an aggregation of data which usually spawns over more than one file. Its role is to provide a logical organization over files. Typically, a data set is composed of a set of files which a user needs to process as a whole, either serving as the input/output of a certain computation process or as the result of a data acquisition process [17].

A.2 What is a jumbo data set?

"Jumbo data set" is not a concept expressed in any of the ATLAS technical documentation, but more an expression frequently used to refer to data sets whose contents exceed tens of thousands of files.

The implementation of the container concept (section B.1), was only done recently (section 3.3.2) and the lack of this feature originated the appearance of very large data sets. These data sets became "famous" due to the fact they caused stability problem in our central services, as described in section 1.2.2. Short after, the term "jumbo data set" appeared and spread over our user community.

A.3 Why do we use data sets?

First of all, because the usage of data sets map to the requirements made in the ATLAS Computing TDR [11]:

> "Here follow the principal design precepts and requirements driving the design and implementation of DQ2.
>
> (...)
>
> - Datasets should be the principal means of file-based data organization and lookup. In order to achieve adequate performance and scalability in lookup and access of data files, and in order to provide the data aggregations required for efficient data handling and analysis in support of the ATLAS Computing Model, the DDM system should work wherever possible with file aggregations (datasets) rather than files.

- The system must provide data management tools to end users at the file level as well as the dataset level. While most end-user data access will be done at a higher (dataset) level, working with files (and collections of files) is important for end users today and will remain so in the future in some contexts. Consequently, while the system should be optimized for dataset-level access, it must also support file-level access.

" [11, 4.6.3 Precepts and Requirements]

For a physicist searching for data with some particular characteristics, they can more easily handle a shorter list of data sets who match their interest than having a long list of files. Also, since the data is organized and transfered using this unit (data set), he can easily determine the completeness of the data at the sites and which ones are the best candidates to run his jobs.

This is not all. For example, the physics simulation software is organized so that a certain analysis task is spawn over a set of jobs that produce a certain number of files containing valuable data.

As described on section 1.1.2, the jobs will run in many different Tier-2 facilities, which will then send their data to their respective Tier-1. To more easily manage the data which is produced and its movement, before the job is sent to the site, it already has a well defined output data set. As files are produced, they are also registered in the job's output data set. In the end of its work, this data set will be registered for transfer to the Tier-1 facility so that, together with other data sets produced in other sites, further processing may be done.

By this example, it's easy to see that a file movement is usually associated with the movement of other files; therefore the concept of data set exists already in the way data is acquired, transfered and processed.

In conclusion, data sets are part of the software requirements, as expressed in [11], and, according to [13, 17], also have the following advantages:

- wraps the data which users manipulate together.
- the order of magnitude of data sets is O(100,000) which is less than the order of magnitude of files O(10,000,000).
- a data set can be thought as the basic unit of transfer.
- network channels and storage elements can be used more efficiently since more data (files) can be transfered using bulk operations instead of a file-by-file approach.

Appendix B

Container

B.1 What is a data set container?

> "Files are aggregated into datasets, datasets are aggregated by dataset containers, and dataset containers can be organized in a hierarchy to express flexibly layered levels of containment and aggregation." [11, section 4.6.5 System Architecture]

Having stated this, container is a logical aggregation of data sets or, in other words, it's higher layer than data sets to aggregate files.

B.2 Why do we use containers?

As explained above, the container concept is part of the requirements stated in the ATLAS Computing TDR [11]. Therefore, it has to be implemented in the system.

Still, experience has shown that without this concept, the system will not be able to scale, as well as it could (section 1.2.2).

B.3 Use cases

The use cases for the container catalog are described on this part of the report, still they are also publicly available in [32].

A UML use case diagram can be found on Figure B.1, page 54.

create container

Use case:create container

Version: 1.0

Summary: user sends a request to register a data set container.

Dependency:

Actor: DQ2 User

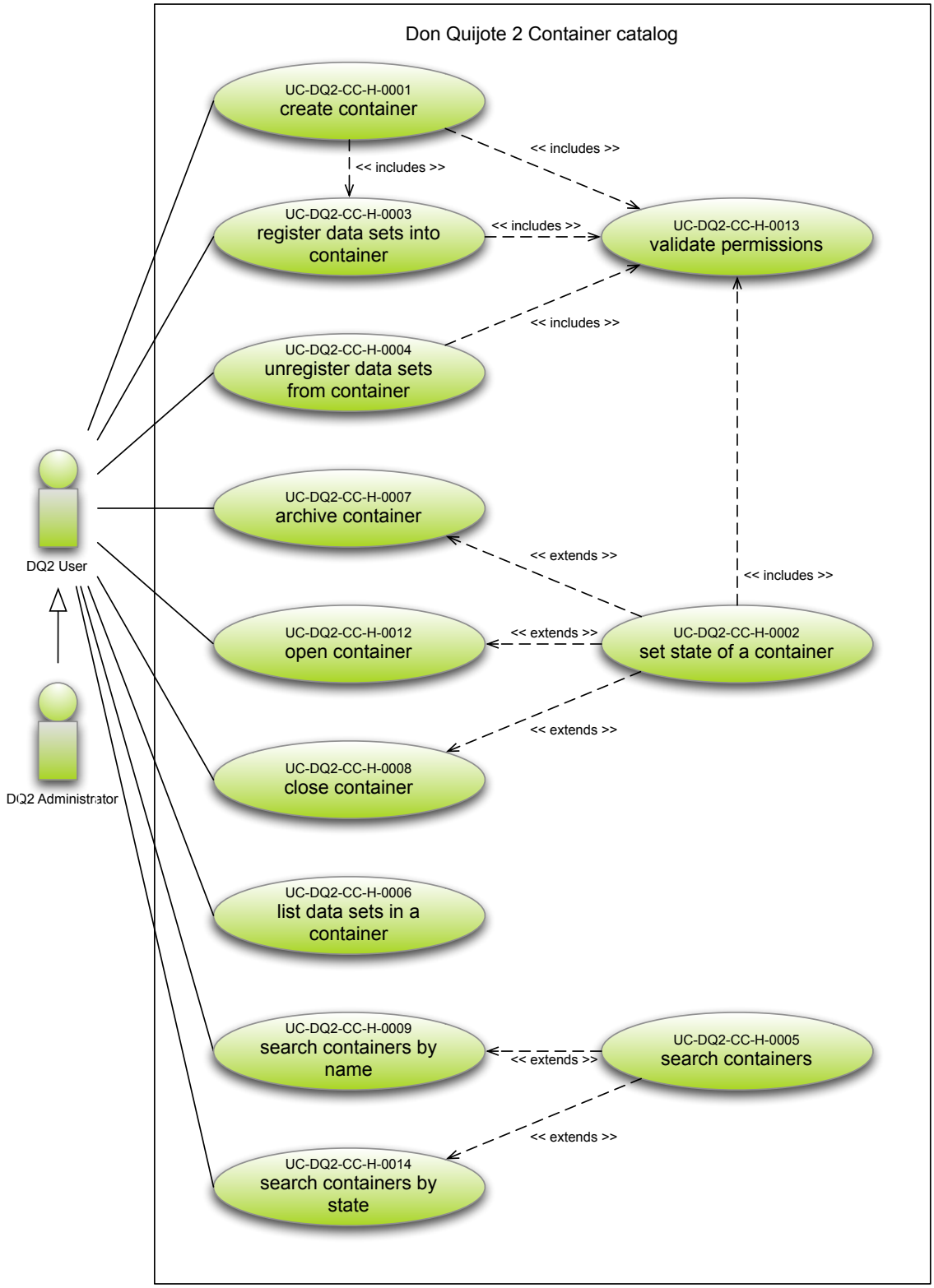

Figure B.1: UML Use Case diagram of the DQ2 container catalog (v1.1, July 2008).

Precondition:

Description: 1. the use case begins when the DQ2 user requests the registration of a container.
2. the system registers the container.
3. the system informs the user the container was registered.

Alternatives: 2a: container with the same name already exists.
2a1. The system informs the user the container already exists.

Postcondition:the data set container is registered and its name is unique.

register data sets into a container

Use case:register data sets into a container

Version: 1.0

Summary: user sends a request to register a data sets into a container.

Dependency:

Actor: DQ2 User

Precondition:

Description: 1. the use case begins when the DQ2 user requests the registration of data sets into a container [validate permissions].
2. the system registers data sets into the container.
3. the system informs the user the data sets were registered.

Alternatives: 2a: data set container doesn't exist.
2a1. the system informs the user the container doesn't exist.

2b: container is closed.
2b1. the system informs the user that the container cannot be modified because it's closed.

2c: container is archived.
2c1. the system informs the user that the container cannot be modified because it's archived.

2d: data set is open or closed or archived.
2d1. the system informs the user that a data set can only be added if it's frozen.

2e: container already contains the data set.
2e1. the system informs the user it already contains the data set and continues fullfilling the request.

2f: data set does not exist.

2f1. the system informs the user the data set does not exist.

Postcondition:a data set container can only have frozen data sets.

remove data sets from container

Use case:remove data sets from container

Version: 1.0

Summary: user sends a request to remove data sets from a container.

Dependency:

Actor: DQ2 User

Precondition:

Description: 1. the use case begins when the DQ2 user requests the removal of data sets from a container. 2. the system removes the data sets from the container. 3. the system informs the user the data sets were removed.

Alternatives: 2a: data set container doesn't exist.
2a1. the system informs the user the container doesn't exist.

2b: container is closed.
2b1. the system informs the user that the container cannot be modified because it's closed.

2c: container is archived.
2c1. the system informs the user that the container cannot be modified because it's archived.

2d: container doesn't have the data set.
2d1. the system informs the user the container doesn't contain one of the data sets.

2e: data set does not exist.
2e1. the system informs the user the data set does not exist.

Postcondition:all data sets sent by the user aren't part of the data set container.

set state of a container

Use case:set state of a container

Version: 1.0

Summary: user sends a request to change the state of a data set container.

Dependency:

Actor: DQ2 User

Precondition:

Description: 1. the use case begins when the DQ2 user requests a change on the state of a container.
2. the system changes the state of the container. [archive container][close container][open container]
3. the system informs the user the state has been changed.

Alternatives: 2a: data set container doesn't exist.
2a1. the system informs the user the container doesn't exist.

2b: user is not the owner of the data set container.
2b1. the system informs the user he must the owner of the container to be able to change it.

2c: container is archived.
2c1. the system informs the user the container is árchived.

Postcondition:the data set container state will be changed.

archive container

Use case:archive container

Version: 1.0

Summary: user sends a request to archive a data set container.

Dependency: set state of a container

Actor: DQ2 User

Precondition:

Description: 1. the system changes the state of the container to árchived.

Alternatives: 1a: container is already archived.
1a1. the system informs the user the container is already árchived.

Postcondition:.

close container

Use case:close container

Version: 1.0

Summary: user sends a request to close a data set container.

Dependency: set state of a container

Actor: DQ2 User

Precondition:

Description: 1. the system changes the state of the container to ćlosed́

Alternatives:

Postcondition:the data set container is marked as ćlosed́

open container

Use case:open container

Version: 1.0

Summary: user sends a request to open a data set container.

Dependency: set state of a container

Actor: DQ2 User

Precondition:

Description: 1. the system changes the state of the container to ópeń

Alternatives: 1a: container is already open.
1a1. the system informs the user the container is already ópeń

Postcondition:the data set container will be marked as ópeń

search containers

Use case:search containers

Version: 1.0

Summary: user sends a request to search for containers.

Dependency:

Actor: DQ2 User

Precondition:

Description: 1. the use case begins when the DQ2 user searches a list of containers by a certain criteria.
2. the system will perform a search for containers (in open or closed states, by default) who match the given criteria [search by name][search by state]
3. the system will present the results to the user.

Alternatives:

Postcondition:

search containers by name

Use case:search containers by name

Version: 1.0

Summary: user sends a request to search for containers by name.

Dependency: search containers

Actor: DQ2 User

Precondition:

Description: 1. the system will perform a search for containers who match the name.

Alternatives: 1a: name contains śtarćharacter.
1a1. the system will perform a search for containers who match the given wildcard expression and present them to the user.

Postcondition:

search containers by state

Use case:search containers by state

Version: 1.0

Summary: user sends a request to search for containers by state.

Dependency: search containers

Actor: DQ2 User

Precondition:

Description: 1. the system will perform a search for containers who match the given state.

Alternatives: 1a: state contains ˝ćharacter.
1a1. the system will perform a search for containers who don't match the given state.

Postcondition:

list data sets in a container

Use case: list data sets in a container

Version: 1.0

Summary: user sends a request to list the data sets in a container.

Dependency:

Actor: DQ2 User

Precondition:

Description: 1. the use case begins when the DQ2 user requests a list data sets in a container.
2. the system will retrieve the data sets of the container.
3. the system will present the results to the user.

Alternatives: 3a: container doesn't exist.
3a1. the system will inform the user the container doesn't exist.

3b: container is archived.
3b1. the system will inform the user the container is archived.

Postcondition:

validate permissions

Use case: validate permissions

Version: 1.0

Summary: users sends a request to modify a container which triggers this validation.

Dependency:

Actor: DQ2 User

Precondition:

Description: 1. the use case begins when the DQ2 user makes a request to change a container.
2. the system will check if the user is an administrator or the owner of the container.
3. the system will let the request be fulfilled.

Alternatives: 2a: container doesn't exist.
2a1. the system will inform the user the container doesn't exist.

2b: user is not administrator or the owner of the container.

2b1. the system will inform the user he cannot change the container and will stop fulfilling the request.

Postcondition:

Container

Appendix C

Others

C.1 What is a wiki?

Wiki is a type of collaborative software, which allows anyone who accesses it, typically using a web browser, to create, edit and delete web pages, using a simplified markup language [51].

C.2 Why do we use wiki?

In the ATLAS Distributed Data Management team, we use wiki to store information of our daily work, namely publish technical documentation, development notes, internal meeting minutes, internal procedures for user and service support, manage our hardware with links to monitoring systems, references to international publications, among others.

We use some of our wiki "advanced" features like wiki forms, to more easily structure our data in pre-determined way, as shown in the container catalogs use cases [32].

We also use the built-in permissions mechanism to allow or disallow a group of users to edit or rename our topics and the ability to include information of a particular section of another wiki page, to reduce the overhead of maintaining and referencing the same information in several pages, as shown in https://twiki.cern.ch/twiki/bin/view/Atlas/DistributedComputingMachines and https://twiki.cern.ch/twiki/bin/view/Atlas/DistributedDataManagementARDAMachines.

C.3 Calculating the average size of integer type

```
INSERT INTO vsize_test
(a)
VALUES
(1);

INSERT INTO vsize_test
(a)
VALUES
(2);
```

```
INSERT INTO vsize_test
(a)
SELECT a+2
FROM vsize_test;

INSERT INTO vsize_test
(a)
SELECT a+4
FROM vsize_test;

INSERT INTO vsize_test
(a)
SELECT a+8
FROM vsize_test;

INSERT INTO vsize_test
(a)
SELECT a+16
FROM vsize_test;

INSERT INTO vsize_test
(a)
SELECT a+32
FROM vsize_test;

INSERT INTO vsize_test
(a)
SELECT a+64
FROM vsize_test;

INSERT INTO vsize_test
(a)
SELECT a+128
FROM vsize_test;

INSERT INTO vsize_test
(a)
SELECT a+256
FROM vsize_test;

INSERT INTO vsize_test
(a)
SELECT a+512
FROM vsize_test;

DELETE FROM vsize_test
WHERE a > 999;
```

```
SELECT AVG(VSIZE(a))
FROM vsize_test;

2.891891891891891891891891891891891891891
```