Feicim: A browser and analysis tool for distributed data in particle physics

Zoltán Máthé

UCD School of Physics

A thesis submitted to University College Dublin
for the degree of Doctor of Philosophy

Head of School: Prof. L. Hanlon

Supervisor: Dr. R. McNulty

May 2012

Abstract

The Grid was developed to solve large-scale scientific problems by providing access to wide-area distributed computing resources. The Grid is used by various communities: high-energy physics, earth observation, biology, etc.

LHCb is one of the four Large Hadron Collider (LHC) experiments based at the European Organisation for Nuclear Research (CERN). The LHCb experiment produces a huge amount of data. As a single institution does not have the computing resources to handle this data, Grid computing is adopted to store and process the data. Managing PetaBytes of data in distributed environments provides a large scale of challenges related to performance, reliability and scalability. PetaBytes of data are spread over several Grid sites. The users and applications need an efficient mechanism to locate datasets based on their content, which can be achieved by associating descriptive attributes to these datasets and providing the associated information to the users. A Grid service, called the Metadata Catalogue was introduced to perform these tasks.

In physics very complex programming environments are used which are becoming ever more complex. Consequently, a lot of time must be invested in learning how to use the existing Grid resources for data analysis.

Feicim is a distributed computing tool that integrates a Bookkeeping Metadata Catalog and a representation of data and analysis. Feicim provides datasets, data-content discovery and analysis through the use of a Graphical User Interface.

This thesis presents my research into the functionality and design of Feicim. I present my current effort in the LHCb experiment to use data-content, dataset discovery and analysis within the Feicim framework.

Declaration

This thesis is the result of my own work, which was performed between 2008 and 2011 in the Experimental Particle Physics Group in University College Dublin. The Feicim tool which is presented in Chapters 5, 6 and 7 are my own work. I made explicit references to the work of others. I declare this thesis is not submitted to another qualification or any other university.

Zoltán Máthé

Acknowledgements

I would like to thank Dr. Ronan McNulty and Dr. Tahar Kechadi for the opportunity to work in the particle physics group at University College Dublin. In addition I would like to express my sincere appreciation to Dr Ronan McNulty for guidance during this undertaking who also had to read and re-read my thesis during the past few months and I am very grateful for all his efforts. A big thanks must also go to Dr. Zsolt Lazar who provided me an excellent starting point.

At CERN I would like to thank my third supervisor Dr. Philippe Charpentier for allowing me to work with the LHCb computing group. I am especially grateful to him as he guided me during this period and provided many very helpful ideas.

I would like to thanks all the DIRAC team whose were very helpful: Dr. Andrei Tsaregorodtsev, Dr. Joel Closier, Dr. Ricardo Graciani, Adria Casajus, Dr. Stuart Paterson, Dr Andrew Simth, Dr. Federico Stangini, Dr. Elisa Lanciotti.

I would like to thanks Dr. Marco Clemencic who shared very useful information which helped to integrate various LHCb software components.

I would like to thank my friends who encouraged me. We spent very good time together. A big thanks must also go to the other students at University College Dublin. Together with them we spent great times in Dublin and Geneva and they always helped to improve my English knowledge.

Finally I would like to thanks my parents and my sister who have encouraged and gave emotional support.

Contents

To Réka

1. Introduction

Computing has played an increasingly significant role in science over the years and scientific communities have initiated a new approach which allows using wide-area computing resources as a large virtual computer for solving large-scale scientific problems. The problem of giving access to wide-area computing resources is a principal issue in the contemporary scientific communities. For example, collaborations in physics, astrophysics, biology, medicine and Earth science need to store and analyse huge amounts of data. Consequently, the analysis of petabytes of data requires intensive computations. The appropriate computing and storage resources can not be ensured by one research centre. The modern approach to the solution of this problem is to utilise the computational and data storage facilities of the centres participating in the collaboration. The most advanced implementation of this approach is based on Grid technologies, which enable effective work by the members of collaborations regardless of their geographical location. Currently, the Grid technologies are used implemented in various fields of science all over the world. One of the largest Grid infrastructures is the Worldwide Large Hadron Collider Computing Grid (WLCG) which provides the production and analysis environments for four Large Hadron Collider (LHC) experiments-ALICE, ATLAS, CMS and LHCb at CERN.

The LHCb detector produces approximately 1 petabytes of data per year which has to be stored, reconstructed and analysed. In the LHCb experiment where information is dynamic in nature and not centrally managed, scalable and robust metadata management tool is essential. The Bookkeeping Metadata Catalogue of the LHCb experiment stores structured information that describes the datasets and consists of attributes such as name, time of creation, size on the disk, details of the process that produced the data, data provenance, information about the detector, data taking, information about Monte Carlo simulation, etc. A set of robust and user friendly tools must be available that allows users to manipulate the metadata information.

In addition a dedicated workload management which performs data processing (reconstruction, stripping, etc.) is required; it is called **DIRAC** (Distributed Infrastructure with Remote Agent Control). DIRAC is a Grid middleware component which manages data simulation, reconstruction and analysis on the Grid. The simulation, reconstruction and analysis tasks, or in other words, 'jobs', contain associated information which, at the end of a Grid Job, is reported to the Bookkeeping Metadata Catalogue.

The goal of the physicist is to make physics measurements using the data produced by the LHCb experiment. The physicists must extract, interpret and filter relevant information from the data in order to make physics measurements. The Bookkeeping Metadata Catalogue is used to define the input data using metadata information that describe the dataset. In particle physics the process and understanding of the data requires several processing phases and various algorithms which perform different applications. However, as programming environment become larger and more complex due to the various applications, access to the information of interest become more and more difficult. In addition, to prepare a job and run this analysis job on the Grid is not an easy task for the users.

We designed and developed a distributed analysis tool called **Feicim** (Feicim is a Gaelic word which means 'I see') that provides a visual representation of datasets, data content discovery and analysis through the use of Graphical User Interface. Feicim unifies various resources, and gives access to these resources in a user friendly environment that is robust, adaptive and scalable. It hides the complexity of the Grid and provides links between the people with Grid experience and the physicist who does not dispose a prominent knowledge of Grid computing.

We have redesigned the previous **LHCb Bookkeeping system** based on Feicim and developed a new LHCb Bookkeeping system which is currently used by hundreds of users. The LHCb Bookkeeping System integrates the Bookkeeping Metadata Catalogue, Bookkeeping Service and User Interfaces which are used to manipulate the metadata. The Bookkeeping Service provides access to the Bookkeeping Metadata Catalogue in an efficient way, while the User Interfaces (Command Line Interface, Graphical User Interface, Web User Interface) provide a visual representation of the datasets stored in the Bookkeeping Metadata Catalogue in a hierarchical format.

We have designed and implemented the **Feicim Data Browser** component which visualise the data file content, and extract relevant information. This information is used to define the different options which are essential to run data analysis. We pay particular attention to designing the *data browser algorithms* which aims to browse and extract information from event data. We designed and implemented the Feicim Plots Factory which provides an algorithm for the creation of different type of plots and presents the extracted information to the physicists. This component also allows to be defined filtering conditions on the visualization of the data.

We have designed and implemented the **Feicim Data Analysis** component which is responsible to create and submit jobs to the Grid or local machine using DIRAC or Ganga. This component automatically generates the algorithms, which perform on the dataset and collect relevant information from the event data. In addition, it is responsible to generate specific script (DIRAC or Ganga) which creates a job based on the automatically algorithms.

1.1. Outline

In Chapter 2 we present a comprehensive introduction to Grid computing and the LHCb experiment. In addition we describe the general components of the widely used Grid middleware. We pay special attention to the Metadata Services. Chapter 3 gives a general description of the Distributed Data Analysis and the existing analysis tools and metadata catalogues which are used to store metadata information about the datasets. We introduce the metadata catalogues of various scientific communities and in particular the metadata catalogues of the LHC experiments. Chapter 4 presents the LHCb Distributed Analysis by describing its important physics applications. We present the LHCb official Distributed Analysis tools DIRAC and Ganga. In Chapter 5 we introduce the Feicim tool which provides a user-friendly Command Line Interface or Graphical User Interface which aims to join together the different LHCb applications used for data analysis. In this chapter we introduce the Feicim design principles that play a vital role for joining together different applications. In Chapter 6 we present the Feicim Data Browser component of Feicim which can be used to discover data file content containing different objects. We present different algorithms which are used to discover the data file content and to present it to the users as a tree (or Virtual File System). The users can browse in the data file content and they can select different attributes. Using these selected attributes various plots can be made. In addition in this we present the Feicim Data Analysis component of Feicim which can used to define tasks and execute these tasks on the Grid or local machine. Chapter 7 presents the LHCb Bookkeeping System which stores metadata information, data provenance and provides the access to the stored information. These data are stored in a database and we describe how these data can be converted to a tree structure in order to form a Virtual File System.

2. Grid Architecture and the Worldwide LHC Computing Grid

In High Energy Physics (HEP) computing has played a vital role to process huge amounts of data, collected by different experimental laboratories such as Brookhaven National Laboratory, DESY, KEK, CERN etc. The Large Hadron Collider (LHC) introduced in section 2.1.1 is built by CERN and is the world's largest particle accelerator. Section 2.1 gives a brief description of the LHCb experiment and its key components. An appropriate computing platform is required where the scientists can execute their data intensive applications. The Grid is developed to provide this platform in order to process data which are distributed in millions of files. Section 2.2 provides a historical view of the Grid computing paradigm from the beginning to the present. In section 2.3 we formulate a few arguments for the importance of Grid Computing. Taking into account the functional aspects of the Grid we distinguish between different Grid types which are presented in section 2.4. Overviews of Grid architecture are mentioned in section 2.5 along with a review of the current standards. In section 2.6 we give special emphasis to the gLite middleware that is being used by four LHC experiments at CERN and also by other collaborations. Section 2.7 gives a brief description of the metadata service required to manage the metadata information associated to the data and provide a reliable interface to the users in order to access the metadata information in a user-friendly manner. In addition it describes the gLite Metadata catalogue. Section 2.8 presents the WLCG project that provides the data storage and analysis infrastructures to the LHC experiments.

2.1. Introduction to LHCb

The LHCb experiment is the smallest LHC experiment based at the CERN laboratory. It is an international collaboration, consisting of approximately 760 scientists from 54 institutes, representing 14 countries around the world. The LHCb experiment produces about 5 PB [1] of data per year. In order to process this huge amount of data, LHCb uses the WLCG Grid infrastructure that provides the data storage and analysis capabilities also used by the other LHC experiments.

[1]PB (petabyte) is a measure of computing resources (memory, disk, storage capacity) and it is 2 to the 50th power bytes.

2.1.1. The Large Hadron Collider (LHC)

The **Large Hadron Collider** (LHC) is a proton-proton collider that currently operates at a centre of mass energy of 7 TeV. The LHC resides in a 27km tunnel underneath the Franco-Swiss border near Genava.

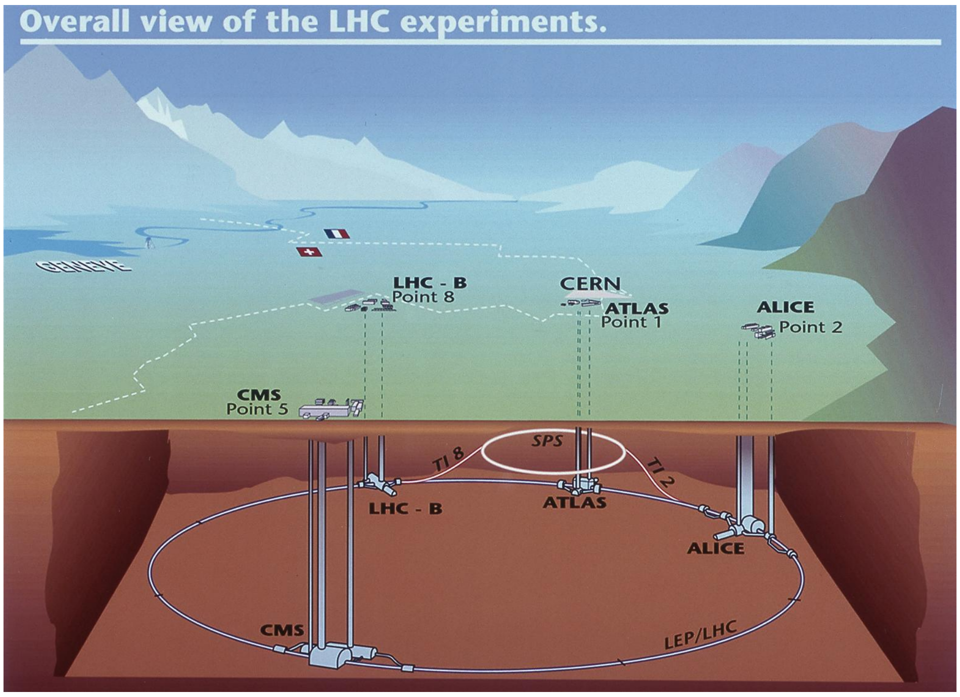


Figure 2.1.: The LHC - An overview of the SPS, LHC and its experiments, from[3]

Four major detectors have been constructed at the LHC: ALICE, ATLAS, CMS and LHCb. Figure 2.1 illustrates the layout of the LHC and its experiments. The current theory which describes the fundamental properties of matters is called the Standard Model[1]. The aim of the four experiments is to explore the current understanding of the Standard Model, to search for the Higgs particle (the missing component of the Standard Model) and to search for new physics effects beyond the Standard Model.

2.1.2. The LHCb detector

Large Hadron Collider "beauty" (LHCb) is a specialised B physics detector which has been designed principally to study CP violation used to make an absolute distinction between matter and antimatter in the b-quark sector at LHC. With large event statistics, LHCb is able to investigate rare decay processes with high precision.

The **LHCb detector**[4], shown in Figure 2.2 is a forward single arm spectrometer which consists of a dipole magnet, two Ring Imaging CHerenkov (RICH) detectors, a Vertex Locator (VELO), an electromagnetic and hadronic calorimeter, five muon chambers and four tracking

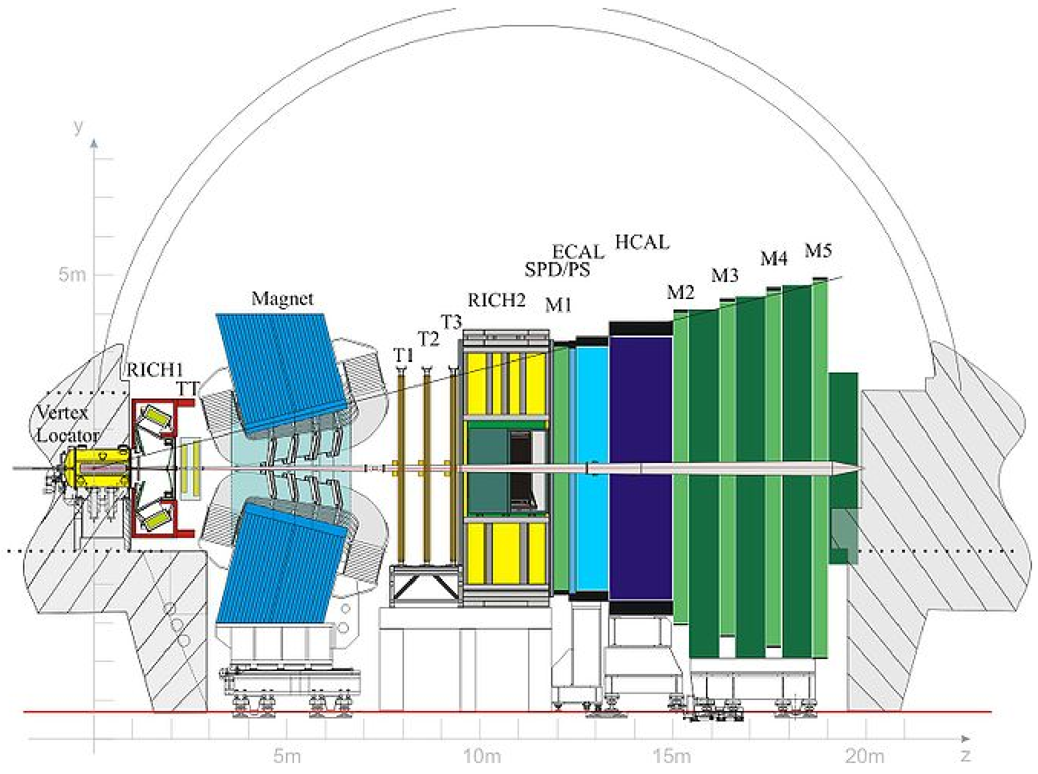


Figure 2.2.: The LHCb detector - Figure taken from[2]

stations. The LHCb detector is composed of several sub detectors that can be divided in two types depending on their task.

The **Tracking system** reconstructs particle trajectories across the spectrometer, and determines the position of the primary and secondary vertices produced in the collisions. It consists of the *Vertex Locator* [5] (VELO) which is located next to the beam pipe, close to the proton-proton interaction point. The VELO measures very precisely the tracks of charged particles allowing reconstruction of primary and secondary vertices. The three Tracking Stations (T1-T3) are divided into a central Inner Tracker (IT)[9] station surrounded by an Outer Tracking (OT)[10] stations. The IT and OT detectors are used to find charged particle tracks and measure the particle momentum which is important to precisely resolve the invariant mass of the reconstructed B mesons[8]. The *Tracker Turicensis* (TT) improves the transverse-momentum measurement. It covers the full detector acceptance and it is also used to measure the momentum of slow particles which do not reach the *Tracking Stations*[6, 7].

The **Particle Identification** (PID) system uses three detection technologies. The LHCb particle identification strategy involves two RICH detectors[11] and uses Cherenkov effects to determine the speed of the particles. The *RICH1* detector is used to measure low momentum particles, while the *RICH2* is used for higher momentum particles. The energy of photons and electrons is measured by the *Electronic CALorimeter* (ECAl) while the energy of kaons, charged pions and protons is evaluated by the *Hadronic CALorimeter* (HCAL)[12]. The *Muon system*[13] is made of five Stations of rectangular shape to identify the particles which penetrate through the calorimeters.

2.1.3. The LHCb trigger system

One of the key component of LHCb is the **trigger system**[14], which aims to reduce the 40 Mhz proton-proton collisions in order to keep only the relevant events. The data rate is reduced by a two-level trigger system to roughly 2kHz of particle collisions. The first level or *Level0* (L0) hardware trigger uses information from calorimeters and the muon system to select particles with high transverse energy and momentum, reducing the rate of accepted events to 1Mhz. The second level trigger or *High Level trigger* (HLT) is a software based trigger. The purpose of the HLT is to perform different algorithms using dedicated clusters to reduce the rate to 2 Khz. This RAW event data is temporary written to the Online system and it will be transferred to CERN computing center for future processing[15]. According to the size of the dataset, it is not efficient to use only one computing centre. The LHCb experiment uses the Grid to distribute the data around the world in order to be able the process of data in a reasonable time.

2.2. Grid Computing

Different scientific communities in physics, chemistry, genetics, mathematics etc. use extensive computing and data storage infrastructures for modelling and analysing data. To process these huge amounts of data it is not always possible to have all the necessary computing power and storage capacity in one geographical location. **Metacomputing** is a technology intended to integrate multiple computing resources, which are used by various applications as a heterogeneous computing resource. It is the ancestor of the Grid. The term Metacomputing was coined around 1987 by NCSA Director, Larry Smarr[16]. Around 1990 scientists evaluated the existing computing infrastructures and tried to define a better computing infrastructure. One of the first project in this area was named **I-WAY** (Information Wide Area Year) which aimed to virtualise resources over 16 participating sites using varying supercomputing resources and Virtual Reality display environments[17]. The I-WAY was successfully demonstrated at the **Supercomputing** '95 conference. This project strongly influenced later Grid computing activities. Ian Foster, as project leader of I-WAY, and Carl Kesselman published in 1997 a paper [18] that introduced the **Globus Toolkit**[19] which is an open source software toolkit implemented by the **Globus Alliance** for building Grids with Metacomputing. This paper becomes the pillar of current Grid projects. Independent of the architecture and technologies, the Grid was defined by Ian Foster and Carl Kesselman as[20]:

> "a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities"

A more recent definition from Wolfgang Gentzsch is[21]:

> "A Grid is a hardware and software infrastructure that provides dependable, consistent, and pervasive access to resources to enable sharing of computational resources, utility computing, autonomic computing, collaboration among virtual organizations, and distributed data processing, among others"

Therefore, the Grid combines heterogeneous and distributed computational resources to a large virtual computer through a middleware that can be used to solve various scientific problems. The benefit is faster, more efficient processing of different tasks. One of the first Grid initiatives was the loosely connected network of independent desktop computers **SETI@home** [41]. The implementation of this network was based on the **Berkeley Open Infrastructure for Network Computing** (BOINC)[42] ”volunteer computing” and Grid computing platform. SETI@home is used for data analysis searching for extraterrestrial life by running free software on several millions computers. Grid technology has the potential to affect other areas of study with heavy computational requirements, such as urban planning[31]. Computer graphics is another potentially important area for the Grid technology, because it requires large amounts of computational power.

The **Global Grid Forum** (GGF) attempted to define Grid standards and standardize Grid development. Several research and working groups approached this task, and later the **Enterprise Grid Alliance**[22] was formed to develop enterprise Grid solutions and accelerate the development of Grid computing. In June 26, 2006 the Enterprise Grid Alliance and The Global Grid Forum merged to form the **Open Grid Forum** (OGF)[23]. The OGF aims to lead the global standardization effort for Grid computing and to accelerate the adoption of Grids worldwide. Several groups attempted to define a Grid and in the end they agreed that the adoption of the **Open Grid Services Architecture** (OGSA) will define a Grid[24]. The OGSA is developed by the Global Grid Forum working group[25] and combines the emerging strengths of Grid technology by using Web Services concepts and technologies of **Web Service Definition Language**(WSDL)[27,28] and **Simple Object Access Protocol** (SOAP)[30]. These concepts and technologies are developed to deliver a coherent architecture which are defined in terms of **Grid Services**. The driving force of the Grid Services is the Web Services that conform to a particular set of conventions (interfaces and behaviours). Consequently, the Web Service is the core of the Grid Services. The **Open Grid Service Infrastructure** (OGSI)[29] is designed to provide an infrastructure layer for the OGSA to support reliable Grid services, but this has now been replaced by the **Web Services Resources Framework** (WSRF)[29]. The Open Grid Forum officially launched the **Open Cloud Computing Interface Working Group** (OCCI-WG)[93] in spring 2009 which aims to describe an API specification for remote management of Cloud Computing infrastructures. The scope of the specification will be high level functionality required for the life-cycle management of virtual machines (or workloads) running on virtualization technologies (or containers) supporting service elasticity[94].

2.3. Why is a Grid significant?

Many computer scientists have formulated the significance of a Grid. We try to summarize it in the following lines:

- Different user communities can quickly and easily create a large-scale computing infrastructure that was formerly impractical or unfeasible due to the physical location of indispensable resources.

- The Grid is a heterogeneous computing environment which mostly consists of thousands of pieces of commodity hardware that can produce similar computing resources as a multiprocessor computer, but at lower cost.
- The Grid provides a real-time response to analyse complex problems.
- Computing intensive applications that were previously inhibited by constraints on computing power, become possible using the Grid.

2.4. Grid Infrastructures

We are interested in the functional aspect of Grid infrastructure. From a functional point of view there are two different types of Grids: **Computational Grids** and **Data Grids**.
The early Grid developments were focused on computation, but Data Grids[43] are becoming more important as they provide tools for easy access and manipulation of large and shared datasets.

- A Computational Grid is a collection of distributed computing resources, within or across sites, which are aggregated to act as a unified processing platform[20]. A Computational Grid provides an architecture for transparent access to distributed heterogeneous *Computing resources*. For example: TeraGrid [35] is a Grid infrastructure which provides distributed access to United States and European supercomputing facilities. Grid-Ireland [36] provides a research platform for scientists to use computing resources throughout Ireland and worldwide.
- Data Grid technologies provide the infrastructure for users to process *large volumes of data* across geographically distributed storage sites. For example: OSG (Open Science Grid) [37] provides the computing infrastructures for different scientific communities. The DataGrid[38] project objective is to build the next generation computing infrastructure providing intensive computation and analysis of shared large-scale databases, from hundreds of TeraBytes to ExaBytes, across widely distributed scientific communities.

2.5. Grid Architecture

We now discuss the general features of a **Grid architecture**. A specific implementation will be presented in the next section. An **architecture** [39] is a formal description of a system, and defines its purpose, functions, externally visible properties and interfaces.

The first definition of the Grid architecture was presented in the article 'The Anatomy of Grid' [44]. The Grid architecture identifies fundamental system components, specifies the purpose and function of these components, and indicates how these components interact with one another[44]. It defines standard protocols and APIs to help the creation of cooperative Grid systems and portable applications. The Grid architecture is defined in terms of layers, where each layer has a dedicated scope.

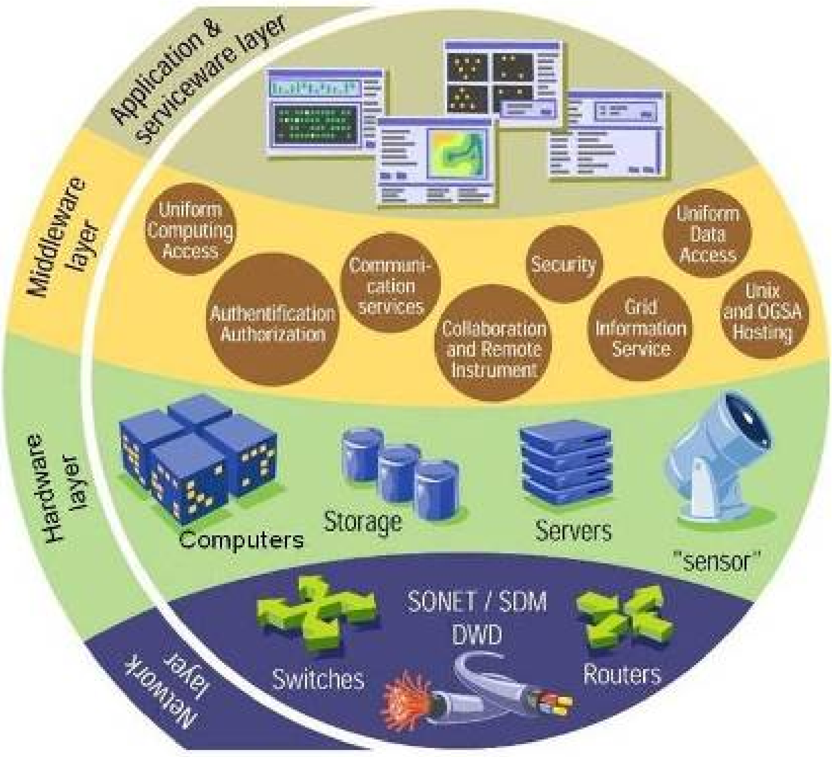


Figure 2.3.: The Grid architecture - Figure taken from "WLCG" web page[86]

Following Figure 2.3 top-down, the first level of the Grid architecture is the **Application and Service** layer. The Application and Service layer is defined by application and service software which includes portals and development toolkits as well. This layer provides many management-level functions such as accounting, and measurement of usage metrics.

The Application and Service layer is followed by the **Middleware** layer which consists of a software system that located between applications and the operating system providing protocols that enable multiple elements (network, storage, servers, etc.) to participate in an integrated Grid environment. The Grid middleware purpose is to define standard protocols and standard APIs realizing communication and data exchange between the various elements. It enables virtualization which is used to mask the heterogeneous computing resources.

The **Hardware** layer (or resource layer) [46] consists of the actual resources that are part of the Grid, including primarily servers and storage devices while the **Network layer** is the underlying connectivity for the resources in the Grid.

Since one of the main subjects of this thesis is related to the Grid computing, we discuss this part of the Grid architecture in more details focusing one by one on each service.

2.5.1. The Information System

The Grid **Information System** interconnects various Grid Middleware components by enabling dynamic and static flow of information. It provides information about Grid Services needed for various different tasks, such as discovery and monitoring features, by allowing the registration of available resources and services of member sites of the Grid.

Hierarchical directory services are used to define APIs and protocols for the Grid Information Service. The first prototype, based on the **Lightweight directory Access Protocol** (LDAP), was implemented in the Globus Toolkit[47].

The **Grid Laboratory for Uniform Environment** (GLUE)[49] defined a schema that fully describes Grid resources and services and their properties which make a Grid Information Service effective. The **GLUE Working Group**[48] which was a joint effort between Grid projects in Europe and the USA together with the OGF working group worked on the standardization of the schema. The schema which is independent of implementation, describes computing, data and storage resources and services in a uniform way.

2.5.2. Authentication

One of the most important issues is to allow for resource access without imposing complicated and site specific authentication and authorization procedures. While **authentication** is the process of attempting to gain access by allowing the users to prove their identity, **authorization** is needed for granting the user or service the possibility to perform the requested task.

Usually, the standard authentication mechanism in Grid computing uses the **public key infrastructure** (PKI) [50]. Authentication and authorization protocols are parts of the Grid middleware layer. When the PKI based Grid authentication mechanisms used, the Grid users must have a **user certificate**, which is used to generate and sign a temporary, so called *proxy certificate*, often referred to as *proxy*. The proxy certificate generated using X.509 encryption are signed by **Certification Authorities** (CA). Certificates are a combination of public key and a password protected private key pair which can be used when authenticating with remote services.

2.5.3. Workload Management System

The **Workload Management System** (WMS) defines and implements an architecture for distributed scheduling and resource management in a Grid environment by allowing participants to allocate a number of resources and then schedule the tasks on those resources. The specific kinds of tasks which require computation are usually referred to as *jobs*. The WMS is a very complex system because it has to take decisions based on the status of the Grid resources at the time a job is to be executed while hiding most of the complexity of the Grid from users.

The local resource management system/scheduler (such as **Condor**[2][53], **LSF**[51], **PBS**[52]) is responsible for scheduling task in a cluster or farm according to specific allocation policies.

A **Computing Element** (CE) acts like a Grid batch queue. It is a gateway to local nodes of a cluster or a farm. These nodes which belong to a computing farm or cluster and are attached to the different CEs, are called **Worker Nodes**.

The WMS interacts with other Grid services such as Data Management Service, Information Service, Logging and Bookkeeping Service, etc.

2.5.4. Data Management and Replication

Access to data in a Data Grid infrastructure must be transparent and efficient. In High Energy Physics the data which, were produced by the particle detector or analysis, originally stored in one location which may penalize researchers sitting in a different lab. Because, the access of the data from different labs can be influenced by the network. In order to avoid this the data must be replicated to some well defined and chosen locations. **Data Replication** is a well known and accepted technique for replicating data in order to optimize data access. We only replicate read only dataset which content will not change, otherwise the content of the replicated data will be different than the original dataset.

The variety of storage resources which are available in the hardware/resource layer requires a standard protocol to manage these resources. The **Storage Resource Manager** (SRM)[54] interface provides a middleware layer between clients and underlying resources. The SRM includes storage, collection, backup and recovery of data, but does not define a transfer protocol. The SRM allows the use of any transfer protocols (bbftp[55], gridftp[57], ftp[56]) which are supported by the clients and servers. The advantage of the protocol negotiation mechanism is that the user can control which protocol is used and it easily allows the implementation of a new protocol. The GridFTP protocol is used for data transfer over a wide area network. It uses **Grid Security Infrastructure (GSI)** (more details in Section 2.6.2) for secure data transfer and is built on the FTP specification by allowing multiple data channels for network utilization. The **Grid File System** (GFS) provides this logical hierarchical view of data and other digital entities in a Data Grid[58]. The GFS creates a logical resource name space which is independent of location and infrastructure details allowing the data to be replicated/migrated to any file system resource at any Grid site[59].

2.5.5. Logging and Bookkeeping Service

The Logging and Bookkeeping Services (LB) is used to solve the problem of job monitoring within the heterogeneous Grids environment. LB collects important events during the lifetime of the Grid jobs and stores this collected information in a reliable way. One event indicates

[2]Condor was also taken as the software core of the schedulers within the WMS, which implements scheduling tasks between the Grid sites.

a change of the job status. The jobs can have the following states: submitted, waiting, ready, scheduled, running, done or aborted, cleared. The LB has been developed within the framework of the Workload Management Service.

2.6. gLite Middleware

Various Grid middlewares have been implemented during the past few years such as Condor, gLite, ARC, etc. A special implementation of Grid Architecture is provided by gLite and we look at this in some detail because it is widely used by the LHC experiments. gLite is a Grid middleware which has been developed by the EGEE (which finished in 2010) [60] project. Currently, gLite is maintained by the European Grid Infrastructure (EGI)[61] which enables access to computing resources for European researchers. It is based on a service-oriented architecture (SOA)[62] providing the following main services: Information, Security, Workload Management, Data Management, Logging and Bookkeeping.

2.6.1. Information

The Information system aggregates information from different resources. The information are consistent with the GLUE Schema. gLite uses two implementations of the Information system: the **BDII** which is an evolution of **Monitoring and Discovery Service** (MDS) which is utilised to publish the status of resources, and discover new resources; and the **Relational Grid Monitoring Architecture** (RGMA) which is needed for publication of user level information, monitoring and accounting.

The Information system has a hierarchical structure.

Figure 2.4 shows three levels of the Information system: The users query the **top BDII** to find the information that they require. The **site level BDII** aggregates the information from all the resource level BDIIs. The **resource level BDII** provides information about Grid services running at the site. **FCR** refers to Freedom of Choice for Resources, and is a mechanisms which allows to mask sites or services which are working correctly.

R-GMA is an implementation of the **Grid Monitoring Architecture** (GMA). It provides a uniform method to access and publish information and monitoring data. Figure 2.5 shows the main components of R-GMA. The data produced by producers (i.e. the data providers) is written into a virtual database and read by consumers. The registry mediates the communication between the producers and consumers.

The Information System is used by users, site managers and middleware. The users retrieve information about geographically dispersed resources which are provided by site managers and the middleware matches job requirements and allocates the resources.

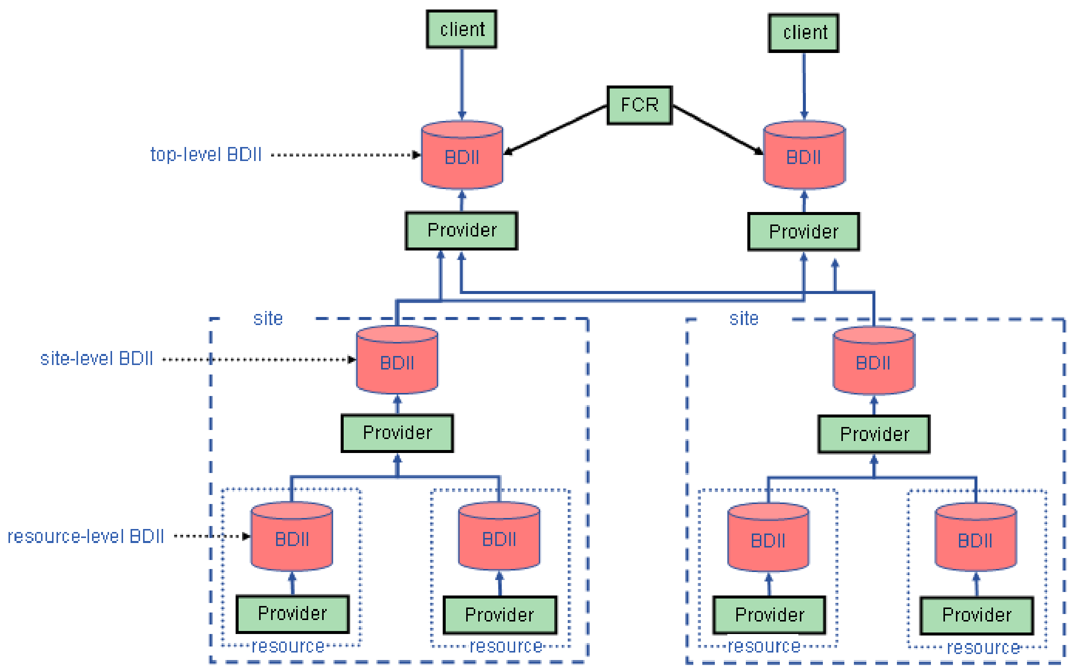


Figure 2.4.: Information system hierarchical structure - Figure taken from[63]

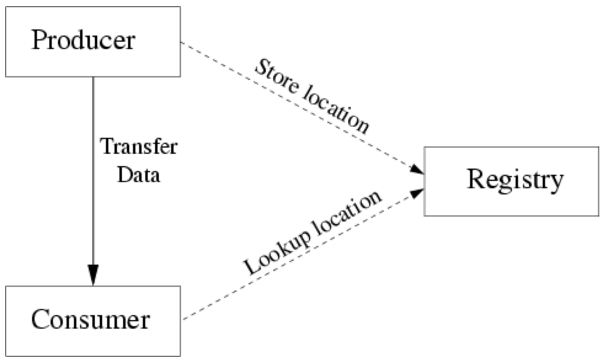


Figure 2.5.: The R-GMA architecture - Figure taken from[64]

2.6.2. Virtual Organization

We introduce the notation of a **Virtual Organization** (VO)[40] which is a new type of collaborative community to utilise geographically distributed resources. It is a set of individuals and/or institutions defined by specific applications, data and resources sharing rules. The users must join a VO in order to be authenticated and authorised to access different Grid resources. gLite uses **Grid Security Infrastructure** (GSI) for authentication and communication. GSI is based on x.509 certificates. The authorization of a user on a specific Grid resource can be done in two different ways.

- The first and simplest solution relies on the so-called **gridmapfile** mechanism. The Grid resource has a local file, which maps the user certificates to local accounts[65].

- The second way relies on the **Virtual Organisation Membership Service** (VOMS) and the **Local Centre Authorisation Service** (LCAS)/**Local Credential Mapping Service** (LCMAPS) mechanism, which allows for a more detailed definition of user privileges.

VOMS maintains and verifies user groups and role attributes. The user resource level authorization information is extracted from the proxy and processed by LCAS and LCMAPS services. The LCAS make binary authorizations at the site and resources level, and checks if the user is authorised (currently using the grid-mapfile). The LCAMPS maps Grid credentials to local credentials and maps VOMS groups and roles. The proxy lifetime is limited to a user specified certain period. MyProxy is a credential repository for the Grid that is used to store long life proxy certificates. The **File Transfer Service** (FTS) (which will be described in section 2.6.4) uses MyProxy to validates user requests and eventually renew proxies.

2.6.3. Workload Management

The gLite WMS[66] is responsible for the distribution and management of tasks (jobs) across Grid resources. It provides the job management services for matching users tasks (jobs), submitting them, monitoring their state and retrieving their output. The two core components of the gLite Workload Management System are the **Workload Manager** (WM), whose purpose is to accept and satisfy requests for job management coming from its clients and the **Computing Resource Execution And Management** (CREAM) computing element. The other fundamental component is the Job Logging and Bookkeeping Service whose purpose is to keep tracks of the tasks (jobs) (see section 2.5.5). The WMS services run a machine which is called Resource Broker (RB). The clients of the gLite WMS describe their computing tasks (jobs) using the gLite **Job Description Language** (JDL). The JDL is a specific language which is based on the Condor **ClassAd**[67] language. The ClassAd defines job attributes, data attributes and job requirements. For a user task (job) there are two main types of request: submission and cancellation. The submission passes the responsibility of the job to the WM and the WM will pass the job to an appropriate CE for execution, taking into account the requirements and the preferences which are defined in the JDL. The **matchmaking** process takes the decision about the resources to be used. The scheduling (some prefer to call it planning, or meta-scheduling) algorithms[68] for matching a job to a resource can be performed in two ways:

- **eager scheduling** matches the best resource for the job based on the requirements. This mechanism is usually referred to as ”push mode”.
- **lazy scheduling** where the job is held by the WMS until a resources becomes available and matches the most appropriate job from the task queues. This is called ”pull mode”.

These approaches are symmetrical: eager scheduling implies matching a job against multiple resources, however lazy scheduling implies matching a resource against multiple jobs. The gLite WMS also offers real-time job interaction, bulk submission, output peeking and proxy renewal.

CREAM CE

The CREAM CE[69] is a simple, lightweight service for job management operation at the Computing Element (CE) level. It implements direct job submission to the CE and jobs submission via WMS.

The management of tasks can use a **legacy one** (Web Service) and the **Basic Execution Service** (BES) interface. The **CE Monitor** (CEMON)[70] service is responsible for providing jobs state information by implementing a call-back mechanism to clients. The **Journal Manager** (JM) stores the user job commands on persistent storage to preserve them in case of system failure. The Journal Manager uses a **Batch-system Local ASCII Helper** (BLAH) interface to translate the user requests to computing resource client commands by interacting with the underlying LRMS. The gLite WWS has an additional component which is called the **Interface to Cream Environment** (ICE). The ICE is responsible for directly interacting with CREAM. The ICE receives the tasks from the WMS component and uses the appropriate CREAM methods to perform the requested operation.

2.6.4. Data Management

The gLite Data Management components provide tools for storing, cataloguing, transferring and accessing files in a Grid environment. The **Storage Element** (SE) is the service which allows a user or an application to store data for future retrieval. The files in the Grid are identified by **Logical File Names** (LFNs). Each LFN has a certain number of replicas at the same or different Grid sites, which have corresponding **Physical File Names** (PFNs) associated with them. The LFN name space is hierarchical, like a file system. Each LFN also has a **Global Unique Identifier** (GUID) which guarantees the uniqueness of the LFNs. The logical name space provides the concept of **Logical Symbolic links**; their semantics is similar to Unix file system. The **Storage URL** (SURL) specifies a physical replica of a file, while the **Transport URL**s (TURLs) are used to access files. The TURLs include a protocol determining how the files can be accessed and understood by SEs.

The gLite data services are the following:

- **Storage**: gLite supports CERN Advanced STORage manager (Castor)[71], dCache[72] and the gLite **Disk Pool Manager** (DPM)[73]. Castor is a hierarchical storage management system designed to handle millions of files which can be stored, listed, retrieved and remotely accessed using Castor command line tools. DCache is a system for storing and retrieving huge amounts of data. The DPM has been developed as a lightweight solution for disk storage management. The DPM offers a security enabled RFIO interface for posix like data access and gridFTP for data transfer. The DPM[74] architecture consists of tree components: **head node**, **disk node**, **client(s)**.

  The **head node** includes SRM, DPNS and DPM demons, plus the database service. The DPM demon handles files access requests. The DPNS handles all file and directory related metadata operations. The SRM demon exposes the SRM interface to the clients.

The **disk node** hosts the actual data and provides remote access to the data. The following data access demons run on each of these nodes: rfio[76], xrootd[77], gridftp[75] and nfs4.1[78].

The **client** talks to the *head node* and *disk nodes* for metadata operations and data access. The DPM services can be installed on a single machine or all services (including the database server) can be distributed across different machines. gLite uses the **Grid File Access Library** (GFAL) and the **LCG Utils** (lcg utils)[79] clients for data management. The **POXIS** interface of GFAL allows users to open/read/write/close files locally or remotely. The GFAL includes SRM client capabilities. The lcg-utils toolkit is built on the GFAL libraries to allow file replication with SRM based storages.

- **Catalogues** keep track of where data is stored (more details below).
- **File Transfers**: The gLite **File Transfer Service** (FTS) is a low level data movement service which provides reliable point-to-point file transfers. The *File Transfer Interface* is used to submit File Transfer jobs and monitor them, to cancel transfers, to set priority of transfers and to add, remove and list VO managers. The *Channel Management Interface* is used to set channel parameters and to add, list, or delete channels for the FTS instance. The *Status Interface* is used for monitoring the transfer activity. The clients interact with the FTS web service to submit transfer requests, which are then assigned to channels where the GridFTP transfers are executed by transfer agents. The FTS interrogates the SRM to ensure appropriate authorization and authentication.

Catalogues in gLite

Usually, if datasets are heavily used, they maybe replicated at many Grid sites. The user application does not need to know about the dataset location. Therefore the dataset name, which the user refers to, has to be a location independent LFN. These LFNs are kept by different catalogues which stores the location(s) of their files and replicas. We distinguish four catalogues (Figure 2.6):

- **File Catalogue** (FC) - manages the logical name spaces, making directories, renaming files and creating symbolic links operations.
- **Replica Catalogue** (RC) - The RC stores the replicas of Grid files. The RC interface provides the operations to add, remove, and list replicas for a file.
- **Metadata Catalogue** (MC) - Metadata is data about data. It provides additional information about the datasets. Each LFN may have additional metadata associated with them. The MC interface provides the operations to set, get and query the metadata.
- **Combined Catalogue** - maintains a persistent state of all operations which are performed across catalogues.

Each catalogue provides a well-defined interface which is used to execute a set of operations. The File Catalogue and Replica Catalogue make mappings of GUID-LFN-SURL. The

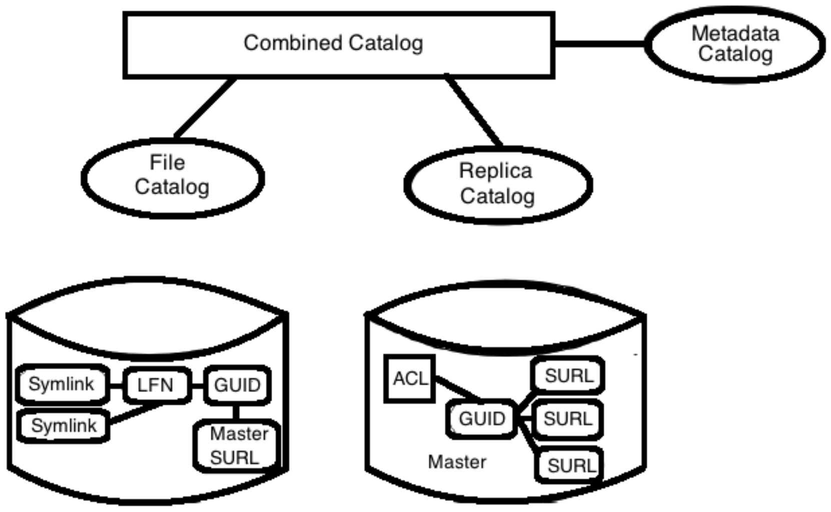


Figure 2.6.: Catalogue services with the accessible mappings. - Figure from[80]

File Catalogue allows its users to perform various operations such as creating files, renaming directories etc., while the Replica Catalogue gives access to GUID-SURL mappings by identifying all replicas of a given GUID. The Metadata Catalogue is accessible through the Combined Catalogue interface. The File Catalogue, Replica Catalogue, or most commonly the Metadata Catalogue, might be implemented and controlled by the VO directly.

LCG File Catalogue

The LCG File Catalogue (LFC)[81] is a lightweight and scalable file metadata and replica catalogue. The LFC provides a hierarchical view of the logical file name spaces. Each logical file entry has associated metadata and replica information. The LFC maps LFNs or GUIDs to a SURL. It is a high performance file catalogue which is built on Oracle and MySQL database backends and it is integrated with the GFAL interface. It offers secure authentication VOMS based authorization, and POSIX Access Control Lits (ACL) to define ownership.

2.7. Metadata Service on the Grid

The Grid allows millions of files to be spread over several storage servers on various Grid sites. The users and the applications need an efficient mechanism to describe and locate datasets based on their content. This can be achieved by associating descriptive attributes to datasets and providing this associated information to the users. The Metadata service provides information about files; it can describe any Grid entry or object. The metadata can provide monitoring information for running applications because the results from the running jobs can be published on the metadata server. The official metadata service for gLite is the **ARDA Metadata Grid Application** (AMGA)[82] metadata file catalogue.

2.7.1. AMGA metadata catalogue

The AMGA metadata service provides database access for Grid applications [83]. AMGA allows transparent storage and retrieval of metadata about data files, jobs, or general information. AMGA provides a Grid style authentication mechanism and uses streaming for data transfer that allows the retrieval of information at high speed. It gives database independence and works with any supported back-end such as Oracle, PostgresSQL, MySQL, SQLite. Currently AMGA is being used by several groups on different user communities, including High Energy Physics, Biomed[84] and UNOSAT[84, 85].

2.8. Worldwide LHC Computing Grid (WLCG)

The **Worldwide LHC computing Grid**[86] is a distributed computing infrastructure to provide the production and analysis environment for the four LHC experiments at CERN. The aim of the WLCG project is to use a world-wide Grid infrastructure of computing centres to provide sufficient computational, storage and network resources an order of magnitude higher than previous particle physics experiments.

The processing of this data requires huge computational and storage resources, which do not reside on one site (computing centre), therefore the WLCG computing services are implemented as a geographically distributed Computational Data Grid. Currently WLCG contains more than 140 computing centres in 35 countries[87]. The WLCG project was started in 2003 and has been extensively used by the four major experiments at CERN during 2009, 2010 and 2011 data-taking periods. The WLCG communities achieved excellent results for high-speed data transfer, distributed processing and storage. The LHC used approximately 92 million HS06[3] days of processing power and approximately 44 PB of disk and 52 PB of tape storage during 2010[88] and it used approximately 129 million HS06 days of processing power and 60 PB of disk and 80 PB of tape storage from January till September in 2011[89].

WLCG contains the gLite middleware component which gives the impression to the users that all of the computing resources are available in one coherent virtual computer centre.

The WLCG is built by using software which comes from various European and American Grid projects:

- **Globus:** The Globus Toolkit[91] (GT) was an early implementation of the Grid infrastructure. The Globus Toolkit is coordinated by the Globus Alliance, which was the first attempt to define Grid standards. The most popular GT2 was replaced by GT4[90]. The

[3]The HEP-SPEC (HS06) is the new High Energy Physics-wide benchmark for measuring CPU performance. For example: Intel Xeon 3 GHz and 2 core machines provide approximately 11.51 HS06.

GT services are used by other projects such as Grid File Transfer Protocol, Grid Security Infrastructure, etc.

- **GriPhyN:** The Grid Physics Network (GridPhyN) is based in U.S. and provides the important infrastructures for astronomy and particle physics to perform distributed, collaborative analysis of data. To support these tasks gridPhyN has developed the Virtual Data Toolkit (VDT).
- **iVDGL:** The International Virtual Data Grid Laboratory (IVDGL) aggregates heterogeneous computing and storage resources in Europe and Asia. iVDGL provide a data-intensive scientific computing in a single system.
- **European DataGrid** (EDG): The European Data Grid[92] project goal was to support the data access and computation needs of demonstration projects in High Energy Physics, Earth Observation Data and Biosciences.
- **PPDG**: The Particle Physics Data Grid (PPDG) is involved in several experiments in particle physics. PPDG is a joint project together with iVDGL and GridPhyN and funded by a U.S. Grid project for physics.
- **EGI:** The gLite middleware is supported by the European Grid Infrastructure (EGI).

The WLCG is currently the worlds largest Grid[97]. The WLCG infrastructure interoperates with other Grid infrastructures such as the Open Science Grid (OSG)[95] and the Nordic Data Grid Facility (NDGF)[96].

2.9. Summary

Grid technologies are being adopted in various fields of science and industry to fulfil a computational need for solving complex generic problems. The numbers of existing Grids are getting comparatively high, as research organizations, academic institutions and commercial enterprises take advantage of the existing Grid infrastructures and Grid technologies. **Cloud Computing** has been started as a new computing paradigm and has become popular in the past few years. In an article “A Break in the Clouds: Towards a Cloud Definition” Cloud Computing is defined as follows[32]: “Clouds are a large pool of easily usable and accessible virtualised resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization.” Various Cloud computing projects are being started in order to increase the computing capacity of the existing Grids[33]. Most of the time the applications which are used on the Grid are dependent on specific operating system versions and have a large number of complex software dependencies. Virtual machines have been introduced to resolve this requirement by containing applications with complete execution environments to run on resources that do not meet their operating system requirements. The Globus Virtual Workspaces project was initialised to manage ”virtual” resources[34] using standard Grid technology. **Virtualization** introduces a layer of abstraction which provides a dynamically controlled environment to the users, and provides great capabilities in managing and moving operating systems onto different hardware resources.

3. Distributed Data Analysis

Different scientific areas produce a huge amount of data which has to be analysed. **Distributed Computing** enables the users to handle this amount of data with less complexity and enough resources to its manipulation. Currently, there are two important streams of distributed computing technology development and research. The first stream is the so called *Cluster based computing* which combines large amounts of standard computers into high performance computing resources. However from our point of view the second stream *Grid Technology* is more important. It makes data storage and computing facilities, that can be geographically widely separated, available for a common, global data analysis. In addition to Grid Technology the Cloud Computing is being adopted by different scientific communities for distributed data analysis.

Different scientific communities have developed user friendly applications for job definition and management, which allow easy access to the Grid resources by hiding the Grid complex functionality.

This chapter presents four different systems that have been used to store and analyse information produced by different scientific communities. Section 3.1.1 presents a distributed computing framework for data sharing and analysis produced by various neuro-imaging scanning technologies. Special attention is paid to describe the biomedical federated database system and its data flow. Section 3.1.2 presents the Distributed Aircraft Engine Diagnostics which brought together university researchers and commercial collaborators to design and implement a fault diagnosis and prognosis system over the Grid computing infrastructure for aircraft maintenance. It provides a real time search on large distributed engine data archives through the use of sophisticated pattern matching techniques. We describe a common hierarchical architecture in section 3.1.3 for data store, data access and analysis of astrophysics data. Finally, the huge data volumes of LHC experiments also require an efficient analysis environment. To achieve this the LHC experiments designed different system for data storage and data analysis tools which are discussed in section 3.1.4. Since the analyses require input data together with the associated information (so called metadata), robust metadata management tools are essential. Each experiment has designed their Metadata Catalogues and tools needed to manipulate these data. Section 3.2 presents each experiment's database systems which were used to develop their Metadata Catalogue.

3.1. Distributed data analysis in different scientific communities

We have studied the designs and implementations of distributed data intensive applications used by different scientific communities. The data intensive applications of these experiments use huge amounts of data that have to be analysed in order to produce different measurements. When the experiments can not handle the data themselves, they either need to use external computing resources or buy new computing resources. The latter solution is very expensive and can be avoided using the Grid.

They have adopted Grid technology, re-used existing Grid services or implemented experiment specific services to provide a reliable distributed computing framework for the sharing and analysis of data. We distinguish between three types of experiment according to how the data is stored:

1. Experiments that stores unstructured data: The Storage Resource Broker (SRB)[102] logical distributed file system is adopted to store unstructured data in addition to metadata information. In addition they use various Grid systems (Information System, Data Management System, etc.) Examples of these are the Biomedical Informatics Research Network and Distributed Aircraft Engine Diagnostics described in section 3.1.1 and 3.1.2 respectively.

2. Experiments that store the data in a Relational Database Model: Relational Database Management Systems (RDBMS) are used as a data repository that can store information. The data repositories are distributed across different laboratories. On top of these repositories a service is implemented which allows users to interact with the database to retrieve datasets. An example of this is the World-Wide Telescope described in section 3.1.3.

3. Experiments that store data using Grid Storage element. An example of this is given by the LHC experiments described in section 3.1.4.

Various methods such as data mining[114], pattern matching, etc. can be performed on the data. Consequently, a service is required that supports different data mining techniques such as classification, regression, clustering etc. Taking into account the experiment requirements, they could decide to use existing services, or instead design and develop their own data mining service.

In addition, in order to manage the large datasets efficiently each experiment uses a metadata service, which provides descriptive information or metadata about the datasets that need to be managed.

The Metadata Catalogues are based on a RDBMS such as Oracle, MySQL. In addition, NoSQL database management systems (such as MonogoDB[103], ChouchDB[104], etc,) have been introduced and used by two LHC experiments (ATLAS and CMS) .

Each experiment has different information that can not be stored using a general Data Model which can store their experiment specific information. Consequently, using a common Metadata Catalogue is not always trivial. Therefore, each experiment designed and developed their own Metadata Catalogue that conforms to their requirements.

They implemented different types of User Interfaces (more detailed description found below) that allows users to interact with the system.

Each experiment has common characteristics:

- They have to handle huge amount of data in a reasonable time.
- They adopted to Grid technology.
- They have different catalogues to store metadata information.

More detailed information about how the experiments handle the data will be introduced in the next sections. The experiment specific services, applications, etc. will be presented. In addition, the answer of why they adopted the Grid and how they using the Grid are provided.

3.1.1. Biomedical Informatics Research Network

The Biomedical Informatics Research Network (BIRN) [98] was designed as the first national cyberinfrastructure for biomedical research. The Morphological BIRN, the Functional BIRN and the Mouse Brain BIRN produce 500 GB data/day [99]. Three core components are implemented for managing and documenting acquired and derived data in a federated environment: the **Human Imaging Database** (HID) [100] for distributed/federated relational database support and web-enabled GUI, the **XML-Based Clinical experiment Data Exchange** (XCEDE2)[101] for structured data or metadata storage and exchange, and the **data publication scripts** to organize and transfer data to the distributed file system and send the appropriate URL links to the HID database. The **BIRN Portal and Mediator** interface are used to access the distributed databases, query the distributed data stores, and provide Grid computing and application interface support.

Data can be produced by **scanner application** or **Data Input terminal**. The scanner application data is uploaded to the SRB and the metadata information sent to the HID database. The Data Input terminal allows the researchers to define their clinical data which was collected by the imaging experiment, and the user defined data will be uploaded to the HID database. The image analysis software can be used to extract the images from the SRB by using HID database queries, execute the analysis, upload the results to SRB, and send the metadata information to the HID database. The external investigators can query the BIRN portal and Mediator to extract datasets, perform their analysis, and send the results back to the federated database system.

3.1.2. Distributed Aircraft Engine Diagnostics

The **Distributed Aircraft Maintenance Environment** (DAME)[113], brought together university researchers and commercial collaborators to design and implement a fault diagnosis and prognosis system over the Grid for aircraft maintenance. It has to handle terabytes of vibration and performance data per year.

DAME has implemented a **Data Mining Service**[117] which consists of the **Advanced Uncertain Reasoning Architecture** (AURA)[116] pattern match system. AURA provides a real time search on large distributed engine data archives through the use of sophistical pattern matching techniques. The **Data Repository** stores all raw engine data along with any extracted and AURA encoded data[115]. A **Metadata Catalogue** is maintained which maps logical handles to physical file locations. The system specific metadata can be added to each record which means that the Metadata Catalogue can be queried in data centric way, rather than in a location centric fashion.

3.1.3. The World-Wide Telescope

The World-Wide Telescope[105] (also called the Virtual Observatory) seeks to unify the world's astronomy archives, which were collected by various telescopes, into a giant database and provides software tools to manipulate these data.

The **archives** store the astronomical data (text, images and raw data or files) in relational database as well as metadata[106] which describe not only their physical units but also the provenance of the data. The archives provide a **web service** interface for data query which involves data transfer and heavy computation. Each archive declares its services with one or more registers that will be used by **portals**:

- **M**ultimission **A**rchive at **ST**ScI (MAST) portal: is provided by National Aeronautics and Space Administration (NASA) and supports a variety of astronomical data archives [108]. The data archives are provided and distributed by the Space Telescope Science Institute (STScI)[109].
- Open SkyQuery portal: allows the scientists to query the astronomical catalogues using SQL[112] query language[111]).

The portals purpose is to answer user queries by integrating data from many archives.

The portals allow the scientists to generate and execute their queries. This requires a basic knowledge of the SQL language.

The astronomy data on the Grid generally resides in read intensive relational databases that are accessed by associative query interfaces that represent subsets of data.

The AstroGrid[110] project provides a set applications to enable astronomers to manipulate the data (find data, share files, query databases, build and run scripts etc.).

3.1.4. LHC distributed analysis

Distributed data analysis using Grid resources is core to the success of the current particle physics experiments. The LHC experiments are using two approaches for data analysis: *asynchronous* (analysis on the Grid) and *synchronous* (interactive) analysis through the experiment's computing frameworks. Huge data volumes of LHC experiments (ALICE, ATLAS, CMS, LHCb) require an efficient analysis environment to achieve large-scale data-intensive analysis. The most natural approach is to move the processing close to the data which means to run analysis program at a computational centre where data is kept.

The Computing Model of the LHC experiments

The **Models Of Networked Analysis at Regional Centres** (MONARC)[118] for the LHC Experiments project proposed the use of a hierarchical model, that could share the processing and storage responsibility across member countries and institutes. MONARC introduced the concept of a multi-tiered computing infrastructure with well defined functionality and connectivity at each tier. The central site in the hierarchical model is the Tier-0, which has a strong computing facility and from which the experimental data originates. The **Tier-1s** are large regional centres which represent a country or geographical region and typically have a national scope. The **Tier-2s** are smaller computing centres such as universities or institutes which represent a part of a country or geographical region. The **Tier-3s** have more limited computing capability. The Tier-3s include individual computing clusters that belong to physics groups which are focusing on the solution of one or a few specific computing problems. The **Tier-4s** are individual machines which are desktops or laptops of group members where data analysts actually work.

Each LHC experiment has a **Computing Model** that is based on the MONARC report, and whose purpose is to define the expected **data flow** and the estimated computing and storage requirements which is required at each Tier while the **work flow** describes how the experiments process the data. The main concept of the Computing Model is that every physicist should have equal access to the computing and storage resources which are needed for the processing and analysis of the experiment data.

The Computing Model of the ALICE, ATLAS and CMS experiments are similar: the data recorded by the detectors are immediately transferred to the CERN Tier-0 and distributed to Tier-1 sites. The first reconstruction takes place at CERN and after these reconstructed data are distributed to the different Tier-1's where the second and third reconstructions take place. The Tier 1 and 2 sites provide the resources for data reduction, analysis and Monte Carlo simulation. In particularly the Tier-2's sites are used for Monte Carlo production and analysis. The Tier-3 centres are used on demand for physics analysis. The datasets in these sites are handled locally; they are not available to the whole collaboration. In addition each experiment has decided where their calibration, detector optimization will take place. In most cases they use the Tier-0.

The LHCb Computing Model is different than the models described above. A more detailed description is provided in chapter 4.

Computing frameworks used by the LHC experiments

The experiments use the gLite middleware, but due to their specific tasks, they have built their own frameworks to support their data flow. The LHCb experiment has built a common framework, which is called **DIRAC** to provide data and workload management system to fulfill all terms of the data flow that requires the execution of computing tasks in the distributed environment[119]. The ALICE experiment also developed a single framework within the **ALIEN** [121] project, which is an implementation of distributed computing infrastructure needed to manipulate the ALICE data flow [120]. The ATLAS [122] and CMS [123] experiments are using most of the computational and storage Grid resources. To split the responsibility of their Grid computing project, they have adopt specialised systems for different aspect of their computing models. The ATLAS communities are using the **Production and Distributed Analysis** (PANDA) [124] framework which is based on DIRAC to allow more effective resource management. The CMS experiment are using **ProdAgent** [126] and **CMS Remote Analysis Builder** (CRAB)[127] systems which are developed to fulfil all aspects of the data flow. The **Batch Object Submission System** (BOSS)[125] is used by CRAB and ProdAgent to provide job submission and monitoring. ALICE, ATLAS and CMS distributed analysis will be discussed briefly below in the next sections while LHCb will be discussed in detailed in chapter 4. We focus on their Metadata Catalogues in section 3.2.

ALICE distributed environment

The ALICE collaboration has developed the AliEn (AliCe Environment) framework which is a set of middleware tools and services that implement a Grid infrastructure to fulfill all aspects of their Computing Model[120].

The analysis and reconstruction algorithm affects the time which is needed to analyse and reconstruct events. To define the input data, physicists have to query the **AliEn File Catalogue** which is used to store the files and associated metadata. Different algorithms can run on the selected input data by using the AliEn framework to submit interactive or Grid jobs. The other possibility is to run fast analysis and reconstruction using the **Parallel ROOT Facility** (PROOF) [128] infrastructure. PROOF enables interactive parallel processing of data on clusters of computers or multi-core machines. The ALICE collaboration has developed the **Monitoring Agents in A Large Integrated Services Architecture** (MONALISA)[129] distributed computing oriented monitoring system to monitor the computing resources.

ATLAS distributed environment

ATLAS splits the Grid computing projects into four areas: Tier-0 processing, *Distributed Analysis, Distributed Production and Distributed Data Management*. Since the main purpose of this section is distributed analysis, we concentrate on **ATLAS Distributed Analysis** (ADA).

ADA is based on a client-service architecture and is required to work with different Grid infrastructures. The analysis services manage the processing of data while the catalogue service records data and their provenance information. **Distributed Interactive Analysis of Large Datasets** (DIAL)[130] is a project that was originally founded to provide interactive analysis of large datasets. The main purpose of this project is to allow users to submit and monitor their jobs.

Another production and distributed analysis system has been developed to meet ATLAS requirements for petabyte scale production and distributed analysis processing which is called PANDA[131]. PANDA integrates the ATLAS **Distributed Data Management** (DDM) system, a monitoring system for production and analysis operations. PANDA has a generic, high level data-driven PANDA **Workload Management System** which meets the requirements of large scale data processing. PANDA became the main analysis and production system for the ATLAS experiment. Consequently, DIAL provides an interface for job submission to the PANDA middleware.

Two main frontends have been developed for job submission which allow the users to submit analysis jobs to different computing resources. pAthena is a glue script to create and send jobs to the PANDA Workload Management System[132]. The other application is called Ganga [133] which provides a simple and consistent way of preparing, organizing and executing analysis tasks within PANDA and different Grid resources.

CMS distributed environment

The CMS experiment has developed a set of tools for distributed analysis. The CMS **Workload Management System**'s purpose is to interpret user requests, create jobs that process data, submit them to local or distributed systems, provide monitoring tools to monitor their status and allow users to retrieve their outputs[134].

The **Production Agent** (ProdAgent)[136] is a tool to perform these tasks in a controlled environment, and provides a capable large scale coherent production system which meets the simulation and data processing requirements of the CMS experiment while the **CMS Remote Analysis Builder** (CRAB) [135] has been developed as a user friendly interface to handle analysis in a local or distributed environment. The CRAB hides the complexity of interactions with the Grid and CMS services, providing data discovery and location, job preparation, job splitting, job submission, job monitoring and output data retrieval functionalities. The interaction with the Grid can be either direct with a thin CRAB client or using an intermediate **CRAB Analysis Server** which is made of a set of independent components for managing user tasks.

The **CMS Dashboard** monitors the activities of the CMS experiment on the distributed infrastructure by providing a uniform and complete view of job processing and data movements. The CMS Dashboard has a set of tools that provide access to the information that was collected from various sources through web interfaces.

3.2. Metadata Services on the LHC experiments

In the LHC experiments where information is dynamic in nature and not centrally managed, scalable and robust metadata management tools are essential. The Metadata Catalogues of the LHC experiments store structured information that describes the datasets and consists of attributes such as name, time of creation, size on the disk, details of the process that produced the data, data provenance, information about the detector, data taking, etc

Sets of tools are available that allow users to manipulate the data e.g. to:

- query the dataset;
- insert, delete a dataset;
- retrieve different statistics; and
- determine the history of the dataset.

Each experiment designed and implemented various User Interfaces that can be used to discover the Metadata Catalogue content. The user who queries Metadata Catalogues is not expected to have knowledge about the database structure (relation between different database tables etc.) or database locations, because the architecture allows them to express queries in terms of the application semantics.

Since Metadata Catalogues are central to this thesis we now describe the Metadata Catalogues for ALICE, ATLAS and CMS. The Metadata Catalogue for LHCb will be described in Chapter 7.

3.2.1. AliEn File and Metadata Catalogues

The ALICE File and Metadata Catalogues has been implemented within the AliEn framework. The File and Metadata Catalogue is designed to allow each directory node in the hierarchy to be supported by different database engines. The AliEn relational databases can be installed on different machines, Figure 3.1 shows the structure of the AliEn File and Meta Catalogues. It stores the LFNs and provides the mapping to the corresponding PFNs. The PFN entries in the catalogue describe the physical location of the files and contain the name of the storage element and the path to the local file. The AliEn Metadata Catalogue has a specific functionality, and keeps the job output for future retrieval.

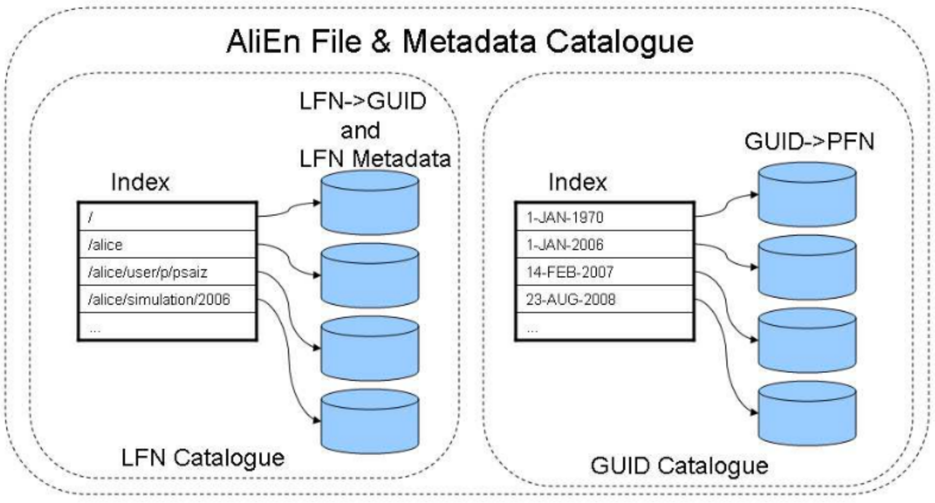
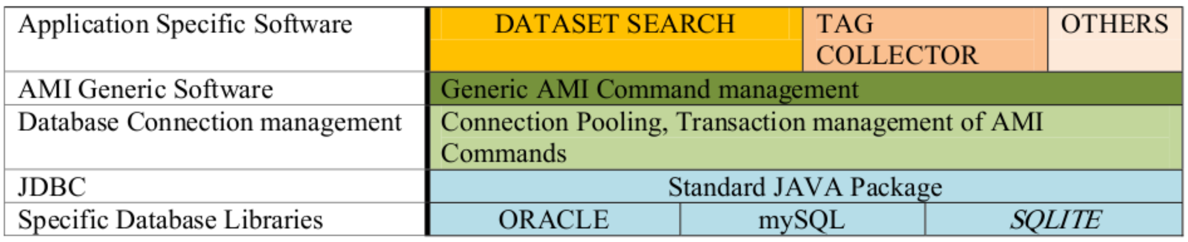


Figure 3.1.: The AliEn File and Metadata Catalogue - Figure taken from[137]

The interface to the File Catalogue is similar to a UNIX file system and is used by almost all components of AliEn. The interaction can be carried out from a command line interface with the most common UNIX shell commands or via a Web portal. AliEn provides a Perl, C and C++ API to access the AliEn framework[138].

3.2.2. Atlas Metadata interface (AMI)

AMI was chosen as the ATLAS dataset selection interface to provide all the necessary functionality for the users to discover the event data by using metadata criteria.



Application Specific Software	DATASET SEARCH	TAG COLLECTOR	OTHERS
AMI Generic Software	Generic AMI Command management		
Database Connection management	Connection Pooling, Transaction management of AMI Commands		
JDBC	Standard JAVA Package		
Specific Database Libraries	ORACLE	mySQL	*SQLITE*

Figure 3.2.: The AMI architecture - Figure taken from[140]

AMI has been implemented as a generic database management framework which allows parallel searches over many catalogues[139]. Figure 3.2 shows the AMI layered architecture. The three lower level packages wrap the connections to the databases managing transactions: connection pooling; transmission of SQL commands; and recuperation of query result. The top layer of the software contains application specific packages with knowledge of the application semantics[140],[142].

AMI currently uses MySQL and Oracle database backends which are deployed on geographically distributed servers. AMI is a web application which is implemented in Java but also provide a SOAP Web service for clients. A python interface, which is called pyAMI[140], is integrated into the Ganga distributed analysis tool.

The purpose of AMI is to extract and correlate the information which is needed for dataset selection from different *data sources* such as ATLAS production database, ATLAS Tag databases [141] etc. This extraction of data is done by a special task server which runs tasks as Java threads.

The AMI web interface provides several types of *dataset search*: a simple search on the dataset name or on a selection of fields; a more complex search which allows the users to specify various requirements; and a hierarchical search for datasets, where the user selects one parameter at a time. Each user has different rules such as read, write or update to access to the catalogues using a personalized home page. AMI has implemented an internal bookmarking system to save the user queries. The results of the queries are always recalculated dynamically by AMI tasks which ensure that the users always get the latest information in a reasonable time.

3.2.3. The CMS Dataset Bookkeeping Service (DBS)

The Dataset Bookkeeping Service[143] has been developed to catalogue all data produced by the Monte Carlo productions and the CMS detector.

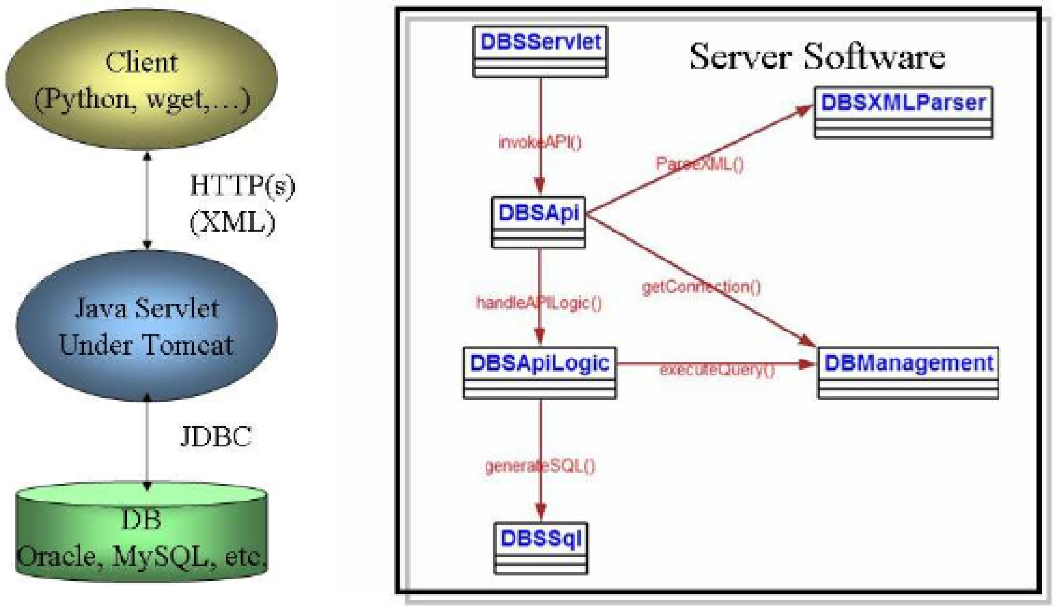


Figure 3.3.: The DBS system architecture and server software design - Figure taken from[143]

The database schema is designed to store all the data processing history and event selection criteria. Each datasets in the database belongs to different categories such as primary dataset, processed dataset, analysed dataset etc. which are crucial for the data processing. The DBS system is a multi-tier web application that has been designed to allow the usage of various database technologies (Oracle, MySQL, and SQLite). Figure 3.3 illustrates the DBS architecture which consists of several servlet[144] modules. DBS supports a hierarchical deployment model which is convenient for working groups and users allowing them to work within the local or global database instances.

DBS consists of a database and the services used to store and access metadata. The query system is based on a query language that hides the complexity of the underlying database structure. The DBS Query Language (QL)[145] is used to translate the user request into the SQL language which can be executed on the database backend.

On top of the existing database services the CMS **Data Aggregation System** (DAS) has been implemented to support keyword based search queries with the ability to use conditional operators. The DBS Query Language has been adapted to the CMS DAS system in order to build various database queries.

The CMS DAS system uses a non-relational data structure that describes the Data Model. It collect metadata information from various CMS catalogues[123, 146] and stores them in a document oriented databases which is called MongoDB[103]. Several systems are used for reliable data retrieval: *DAS Mapping DB* is used to track and collect information about data services, translate the input and output parameters into a DAS keys, and stores mappings between CMS DAS records and their user interface representation; the CMS *DAS Analytic DB* collects information on user requests against the system; while the *DAS caching system* is used to dynamically fetch and aggregate data upon user request using the DAS cache and the DAS merge independent databases[147].

3.3. Summary

Different scientific communities use distributed computing infrastructures to analyse their datasets. Each experiment evaluated the exiting technologies used for data processing. Depending on the size of the datasets of the experiments, they adopted the most appropriate infrastructures which fulfill their specific requirements. Most of the cases the Grid or some specific Grid services such as resource and data management, security, information services, etc. are adopted to their systems.

To handle large datasets efficiently required a metadata catalogue which keeps information about the datasets. The datasets of the experiments are different and very complex. Consequently, it is not always possible to implement a common metadata catalogue. They developed their own specific metadata catalogues. The metadata catalogues of the experiments based on RDBMS, therefore NoSQL adopted when the RDBMS was not efficient. In addition, they developed different tools and services to handle the metadata. When the experiments only store very basic metadata information they use AMGA metadata catalogue which is a specific service of gLite.

Each experiment developed their experiment specific applications to process the datasets. These applications perform on the Grid and analyse the data which is stored by different systems: RDBMS or a logical distributed file system.

4. Distributed analysis in the LHCb experiment

LHCb data can be produced by Monte Carlo(MC) simulation or by the LHCb detector. These data have to be stored and processed using different applications. Section 4.1 presents the Gaudi framework which provides a common framework to the LHCb applications. A detailed overview of these applications is presented in section 4.2. The Work flow describes how the data is processed using these applications. The Data flow describes the ‘flow’ of data from the detector to the final destination. A brief description of the Work flow and Data flow is given in 4.3. In section 4.4 the LHCb Computing Model is introduced which is based on a distributed multi tier centre and describes the way LHCb uses the resources at the collaborating sites. Section 4.5 presents the computing resources used in 2010, 2011 and the estimated resource requirements from 2012 to 2013. Section 4.6 describes the DIRAC software which is a middleware component which manages the LHCb activities on the Grid. In section 4.7 the Ganga user friendly interface is presented, and is widely used to allow access to the geographically distributed computing resources.

4.1. Gaudi framework

Gaudi[148] is a software framework which was initially developed by and for the LHCb experiment and has since been adopted by ATLAS together with other experiments (GLAST [149], HARP[150], DayaBay[151], MINERvA[152]). Gaudi provides a flexible framework which fulfills various criteria of the computing activities such as data processing, simulation, reconstruction and analysis. The architecture ensures changing requirements can be adapted to over the lifetime of the LHCb experiment. Gaudi provides a common set of services and components with clearly defined interfaces which are used by all event processing applications. Figure 4.1 shows different Gaudi categories of components: algorithms, services and converters.

The Gaudi **Interface Model** provides well defined interfaces that separate the components from each other. This approach allows transparent inclusion of new software or technologies which are developed on the same interface. The well defined interfaces allow run-time dynamic library loading to be used in the software applications. A clear separation exists between **Data** and **Algorithms** which means an important role of the Gaudi architecture is the de-coupling of objects describing data and the methods for manipulating the data. *DataObjects* are containers of data quantities which can be persistent or transient. The **Algorithms**

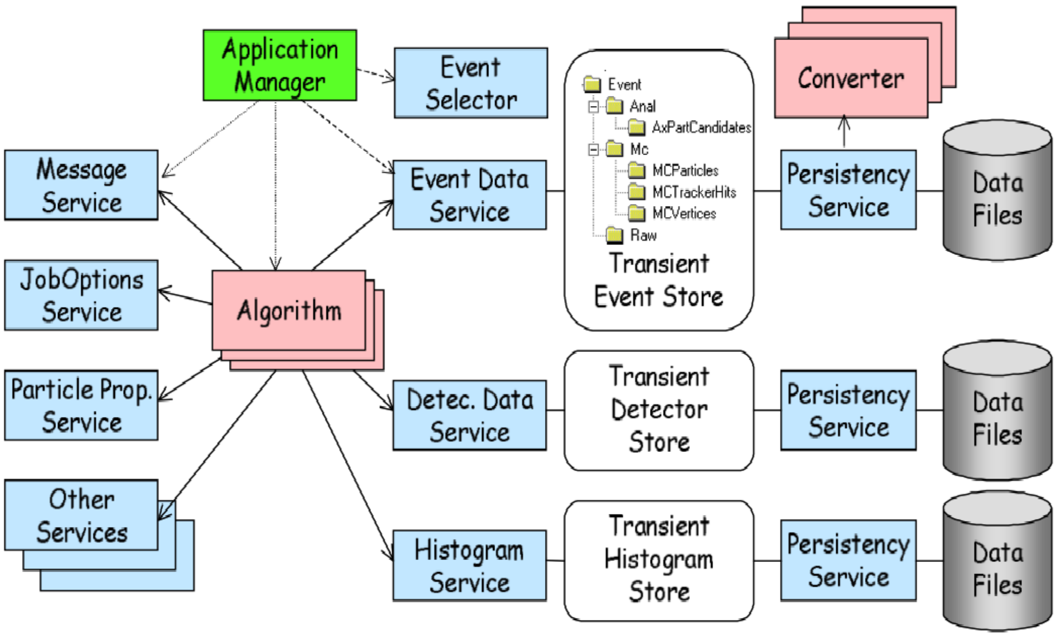
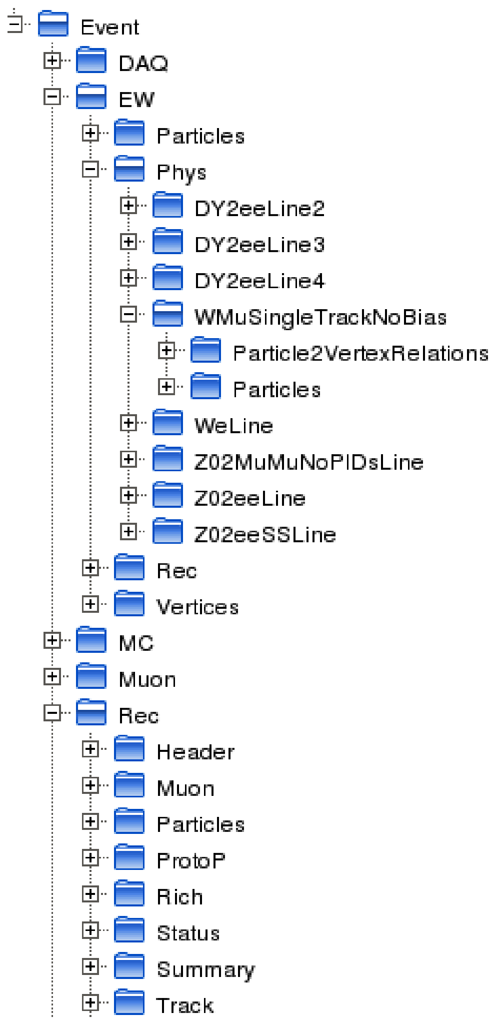


Figure 4.1.: The Gaudi framework architecture - Figure taken from[153]

perform event simulation, reconstruction and analysis on the *DataObjects* which are organized in various **Transient Data Store**s. The Algorithms can be instantiated and executed based on requirements of applications. They get access to the data to be processed through the relevant Transient Data Store. The data are distributed over three stores:

- **Transient Event Store** (TES) stores the event data which is only valid during the time which is needed to process one event.
- **Transient Detector Store** (TED) contains the detector data which includes the detector description such as geometry, calibration, etc, which are valid for the time to process a series of events.
- **Transient Histogram Store** (THS) stores the statistical data produced during event processing.

Gaudi has the following basic containers: *KeyedContainer, SmartRefVector, vector, ObjectVector*. It provides various components which offer all the **Service**s needed to manipulate the data object in the Transient Data Store. For example: the Event Data Service is used to retrieve a *DataObject* from the Transient Event Store. To read/select events the Gaudi *Event Selector* is used.



```
Event
  DAQ
  EW
    Particles
    Phys
      DY2eeLine2
      DY2eeLine3
      DY2eeLine4
      WMuSingleTrackNoBias
        Particle2VertexRelations
        Particles
      WeLine
      Z02MuMuNoPIDsLine
      Z02eeLine
      Z02eeSSLine
    Rec
    Vertices
  MC
  Muon
  Rec
    Header
    Muon
    Particles
    ProtoP
    Rich
    Status
    Summary
    Track
```

Figure 4.2.: Transient Event Store - an example of how the data is stored in the TES; each folder is the name of a given DataObject. For example: Event is a folder which identifies a data object that also contains identifiable objects (DAQ, MUON, REC, etc).

Event Model

The Event Model[154] is an LHCb-specific component within the Gaudi framework which describes the LHCb event data structure, both simulated and real. The LHCb Event Model is defined as the set of classes and relationships between classes which allow persistency. The objects can be written on the storage and read back by another program. The Gaudi TES is used to exchange event data inside the event processing loop. Algorithms simply retrieve their input data from the TES and publish their output data to the TES. Data in TES is organized in a tree-like format. Figure 4.2 gives an overview of the tree format.

4.2. Physics Applications

Data processing applications are collections of software packages which perform particular steps in the work flow. The LHCb applications (Gauss, Boole, Brunel and Davinci) are built within the Gaudi framework and can be executed in a standard environment. Each application shares and communicates via the Event Model and produces and/or consumes data. The following sections give a short description of these applications.

4.2.1. Gauss

The LHCb detector simulation application is called Gauss[155, 156] and is the first step in the simulation of physics data. Gauss is used to study the performance and the behaviour of the detector in response to proton-proton collision events. Within Gauss there are two independent phases: Generator Phase and Simulation Phase. The **Generator Phase** is split into two parts[157]. The event generation of the proton-proton collisions uses various generators such as PYTHIA[158, 159], HIJING[160] etc. while b particle decays use the EvtGen[161] generator. The **Simulation Phase** uses Geant4[162] to simulate the detector response to particles produced by the Generator Phase. The output of the Simulation Phase uses the Event Model format.

4.2.2. Boole

The purpose of Boole is to provide a simulation of the response of the LHCb detector to the simulated physics event. Boole[163] simulates the digitization of the detector using hits generated by the Gauss application. The Boole step includes simulation of the detector response and of the readout electronics, as well as of the L0 trigger hardware. The output produced by Boole is in the same format as real data coming from the detector after the L0 trigger.

4.2.3. Brunel

The Brunel application reconstructs the digitised output from the Boole application or real data from the LHCb detector. Brunel integrates complete pattern recognition and performs particle identification. It produces output files containing all reconstructed items such as tracks, clusters and performs particle identification. The Brunel application consists of a set of *sequential phases* (Initialisation, Reconstruction, Finalisation) and a series of independent *processing phases* (Reconstruction, Relations and Monitoring). The sequential phases are used for sequencing reconstruction algorithms. Each sequential phase can contain sequences of subdetector algorithms. The processing phases use algorithms which can run on both real and simulated data. The **Reconstruction phase** begins with clustering in the tracking detectors. The tracking pattern recognition proceeds in several steps and uses the clusters as an input. For simulated data, the reconstruction phase is followed by the **Relation phase** where clusters are associated with Monte Carlo particles. The **Monitoring phase** is used to study a specific sub-system's performance and may be executed selectively.

4.2.4. Davinci

Davinci is the LHCb experiment analysis framework which supports selection of events and analysis on real and simulated data. The selection of events can be specified through job options or by using user supplied or predefined algorithms. These algorithms manipulate physics event objects that are described in terms of particles and vertices. Davinci is configurable to use several output formats: an output file containing event data selected to be used for later processing or Analysis Object Data (Ntuples) files containing physics objects for later processing.

4.3. The LHCb experiment Data flow and Work flow

The processing of event data occurs in several phases which normally follow each other in a sequential manner. The Work flow Model describes how the phases follow each other while the data flow describes the flow of the data from the detector to the final destination. The terminology of each step is discussed bellow.

Simulated data

The simulated data are produced from a detailed Monte Carlo model. These RAWmc datasets contain simulated hit information and extra 'truth' information which is used to record the physics history of the event and the relationship of hits to incident particles.

The simulated datasets are larger than real data but nevertheless have an identical format and are processed using the same reconstruction software.

RAW data

As mentioned in Section 2.1.3, data is collected by the LHCb detector with the LHCb Trigger System to select events of interest. RAW data are transferred to the CERN Tier-0 computing centre in quasi-real-time for archiving and future processing.

Data Reconstruction

Once the RAW data has been created, whether real or simulated, it is reconstructed using the Brunel application in order to provide physical quantities and information about particle identification. The reconstructed events are typically 50 Kb each and are written to output files called *reduced* Data Summary Tapes (rDST) or *slim* Data Summary Tapes (SDST). The rDSTs contain all the information which are needed by the next step, while the SDSTs do not.

The reconstruction is performed after the RAW data is transferred to CERN Tier-0 computing centre and distributed to the largest Tier-1 computing centres.

Data Stripping

The events stored in the SDST or the rDST are analysed in order to produce different streams for future analysis by using pre-selection criteria proposed by physics working groups, called stripping. The events are processed using these preselection algorithms, using Davinci. The output of the stripping is a different DST stream which belongs to a physics working group. Stripping is performed when the reconstructed data becomes available and is repeated for any reprocessed reconstructed files (SDST,rDST) or when the preselection algorithms has changed.

Merging

The events in the different DST streams are stored on various Storage Elements. Sites require file size are grater than 2 GB file which imply these files must be merged in a proper way. The Merging is performed in real time when enough stripped files are available from the previous step.

Data Analysis

Physics analysis uses the DSTs produced by the previous phase. Data Analysis is performed using the Davinci package and produces personal Ntuples or DSTs, which are analysed in order to obtain final physics results. It is assumed that each physics working group or physicist is performing a separate analysis on a specific stream. These outputs can then be shared by physicists collaborating across institutes and countries.

4.4. The LHCb Computing Model

The LHCb computing model is based on a distributed multi-tier regional centre model and is designed to maximise the use of resources available to the collaboration. LHCb uses the resources which are located at national Tier-1 and regional Tier-2 centres in order to perform data processing and analysis. The LHCb member states pledged resources to WLCG which is used for data processing. Figure 4.3 shows the role of each Tier of computing centres.

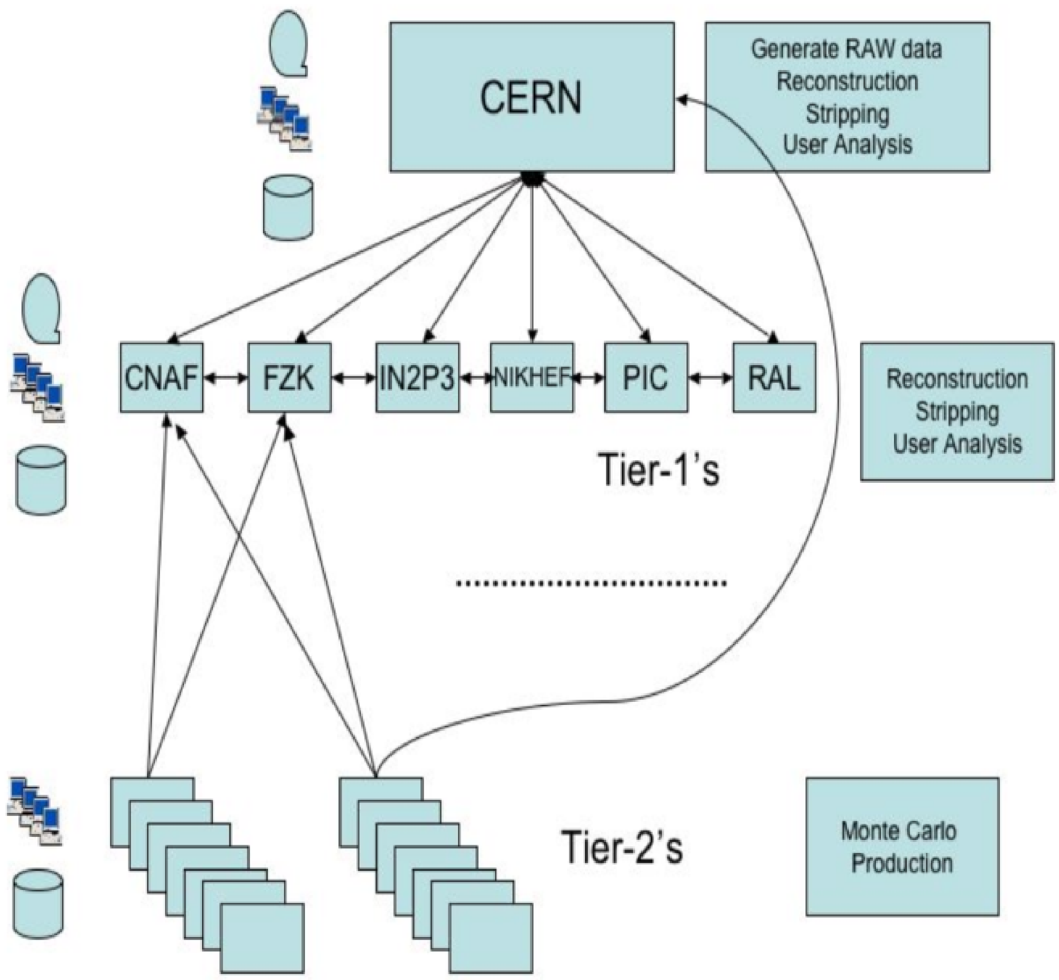


Figure 4.3.: LHCb Computing Model Schematic - Figure taken from[119]

Tier-0

CERN is the central production centre and is responsible for storing and archiving the RAW data which is produced by the LHCb detector, and for distributing the RAW data in quasi-real time to the Tier-1 centres. It participates in all the activities of the data processing.

Tier-1

CERN can also be considered as a Tier-1 centre, because it has enough CPU and storage resources for data processing. In addition there are six external Tier-1 centres: CNAF (Italy), FZK (Germany), IN2P3 (France), NIKHEF (Netherlands), PIC (Spain) and RAL (United Kingdom). These centres provide the necessary resources for data processing. The RAW files are replicated in quasi real time to the SE of one of these external centres, where it is reconstructed according to pledged computing resources. The reconstructed data is archived at the Tier-1 where it was produced and subsequently stripped and merged. The final DST files are replicated to the four Tier-1 centres to ensure high data availability for analysis. Simulation also takes place at Tier-1 when the reconstruction, stripping and merging are not using all the computing resources. The user analysis activity is performed at all Tier-1s where the data is produced.

Tier-2

The Tier-2 centres are primarily MC production centres. The data produced at these sites is uploaded to the associated Tier-1s. The biggest Tier-2 sites can be attached to a Tier-1 site and they can participate during data reprocessing.

4.5. Resource requirements

LHCb utilises the resources of the Tier-0, Tier-1 and Tier-2 centres to perform the LHCb computing activities. Each phase of the LHCb Work flow requires certain CPU and storage resources. This section describes the resources which have been used from 2010 to 2013.

The following formula is used to calculate the CPU requirements:

$$x = y * n \tag{4.1}$$

where x is the CPU requirement in a given period in HS06; y is the CPU time (second, day, week, month, year) in HS06 which can be used to process one event. Each LHCb application introduced in section 4.2 has a certain CPU time. For example: a typical Brunel process requires 12 second in HS06 to reconstruct one event; n is the number of events which can be processed in a given period.
The same formula is used to calculate the storage requirements:
where x is the expected storage requirement in Kb; y is the required storage per event in Kb; n is the number of events produced in a given period.

Every year the experiment requests computing resources calculated from these two formulae.

Resources used during 2010

Figure 4.4 shows the number of running jobs during 2010 separated into different computing activities.

The simulation consumed almost all the computing resources and produced 2,791,308,366 events using Tier-0, Tier-1 and Tier-2 resources. Table 4.1 gives a short summary of the usage of the computing resources, separated into CPU and storage.

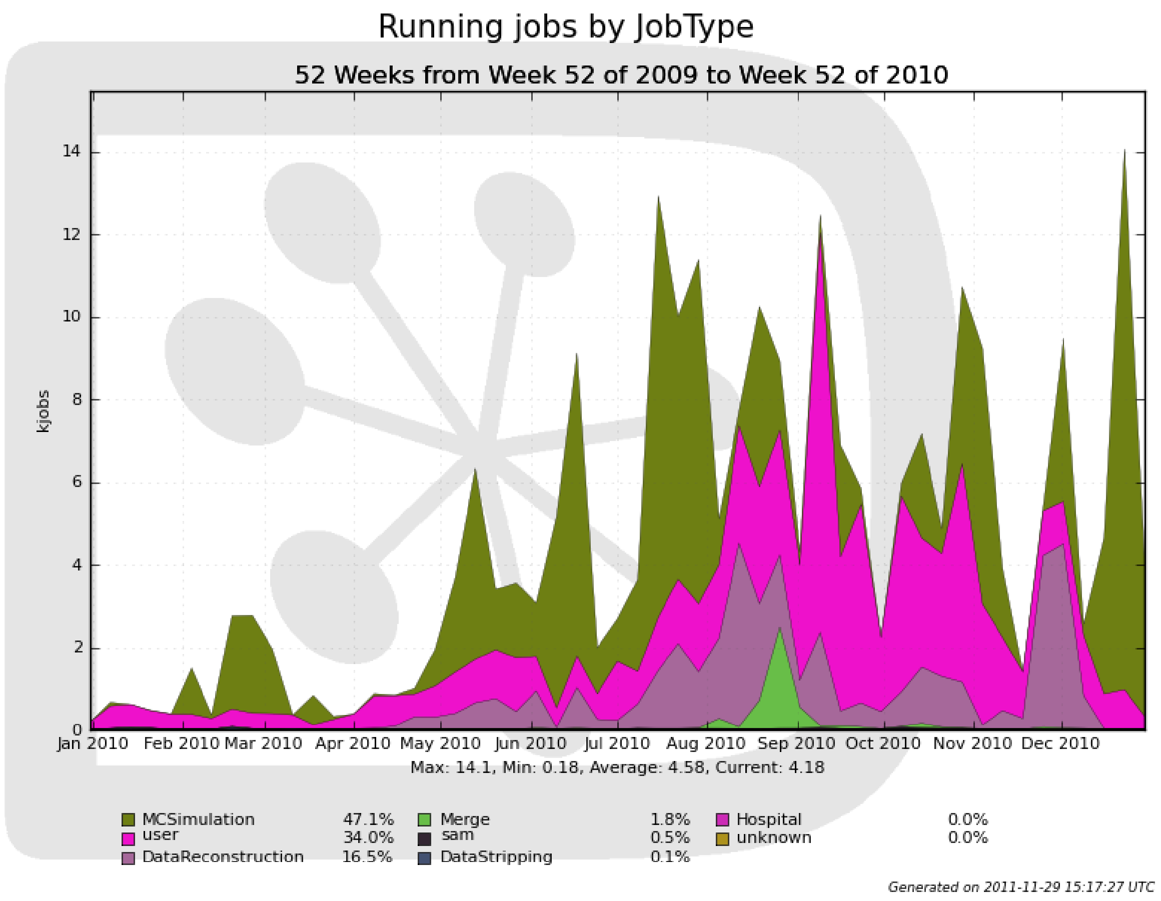


Figure 4.4.: Executed jobs which corresponds to a data processing activity during 2010 - Figure taken from the DIRAC accounting page

Site	CPU (HS06.year)	Disk (TB)	Tape (TB)
Tier-0	7166	922	844
Tier-1	18148	2096	903
Tier-2	24273	0	0
All	49587	3018	1747

Table 4.1.: Computing resources (CPU, disk,tape) used in 2010 (Numbers taken from[88, 164, 165]).

Resources used during 2011

The LHCb detector collected 1.12 x 10^{10} physics events during 4.6 x 10^{6} seconds of LHC collisions in 2011[164]. In parallel Monte Carlo simulation production generated an additional 1.17 x 10^{9} events. Figure 4.5 shows the data processing activities during 2011 which were much higher than the previous years. The CPU, Disk and Tape required in the Tiers are given in Table 4.2.

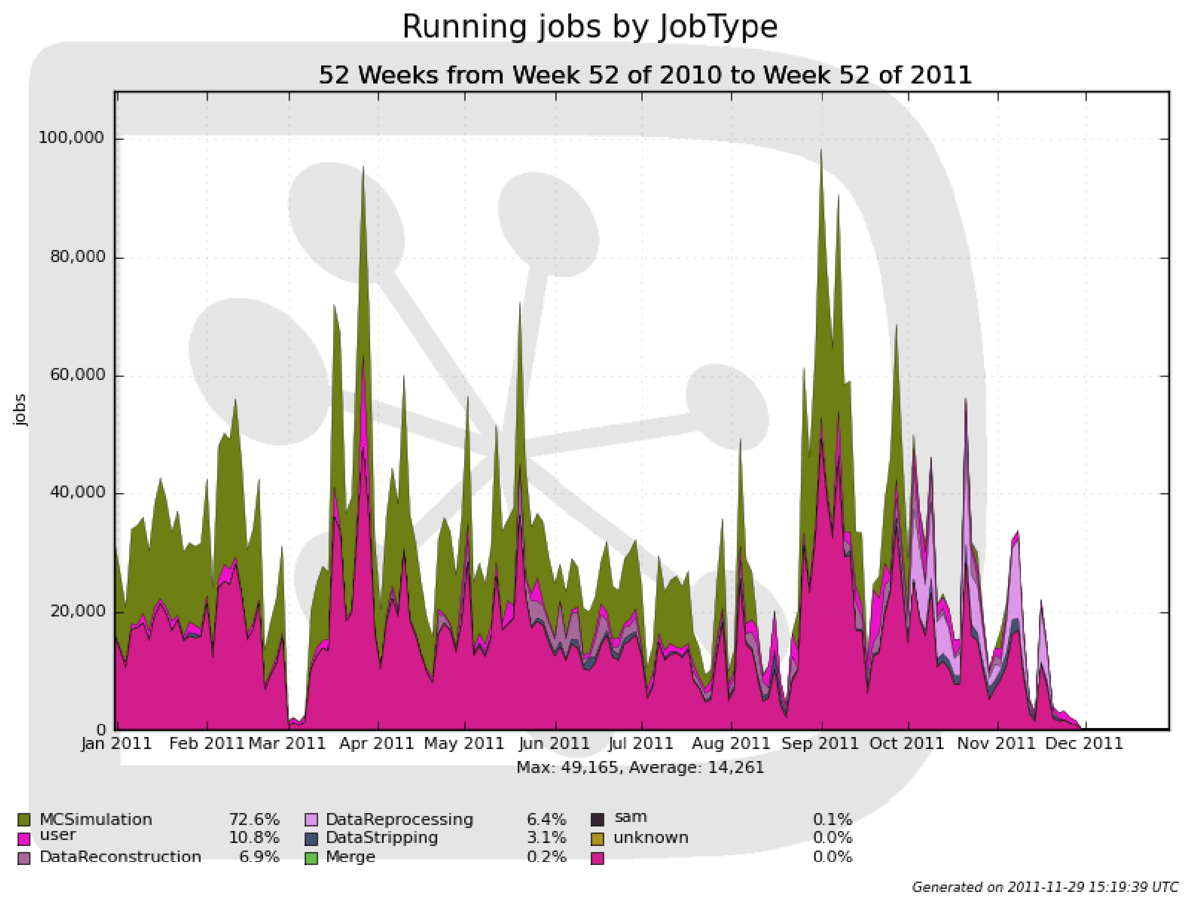


Figure 4.5.: Executed jobs which corresponds to a data processing activity during 2011 - Figure taken from the DIRAC accounting page

Site	CPU(HS06.year)	Disk (TB)	Tape (TB)
Tier-0	7200	1200	2100
Tier-1	35800	2700	3300
Tier-2	47000	0	0
All	89200	4000	5500

Table 4.2.: Computing resources (CPU, disk,tape) used from 1st of January to 30th of October in 2011. The numbers are from[89, 165].

Resource required to 2012 and 2013

The expected data in 2012 is the same as for 2011 data e.g. since the beam is defocussed to maintain an constant number of interactions and the trigger rate is fixed. In addition, the 2010 and 2011 data will be fully re-processed and re-stripped, together with 2012 data at the end of the data taking period.

The estimated computing resources for 2013 will be dedicated to re-processing and re-stripping, because there will be no data taken throughout the year. The full 2010-2012 data will be re-stripped twice and fully reprocessed during 2013. LHCb expects to produce more Monte Carlo data in 2013, equivalent to 1000 M event[164]. This extra Monte Carlo data will be produced at the Tier-0,Tier-1 and Tier-2 resources.

The CPU resources pledged at each site for LHCb activity during the 2012 and 2013 data taking period is summarized in Table 4.3. The disk and tape resources required at Tier-0 and Tier-1 is given in table 4.4 and 4.5.

	2012		2013	
Site	CPU(kHS06.year)	%	kHS06.year	%
Tier-0	34	18	33	18
Tier-1	113	59	110	59
Tier-2	43	23	43	23
All	190	100	186	100

Table 4.3.: LHCb 2012 and 2013 estimated CPU requirements at Tier's. The numbers are from[165].

	2012		2013	
Disk	TB	%	TB	%
Tier-0	3500	27	4000	26
Tier-1	9500	73	11100	74
All	13000	100	15100	100

Table 4.4.: LHCb 2012 and 2013 estimated disk requirements at Tier-0 and Tier-1 centres. The numbers are from[165].

	2012		2013	
Tape	TB	%	TB	%
Tier-0	6400	51	7700	49
Tier-1	6200	49	8000	51
All	12600	100	15700	100

Table 4.5.: (LHCb 2012 and 2013 estimated tape requirements at Tier-0 and Tier-1 centres. The numbers used from[165])

4.6. Dirac

The **Distributed Infrastructure with Remote Agent Control** (DIRAC) is a grid system which allows a community of users to access the distributed computing resources in a user-friendly manner. As a complete Grid solution, the DIRAC project was designed to make available heterogeneous computing resources and supports all the aspects of the LHCb Computing Model[167]. In this section we present the Dirac systems which plays an important role in providing a reliable heterogeneous computing system.

4.6.1. Dirac design and implementation

To keep the system generic DIRAC was designed to use pluggable modules. These modules allow DIRAC to support future use-cases. The architecture was designed to ensure that DIRAC could manage all the loosely coupled services required to perform distributed computing on heterogeneous resources. Consequently, DIRAC can be used as a standalone environment or on top of existing Grid middleware (in our case gLite) without any development or knowledge of specific configurations. Therefore, when DIRAC is running within existing Grid infrastructures, no additional services are required to be installed at the site.

DIRAC itself is lightweight and portable and can be used with many operating systems and architectures. It is implemented in Python. Because this is an object oriented and interpreted language, it provides the ability to rapidly design and implement applications. In addition, due to its portability, it is available in many computing platforms, and has a rich set of libraries and modules which can be used by DIRAC.

4.6.2. DIRAC Architecture

The DIRAC architecture is based on a **Service Oriented Architecture** (SOA) which can be decomposed into four categories: Services, Resources, Agents and Interfaces as shown in Figure 4.6.

Resources in the DIRAC perspective are the underlying computing and storage facilities provided by the computing centres. DIRAC does not provide a complex Storage Element but it implements a wrapper to the SRM standard interface as well as the most important data access protocols (gridftp, xroot, etc). The DIRAC **Agents** interact with the Resources to perform specific functions within the system[167]. The **Services** perform operations or requests. A Service can be stateful or stateless. Each Service usually interacts with a RDBMS [1] which stores the state information. The Agents also interact with DIRAC **Services** to perform different operations which are requested by users or another agent. The services are centrally managed in a controlled environment, while the agents can be executed anywhere in the distributed environment. Each Service offers a client interface which can be used by Agents or

[1]DIRAC Systems use MySQL databases except the LHCb Bookkeeping System

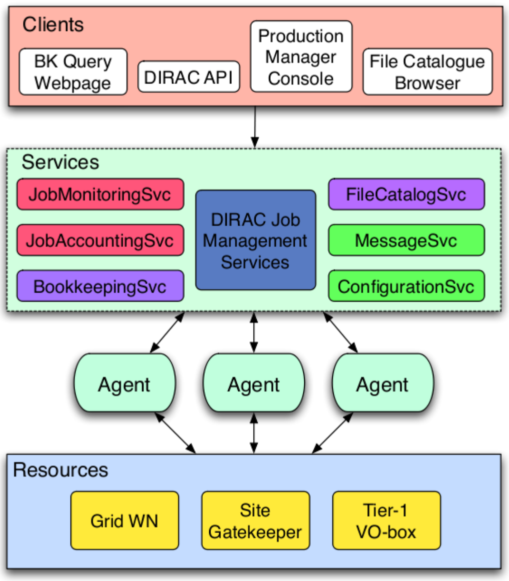


Figure 4.6.: Dirac services, agents, resources and their interactions - Figure taken from[166]

by DIRAC **Interfaces**. DIRAC provides different Application Programming Interfaces (API)s which provide various functionality to the users in order to execute their operations or requests. The APIs or the Web interfaces are used to ensure access to the DIRAC services and are part of the DIRAC Systems.

4.6.3. DIRAC Systems

The DIRAC System is composed of various services, agents, and clients which are destined to support required functionalities. The main DIRAC systems are the Workload Management System (section 4.6.3), the Data Management System (4.6.3), the Production Management System (4.6.3) and the Bookkeeping System (chapter 7). Each DIRAC system has a dedicated scope and they are built on a single secure DIRAC framework.

Framework

DIRAC has to manage a large number of distributed resources that require secure communication. It has to provide the infrastructures which are needed to build services and agents easily. Therefore, a powerful and flexible framework is essential. The framework contains a **DIRAC SEcure Transport** (DISET) protocol for secure communication, **Configuration System** (CS) for providing configuration parameters to DIRAC components, **Logging and Monitoring System** (LMS) for providing a report to all DIRAC components and a Web Site for providing access to the systems and ensuring a visual representation of their status and activities[184]

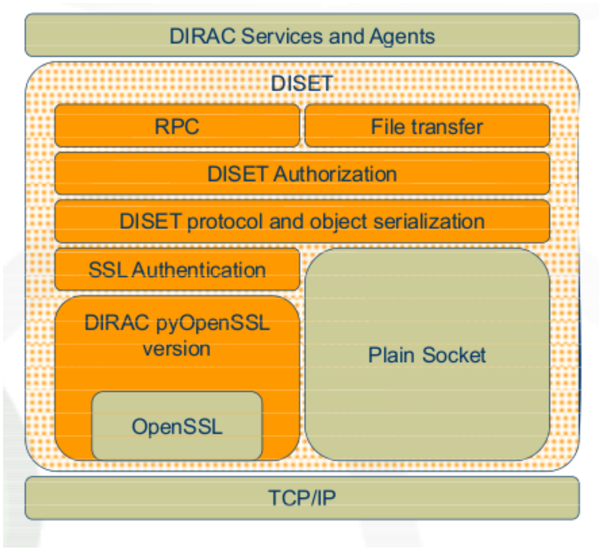


Figure 4.7.: DISET layered architecture - Figure taken from[184]

The lowest layer of the DIRAC software is DISET. It is the DIRAC secure transport layer which manages the communications between DIRAC components using OpenSSL to perform authentication and encryption based on the X509 certificates. As the core of the DIRAC framework provides network connectivity using standard TCP/IP protocol, a **Remote Procedure Call** (RPC) and *file transfer* functionality are embedded into the services and clients using multiple threads. Authorization rules can be specified to users and groups that make the system more secure. DISET automatically checks the types of the call arguments and in case of problems it returns an error which is advantageous for developers. Figure 4.7 show the DISET layered architecture.

On top of this, the DIRAC framework provides a set of utilities which are required in order to fulfil the application programming environments. The Services, Resources and Agents require configuration parameters or information which have to be stored and accessible in a reliable way.

The Configuration System (CS) stores configuration data in a hierarchical structure and provides these stored data to the DIRAC components using a mechanism consisting of a single master with multiple slaves[185]. These are synchronised automatically with the master CS and ensure the availability of configuration data to DIRAC services and agents.

The remaining framework utility is the Logging and Monitoring System that represent the status of the DIRAC system. DIRAC services and Agents send their activity reports to the Monitoring system, and send important messages such as errors or failures to the Logging system.

Workload Management

The DIRAC Workload Management System (WMS) is the central service of the DIRAC system for distributed computing. It provides capabilities to the different users to submit and monitor their computation tasks (jobs) and to retrieve their output. The WMS provides the

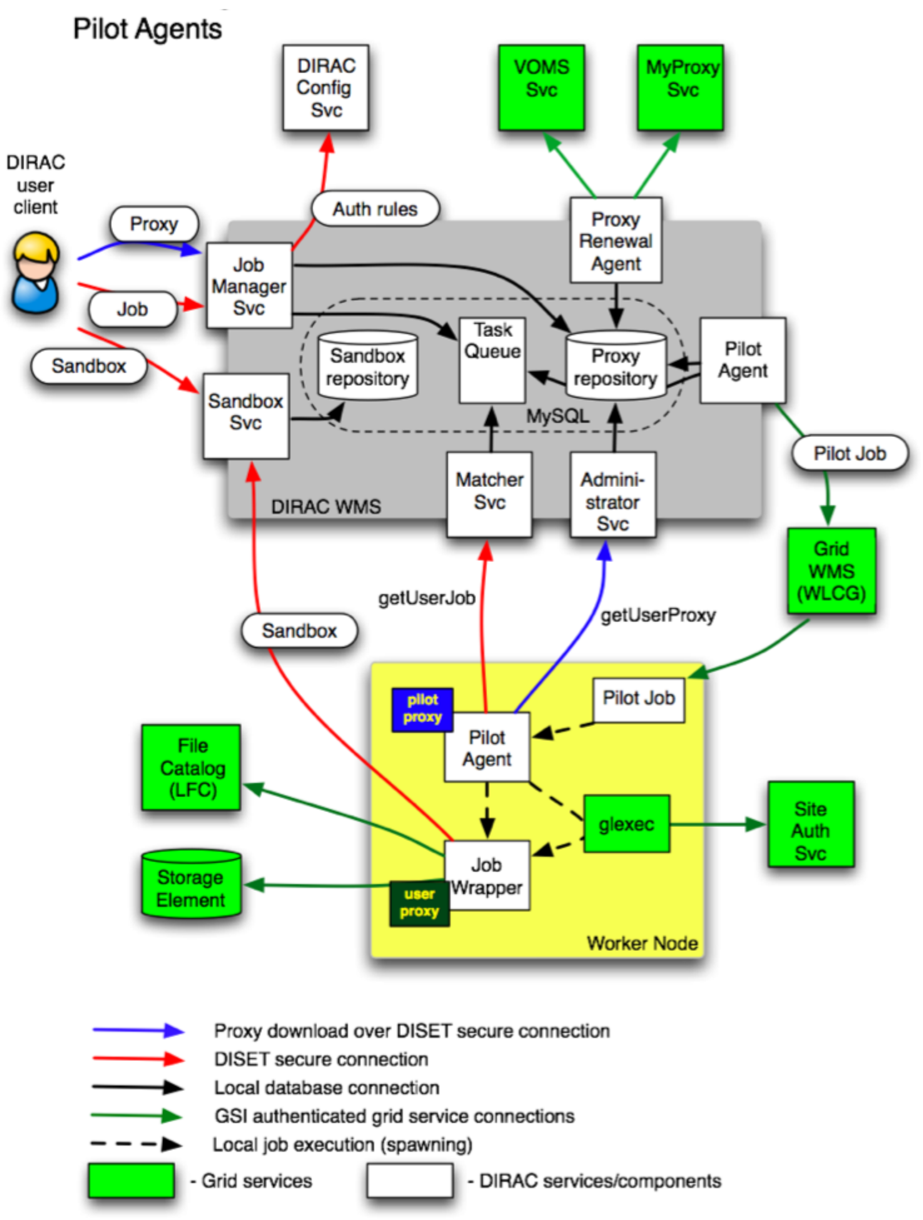


Figure 4.8.: DIRAC WMS - Figure taken from[166]

various computing facilities to its clients and increases the stability of the system by hiding the transient failures on the Grid. Increased stability is achieved by *rescheduling* failed tasks that are automatically retried by the DIRAC system.

The WMS is based on the *pull scheduling* paradigm which ensures stability and efficiency. Pull scheduling uses the execution environment found on the underlying computing resources and tasks can only be executed on those resources which match the required environment. The decision of the task execution is made when the resources are available which means a scheduling can be applied based on priorities. DIRAC uses light agents or *Pilot jobs* which are sent to the underlying computing resources in order to execute the tasks. These regular WLCG grid jobs (pilots) which do not require any special handling tools, check the execution environments and requests an eligible job from the DIRAC WMS. These pilots have an allocated CPU and wall clock time limit when they are executed within a batch system. Therefore, DIRAC is capable of optimizing the use of a batch slot according to the CPU and wall clock time limit. Consequently, multiple jobs may be run within the time limits, increasing their usage efficiency.

Figure 4.8 shows the major services and agents of the WMS which together make up the pull scheduling architecture. The users must have a valid Grid certificate in order to be able to interact with the DIRAC services through the DIRAC API. The DIRAC API is able to formulate users tasks which are translated into the Job Description Language (JDL) format and are provided to the **Job Manager** service through a secure RPC connection. The Job Manager service is the entry point for clients and is responsible for creating a new entry which describes the requirements of the jobs, the identity and group of the job owner. This job-related information is stored in a relational database backend.

When a user runs an application on the Grid, in most of the cases the application requires the use of small files which are less then Megabytes in size. These files are called the *input sandbox* while the term *output sandbox* refers to similar small output files of a job, which do not require permanent storage on the Grid. These sandboxes are stored in a database and the **SandBox** services manage (uploads or downloads) these small files in a reliable way. The users use the DIRAC API to specify the input sandbox files to be used during job execution time.

The newly entered job is treated by a series of agents before being assigned to a **TaskQueue**. The TaskQueue contains tasks with identical execution requirements and keeps the list of pending tasks. Based on this list, the Director Agents, called **TaskQueueDirector** are used to populate the available computing resources with Pilot Jobs. The pilot checks the environment on the worker node and requests a job using the **Matcher Services**.

The Matcher service receives requests from the pilot jobs, checks available jobs in the Task Queues and makes a decision according to the requirements presented by the pilot job.

Job Wrappers prepare the job execution on the worker node and provide all the data required by the job.

During execution the job periodically reports back its status and progress information to the **Job State Update** service. When a job has completed, the pilot job checks the time remaining in the batch slot and attempts to match another job with the updated CPU limit requirement.

The application may produce output files which must be stored. Two possibilities are available. Small files can be stored in the output sandbox while big files can be stored in the Grid storage element.

Data Management

A reliable **Data Management** system is essential in order to manage file transfer and cataloguing of LHCb data according to the LHCb Computing Model. The DIRAC Data Management system core resources are the Storage Elements (SEs) and File Catalogues. The Replica Manager using the File Catalogues and SEs in order to perform simple file management on the Grid.

Storage Element

The **DIRAC Storage Element** is an abstract layer on top of the storage resources used to mask underlying storage technologies and protocols. It stores directories and files on the Grid SEs and allows to upload, download and replicate these files and directories using different protocols. The DIRAC DMS determines the protocols which are available in a particular resource and ensure these protocols will be used in an efficient manner. The descriptions of each SE for a given site are stored using the Configuration Service. This has a functionality similar to the SRM service with the aim of providing transparent access to various SE implementations. The access protocol relies on plugin modules, which represents disparate mechanisms of data access. The SRM2 plugin is used by the SE with a GFAL python binding to contact the SRM services and the lcgutils package is used to perform file transfers.

Catalogues in DIRAC

DIRAC uses different catalogues such as metadata and replica catalogues. The transparent usage of the catalogues is essential for efficient data access. The catalogues can be used by various types of users.

- The production jobs have to store and access the metadata, the location and replica(s) of a file in an efficient and easy way.
- The distributed analysis jobs need to access the metadata and replica catalogues without knowing in advance where the job is running.
- The Services or Agents use the metadata and replica catalogues to ensure data availability and data consistency.

DIRAC uses the following catalogues:

- Bookkeeping Metadata Catalogue stores files and jobs metadata and job provenance information (more details in section 7.4.2).

- LFC catalogue is a replica catalogue. It is used to keep files and it's attributes (such as size, creation time, owner etc), in addition to it's replica location(s).
- DIRAC-FC is a replica and metadata catalogue. The DIRAC-FC is being used by the non-LHCb community in order to replace the LFC catalogue.

DIRAC provides an abstraction layer called FileCatalog which provides a single point access to these catalogues through different plugins which expose its functionalities. It uses the Configuration Service as the SE to store the available catalogues and their attributes, but the philosophy is different from a normal SE because each operation must be carried out on all available catalogue plugins to maintain their consistency.

Replica Manager

The **Replica Manager** (RM) combines the functionalities of the Storage Element and DIRAC catalogues in order to perform file based data management and accounting. The RM implements methods which are used by clients for the manipulation of files on the Grid. The RM interface allows users to upload and download data which is described by LFN to Grid SEs and register their contents to the various catalogues (such as Bookkeeping Metadata Catalogue, LFC). In order to have efficient data analysis, the datasets can be replicated using RM to other Grid SEs and can be registered to the DIRAC catalogues. Like a File System, it

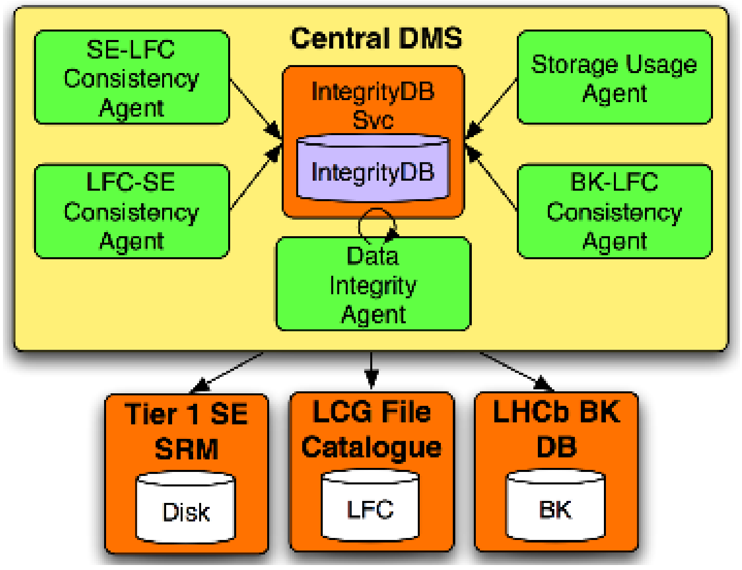


Figure 4.9.: DIRAC Data Integrity tool architecture - Figure taken from[186]

allows the removal of files and replicas from Grid SEs and catalogues where these files were registered. Figure 4.9 shows the *data integrity* suite architecture used to ensure the data integrity. The DIRAC agents check the different catalogues and SE in order to discover data consistency and report the status to the **IntegrityDBSvc**. A set of clients extract the IntegrityDBSvs service functionality. RM operations such as replication, removal, registration etc.

may be persisted as a request in a **Request Database** in case of failure in a generic XML format for later retry. The requests which are in the Request Database are retrieved and executed by a series of agents: the transfer requests are retrieved and executed by the **Transfer Agent**; the removal request are retrieved and executed by the **Removal Agent** and the registration requests are retrieved and executed by the **Registration Agent.** The DMS consists of a **bulk transfer framework** which handles the transfer requests by submitting a job to gLite File Transfer service (FTS). This framework also provides capability to monitor the submitted jobs.

Production Management

While the WMS gives users the ability to run their jobs, the **Production Management** system is designed to centrally manage large productions used to generate Monte Carlo data and to process real data produced at the LHCb detector. The Production Management is built on top of the Workload Management system to perform definition and submission of the production requests in a fully automatic data driven way. The production requests are stored in the database backend. Each production request is described by a production and the production is defined by a *Work flow*. The Work flow describes the *steps* which correspond to the applications and configurations to be run, the input data requirements, the destination and the data to be produced. The main components of the Production Management system such as **Production Database** and associated services and agents are shown in Figure 4.10.

When a production request is defined and submitted to the production team, a *Data Processing* type production is created by the production manager. The input data is retrieved from the LHCb Bookkeeping File Catalogue by the **BookkeepingAgent**. These retrieved input files are stored in the Production Database and assigned to a task queue with matching input data requirements. The **TransformationAgent** periodically checks the active productions in the system in order to create new jobs which are assigned to particular sites according to the LHCb Computing Models. The agent uses two different criteria to create jobs for Monte Carlo productions and for Processing productions:

- For MC production the agent creates jobs which correspond to the number of events requested by the user.
- For Processing the agent groups the input files using a *transformation plugin* (e.g. by size, by file type) according to their location and creates jobs to process.

The created jobs will be submitted by the **JobSubmissionAgent** to the DIRAC WMS. The **ProductionStateAgent** monitors the production progress which can be followed using a command line or Web interface.

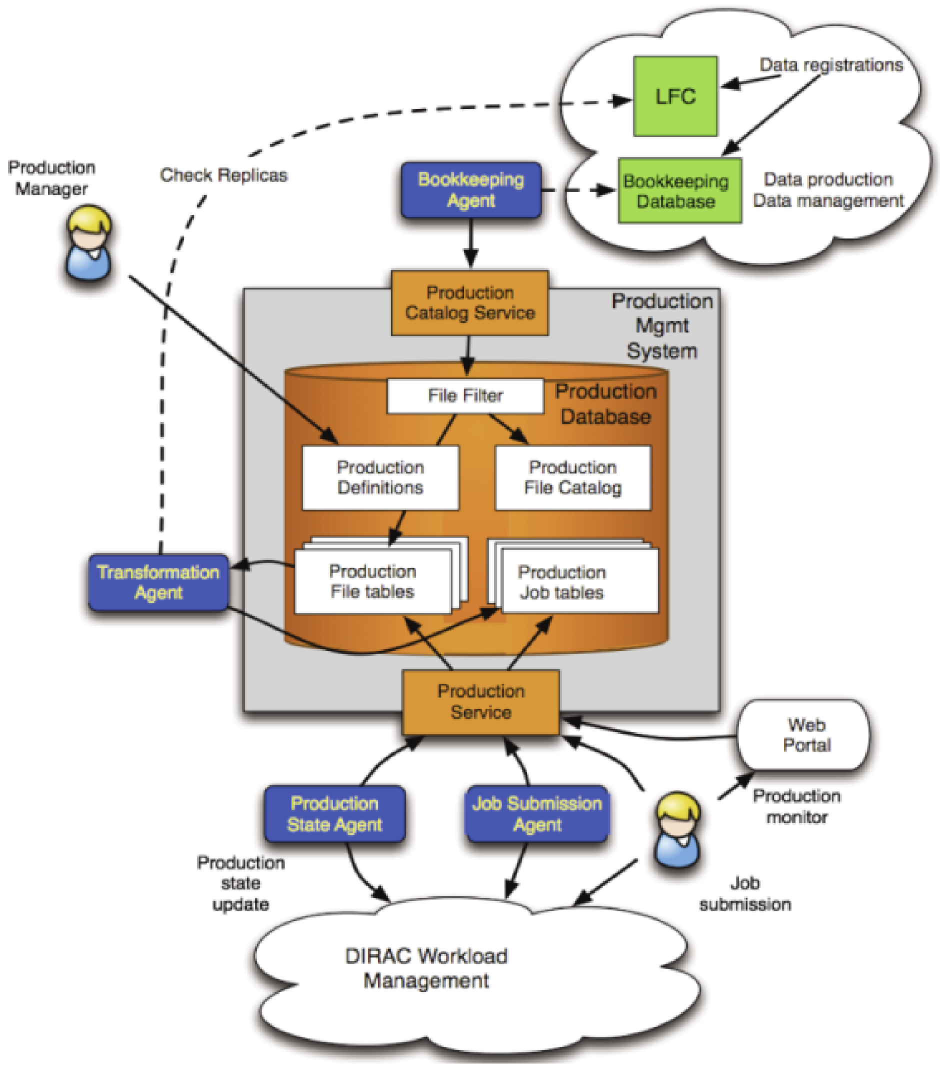


Figure 4.10.: Schema of the Production Management system - Figure taken from[187]

4.7. Ganga

Ganga is a component-based system providing a user friendly interface for the configuration, execution and management of computational tasks in distributed environments. Ganga was developed to support distributed data analysis within the ATLAS and LHCb experiments where large communities of physicists need access to Grid resources. This section presents the design and implementation of the main Ganga components which are parts of the Ganga software.

4.7.1. Overviw of Ganga architecture, design and implementation

The design concept of Ganga is to develop a flexible tool which integrates new functionalities using an object oriented paradigm. Since Python supports object oriented paradigms and has many advantages over other programming languages, the Ganga developers decided to use this programming languages. Ganga is based on plugin architecture which is composed of various components which can be used through a text based Command Line Interface (CLI), a file scripting interface and a Graphical User Interface (GUI). The architecture of Ganga is shown in Figure 4.11. The user interfaces are built on top of the **Ganga Public Interface** which is a liaison between the **Ganga Core** and the user interfaces. The GPI produces a

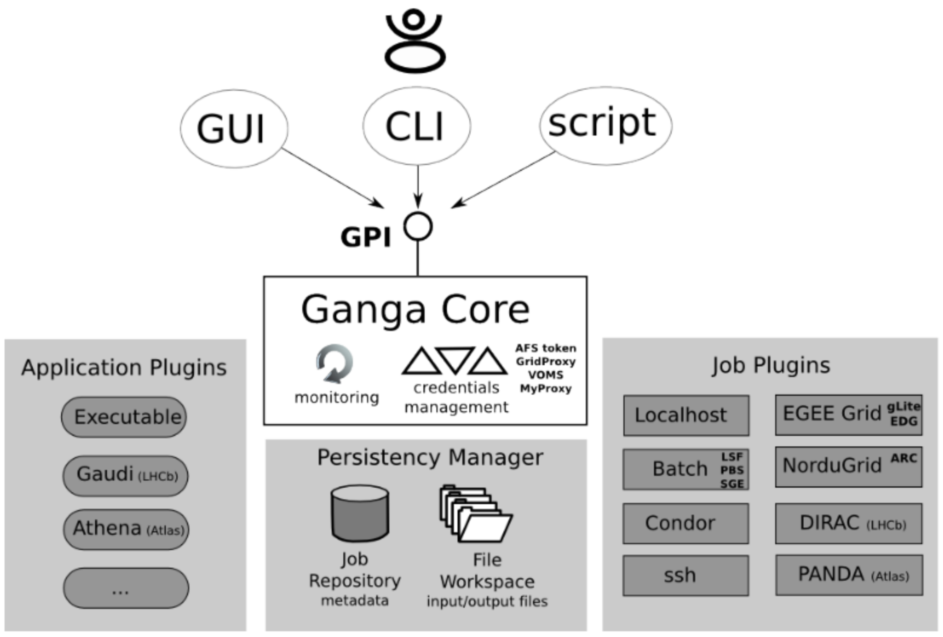


Figure 4.11.: The architecture of Ganga - Figure taken[193]

simplified view for users, but at the same time more detailed information can be exposed from the underlying computing resources. The Ganga Core provides **monitoring** capability which plays a vital role during job execution. Each time the users want to know their job status, they query the job **repository** which is built on the database backend. This repository stores the job statuses such as submitted, running, completed, failed and killed. The input and output data associated to the jobs may be stored in the local machine or the repository. Ganga takes care of handling user credentials, including Grid proxies, and Kerberos[194] tokens which are used by the Andrew file system (AFS)[195]. More details of the different components are given below.

4.7.2. Ganga Components

Ganga implements a set of components which are used to construct a job. All jobs are required to have an **application** component and a **backend** component. The application component defines the software to be run during the execution of computational task while the backend component defines the underlying computing resources to be used by the application component. Most of the jobs require input data and may produce output data. The **input dataset** component is used to specify the job input data which is to be read and processed. The **output dataset** component is used to specify the data to be produced. If the job has a huge amount of input data to be processed, a **splitter** component is used to divide the job into independent sub jobs and a **merger** component to aggregate the output of each sub job.

Application components

The type of the computational task is described by the application component. It defines the options and settings of the software and provides methods for specifying actions to be taken before and after the job is processed. Before the application is executed, intermediate configuration may be required. These intermediate configurations are stored in files which are created automatically by the application component. The configuration method carries out integrity checks in order to ensure the sanity of the jobs. The **Executable** is the simplest application component which has an *exe* property for providing the path to an executable binary or script, an *args* property for storing a list of arguments to be passed to the executable, and an *env* property for storing a dictionary of the environment which is to be defined before the executable starts.

Backend component

The backend component is an abstraction layer for a range of widely used processing systems which can be a local host, batch systems such as Portable Batch System (PBS), Load Sharing Facility (LSF), Sun Grid Engine (SGE), Condor or Grid systems such as gLite, ARC, OSG. The Backend component has plugins to DIRAC and PANDA-specific middlewares which means the users can submit their jobs to DIRAC or PANDA Workload Managements Systems without having any specific knowledge of the Grid systems. The Backend component defines properties which can be set by the users. For example LCG has the following properties: middleware, actualCE, requirements (memory,cputime, walltime,software), status, id. The Backend component provides various methods to be used to submit, resubmit, kill, delete or cancel a job. In addition it provides monitoring capability for the existing jobs. It provides a method which is used to retrieve the output files of a job.

Dataset components

Dataset components generally define properties used to describe a particular collection of data which have to be read and processed. They also provide methods for obtaining information about the data. For example the users might be interested to know the data processing phases or location where the data is stored. The details of how data collections are described can be performed using generic dataset components. The datasets differ significantly between experiments. Each experiment has their own catalogues system which allows the users to query these catalogues and find data collections which will be stored by an experiment specific component. For example the LHCb Dataset component uses the LHCb Bookkeeping System to retrieve datasets.

The users can specify their sandbox files which are stored externally. These sandboxes are transferred from the user file system together with the job to the computing resources where the job will be executed.

Splitter and Merger components

In many cases the computing tasks are too big to be carried out on a single processor. Consequently, the users may want to split their jobs into sub jobs, to get the result in a faster time. One very common usage pattern is to run the same code several time with slightly different arguments. This can be achieved using **GenericSplitter** which specifies the number of sub jobs to be created, and the way in which are sub job differs from another. Different splitters have been implemented such as *ExeSplitter* which splits a list of executable into different sub jobs, and *ArgSplitter* which deals with executing the same application many times, but with different arguments. Most of the experiments have splitter components inherited from the GenericSplitter to deal with creating sub jobs that process different parts of a dataset.

The outputs of the sub jobs may need to be aggregated. To perform this requirement Ganga developers implemented so called Merger components. These components combine the output of sub jobs at a later stage. A few predefined mergers are available in Ganga such as *MultipleMerger*, *SmartMarger*, *RootMerger*, *TextMerger*. The merging can be performed automatically after the job and its sub jobs have finished. The users may want to merge files after the job that created these files no longer exists. The merging of these files can be achieved manually by the users.

4.8. Summary

The LHCb experiment uses a set of application to process data taken by the LHCb detector or produced by the Monte Carlo simulation productions. The applications are based on a common Gaudi framework. The data taken by the detector or simulated by the Monte Carlo productions is reconstructed, stripped and merged. The data can be stripped or reprocessed more than one time during a year. DIRAC is implemented to process the data in a reliable way taking into account the experiment specific requirements. It provides the capability to execute different data processing jobs in the Grid or local cluster. Ganga implemented as a user-friendly interface for configuration, execution and management of jobs in a distributed environment. Ganga provides the possibility to submit jobs to the DIRAC system, different batch system, and the Grid.

We can distinguish between production and user activities. The production activities (reconstruction, stripping, etc.) only use DIRAC system for data processing while the users uses Ganga to submit their analysis jobs to the DIRAC system.

5. Feicim

This chapter discusses Feicim, a browser for accessing LHCb data. Feicim, from the Gaelic word meaning “I see”, is a browser which provides a visual representation of datasets, data-content discovery and analysis through a Graphical User Interface. We start with some motivation for the project in section 5.1. A well defined architecture is presented in section 5.2 and this is followed by a brief description of the main components of Feicim along with their scope in section 5.3. Section 5.4 introduces various design patterns which are used to design and implement loosely coupled components. We describe the Creational, Structural and Behavioral design patterns which provide a schema and the relationship of the components. We present different architectural design patterns which are used to describe the schema of complex software systems. We introduce the Composite Model View Controller which is based on existing design patterns such as the Model View Controller and the Hierarchical Model View Controller [196], but we avoid the disadvantages of these patterns in order to have a more flexible and robust design pattern. In section 5.5 we introduce two objects which are used to temporarily store data in memory. Managers present the data in a treelike format which is described in section 5.6.

5.1. Motivation

The key element in high energy physics is the data collected by different experiments. The goal of the physicist is to make physics measurements using this data. The physicist must extract, interpret and filter relevant information from the data in order to make physics measurements. When users run analysis on the data, they have to define the input data, understand its contents and define the different algorithms to be performed. To find the input data is not easy without a robust user-friendly tool. When the data is found the users have to access this data often without knowledge of how and where the data is stored and how it can be accessed. In particle physics the process and understanding of the data requires several processing phases and various algorithms which perform different applications. However, as programming environments become larger and more complex due to the various applications, access to the information of interest become more and more difficult.

The **Data-Grid Environment and Tools for Distributed Management and Analysis of Large Sets of Scientific Data** (DGET)[197] was created around 2005 to provide a framework which can be used by various scientific communities such as particle physics, astrophysics and bioinformatics. DGET unifies various resources, and gives access to these resources in a user friendly environment that is robust, adaptive and scalable. The Feicim browser is developed as a DGET test application in particle physics. Feicim hides the complexities of the Grid and provides links between the people with Grid experience and the physicist who does not need to know about of Grid computing. In 2007 the first prototype of Feicim were presented at CERN. Since this time, Feicim has developed and become more flexible. At the end of 2007 it was decided to use Feicim by the LHCb experiment as the main LHCb Bookkeeping System. At the beginning of 2008, Feicim was re-designed according to the LHCb requirements, but the main principles did not change. In the middle of 2008 Feicim was progressively put in production and used in parallel with the old LHCb Bookkeeping System by the users and the Production Management System. At the end of that year Feicim became the main LHCb Bookkeeping System. In the following years small improvements were made in order to improve the scalability and reliability. The design of Feicim allows components to be extended and changed without modifying the whole system.

5.2. Architecture

The basic concept of Feicim is to provide a visual programming language to the physicists which lets users create their own analysis by manipulating program elements graphically rather than by specifying the programming elements textually. The Feicim architecture performs these tasks. The Architecture is composed of various components which solve specific problems. The design of these components applies different existing design patterns (see section 5.4.1).

The Feicim architecture is defined in terms of components and their interactions. Figure 5.1 shows the Feicim components.

- Interface Manager defines the basic interfaces which are used by other components.

  The Interface Manager consists of the following interfaces: Controllers, Messages, Entities, Items and Managers.

- Communication and work flow manager is responsible for providing the data to the Feicim data browser and the Feicim LHCb Bookkeeping catalogue.

- Application Builder is responsible for building the application managers.

- Explorer Viewer has two viewers:

  1. Feicim Data browser contains widgets and controllers which are used to work with the LHCb data files.

  2. Feicim File Browser contains widgets and controllers which are used to find the data. It contains the Feicim LHCb Bookkeeping catalogue which is used to select data in the Grid Storage Element, and a local file browser which is used to select data in the local machine.

- Desktop contains the different algorithms. The algorithms correspond to graphical elements. The graphical element is a visual representation of algorithms. The users can easily create a complex algorithm structure by using the different graphical elements.
- GUI contains views which are used by the users to perform different tasks.

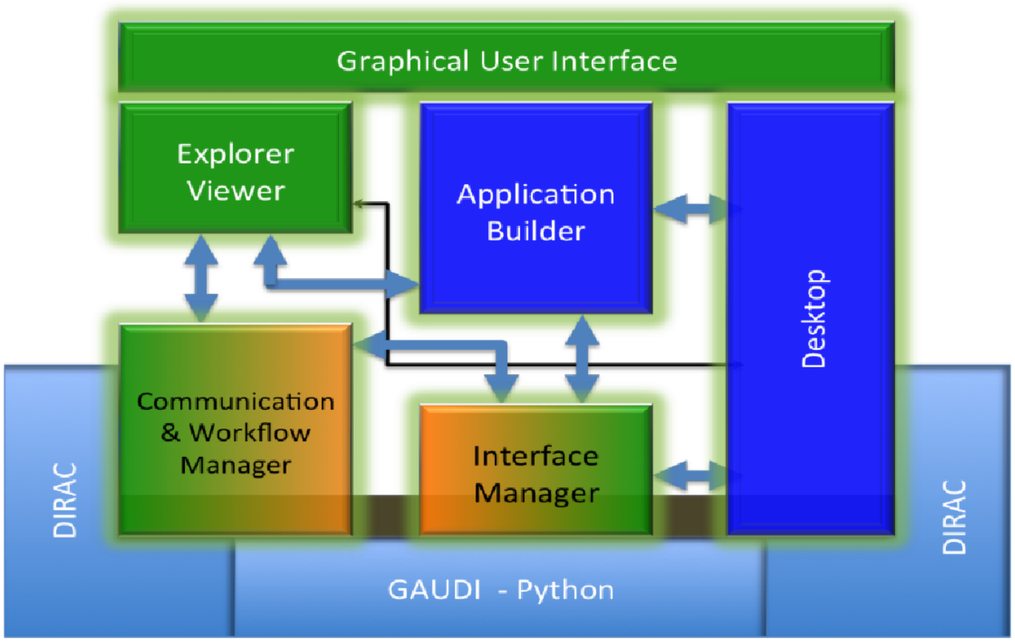


Figure 5.1.: Feicim high level architecture - illustrating the components of Feicim and their interaction.

Feicim is based on loosely coupled components and uses DIRAC/Ganga for distributed analysis using the LHCb Grid infrastructures. Gaudi is used to read events which are in the files and run simple algorithms on the local machine.

5.3. Components of Feicim

Feicim consists of tree components which will be discussed in the next chapters: the LHCb Bookkeeping Metadata Catalogue for data location, the Feicim Data Browser for data-content discovery, and the Feicim Data Analysis for running analysis jobs on the Grid or local machine. Each component has a dedicated role which means they can be used standalone. However, they perform together as a complex system which covers all the LHCb specific tasks needed to run data analysis. Each component based on the Feicim architecture and the Feicim design principles.

5.4. Design and implementation

The design of Feicim takes into account existing design principles which are widely used by software engineers during software development. Feicim is more than a simple application, because it has to meet various criteria which have been achieved by studying existing software technologies and design principles.

Feicim is designed as a platform independent application. We analysed the available capabilities in order to have an optimal design which fulfill the following requirements:

- Re-usability: Feicim components can be easily re-used or re-integrated to another system. In addition, the design principles of Feicim can be used to implement a new system.
- Scalability: We intend to use resources efficiently in order to provide a scalable system, which can handle thousand of user requests without overloading the system.
- Manageability: Various components of Fecim can be manageable without having specific knowledge of the system.

We applied common solutions for common problems and as few technologies as possible to achieve more scalability capable of absorbing new and changing requirements. We attempt to design user friendly views which can be used without specific knowledge.

Design patterns are used to develop the components of Feicim. During the design of Feicim we faced different problems: Feicm has to be able to show a tree structure using different views;the views have to be able to show different types of tree structure without any modification. We studied the Model View Controller and the Hierarchical Model View Controller which can be used for manipulating multiple views on the same data model. If the data model or views are implemented in a different programming environment, these two design patterns can not be used, because a view can not communicate with it's controller and model which is in different programming environment. A new more flexible design pattern is required, which is called the Composite Model View Controller. It has increased functionality compared to existing design patterns, because it separates the data model from the views using different layers to solve different problems. It also allows controllers to use different programming languages.

In order to comply with the requirements above we decided to use the Python programming languages and pyQt (Python-bindings to the Qt graphical toolkit). This allows Feicim to use different external software to solve specific problems. We reuse existing software with small modifications rather than reinvent completely new software.

We now present a general overview of Design Patterns before focusing on the Composite Model View Controller that we have created.

5.4.1. Design Patterns

A design pattern is a general reusable solution to a commonly occurring problem in software design[200]. We decided to adopt various design patterns into our system. Design patterns are used to show relationships and interactions between classes or objects. There are many types of design patterns, among which we used the following: Creational patterns, Structural patterns, Behavioral patterns and Architectural patterns.

Creational patterns

These patterns deal with controlling object creation mechanisms in a manner suitable to the situation[201]. They are categorized into Object-creational and Class-creational patterns. Both categories have dedicated scope; the object-creation deals with creating objects in an efficient way while class-creation deals with class-installation using class inheritance effectively.

Factory Method

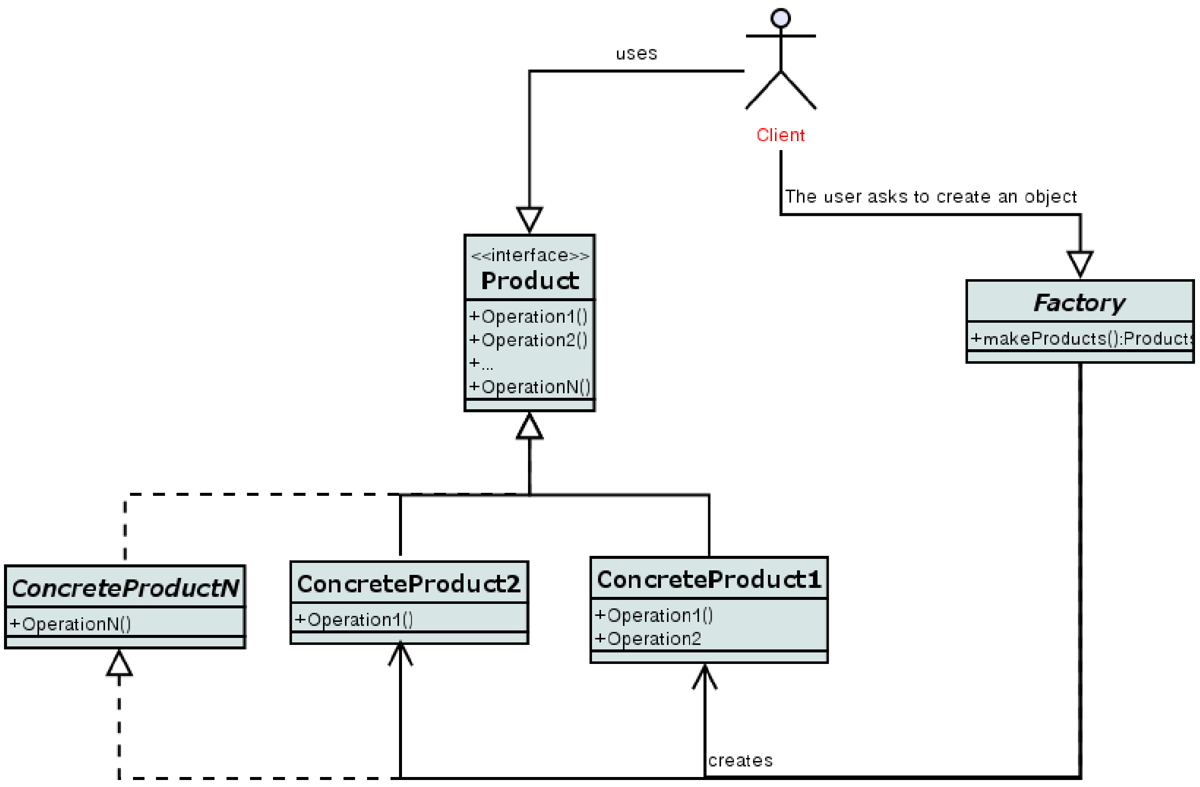


Figure 5.2.: Factory method design patterns - A simple Factory Method with basic classes.

This pattern implements the concept of factories that deal with the problem of creating objects without specifying the exact class of object that will be created[202]. Figure 5.2 shows an overview of the Factory Method design pattern which is described below. Usually, an *interface* is needed to specify the common methods which will be reimplemented by the subclasses in order to perform specific tasks.

The implementations of these subclasses are different and optimized for different kinds of data. The *Client* interacts with the *Factory* Method in order to create a specific *Product* that performs a task. When the product is returned to the Client by the Factory Method, then it can be used by its owner. In this way the Client does not need to know the implementation of each subclass. It only needs one reference to the interface and the factory object. This encapsulates the creations of objects and hides the creation of the very complex processes. In addition a new Product can be implemented and can be created by the Factory Method.

Builder pattern

While the Factory method returns an instance of one of several possible classes depending on the data passed in arguments by the creation methods, the Builder pattern provides different representations of objects which must be constructed by different implementations of abstract steps[203].

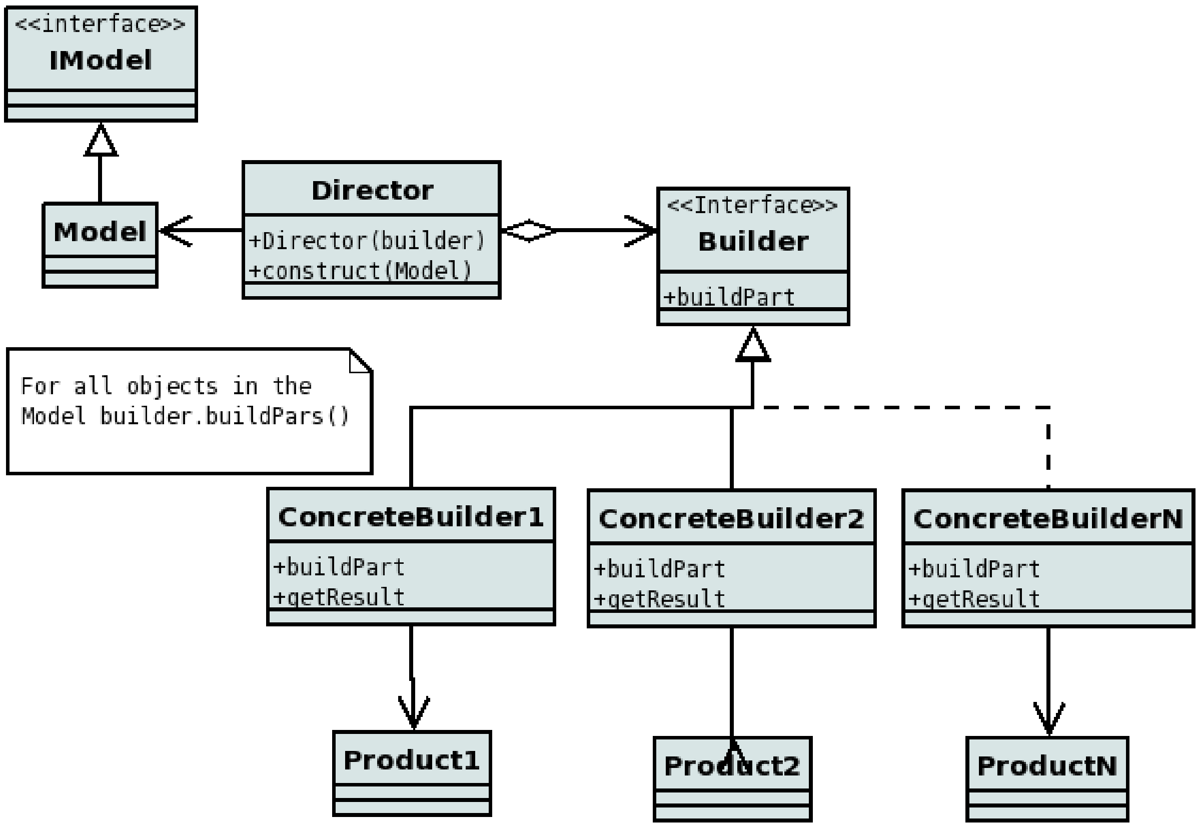


Figure 5.3.: A general overview of the Builder design patterns - A simple Builder Methods with the basic classes

For example: we have a Hungarian text and we want to translate it into different languages. The input text is always the same but the translations are different in each language.

In a case where the developers have to work on complex objects, then the construction of the objects should be separated from its representation. Hence, the same construction process can create different representations of the objects[204].

Figure 5.3 shows an overview of this design pattern. First a *Model* is defined as an implementation of the *IModel* interface. The Model is used to create the *Product* which is a representation of the object that is being built. The *Director* constructs an object using the *Builder* interface. The Builder is an interface which defines the abstract methods that are used to create the parts of a Product object. The *ConcreteBuilder* implements the Builder interface which constructs and combines together parts of the Products.

Structural patterns

The Structural design patterns deal with the composition of Classes and Objects that are used to form larger structures[204]. The Class describes an abstraction and inheritance of various classes that are used to compose an interface, while the Object describes how the objects can be associated and composed to form larger structures.

Composite

When software developers are developing applications, they are usually confronted with an individual or collections of objects which form their application. The Composite design pattern is used to compose these objects into tree structures to represent part or whole hierarchies.

This pattern allows clients to use individual objects and compositions of objects uniformly [205]. The tree-structure data has leaf nodes and branches that make the code very complex, difficult to read and understand. The concept of the tree-structure is to compose one or more similar objects using 1-to-many, “has a” or “is a” relationships into a single interface together with similar functionalities which can be uniformly treated by the clients.

Figure 5.4 shows the basic classes of the Composite design pattern. The *Component* is an abstract interface with well defined methods which perform different operations on the composition of objects that participate in the hierarchical data structure. Each primitive object in the composition has been assigned a well defined behaviour described by a *Leaf* object. As the Leaf object has no children, they can implement services by re-implementing the abstract methods of the Component interface. The *Composite* stores the child components and implements the various methods that are needed in order to perform child related operations. The *Client* manipulates objects using the Composite classes adding various Components or another Composite into it.

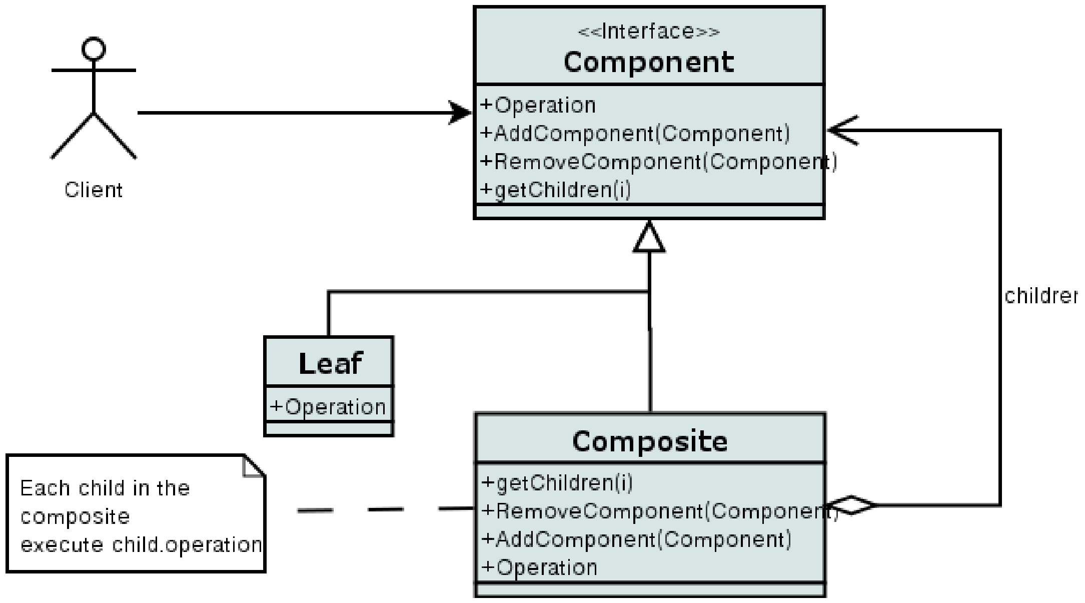


Figure 5.4.: A general overview of the Composite design patterns - A simple Composite pattern with the basic classes.

Behavioral patterns

These patterns deal with the interactions (communications) between objects which have to communicate to each other. These loosely coupled objects can be used as a key to *n*-tier architectures. As the name of this pattern 'Behavioral' suggest, they encapsulate the behaviour of the algorithms into a single object which can perform alone.

Interpreter

This pattern specifies how to evaluate sentences in a language which is described in terms of formal grammars [1]. This design pattern is not really used to solve a complex problem, but it is very useful to solve simple problems in an elegant way. The implementation of this pattern uses the Composite pattern which defines only the structure applied to solve a specific grammar while the Interpreter design patterns defines the behaviour.

For example: suppose we have various quantities in a text: litre, millilitre or decilitre. This pattern can be used to identify these quantities and convert them to a single quantity.

Figure 5.5 shows an overview of the Interpreter design pattern. The *Client* provides an abstract representation of a syntax tree using the grammar in order to represent a particular sentence in the language. In addition, the Client invokes the Interpreter operation. The *Context* contains information that can be globally accessed by the objects. Each *ConcreteInterpreter1..n* implements and interprets one operation by reimplementing the corresponding abstract method

[1]The grammar is a set of rules which describe how to form strings from the language's alphabet that are valid according to the language's syntax[199].

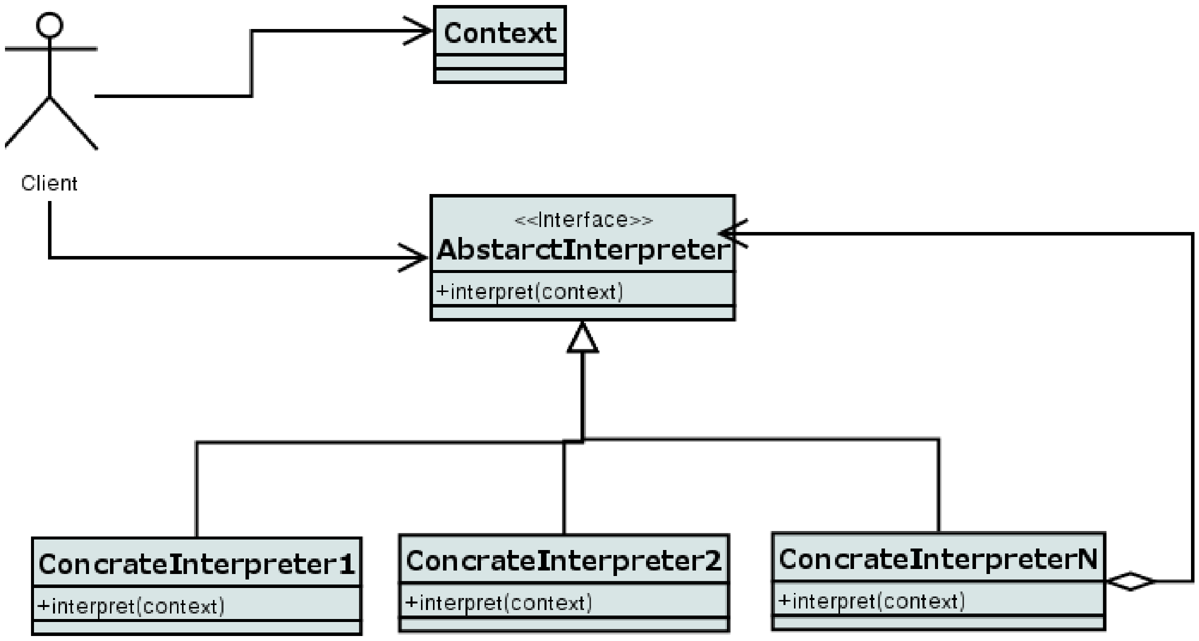


Figure 5.5.: A general overview of the Interpreter design patterns - A simple Interpreter with the basic classes.

which was declared in the *AbstractInterpreter* interface. This interface declares all the methods which will be used during the execution of one operation.

Template Method

In most of the cases where an algorithm is used to perform an operation, the Template Method can be used. The Template Method defines the program skeleton of an algorithm which can be composed by steps that follows various rules[206]. These steps can be overridden by subclasses in order to allow different behaviours without changing the rules which are followed during the creation of the algorithm.

Figure 5.6 shows a simple overview of the Template Method. First an *AbstractClass* has to be created with well defined abstract methods that provide the basic steps of an algorithm design. The AbstractClass defines the skeleton of an algorithm and its abstract methods describe private operations that are implemented by the *ConcreteClass*. This class implements the algorithm's specific steps which perform various operations that were encapsulated in the subclass.

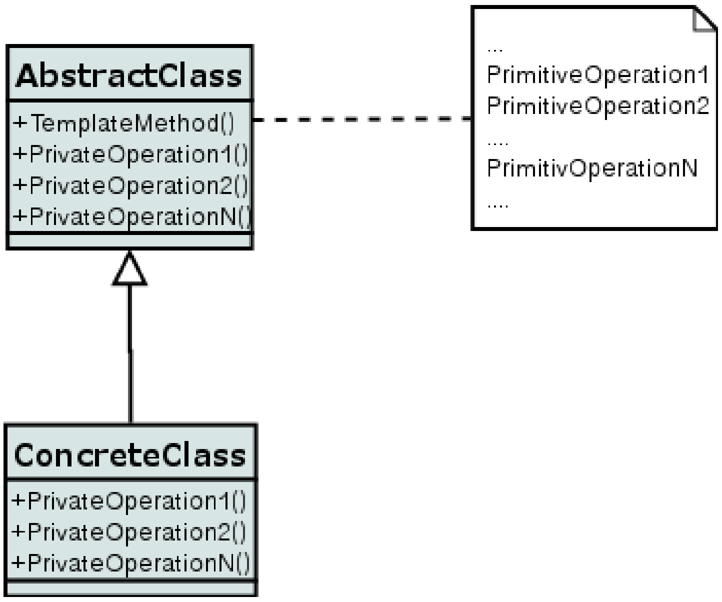


Figure 5.6.: A general overview of the Interpreter design patterns - A simple Interpreter with the basic classes.

Chain of Responsibility

The Chain of Responsibility pattern is used to handle requests (typically events) by using a chain of objects[207]. Usually, each Handler knows its successor which allows it to forward a request through the chain of Handlers until one of them can handle this event.

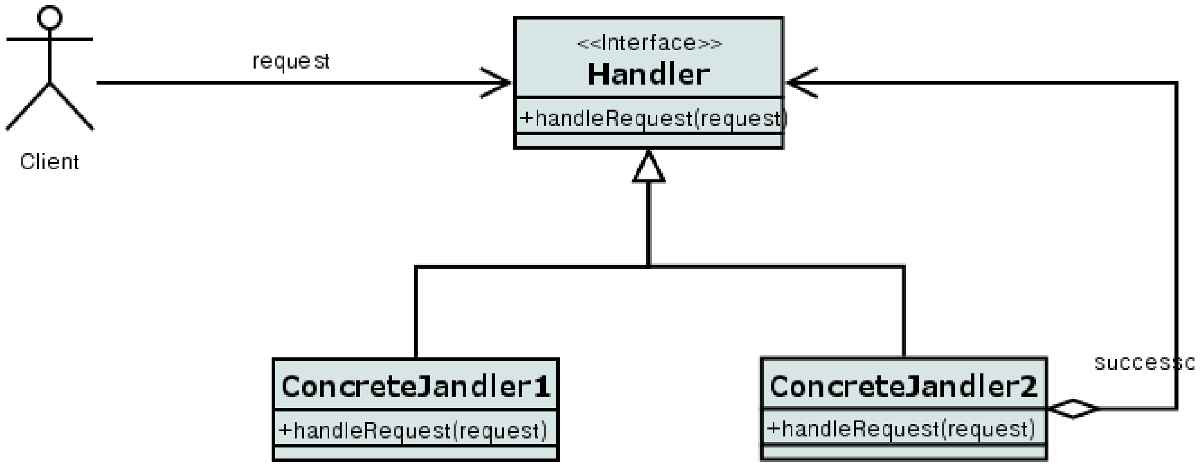


Figure 5.7.: A general overview of the Chain of Responsibility design patterns - It consists of the basic classes which are used to create a Chain of Responsibility design patterns.

Figure 5.7 shows the UML diagram of classes that participate in the composition of the Chain of Responsibility design pattern. The *Handler* defines the interface that is used to handle requests. The *ConcreteHandler* implements the Handler interface in order to handle the requests and is responsible for forwarding the request to its successor if it can not handle the request. The *Client* is the starting point of the request handling procedure, because it sends the request to the first Handler object in the chain that is supposed to handle the request.

We can have many Handlers in a system which are grouped by a common interface. Each group of the Handler can handle specific requests.

The Chain of Responsibility design pattern has a strong advantage, because it promotes the idea of loose coupling[208]. However, the disadvantage of this pattern is that the chain can be broken when we forget to add a few lines to the code which forward the request to another Handler. The result of this is that the system may have unhandled requests.

Architectural patterns

Architectural patterns describe a high-level structure of software systems. They contain a set of predefined sub-systems, which specify their responsibilities and define the relationships between the sub-systems[219]. Several Architectural patterns exist and are widely used. Each of them helps to achieve a specific global system property. We studied the **Model View Controller** and the **Hierarchical Model View Controller** and on the basis of our study we have implemented the **Composite Model View Controller**. A general overview of these patterns is described in the following subsections.

Model View Controller

Model View Controller (MVC) is an architectural design pattern often used by the applications that need the ability to present the same data to their users. Most of the time the MVC is widely used to maintain multiple views that perform on the same data. The MVC hides the complex structure of the object by separating them into three categories:

1. Model manages the data which is required by the application. It disposes the information to the view by providing detailed instructions.
2. Views are typically a user interface element to display all or part of the data which is managed by the Model.
3. Controllers handle different events which can affect the model or the view(s). Usually, the user triggers an event using the View, while the Controller receives the request which is generated by the event and returns a response by initiating calls on Model objects.

Figure 5.8 shows the model/view/controller triads that play a vital role for developing Graphical User Interface elements. Various implementations of MVC have been developed and are widely used by different user communities. We distinguish two different categories depending on where it is used:

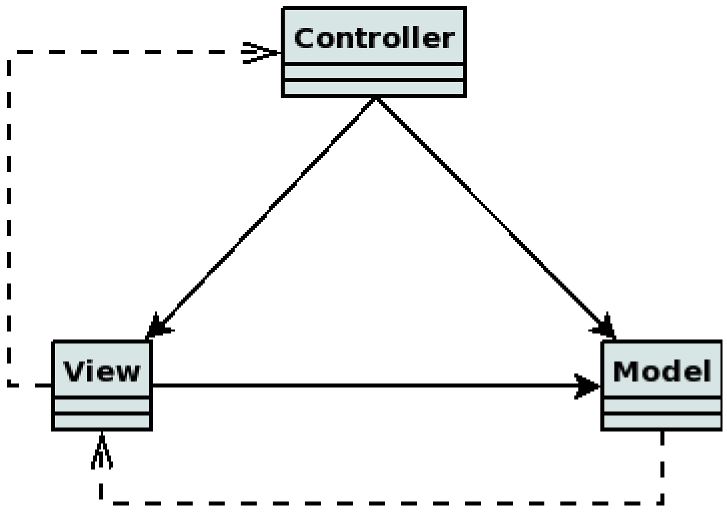


Figure 5.8.: An abstraction of the Model View Controller - Tree classes which are used by the Model View Controller.

1. based on GUI frameworks; such as Qt[209], Java Swing[210], Microsoft Foundation Class Library (MFC)[211], GTK+[212] etc.
2. based on web-based frameworks; such as Spring Framework[213] Django[214], Pylons [215], ASP.NET MVC Framework[216], Struts[217], Oracle Application Framework [218] etc

When the user interacts with the GUI by changing the graphical elements, then one or more event may be generated. The events are treated by the Controllers that then modify the model or the view, or both. Whenever the Controller changes the model data will be modified and the view(s) automatically updated.

Hierarchical Model View Controller

While the MVC paradigm controls GUI elements (widgets), the Hierarchical Model View Controllers (HMVC) introduces a layered design methodology which can be used to develop a complete presentation layer (or client tier). As an extension of the basic MVC, it aims to handle events of all child widgets which have the same parent widget, instead of having an MVC triad for each particular widget[220]. A detailed view of the HVC is given in Figure 5.9.

The *View* is the interaction point of the users. The *container* is the highest level of the GUI. As the name suggests, it contains multiple views which may have relations to each other. Depending on the complexity of the views, it may assign to them an independent Controller and Model.

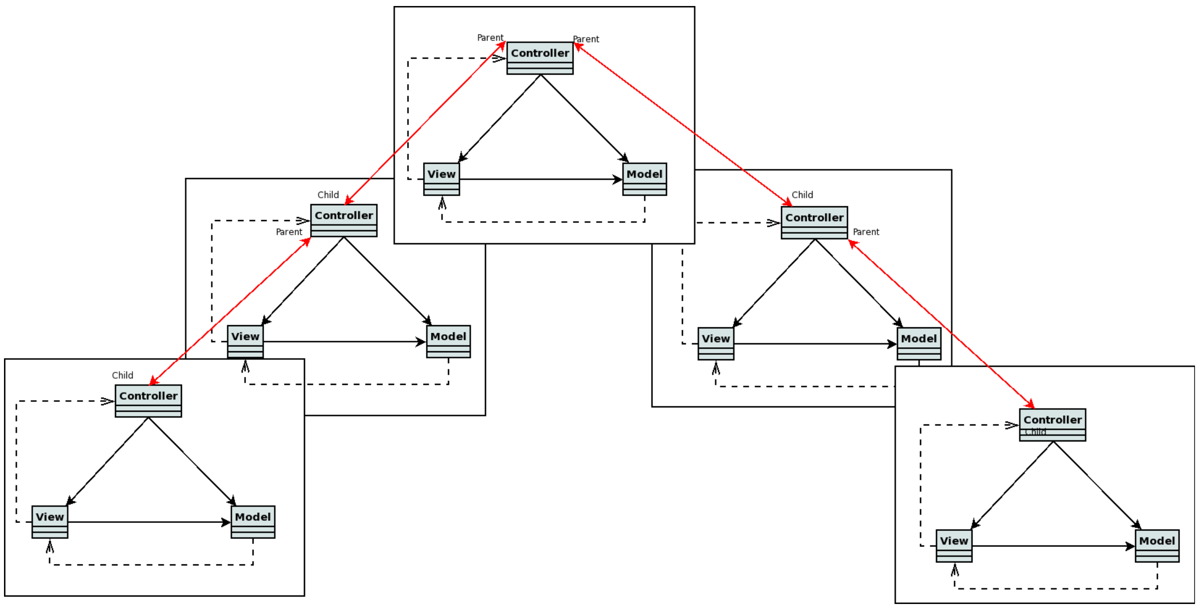


Figure 5.9.: Overview of the HMVC design pattern - This figure shows a hierarchy of controllers which are communicating to each other using the parent-child relationship.

The *Controller* uses the model and provides a distributed control of events through a parent-child hierarchy. It is responsible for responding to all the application-level navigation and data-request events[221]. It implements the Chain of Responsibility design pattern. Consequently, each Controller in the hierarchy knows its successor and that means the parent can send events to the child Controllers to perform request which are generated by the View or other Controller. The HMVC allows the developers to define models at every layer of the hierarchy. Implementing them is not required since this always depends on the application design and requirements. The *Model* disposes the information which has to be displayed.

Composite Model View Controller

The Composite Model View Controller (CMVC) is an architectural design patterns which controls different GUI elements that compose the GUI components in a developer- friendly fashion.

Messages are simple objects that encapsulate information and deliver this information between Controllers. A Message object has two properties: *Action* and *Items*. The Action tells the Controller to perform manipulations on a GUI component or display data which is requested by the user, while the Items are used to encapsulate the data which will be used by the GUI element or the Controller. In addition the Message can contain *multiple actions* that can be performed on different GUI element which are parts of the GUI component.

In our case, a **View** consists of a single or various graphical elements which are stored in a **Container(s)**.

The Model encapsulates and manages the data. The Model only provides data to the Controllers. This allows us to have a complex GUI structure with various components which are distributedly controlled.

The **Controller** is the key element of the CMVC design pattern. The implementation of a Controller is based on the Composite and the Chain of Responsibility design patterns that are used to compose the Controllers into a tree structure. Each Controller has parent and child(ren) controllers and each of them has an associated MVC triad. While the Controllers of the MVC and the HMVC manage the graphical elements through events, the CMVC controls its graphical elements by sending Messages to the appropriate Controllers. The Controllers can send Messages to any other Controller. The *root* Controller deals with the Model and distributes the messages to the other sub controllers. In our case the root Controller only transfers messages to its children; it never tries to interpret the messages or perform one action. The Model are distributed between the child controllers. Consequently, when an action is performed in one GUI element, it will be handled by its Controller. If this Controller cannot handle the action, it will send a message to its parent Controller. This procedure can continue recursively until the appropriate Controller handles the action and returns the result to the requester.

Figure 5.10 shows the UML class diagram of the CMVC design pattern.

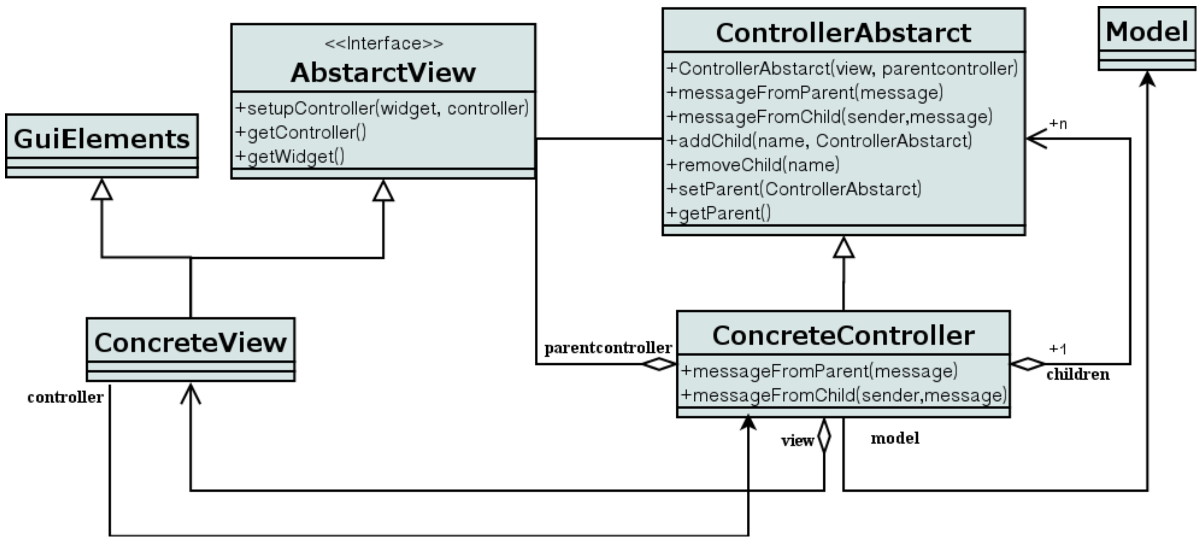

<<Interface>>
AbstarctView
+setupController(widget, controller)
+getController()
+getWidget()
ControllerAbstarct
+ControllerAbstarct(view, parentcontroller)
+messageFromParent(message)
+messageFromChild(sender,message)
+addChild(name, ControllerAbstarct)
+removeChild(name)
+setParent(ControllerAbstarct)
+getParent()
Model
GuiElements
+n
ConcreteView
ConcreteController
+messageFromParent(message)
+messageFromChild(sender,message)
parentcontroller
+1
children
controller
view
model

Figure 5.10.: The CMVC design pattern - A general structure of the CMVC.

The *GuiElements* , as the name suggests, consists of different GUI elements which are implemented in a programing language. It displays the Model to its users and it exploits the benefits of the programming language. Actually, each GuiElements has an associated MVC triad (Model, View, Controller).

The *AbstractView* interface declares the abstract methods which must be used to create the CMVC.

The *ConcreteView* implements the AbstractView methods. Each ConcreteView must have an associated Controller which is used to control the GuiElements. The ConcreteView:

- creates its own Controller;
- makes a relation between the newly created and the existing Controller of the MVC, by assigning this new Controller into the MVC;
- sets up the child Controllers if the ConcreteView contains another GUI element which has to be controlled.

The *ControllerAbstarct* declares and implements the interface which is used in the composition.

The *ConcreteController* implements the ControllerAbstract interface. Each ConcreteController has a parent controller and child Controllers. Actually, it stores the Controllers in a hierarchical structure and provides the methods which are needed for communications and for manipulating the tree elements. The ConcreteController knows the view(s)[2], handles the GUI events, and provides the data to the GUI using the Model object.

The **Model** provides the data to the ConcreteController. The data source can be a database, Extensible Markup Language (XML)[222] file, files with objects, file system etc.

The CMVC flexibility

CMVC can be used when we have a GUI and we want to control the GUI elements in an easy user-friendly way. The CMVC allows us to have Controllers which are implemented in different programming language. For example: the GUI might be implemented in Java and the Model might be implemented in Python. That means the Controllers of the GUI are implemented in Java while the main Controller is implemented in Python. Consequently, we have a layered architecture with two layers that are dedicated to support different purposes. The Message plays a vital role when a Python Controller has to communicate to the Java Controller. The Message can be translated to a Java object or vice versa. In addition it can be translated to a literal string which can be interpreted in both programming languages.

The CMVC can support a client server architecture. The Model is located on the server side together with the main Controller which provides data to the client using the Message.

CMVC allows us to distribute the data within various Controllers.

CMVC supports a pluggable architecture. Each GUI component can work standalone with minor modifications. In addition a new GUI component can be integrated without modifying the whole application structure. A new Controller can be easily implemented and added to the hierarchy of Controllers. CMVC is currently implemented in Python, but it can be easily implemented in other object oriented programming languages.

[2]We can assign the same Controller to two different views.

CMVC interaction with Controllers and their role

HMVC sends the events to all the Controllers, while CMVC only sends messages to the appropriate Controller or a dedicated Controller. But if the appropriate Controller can not handle a message, then it will be sent to its parent. The Controller can send a message to its children. Usually when a request has to be performed in different views the main Controller sends a message to its child Controllers.

The Model can be distributed among various Controllers. That means if we have a complicated tree structure, we can split the tree into various sub trees. The root of the sub tree will become the main Controller. This mechanism reduces the number of messages and the communication.

Figure 5.11 shows a general example of a complex Controller hierarchy structure that illustrates the described above statement.

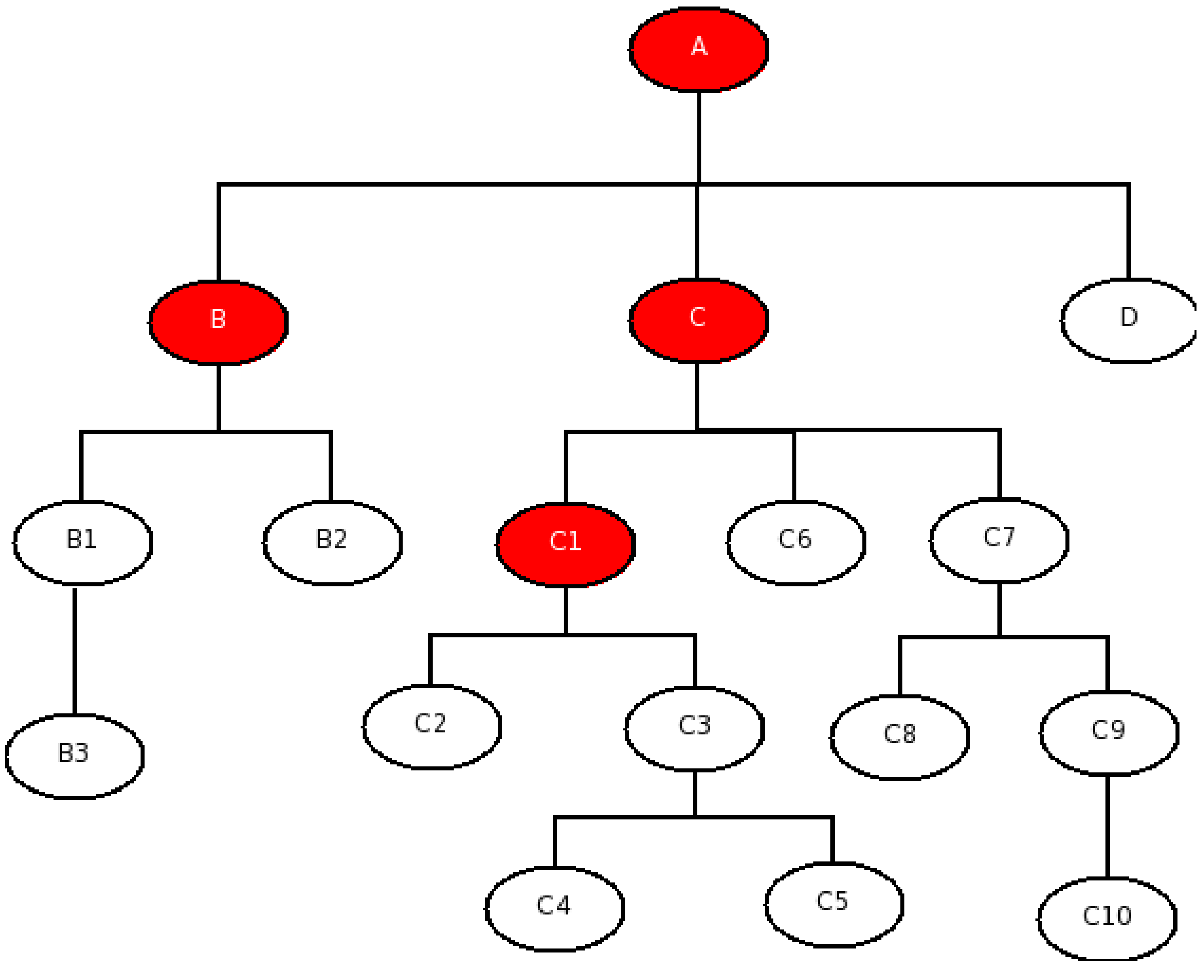


Figure 5.11.: CMVC example - Tree of Controllers

In the figure we distinguish two types of Controllers: main Controllers which are marked in red such as A, B, C, C1 and normal Controllers which are not marked. A is the main Controller which have all the data associated to a Model. The A Controller distributes the Model to the B, C and C1 Controllers and they become main Controllers. The B, C and C1 Controllers consists of 3 different sub trees; that means they provide the data to their children. When the C10 controller has to send a message to the B3 Controller then it will be sent via C9, C7, C, A, B, B1, B3 Controllers.

CMVC knows all the children which means that the main controller can send messages to them. This reduces the number of communications. If the Controllers are located in a server, the load on the servers will be reduced by a large factor.

5.5. Data handling in Feicim

One of the purposes of Feicim is to present data to it's users. This requires that the data is temporarily stored in memory. We introduce the notion of **Entities** and **Items** which store data in memory. The Entity stores the attributes of various objects while the Item stores these entities in the memory in a hierarchical (treelike) format.

Feicim has to work with millions of entries which requires applying different strategies for efficient use of memory:

- Only load data which is needed, since that part of data is small and easy to handle.
- Do not keep data in the memory for a long period. To achieve this requirement, part of the data has to be saved in a temporary file.
- The old memory space should be destroyed before a new memory allocation occurs.
- At any given time, the memory should not have the same information twice.

5.6. Managers

Managers manage the entities and provide methods which are used to browse through the entities. They are responsible for creating tree structures in an efficient way. Usually, a Client is associated to each Manager. The User can interact with the Managers using the Client. Figure 5.12 shows a general UML diagram which describes a Manager.

We distinguish between Clients and Managers. Each Client has assigned to it a Manager. Actually, the Manager is the key element, because it deals with the entities, while the Client provides the access to the Manager functionalities.

The *IEntitySystem* is an abstract interface that describes the common methods that will be reimplemented by the appropriate class. The *BaseESClient* has an associated ESManager that allows the user to interact with its associated Manager. The *ConcreteClient* exposes all the available methods, implemented by the *ConcreteManager*, which manage the entities and creates the tree of entities. The *IEntitySystemManager* defines the methods which are only

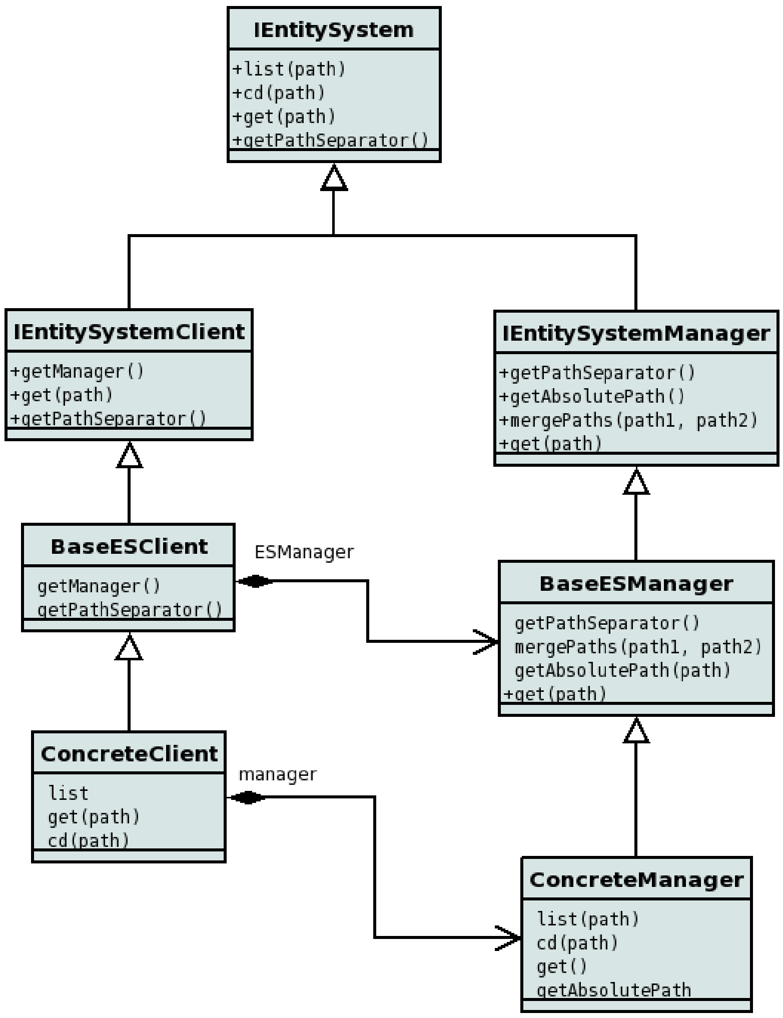


Figure 5.12.: UML diagram of a Manager -

used by the Managers. The *BaseESManager* implements the IEntitySystemManager abstract methods and deals with the methods which are common to other Managers.

5.7. Summary

To analyse huge amounts of data in a easiest way it is not trivial without having appropriate tools. The following steps has to be performed during data analysis:

- understand the content of data which have to be analysed.
- develop or use an existing algorithm(s) which used to extract information from the data, or make different measurements.
- define the input data which will be used by the algorithm.

- create jobs using different input files and run them on the Grid.

In particle physics each step which are described above has an associated application which used to solve specific problems (locate datasets, create a Grid job, create an algorithm, etc.). An tool required which integrates the applications which belongs to the different steps into a single distributed analysis tool. This tool called Feicim. The architecture of Feicim is very important, because it has to provide all the necessary functionalities required to create different components which are used for analysis. In order to control the components of Feicim and ensure the communication between them we are created the Composite Model View Controller. In addition we used various design patterns for developing the components of Feicim.

6. Distributed analysis using Feicim

In this chapter we present the Feicim Data Browser component which is used to discover the contents (i.e. physics information) stored inside the data files. This information can be histogrammed, filtered, and written to a user file for future analysis. A general tree data structure is introduced in section 6.1. Section 6.2 presents different existing tree traversal algorithms and a new Feicim Tree Traversal Algorithm which is used to discover a data file content. In section 6.3 a detailed overview of the Feicim Data Browser is given. The architecture of the Feicim Data Browser components and a python implementation of the Feicim Tree Traversal Algorithm are presented in section 6.3.1. Information from several data files can be combined and analysed together, either locally or using the Grid. This is accomplished using the Feicim Data Analysis which is presented in section 6.4. Section 6.4.1 presents the Feicim Data Analysis architecture which meets the requirements for performing data analysis using distributed computing resources.

6.1. General tree data structure and tree traversal algorithms

A tree is a widely used non-linear data structure that stores elements hierarchically. A general tree structure is shown in Figure 6.1. A *node* is a structure which may stores a value, or a condition, or a separate data structure or a tree of its own.
Definition: A tree[243] is a finite set T of one or more nodes
$T = \{r\} \cup T_1 \cup T_2 \cup ... \cup T_n$, with the following properties:

1. The set consists of a specially designed node r which is called the *root* of T
2. The remaining nodes are partitioned into $n \geq 0$ subsets, $T_1, T_2, ..., T_n$, and each of these sets is a tree. The trees $T_1, T_2, ..., T_n$ are called the subtrees of the root.

According to the previous definition we use $T = \{r, T_1, T_2, ...T_n\}$ to denote the tree T and the following terminology is used:

- child: Each root r_i of subtree T_i of tree T is called a child of r.
- parent: The node r of tree T is the parent of all roots r_i of the subtrees $T_i, 1 < i \leq n$
- siblings: Two roots r_i and r_j of distinct T_i and T_j of tree T are called siblings.

- internal node (also called branch node, inner node): Any node of a tree that has child nodes.
- external nodes (also called leaf node, outer node, terminal node): Any node of a tree that does not have child nodes.
- edge of a T: is a pair of nodes *(u, v)* such that u is the parent of v or vice versa.
- path of a T: is a sequence of nodes connected by edges. If R is a set of nodes of a tree T, then $P = \{r_1, r_2, ..., r_k\}$, is path of T, where $r_i \in R$, for $1 \leq i \leq k$ such that the i-th node r_i in the P, is the parent of the $(i+1)$-th node, r_{i+1} in the P. The length of the path P is $k-1$.
- height of a node is the length of the longest path from the node to a leaf.
- depth of a node is the length of the path from the root to that node.

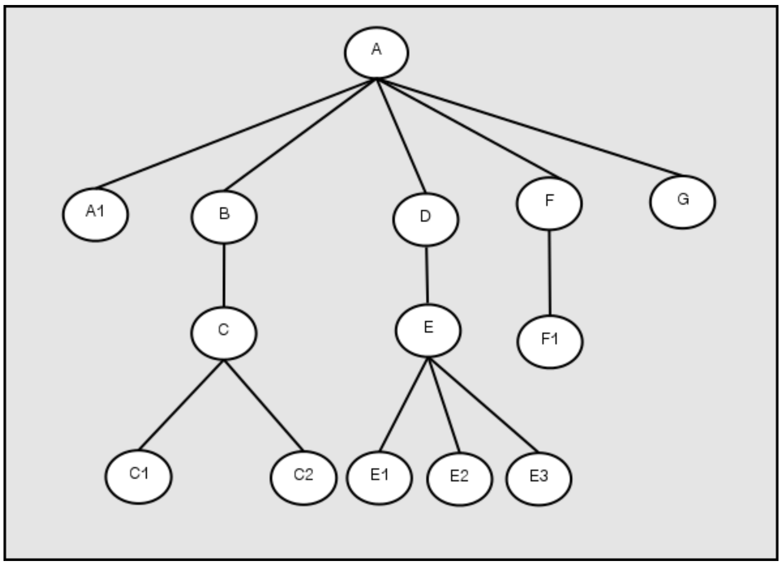


Figure 6.1.: Example of a tree - A general tree T_A

The tree in the Figure 6.1 can be written using the tree definition as:
$T_A = \{A, \{\{A1\}, \{B, \{C, \{C1\}, \{C2\}\}\}, \{D, \{E, \{E1\}, \{E2\}, \{E3\}\}\}, \{F, \{F1\}\}, \{G\}\}\}$

6.2. Tree traversal algorithms

The objective of traversal is to visit (perform some operation at) each node of a tree. The trees can be visited in several different traversal orders. However, all of them have a common characteristic: that they systematically visit all the nodes in the tree. The following types of tree traversal algorithms are available: depth-first traversal and breadth-first traversal.

6.2.1. Depth-first traversal (DFT)

The DFT algorithm traverses the entire subtree of a node before beginning traversal of any other subtree of another node. It tries to go deeper in the tree before exploring siblings. There are three different types of depth-first traversals: pre-order, in-order, and post-order. These types of traversal are usually used to traverse a binary tree. To traverse a generic tree, the pre-order traversal algorithm is usually used. However, depending on the problem in-order or post-order tree traversal algorithm may be required. The depth-first tree traversal algorithm when the tree is generic is the following:

Algorithm 1 depth_first(T)

```
Input: is a tree.
Output: Nothing.
 1: $container \leftarrow root$  // store the root node of $T$ tree in a container.
 2: while $notcontainer.empty()$ do
 3:     // while there are nodes in the container
 4:     $Node \leftarrow container.pop()$  //get the next Node form the container
 5:     for $i \leftarrow Node.getChildren()$ do
 6:         //for each child of Node
 7:         container.put(i)  //store $i$-th child of Node in the container
 8:         visit(Node)  //do some work on Node
 9:     end for
10: end while
```

For example: Given the T_A tree which is shown in figure 6.1 . Algorithm 1 visits the nodes of the T_A in the following order: A, A1, B, C, C1, C2, D, E, E1, E2, E3, F, F1, G

The equivalent recursive algorithm for the one introduced above is the following:

Algorithm 2 depth_first(Node)

```
Input: is the root node of a tree.
Output: Nothing.
1: if $Node <> None$ then
2:     return
3: else
4:     for $i \leftarrow Node.getChildren()$ do
5:         //for each child of Node, visit the $i$-t child of Node (perform some operation)
6:         visit(i)
7:         depth_first(i)  //go to the i-th child of the Node (one level deeper of the Tree).
8:     end for
9: end if
```

For example: The input parameter is T_A which is shown in 6.1. Algorithm 2 visits the nodes of the T_A in the following order: A, A1, B, C, C1, C2, D, E, E1, E2, E3, F, F1, G.

6.2.2. Breadth-first traversal (BFT)

The BFT algorithm visits every node on a level before going to a lower level of the tree T. This algorithm is also called level-by-level traversal algorithm. The differences between the two traversal algorithms is the type of the *container*. The depth-first algorithm uses a stack, while the breadth-first algorithm uses a queue. In algorithm 1 when the container is a queue it visits the tree nodes in the following order: A, A1, B, D, F, G, C, E, F1, C1, C2, E1, E2, E3

6.2.3. Feicim tree traversal algorithm (FTA)

The FTA is a novel traversal algorithm which visits the tree nodes in a given way (order). The way to visit the nodes of a tree is given as a tree path. For example: P = {A,B}, P = {A,B,E}, etc. are paths of the tree T (see. Figure 6.1). We also use the notation /A/B, /A/B/E to describe a path (calling to mind a unix directory structure). The FTA algorithm uses two functions: **isLeaf** and **visit**. The isLeaf (Algorithm 3) function is used to decide the type of a given node of a tree.

Algorithm 3 isLeaf(Node)

```
Input: Node
Output: True or False.
1: if $len(Node.getChildren()) > 0$ then
2:     //if the Node have at least one children, it is not leaf
3:     return False
4: else
5:     return True
6: end if
```

We use the visit function (Algorithm 4) to go one level deeper (in other words to visit the next node) of a given tree. If we visited one node of a tree and we want to visit the node in the next level of the tree, we have to provide the subtree and the name of the node in the next level. For example for a given $T_B = \{B,\{C,\{C1\},\{C2\}\}\}$ which is a subtree of the T_A shown in figure 6.1 and $nextNode = \{C1\}$ the visit(T_B,$nextNode$) output is an empty set $\{\}$. If we change the $nextNode$ value to $nextNode = \{C\}$ and we call again the visit (T_B,$nextNode$), the *visit* returns $\{C,\{C1\},\{C2\}\}\}$ subtree.

The fta(T,P) (Algorithm 5) algorithm uses the *isLeaf(Node)* and *visit(T, nextNode)* functions and visits only the nodes of T which are given by P (path). This generic algorithm can be applied to a particular data structure to visit a node in a sub tree.

The advantage of this algorithm is it is not visit all the nodes in a tree. Consequently, it is faster than the existing tree traverse algorithms in order of magnitude.
For example: For a given tree T_A which is shown in Figure 6.1 and P where:

1. $P = \{A\}$, fta(T_A,P) returns $\{A1,B,D,F,G\}$

Algorithm 4 visit(T, nextNode)

Input: T, nextNode, where T is a subtree and *nextNode* is the name of a existing or non existing node of *T*.
Output: is a subtree.

```
1: $node \leftarrow None$ //We do not have the next node.
2: for $i \leftarrow T.getChildren()$ do
3:    // for each children of the subtree $T$
4:    if $i = nextNode$ then
5:       //If the $i$-th children of the subtree $T$ equals the nextNode, the next node (next subtree) become $i$
6:       $node \leftarrow i$
7:    end if
8: end for
9: RETURN node
```

Algorithm 5 fta(T,P)

Input: is a given path P = $\{r_1, r_2, ..., r_k\}$ where $r_i \in R$, R is a set of nodes, $1 \leq i \leq k$ and T tree.
Output: the child nodes of the r_k.

```
1: for $i \leftarrow P$ do
2:    // for each node in the $P$ container
3:    $node \leftarrow visit(T,i)$ // visit the $i$-th node, the node become the subtree of the $i$-th node
4:    if $isLeaf(node)$ then
5:       if the node does not have any child, return that node.
6:       RETURN node
7:    else
8:       if $len(P) > 0$ then
9:          // If we have node in the $P$ container which are not visited, visit that node.
10:         RETURN fta(node, P)
11:      else
12:         // all the nodes which were in $P$ are visited. The node contains the children of the last node which was in $P$
13:         $nodes \leftarrow []$ // it is a container and used to store the children of the node
14:         for $i \leftarrow node.getChildren()$ do
15:            // for each children of the node
16:            $nodes \leftarrow nodes + i$ // store the node in the container.
17:         end for
18:         RETURN nodes // it returns the children of the last node in $P$
19:      end if
20:   end if
21: end for
```

2. $P = \{A, B, C\}$, the fta(T_A, P) returns $\{C1, C2\}$.

6.3. Feicim Data Browser

In most of the cases in High Energy Physics the users run analysis using different input files. These files are distributed among different Grid sites. The LHCb Bookkeeping System was developed to discover these input files using their associated metadata information.

The Feicim Data Browser is a component used to discover the content of these files. It allows the user to select any data object (e.g. particles, tracks, energy depositions etc.) and to make plots using the attributes of these objects. It hides the complexity of the LHCb software and provides a user friendly GUI to make various selections and plots.

6.3.1. Feicim Data browser architecture

The architecture of the Feicim Data browser is shown in Figure 6.2. It displays three different components based on the Feicim architecture:

- The Application Builder provides the important Gaudi functionality to the Communication and Workflow Manager which are needed to process events.
- The Communication and Workflow Manager creates a Virtual File System from the Event Model, which will be visualized by the Event Viewer.
- The Explorer Viewer provides a Graphical User Interface to browse through the Virtual File System.

Each component is now described.

Application Builder

The **Application Builder** provides access to the Gaudi framework in order to read event data from a file. It initialises the **Gaudi Application Manager** which contains a **Feicim Event Service**. The Feicim Event Service consist of an Event Data Services and different algorithms used to manipulate the data.

Gaudi Application Manager the Feicim architecture allows to integrate different physics software and it exposes their functionality. The *ApplicationManager* is an interface which declares all the methods required to handle event data. The *GaudiApplicationManager* implements the abstract method and creates the relation between Feicim and Gaudi.

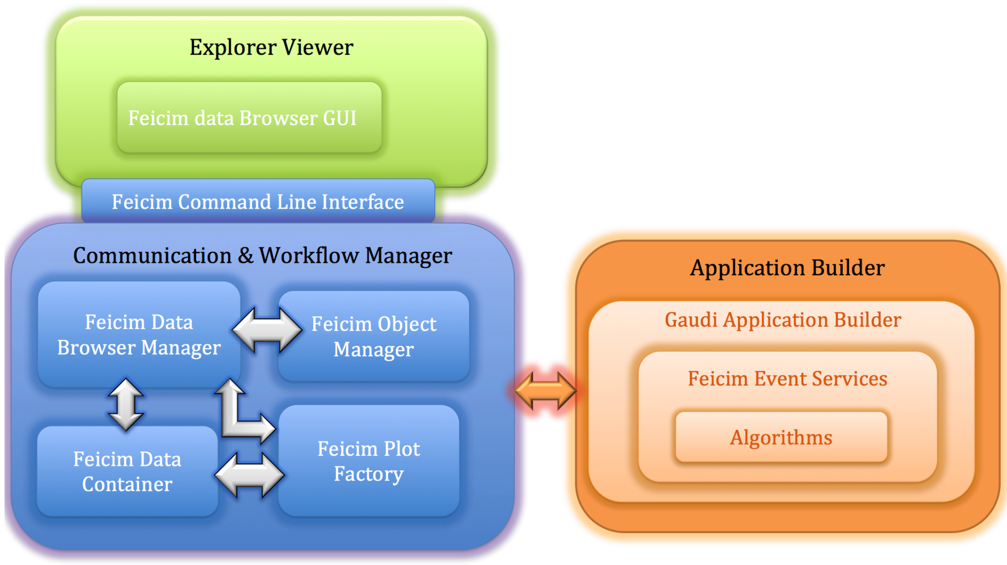


Figure 6.2.: Overview of the Feicim Data browser architecture - This picture shows the components and their relations.

Feicim Event Service The Feicim Event Service is a module which instantiates an Event Data Service and provides algorithms to manipulate the object in the TES.

Algorithms As introduced in section 4.1, the objects in a data file produce a tree structure, which is called an event tree, and it is shown in Figure 6.3. The path is used to identify an object. For example: the EW identifier is '/Event/EW'. A branch node may contains a **DataObject** or a **Container** which are abstract data types:

- DataObjects: They have a single instance per event and they contain another DataObject or a Container. For example: Event (/Event), DAQ (/Event/DAQ), EW (/Eevent/EW), Phys (/Event/Phys) are DataObjects.

- Containers: They store a collection of small objects such particle, track, etc. These objects are called contained objects and they only can be accessed via the container. The Particles (/Event/EW/Phys/WMuSingleTrackNoBias/Particles) node which can be found in Figure 6.3 is a container which is a Gaudi KeyedContainer. The contained objects which are in this container are LHCb::Particle objects. The children of this node are method names (BoostToCM, Coordinates, Vect, Beta, etc) of one object (LHCb::Particle) from the container which returns a value which can be a fundamental data type or an abstract data type.

A leaf contains fundamental data types such as integers, floats, chars, booleans, etc. For example: the leaves E (/Event/EW/Phys/WMuSingleTrackNoBias/Particles/momentum/E) and Et (/Event/EW/Phys/WMuSingleTrackNoBias/Particles/momentum/Et) in Figure 6.3 contain floats for the magnitude of a particle's energy and transverse energy respectively.

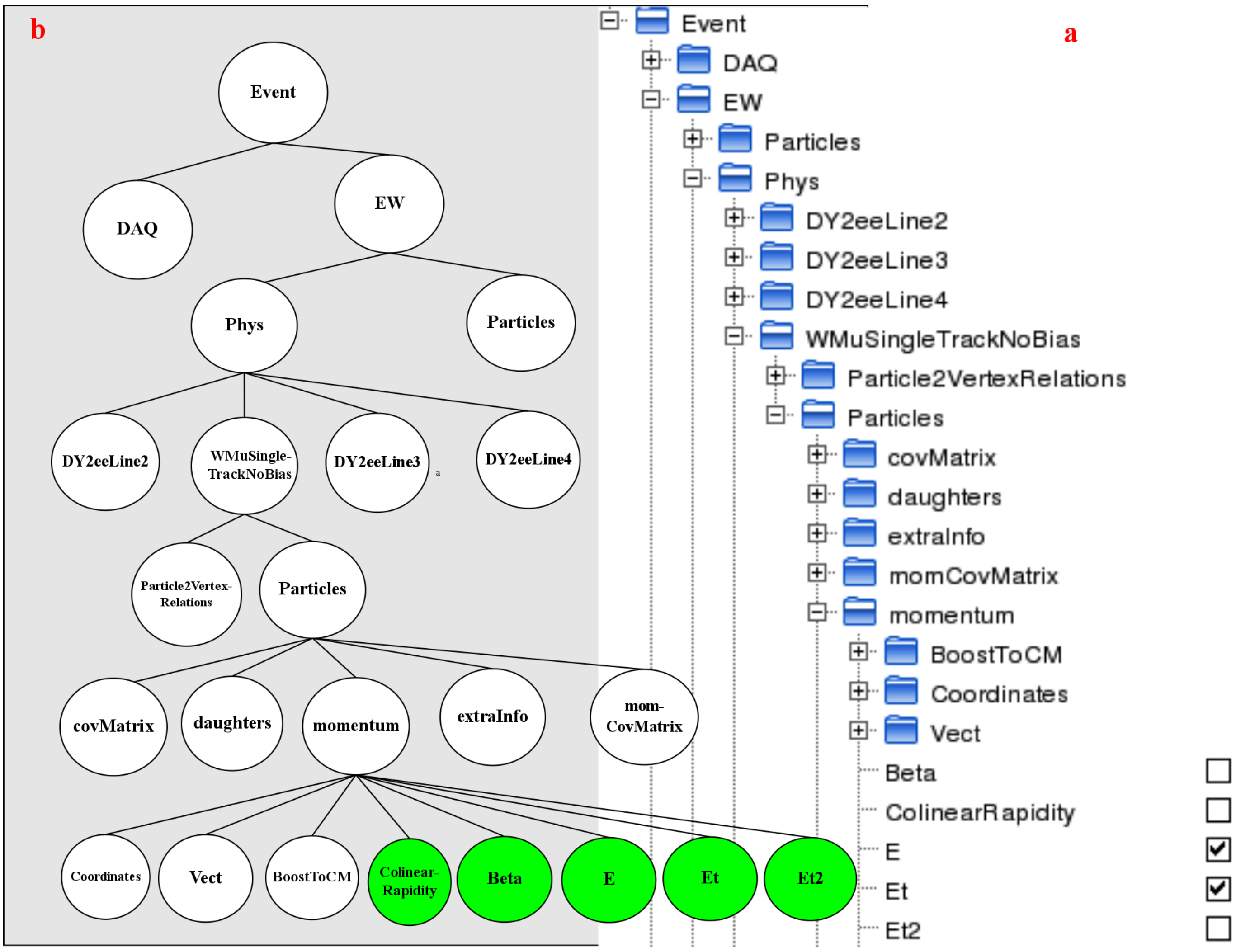


Figure 6.3.: A data file treelike structure which is similar to a file system. - It provides two different view of how the data are stored inside a data file. View *a.)* is produced by using the Feicim Data Browser; the folders are name of an object (such as Event, DAQ, etc) which contains other folders or selectable check boxes which are attributes (such as E, Et, etc.). View *b.)* is a different view of the data file structure. The white circles are folders while the green circles are leaves of the tree.

The root node of an event tree is always called /Event. This component consists of a recursive algorithm which is a python implementation of the **fta** algorithm introduced in section 6.2. This algorithm is used to discover the data file's content by browsing through the data objects.

The isLeaf method is implemented in python and decides the type of a tree node. For example: the nodes which are shaded green in figure 6.3 are leaves, because they store fundamental data type.

The python implementation of the visit method is the following:
The **input parameters** of the method are:

- obj, an object which is one of the nodes of the tree.
- nextNode, the name of the node that has to be visited.
- callMethod, describing how to visit the nextNode.

The **output** of this method is the nextNode object.

```python
##############################################################################
def visit(self, obj, nextNode, callMethod=False):
    node = None
    if callMethod:
       node = self.__objDesc.callFunction(obj, nextNode)
    else:
      node = self.findObject(nextNode)
    return node
```

This method uses the **callFunction** and the **findObject** methods. The callFunction is used to call a method (nextNode) of the object (obj). The findObject is used to retrieve an object from the Transient Event Store. Depending on the input of the visit method, the following examples show how the visit method works using the tree which is shown in figure 6.3:

- n = visit(None, '/Event'); returns the root node. If we call again the same method (n = visit(None,'/Event/EW')), it will return the EW node which is the child of the root node. In these cases the findObject method is used.
- n = visit(obj,'momentum',True); The obj has to be one of the objects from the /Event/EW/Phys/WMuSingleTrackNoBias/Particles container. The momentum has to be a method name of the obj. The visit method result is the return value of the momentum method. In this case the callFunction is used.

The python implementation of the **fta** algorithm is the following:
Input of the algorithm: rootnode, path, newpath, callMethod, description. The rootnode is an object and it is the root node of the tree or a subtree. The path is a list which contains the tree node. It describes how to visit a node. For example: ['Event', 'EW', 'Phys', 'WMuSingleTrackNoBias', 'Particles', 'momentum']. The newpath is a string which contains the name of a visited node. For example: When the EW node is visited, then the newpath is equal to /Event/EW. The callMethod is used to decide how to visit a node. The description variable is used to decide how many elements must be visited in a container. (If the description variable is true, only one element of the container will be visited; otherwise all elements of the container will be visited).
Output of the algorithm: a fundamental data type or an abstract data type.

```
1  ############################################################################
2    def fta(self, rootnode, path, newpath='', callMethod=False, description=True):
3      for i in path:
4           if callMethod:
5             newpath = i
6           else:
7             newpath = "%s/%s" % (newpath , i)
8             path.remove(i)
9           retVal = self.visit(rootnode, newpath, callMethod)
10          if not retVal['OK']:
11             return retVal
12          node = retVal['Value']
13          if self.isLeaf(node):
14             return S_OK(node)
15          else:
16            if len(path) > 0:
17              if not self.__types.isDataObject(node) and self.__types.isContainer(node):
18                callMethod = True
19                items = []
20                for i in node:
21                  retVal = self.fta(i, path, newpath, callMethod, description)
22                  if description:
23                     return retVal
24                  else:
25                    if retVal['OK'] and retVal['Value'] != None:
26                      items += [retVal['Value']]
27                return S_OK(items)
28              elif  self.__types.isDataObject(node):
29                return self.fta(node, path, newpath, callMethod, description)
30              else:
31                rootnode = node
32            else:
33              return S_OK(node)
34     return S_OK(rootnode)
```

From line 3 to 12 it tries to visit the nodes in a given order. If the node is a leaf (line 13), the method returns the current visited node. The algorithm will finish when we have visited all the nodes (line 16). Line 17 decides the type of the node. If the node is a container and it is not a DataObject then the way to visit a node will be changed (line 18). When a node is a container we have to iterate through the objects which are in the container. This mechanism is implemented from line 20 to line 27.

Depending on the output of the **fta** algorithm, we have created the **listNode** and **listObject** methods in order to describe the returned node. The listNode is used when the type of the node is DataObject. It returns the leaves of the node. The listNode method is the following:
Input parameter is an object which is a node of the event tree and its type is DataObject.
Output parameter is a list of node names.

```
1  ############################################################################
2    def listNode(self, node):
3      result = S_ERROR()
4      if node != None:
5        nodes = []
6        for i in self.__evtSvc.leaves(node):
7          nodes += [i.name()]
8        result = S_OK(nodes)
9      else:
10       result = S_ERROR("The node is NULL")
11     return result
```

Line 6 retrieve the leaves of a node from the TES using the Gaudi Event Service.

The listObject method is used when a node is a contained object. The implementation is the following:
Input parameter is a contained object such as a particle (LHCb:Particles).
Output parameter is a dictionary with the method names of the objects and their returned values. The keys of the dictionary are the method names of the obj which return a value while the values of the dictionary are those returned values.

```
 1 ###########################################################################
 2   def listObject(self, obj):
 3 
 4     if self.__types.isContainer(obj) and obj.size() > 0:
 5       obj = obj[0]
 6 
 7     retVal = {}
 8     fs = inspect.getmembers(obj)
 9     for i in fs:
10       res = self.__objDesc.callFunction(obj,i[0])
11       if res['OK']:
12         val = res['Value']
13         if val:
14           retVal[i[0]]=val
15     return S_OK(retVal)
```

Line 4 decides the type of the object. (If an object is a container we do not want to return the container method names and their returned values). In order to list all the available methods of a contained object, we use the inspect module of python[244]. Line 8 lists the object members (method names) of the obj. Line 9 iterates on the method names. Line 10 tries to call the method. If the method has a return value (it is a function), the method name and value is stored in the retVal dictionary (Line 14).

Communication and Workflow Manager

In order to show the content of the database (or a data file) as a File System, we implemented various modules which are part of the Communication and Workflow Manager component. This component deals with the creation of the Virtual File System and provides all the methods which are needed to perform different operations. The Communication and Workflow Manager contain different modules:

- The Feicim Data Container is used to temporarily store the data in memory.
- The Feicim Plots Factory creates different types of plots.
- The Feicim Object Manager is a module which manages the Feicim Event Services.
- The Feicim Data Browser Manager creates the Virtual File System.
- The Feicim Command Line Interface provides a User Interface to work with the Virtual File System.

Feicim Data Container The Feicim Data Container (FDC) is a data structure which is used to *temporarily store* the data in memory.

The data files are large (more than 4 GB). Consequently, it is not advantageous to read all the file content into memory. As at any one time the users only work with a part of data, this is kept temporarily in the memory. We designed the FDC to be able to store the event data and additional information that are used to create plots and provide all the methods needed to manipulate these data (for example: add elements to the FDC, remove elements from FDC etc.). The following information is stored by the FDC:

1. Tree: It stores the values of the selected leaves and their corresponding paths which identify them. For example: If we select the /Event/EW/Phys/WMuSingleTrackNoBias/Particles/momentum/x (figure 6.3), it stores the name and value of /Event/EW/Phys/WMuSingleTrackNoBias/Particles/momentum/x.

2. FileName: the content of the memory can be saved to a file.

3. PlotTypes: Various plots can be made using the data. The name of the plot types is stored in the PlotType data structure.

4. Options: Plotting options such as logarithmic scale (*logX, logY*) are used

5. Ranges: It stores the *xmin,xmax,ymin,ymax* values used for axis scaling and appearance.

6. Conditions: It stores numerical expressions which are used to filter the data. For example: $x > 500$ (where x is /Event/EW/Phys/WMuSingleTrackNoBias/Particles/momentum/x). Only particles will be selected which fulfill this condition.

The FDC is directly used by the Feicim Data Browser algorithms and indirectly by the users through the Explorer Viewer.

Feicim Plots Factory The Feicim Plots Factory is a module which is used to generate plots using the selected data which are stored in the FDC. Feicim consists of a layer which integrates qtROOT[245] which is implemented as a layer of the ROOT[242] framework that is used to make plots. Feicim Plots Factory module makes one or two dimensional plots. The **Template Method** design pattern describes the creation procedure of the plots while the **Factory Method** design pattern creates the plots. Figure 6.4 show the Factory Method and the Template Method design pattern UML object diagrams.

The *QMainwindow* is a Qt implementation of a window. The *PlotView* inherits from the *QMainwindow* and contains the ROOT widgets (for example canvas, buttons, etc.) which are needed to make and show a plot. The *Plot* defines the interface of objects. It has the following methods: Draw (build a plot), *getModel* (the records which have to be histogrammed), *getRootTreeName* (the name of the ROOT tree which can be saved to a ROOT file.), *getFileName* (The name of the ROOT file). Each class which inherits from the plots must implement the Draw method. The *ComplexPlot* and *SimplePlot* implement the *Plot* interface and in addition inherits from the *PlotView* class. The *PlotsFactory* declares and implements the *create_plot* factory method which is used to create *SimplePlot* or *ComplexPlot*.

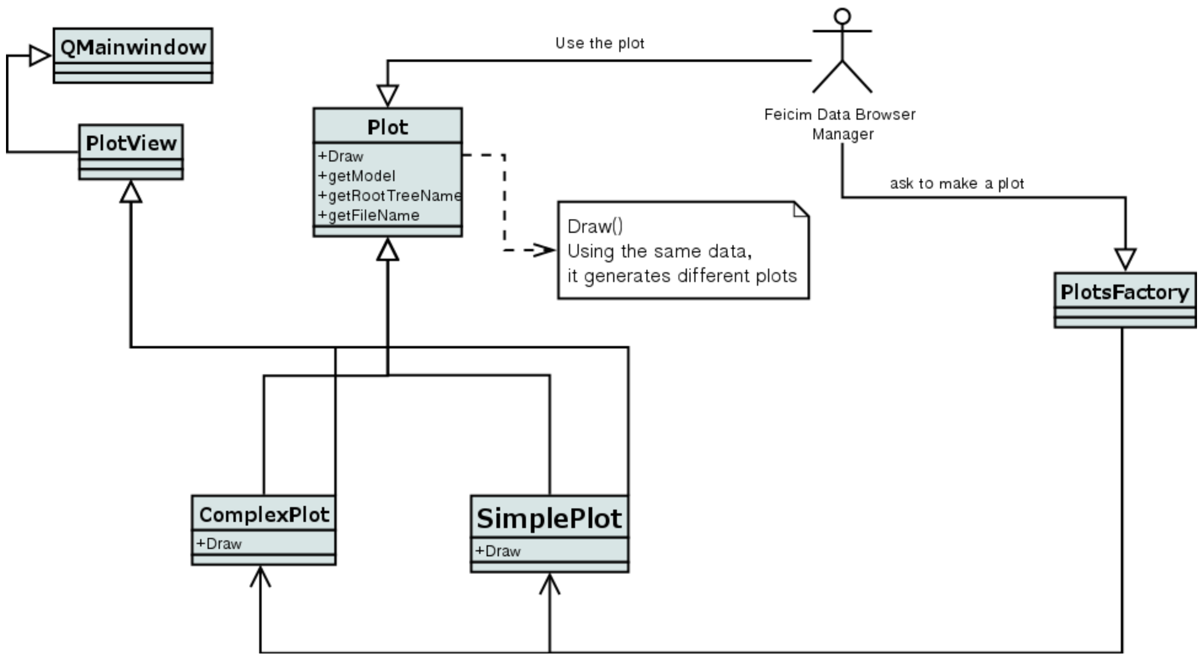


Figure 6.4.: UML class diagram - Plot Factory Method and Template Method design patterns

Feicim Object Manager The Feicim Object Manager is a module which initialises an Application Builder component and manages the Feicim Event Service. The Feicim Object Manager is used by the Feicim Data Browser Manager to create a Virtual File System by using the Feicim Data Browser algorithms.

Feicim Data Browser Manager The Feicim Data Browser Manager is based on the idea which was introduced in section 5.6. It is used to present the file content to the users as a Virtual File System.

To work with this Virtual File System the following operations are provided by the Feicim Data Browser Manager: *open, list, read, next* and *help*. The *open* command is used to open a file. The *read* command is used to read events from a data file. The *next* command is used to read the next event. To list data file content the *list* command can be used.

The Feicim Command Line Interface provides a User Interface to work with the Virtual File System. The Feicim CLI interacts with the Feicim Data Browser Manager. The users can browse through the Virtual File System using the Feicim Data Browser Manager operations.

Explorer Viewer

The Explorer Viewer is a component which consists of various GUI elements. Like other GUI components of Feicim it is based on the Composite Model View Controller. The Feicim

Data Browser GUI is a module which is implemented on top of the Feicim Command Line Interface. There are five widgets in the GUI and the hierarchy is shown in Figure 6.5.

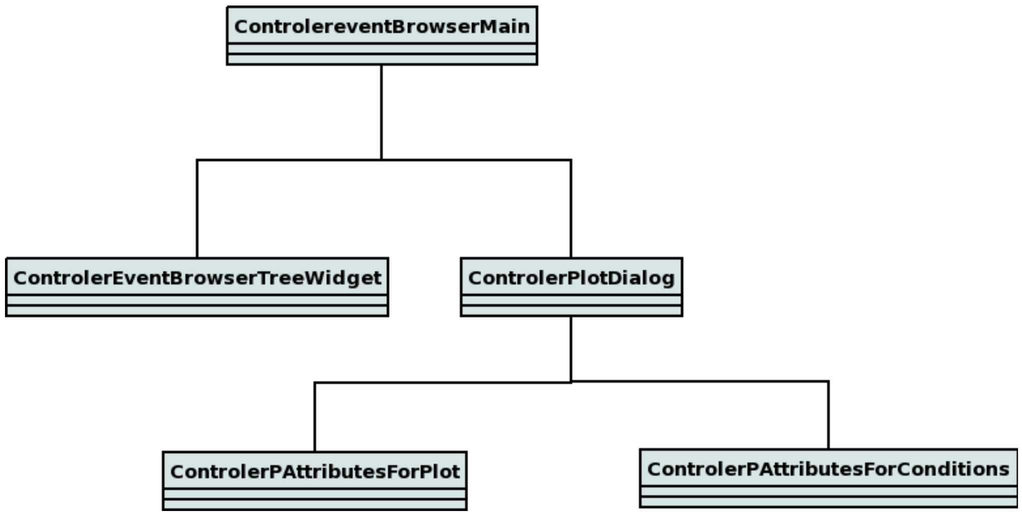

ControlereventBrowserMain
ControlerEventBrowserTreeWidget
ControlerPlotDialog
ControlerPAttributesForPlot
ControlerPAttributesForConditions

Figure 6.5.: Controller hierarchy of the Feicim Data Browser. -

The *ControllerEventBrowserMain* controller deals with the Feicim CLI, Feicim Analysis Model, Feicim Data Container and Jobs submission. It is used to manage the data sent by its child controllers and it is used to provide the models to the appropriate controller.

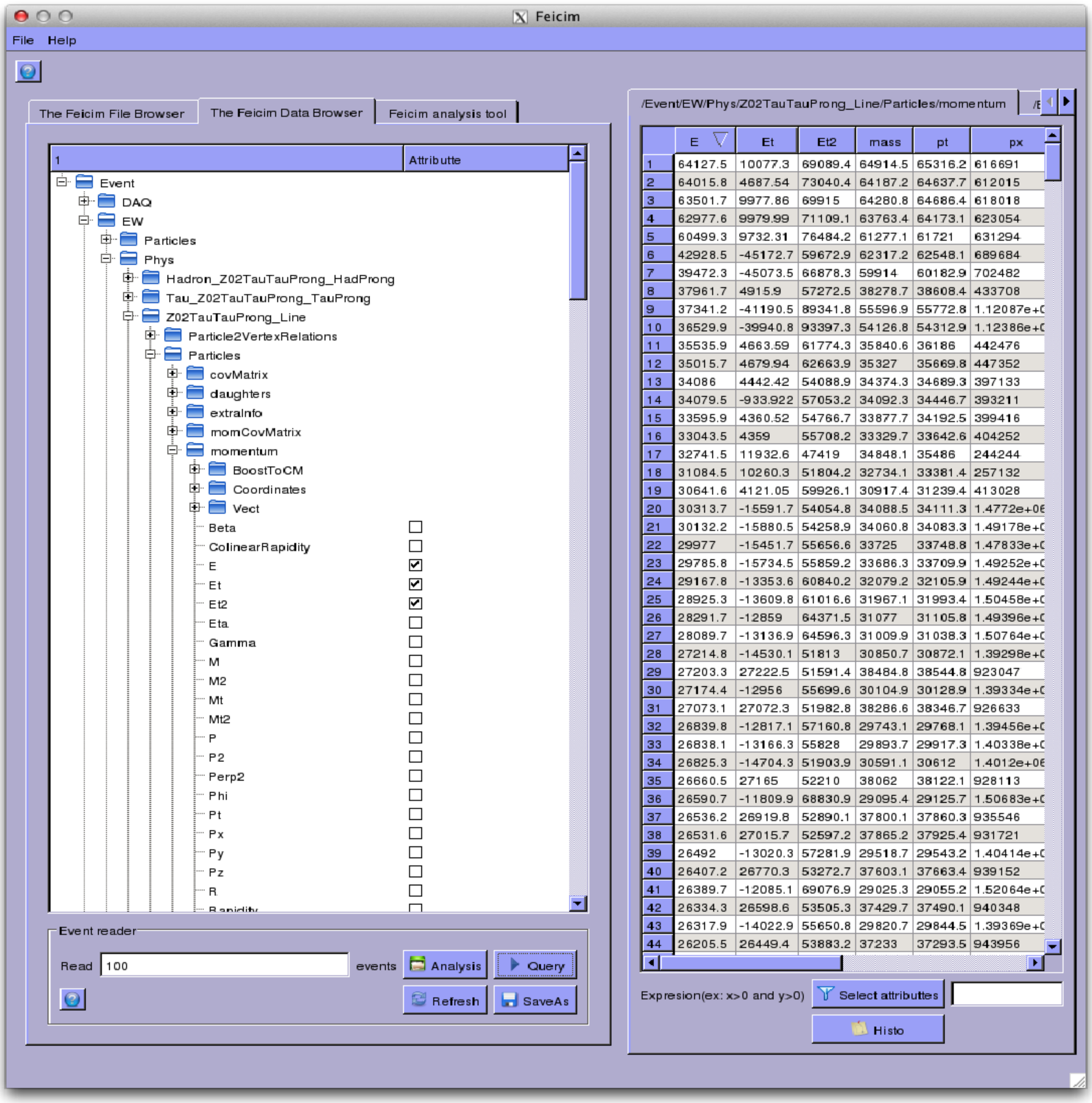
The main widget of the Feicim Data Browser is shown in Figure 6.6. The **FeicimEvent-BrowserTreeWidget** class implements this widget and has **ControlerEventBrowserTreeWid-get** controller.

The FeicimEventBrowserTreeWidget consists of two panels which are shown in Figure 6.6:

1. The Feicim Data Browser is the left panel in the picture. It consists of a tree which represents the data file content and the Event Reader widget which contains a text box and four buttons. The users can set the number of events to read from the file using this text box. The *Refresh* button is used to read only a single event. The *Query* button is used to read events while the *Analysis* button is used to run an analysis. Using the *Save* button the result can be saved to a ROOT file format.

2. The Right panel contains different tabs. Each tab contains a table with different numbers which are the result of the query that has been created in the left panel.

In order to make different plots using Feicim the users have to use the **PlotDialog** widget. Figure 6.7 shows the PlotDialog widget which allows the users to customize their plots and define filter conditions to the selected data. The **ControlerPlotDialog** controller provides all available information such as dimension of a plot, available plot type, the name of the attribute which is used to make a plots, etc. to its widget and forward the user selected information (only a single plot type, and the dimension of the plot, etc.) to its parent controller. When the users press the *Select* button then the **PAttributesDialog** widget appears which contains selectable attributes. Each button has an associated PAttributesDialog which is controlled by



1	Attributte
Event	
DAQ	
EW	
Particles	
Phys	
Hadron_Z02TauTauProng_HadProng	
Tau_Z02TauTauProng_TauProng	
Z02TauTauProng_Line	
Particle2VertexRelations	
Particles	
covMatrix	
daughters	
extraInfo	
momCovMatrix	
momentum	
BoostToCM	
Coordinates	
Vect	
Beta	☐
ColinearRapidity	☐
E	☑
Et	☑
Et2	☑
Eta	☐
Gamma	☐
M	☐
M2	☐
Mt	☐
Mt2	☐
P	☐
P2	☐
Perp2	☐
Phi	☐
Pt	☐
Px	☐
Py	☐
Pz	☐
R	☐
Rapidity	☐

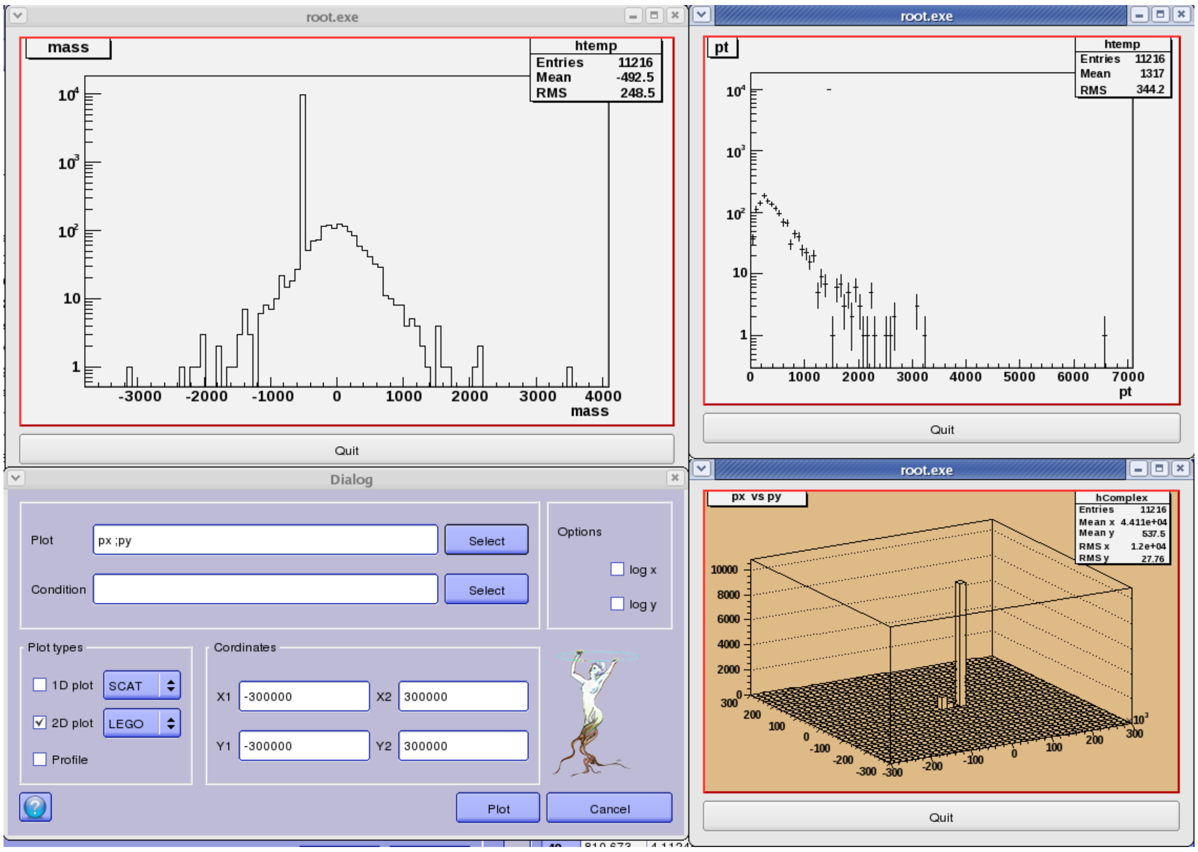
Read 100 events

/Event/EW/Phys/Z02TauTauProng_Line/Particles/momentum

	E	Et	Et2	mass	pt	px
1	64127.5	10077.3	69089.4	64914.5	65316.2	616691
2	64015.8	4687.54	73040.4	64187.2	64637.7	612015
3	63501.7	9977.86	69915	64280.8	64686.4	618018
4	62977.6	9979.99	71109.1	63763.4	64173.1	623054
5	60499.3	9732.31	76484.2	61277.1	61721	631294
6	42928.5	-45172.7	59672.9	62317.2	62548.1	689684
7	39472.3	-45073.5	66878.3	59914	60182.9	702482
8	37961.7	4915.9	57272.5	38278.7	38608.4	433708
9	37341.2	-41190.5	89341.8	55596.9	55772.8	1.12087e+0
10	36529.9	-39940.8	93397.3	54126.8	54312.9	1.12386e+0
11	35535.9	4663.59	61774.3	35840.6	36186	442476
12	35015.7	4679.94	62663.9	35327	35669.8	447352
13	34086	4442.42	54088.9	34374.3	34689.3	397133
14	34079.5	-933.922	57053.2	34092.3	34446.7	393211
15	33595.9	4360.52	54766.7	33877.7	34192.5	399416
16	33043.5	4359	55708.2	33329.7	33642.6	404252
17	32741.5	11932.6	47419	34848.1	35486	244244
18	31084.5	10260.3	51804.2	32734.1	33381.4	257132
19	30641.6	4121.05	59926.1	30917.4	31239.4	413028
20	30313.7	-15591.7	54054.8	34088.5	34111.3	1.4772e+06
21	30132.2	-15880.5	54258.9	34060.8	34083.3	1.49178e+0
22	29977	-15451.7	55656.6	33725	33748.8	1.47833e+0
23	29785.8	-15734.5	55859.2	33686.3	33709.9	1.49252e+0
24	29167.8	-13353.6	60840.2	32079.2	32105.9	1.49244e+0
25	28925.3	-13609.8	61016.6	31967.1	31993.4	1.50458e+0
26	28291.7	-12859	64371.5	31077	31105.8	1.49396e+0
27	28089.7	-13136.9	64596.3	31009.9	31038.3	1.50764e+0
28	27214.8	-14530.1	51813	30850.7	30872.1	1.39298e+0
29	27203.3	27222.5	51591.4	38484.8	38544.8	923047
30	27174.4	-12956	55699.6	30104.9	30128.9	1.39334e+0
31	27073.1	27072.3	51982.8	38286.6	38346.7	926633
32	26839.8	-12817.1	57160.8	29743.1	29768.1	1.39456e+0
33	26838.1	-13166.3	55828	29893.7	29917.3	1.40338e+0
34	26825.3	-14704.3	51903.9	30591.1	30612	1.4012e+06
35	26660.5	27165	52210	38062	38122.1	928113
36	26590.7	-11809.9	68830.9	29095.4	29125.7	1.50683e+0
37	26536.2	26919.8	52890.1	37800.1	37860.3	935546
38	26531.6	27015.7	52597.2	37865.2	37925.4	931721
39	26492	-13020.3	57281.9	29518.7	29543.2	1.40414e+0
40	26407.2	26770.3	53272.7	37603.1	37663.4	939152
41	26389.7	-12085.1	69076.9	29025.3	29055.2	1.52064e+0
42	26334.3	26598.6	53505.3	37429.7	37490.1	940348
43	26317.9	-14022.9	55650.8	29820.7	29844.5	1.39369e+0
44	26205.5	26449.4	53883.2	37233	37293.5	943956

Expresion(ex: x>0 and y>0)

Figure 6.6.: Feicim data browser GUI.



Dialog

Plot: px ;py — Select

Condition: — Select

Options: ☐ log x ☐ log y

Plot types: ☐ 1D plot SCAT; ☑ 2D plot LEGO; ☐ Profile

Cordinates: X1 -300000 X2 300000 Y1 -300000 Y2 300000

Plot Cancel

Figure 6.7.: Feicim plot dialog and Feicim plot attribute dialog. -

the **ControlerPAttributeForPlot** or **ControlerPAttributeForConditions** controllers shown in Figure 6.5.

6.4. Feicim Data Analysis

The users can define and execute very simple Gaudi algorithms using the Feicim Data Analysis component, without writing program by manipulating graphical elements. It can be executed in the local machine or can be submitted to the Grid. These simple algorithms run on huge amounts of distributed data among various Grid sites and select particles from a given container (for example: /Event/EW/Phys/WMuSingleTrackNoBias/Particles in figure 6.3) using different filter conditions. Feicim uses DIRAC/Ganga back-end for job submission. The DIRAC monitoring page can be used to monitor the jobs and different DIRAC Data Management tools can be used to retrieve the job outputs.

6.4.1. Feicim Data Analysis architecture

The Feicim Data Analysis architecture consists of different components which can be used as standalone applications. These loosely coupled components deal with the generation of different scripts and jobs submission. Figure 6.8 shows an overview of the Feicim Data Analysis components.

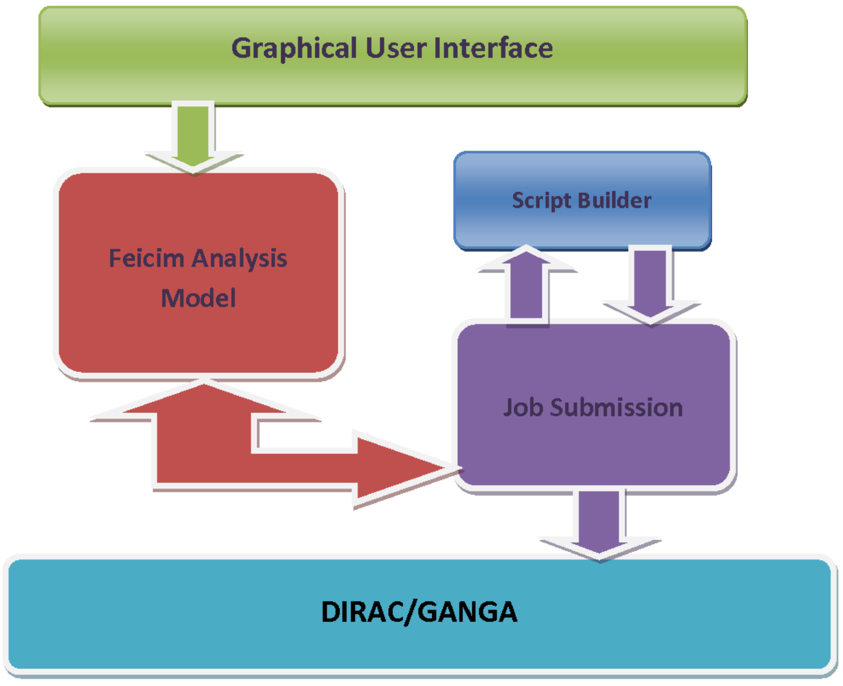


Figure 6.8.: Feicim Data Analysis architecture. -

Feicim Analysis Model

The Feicim Analysis Model stores the information which is needed to run an analysis job in the Grid or Local machine. The following information is required to run a job: algorithm, job requirements.

In order to generate an algorithm some information is required such as Program Name and Program Version, input data (files), a path(s) such as /Event/EW/Phys/WMuSingleTrackNoBias/Particles/momentum/x shown in figure 6.3 which is used to select events which correspond to that path(s) of an event tree (an example is the event tree shown in figure 6.3) and different filters which will operate on the selected events.

To submit jobs to the Grid it is required to know information which is related to the job running conditions such as the job CPU requirement, platform, site where the job will be executed, job name and option files which contain the algorithm requirements.

The users interact with the Feicim Analysis Model through the Graphical User Interface by manipulating graphical elements. The source of the requirements used to run analysis is the Feicim Data Browser component. This component contains the selected path(s) from the event tree (an example of the event tree is shown in figure 6.3) the input data file names, and the output file name produced by the job.

Script Builder

The Script Builder generates the scripts (or option files) using the Feicim Analysis Model used to run analysis in the local machine or in the Grid. It generates two types of script:

- a simple Gaudi script which contains a Gaudi algorithm used to select particles for a given path of the event tree (an example of the event tree is shown in figure 6.3). The users using the Feicim Data Browser select the path interactively.
- a simple Dirac script which prepares a job and submits it to the Grid or local machine.

Figure 6.9 shows the UML diagram of the Script Builder components which generates these two scripts. The Script Builder implements the Builder design pattern which was introduced in section 5.4.1.

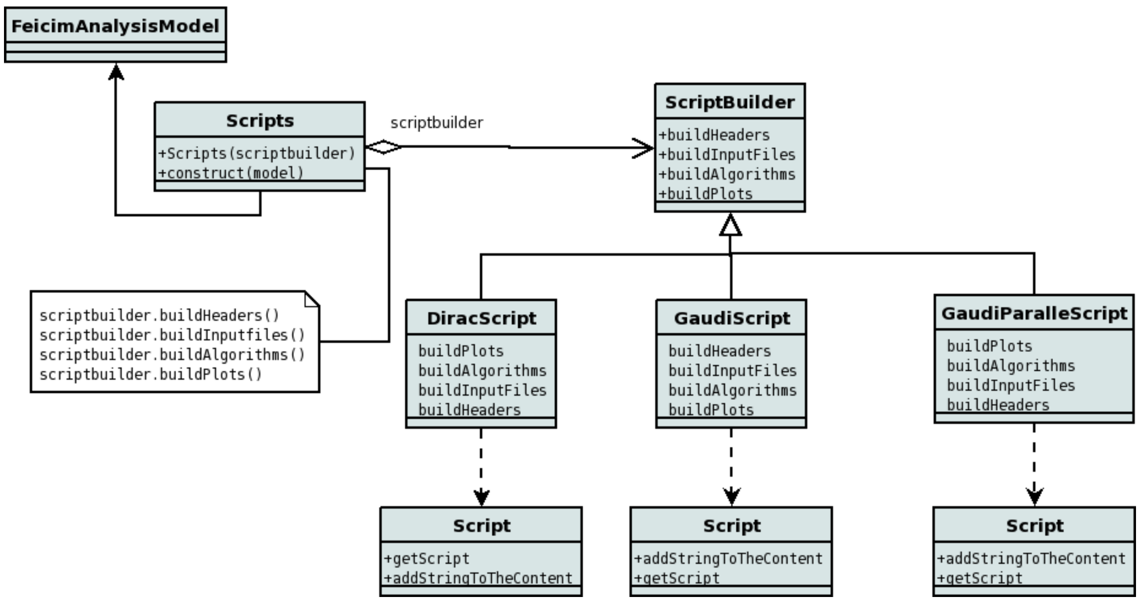


Figure 6.9.: The concrete implementation of the Builder design pattern. -

The **Script** is a class that stores an executable script in a text format which can have different types: Dirac scripts, Gaudi scripts which were introduced above or Ganga scripts.

The **FeicimAnalysisModel** introduced in section 6.4.1 provides all the information required to build a Script.

The **ScriptBuilder** is an abstract interface which declares all the abstract methods needed to generate different Scripts. The **DiracScript** implements the ScriptBuilder abstract methods to generate DIRAC script which will be used to submit jobs to the Grid or local machine. The **GaudiScript** and **GaudiParallelScript** generate the Gaudi algorithm which will be executed in a single core or multi core. It inherits from the ScriptBuilder and generates a python script which can be executed using the Gaudi framework.

The script which submits jobs to the Grid through Ganga is currently not implemented, but it can be implemented as any other script. In addition it is possible to implement classes which can generate other scripts that can be used by other LHC experiments or non LHC communities.

In the future the Gaudi script could be changed in order to combine different complex Gaudi algorithms which are part of a work flow.

Job submission

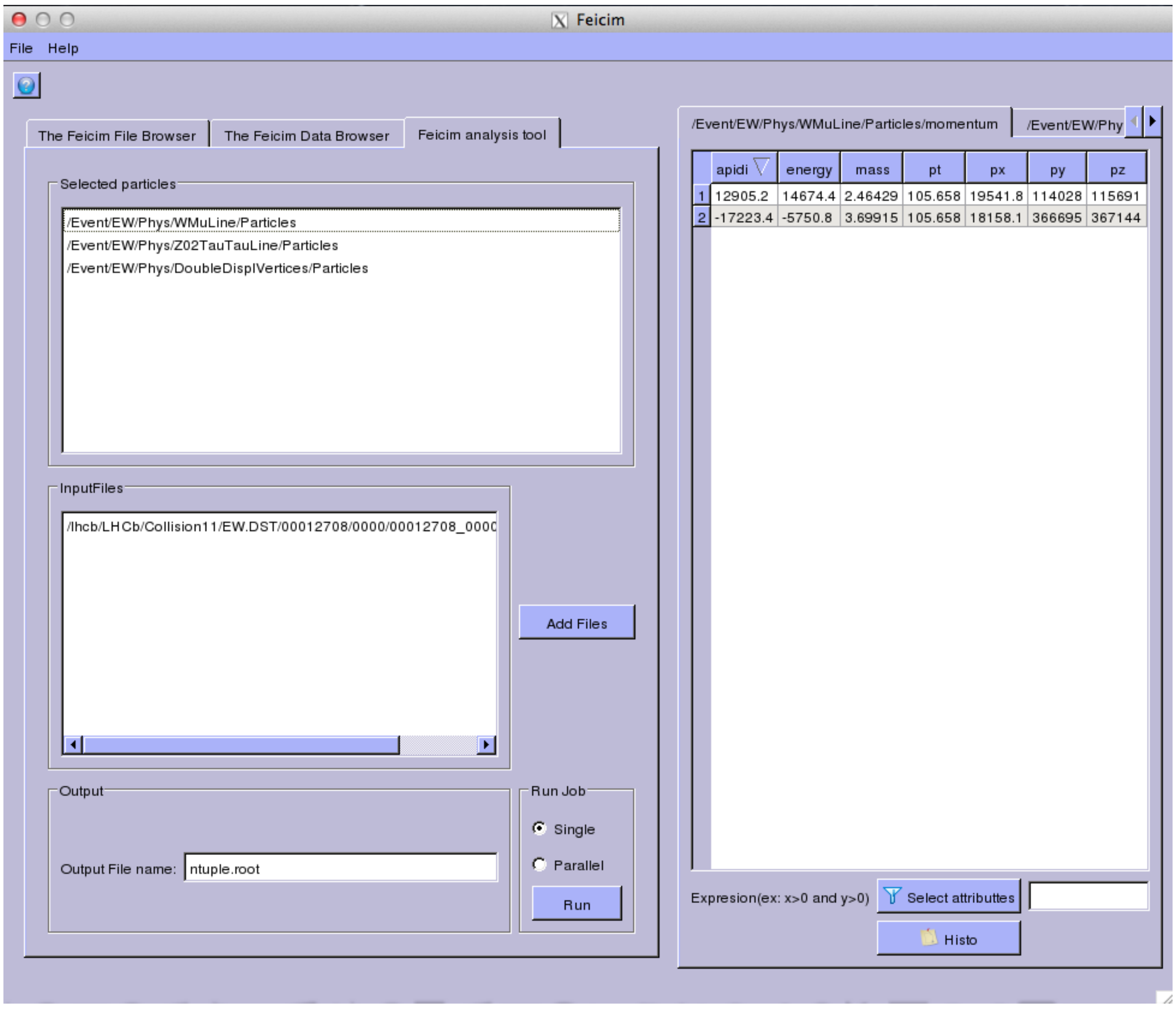
The Job submission component can be use as a standalone application to submit jobs to the Grid or local machine. The Job submission component coordinates the creation of the scripts and interacts with the underlying DIRAC/Ganga frameworks in order to submit the job. It hides the complexity of the job submission and the Gaudi algorithms.

Graphical User Interface

Like other Feicim Graphical User Interfaces, it is based on the Composite Model View Controller design pattern. Figure 6.10 show the Feicim Analysis tool which consists of three GUI element:

- **Selected particles** widget show the selected 'paths' of an event tree (an example of the event tree shown in figure 6.3).
- **Input files** widget contains a list of files which will be used by the algorithms. It allows to add/remove input files from the list.
- **Output** widget contains the output file name. The result of the Gaudi algorithms will be written to this file.
- **Run Job** widget that contains a **Run** button used to run parallel or non-parallel job submissions.

As it is the simplest component of Feicim, it contains only one widget element and only one controller. However, it is very important for performing data analysis.



Selected particles:
/Event/EW/Phys/WMuLine/Particles
/Event/EW/Phys/Z02TauTauLine/Particles
/Event/EW/Phys/DoubleDisplVertices/Particles

InputFiles:
/lhcb/LHCb/Collision11/EW.DST/00012708/0000/00012708_0000

Output File name: ntuple.root

Run Job: [✓ Single] Parallel

	apidi	energy	mass	pt	px	py	pz
1	12905.2	14674.4	2.46429	105.658	19541.8	114028	115691
2	-17223.4	-5750.8	3.69915	105.658	18158.1	366695	367144

Figure 6.10.: Feicim Analysis tool GUI -

6.5. Summary

The events events generated by Monte Carlo productions or taken by the LHCb detector are stored in data files. The data files contain data objects which store the events. The data objects are organised to different hierarchical structure. We implemented a new algorithm called Feicim Tree Traversal algorithm to discover the data file content. This algorithm is different than the existing tree traversal algorithm, because it visits the tree nodes in a given order (or in another word path).

An user-friendly tool is essential which can be used to analyse the data. Feicim Data Browser allows to extract information from a data file using the Feicim Tree Traversal algorithm. It allows to define different filters on the data. In addition, different plots can be made using the data which read by this algorithm.

Feicim Data Analysis creates a simple Gaudi algorithm using the paths extracted from the data file by the Feicim Tree Traversal algorithm. It allows to specify different input whereat the Gaudi algorithm will perform. In addition it allows to create a job using this Gaudi algorithm and submit it to the Grid.

7. The LHCb Bookkeeping System

In the previous chapters we discussed DIRAC and its components which compose the Workload, Production and Data Management Systems as well as the Feicim architecture and design principles. In this Chapter we describe the LHCb Bookkeeping System, which is actually one of the Feicim components. At the beginning of this chapter we present a brief history of the LHCb Bookkeeping system. As the LHCb Bookkeeping System is designed to store metadata information of the datasets, the Data Model which provides a definition and format of these data is crucial. We will point to the Relational Data Model and the Hierarchical Data Model which are used by the LHCb Bookkeeping System in section 7.2. Section 7.2.1 provides a more detailed overview of the Relational Data Model. In order to define operations of the datasets and understand the logic of relations, the use of Relational Algebra is inevitable. In section 7.2.2 we highlight the relation between the relational algebra and the query languages. The architecture of the system is very important to achieve a robust and scalable application which can handle thousands of user requests and in section 7.3 a detailed description is provided. In section 7.4 we present the dataset which will be stored in the Relational Data Model, the Relational Database Schema, and the access protocol to the Bookkeeping Metadata catalogue. To execute user tasks on the database the usage of different relational operation is essential. These relations are dynamically generated based on the users requirements. The mechanism of the generalization of the relational operations will be presented in section 7.5. In sections 7.6, 7.7 the Bookkeeping Service and the Bookkeeping Client, which provide access to the Bookkeeping Metadata catalogue, are introduced. In section 7.9 various User Interfaces are presented which are widely used by the LHCb community. We attempt to design a system which is capable of serving a huge number of user requests. Different scalability tests were carried out to understand the performance of the system. The results of the test are described in section 7.10.

7.1. Evolution of the LHCb Bookkeeping System

The Bookkeeping Working group was formed at the beginning of 1999 to create a bookkeeping system for the data produced by LHCb. They developed a prototype based on the Oracle database[223].

During 2000 and 2001 the LHCb Bookkeeping System was widely used by various users but in parallel many developments took place. This system has well defined functionalities:

- it stores the datasets and their metadata information and provides access to them using a well defined API and a Web User Interface.
- the information about data location and some statistics about the datasets such as number of jobs and their generated files can be retrieved.

In 2002 a new system was introduced because it was hard to manage the previous system. The new system kept the old user interface but in addition provided more functionality to the users:

- replica and Grid management: all the replicas of a file are kept in the system;
- python programming interface;
- failure recovery: in case of error later on the information can be entered to the bookkeeping system; and
- easy extensions: it easily allows to add all sorts of metadata.

The database remained as the old Oracle database; only the database content changed. This system was used during 2002 and 2003 and some improvements took place (query optimization, table partitioning) in order to be ready for the Data Challenge 2004 (DC04).

A new development started in 2005 that adopted the ARDA[169] implementation of the gLite metadata interface called AMGA.

In 2006, several changes took place which affected the user interface. At the end of 2006 the AMGA based LHCb Bookkeeping System was ready and available for the users.

At the beginning of 2007 the Bookkeeping Working Group concentrated on defining the bookkeeping requirements which are needed for data coming for physics collisions (sometimes referred to as 'real' data). Originally the LHCb Bookkeeping System was designed to store MC data, so the new requirements had to be presented to the physics communities and accepted by them. In May 2007 it was decided to redesign the existing system taking into account the Feicim tree-like browser prototype which was developed by the Dublin group and this system and its subsequent development is described in this chapter.

In 2008 this development was started in order to be ready by the end of the year. This new development takes into account all the experiences learned from the existing system. The database schema changed as well as the implementation of the LHCb Bookkeeping System and the data search functionalities. The new system was based on AMGA, but in parallel an Oracle schema was designed. In the middle of 2008 the data was migrated to the AMGA but in parallel the data was copied to the Oracle database. In addition the DIRAC based services were ready and being tested. The first proposal of the User Interface was presented and the implementation of the GUI started. During August a decision was made not to use AMGA any more, because AMGA does not provide all the functionalities like a RDBMS and required extra work in the database level in order to optimize the queries, since this time a pure Oracle database has been used to store the data.

In September 2008 the first Command Line Interface and GUI was ready and presented to the LHCb community. In December 2008, the new standalone GUI and Web interface were available to the users. All the existing functionality was provided by the new system and in addition new functionalities were implemented in order to meet the requirements introduced by real data. The existing system was decommissioned at the beginning of 2009 and since then users have made use of the Feicim scheme.

In 2009 the LHCb Bookkeeping System was extended with new functionalities which are required by the Production Management System in order to allow the users to define different types of requests which describe the conditions and data processing phases used during the data processing. In 2010, 2011 many changes occurred on the database schema and new functionalities have been implemented in order to have a more flexible and robust system.

Currently the LHCb Bookkeeping System is widely used by approximately 500 users and different DIRAC systems such as the Production Management System, Data Management System, Transformation System and Workload Management System.

Metadata

In the LHCb experiment Metadata is used to describe an LHCb dataset. Different kinds of Metadata have been identified such as Job provenance, File and Job Metadata and Bookkeeping information which will be presented in more details in section 7.4.1.

The Job provenance contains information about the history of the data processing such as processing phases, applications etc. The File and Job Metadata consists of information such as execution time, start and end date, location, memory etc while the file Metadata consists of information related to a file such as size, creation date, GUID, etc. The Bookkeeping information is a selection of a dataset on which physics analysis can be performed. The Metadata can be used to identify a dataset or it can be used to make measurements. For example: how many job executed during a period at a specific Grid site? etc.

What is the LHCb Bookkeeping System?

The LHCb Bookkeeping system is a metadata management system which stores the conditions relative to jobs, files and their metadata, as well as their provenance information in an organized way. It provides tools to the users which allows them to get the collections of dataset for analysis making queries on the basis of the metadata.

The LHCb Bookkeeping System contains three key components: the lowest component is the **Bookkeeping Metadata catalogue** which consists of an Oracle database and a module that allows access to the Oracle database. On top of the Bookkeeping Metadata catalogue we have implemented the **Bookkeeping Service** which has been developed inside the DIRAC framework. It integrates the Bookkeeping Metadata catalogue and provides the important data functionality for selecting and inserting data to its clients. The highest component is the **User Interface**.

7.2. A hybrid data model to describe the LHCb metadata structure

The system has to store huge amounts of data and this data has to be accessed in an efficient way. We designed a **Data Model** that provides the definition and format of the data. The Data Model describes the structure of the database. The data that has to be stored are organized as structured data. Various data models are available and can be used to describe the data structure and specify how the data can be stored and accessed. The following data models are available:

- Hierarchical data model is used to store data which is organized in a tree-like structure [224].
- Network data model organizes the data using sets and records. Records contain field and sets which define one to many relationships between records. It allows each record to have multiple parent and child records by forming a generalized graph structures[224].
- Relational data model is based on the concept of relations which will be introduced in the next sections.
- Object-oriented data model combines object oriented programming with database technology. It means the objects, classes and inheritance are directly supported in database schemas and in the query language[225].

A part of metadata information is organised in hierarchical structures. Initially we only used the Relational Data Model to specify data and queries. Because it was not easy to manage (it was very hard to design efficient complex queries), we decided to study the Hierarchical Data Model. As a result of our study we use a mixture of the Hierarchical and Relational Data Models. Using this Hybrid Data Model we can manage the metadata in an efficient way (the queries can be performed very quickly on the data as presented in section 7.10).

7.2.1. Relational Data Model

The Relational Model provides a very simple way of representing data. The idea of the relational model is to organize the data into collections of two-dimensional relations. An (ordered) **n-tuple** is a sequence (or ordered list) of n elements, where n is a positive integer. For example: (8,9,1,3) denotes a 4-tuple. A **domain** D in the relational data model is a set of values. The types of the value (like integer, character, boolean, etc.) in a domain must be identical and they are called data type of the domain D. The name of the domain D helps to interpret its values. For example:

- *EmployeeName* = {Zoltan, Gabor, Gyuszi}; *EmployeeName* is a set of character string that represent the names of people;
- *PhoneNumber* = {(0041)7030-12345, (0041)7200-10005,(0041)7230-10045}; *PhoneNumber* is a set of thirteen digit phone numbers;

In this example the domain names are *EmployeeName, PhoneNumber*. The data type of the *EmployeeName* is character string. The data type of the *PhoneNumber* is a character string which has the following format: (xxxx)xxxx-xxxx, where x is a numeric digit like (0041)7030-12345.

The name of the domains are called **attributes**; we denote an attribute **A**.

In the book of Jeffry D. Ullman the **relation** is defined as a subset of the Cartesian product of domains [226]. Given a sequence of sets of domains $D_1, D_2, D_3, ...D_n$ the Cartesian product, written $D_1 \times D_2 \times ... \times D_n$, is the set of all ordered n-tuples $(d_1, d_2, ...d_n)$ where $d_i \in D_i, i = 1,2,3,...n$. The n is the degree of the relation; another term for degree is 'arity'. For example, consider n = 2; $D_1 = \{a,b\}$ and $D_2 = \{1,2,3\}$; The Cartesian product ($D_1 \times D_2$) is $\{(a,1),(a,2),(a,3),(b,1),(b,2),(b,3)\}$; The set $\{(a,1),(a,2),(a,3),(b,1)\}$ is a relation, a subset of the product $D_1 \times D_2$

A set of attribute names for a relation is called a **Relation schema**. A Relation schema R, is denoted by $R(A_1,A_2,...,A_n)$, where R is the relation name and $A_1,A_2,...,A_n$ are the list of attributes. We denote one attribute from the relation schema R: A_1 or $R.A_1$.

An example of a relation schema for a relation which describes employees, is the following: EMPLOYEES(EmployeeId, Name, BirthDate, PhoneNumber, Salary, Tax). We display the relation as a table, where each tuple is shown as a row and each attribute corresponds to a column name. Each column has a domain which is the set of possible values that can appear in that column. For example: Table 7.1 shows the EMPLOYEE relation.

EmployeeId	Name	BirthDate	PhoneNumber	Salary	Tax
24	Zoltan Mathe	18.09.1983	004278368749	15000	3000
34	John Smith	18.07.1967	0033287568809	13000	2600
50	Johnny Cash	18.08.1900	0041569800101	10000	2000

Table 7.1.: EMPLOYEE relation; The attributes of this relation: EmployeeId, Name, BirthDate, PhoneNumber, Salary, Tax; The tuples of the relation: (24, Zoltan Mathe, 18.09.1983, 004278368749, 15000,3000), etc.

A relation which describes a database table has to observe the following states:

- The tuples are different. That means in a table we cannot have identical rows. At least one attribute value has to be different.

- The order of the rows or columns is irrelevant, because each column and row is identified by its content.
- The attribute of the columns have to be different (Each column has a unique type).
- All entries in a column have same types.

Key of relations

Key is an attribute or a set of attributes which is used to identify a tuple in a relation. The relations must have one or more attributes that serve as a key of the relation. Three types of key are available: Super key, Candidate key and Primary key.

The **relation state** r refers to the present set of tuples in a relation R. We denote the relation state *r(R)*. For example, consider the EMPLOYEE(EmployeeId, EmployeeName, EmployeeAdress) relation. The relation state may include 5 EMPLOYEEs; another includes 100 EMPLOYEEs. An n-tuple t in a relation $r(R)$ is denoted by $t = (v_1, v_2, ..., v_n)$, where v_i is the value which corresponds to attribute A_i, $i = 1,n$[224]. The following notations refer to the values of the tuples:

- $t[A_i]$ (sometimes t[i]) refer to the value v_i in t for attribute A_i
- $t[A_u, A_w, ...A_z]$ where $A_u, A_w, ...A_z$ is a list of attributes from R where $u, w, z \leq n$, refer to the subtuple of values $(v_u, v_w, ..., v_z)$ from t which corresponds to the attributes specified in the list.

For example, consider tuple t =('Zoltan Mathe', '18.09.1983','004278368749',15000,3000) from the Employee relation in Table 7.1. The t[Salary]=('15000'), and t[Name,BirthDate,Tax]=('Zoltan Mathe', '18.09.1983',3000).

The **Super key** of a relation schema R is a set of attributes SK of R with the following condition: for any distinct tuples t_1 and t_2 in any *r(R)*, $t_1[SK] \neq t_2[SK]$[224]

A key K of a relation schema R is a **super key** of R with the additional property that removing any attribute A from K leaves a set of attributes *K'* that is not a super key of R any more[224].

The **Minimal super key** is super key from which we can not remove any attributes and still do not have two distinct tuples in any state r of R.

A key satisfy two constraints:

- Two distinct tuples in any state of the relation cannot have identical values for all the attributes in the key.
- It is a minimal super key.

For example, consider the EMPLOYEE relation of Table 7.1. The key of the relation is EmployeId because no two-employee tuples can have the same value for EmployeeId (Each employee in a company has an unique identifier). Any set of attributes that includes EmployeeId is a super key. For example, consider the EMPLOYEE relation of Table 7.1. The super key of the relation is any set of attributes {EmployeeId, Name, Birthday}, but it is not a key of the EMPLOYEE relation because removing Name or Birthday or both from the set, it still have a super key.

A relation schema *R* may have more than one key. Each of the keys is called a **candidate key**. For example, consider the CUSTOMERS(CustomerName, CustomerAddress, Balance) relation. The candidate key of this relation is CustomerName, CustomerAddress. The **primary key** is a designated candidate key. For example, consider the previous CUSTOMERS relation. The primary key of this relation is the CustomerName, CustomerAddreess.

eId	eName	eBirthDate	eAddress
11	Zoltan Mathe	18.09.1983	1. St. Genis, Fr
12	John Smith	18.07.1967	2. St. Genis Poully, Fr
14	Johnny Cash	18.08.1900	4. Carouge, CH

Table 7.2.: Employee relation

eId	Salary
11	13000
12	5000
14 Cash	10000

Table 7.3.: Salary relation

eId	eName	eBirthDate	eAddress	Salary
11	Zoltan Mathe	18.09.1983	1. St. Genis, Fr	13000
12	John Smith	18.07.1967	2. St. Genis Poully, Fr	5000
14	Jon Cash	18.08.1900	4. Carouge, CH	10000

Table 7.4.: New Employee relation

It is not acceptable to have two relations which contains one or more common attributes, because the cost of duplication influence the relation operations. In order to avoid this, we can combine the attributes of two relations and we can replace the two relations by one relation. This new relation is the union of two sets of attributes. For example: consider the relations Employee and Salary from Tables 7.2 and 7.3 that have the same eName common attribute. These two relations can be replaced by only one relation (see Table 7.4) which contains once the *eName* attribute.

In the next section we will use the following relational algebra operations: Select, Project, Union, Cartesian product and Join. A detailed description of them can be found in Appendix A.1.

7.2.2. Query languages and Relational algebra

This subsection presents the relation between the relational algebra and the query languages. The **query language** describes the queries of a relational database. Various query languages exists. One of the most widely used query language is the Structured Query Languages (SQL) which is originally based on relational algebra and tuple relation calculus[229]; it manages data in a Relational Database Management System (RDBMS). We can use projections and selections within different relations in order to know natural questions about a single relation. For example: Consider the *Employee* relation shown in Table 7.4. If we want to know which employee earns more than 5000, we can write the following query:

```
SELECT e.eName
FROM Employee e
WHERE e.Salary > 5000
```

This query can be translated into the following relational algebra expression:
$\prod_{eName}(\sigma_{salary>5000}(Employee))$

It can be presented also as a query tree shown in Figure 7.1.

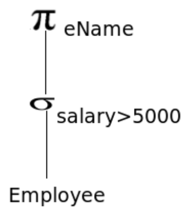


Figure 7.1.: Query tree - representation of a relational algebra expression

7.3. Design and architecture of the LHCb Bookkeeping System

The system design is based on loosely coupled components. We apply the different design principles of software architectures which were introduced in section 5.4. In addition we distinguish between different layers which compose the architecture of the system: Data Layer, Query Generation Layer, Service Layer, Client Layer, Manager Layer and Presentation Layer. Each layer has dedicated scope and is designed to solve specific problem. Figure 7.2 shows the layered architecture. In the next sections we present an overview of the responsibilities of each layer and the components that compose each layer.

7.4. Data Layer

The **Data Layer** is the lowest layer and consists of database specific subroutines which are required by the applications in order to access the database resources. These subroutines are

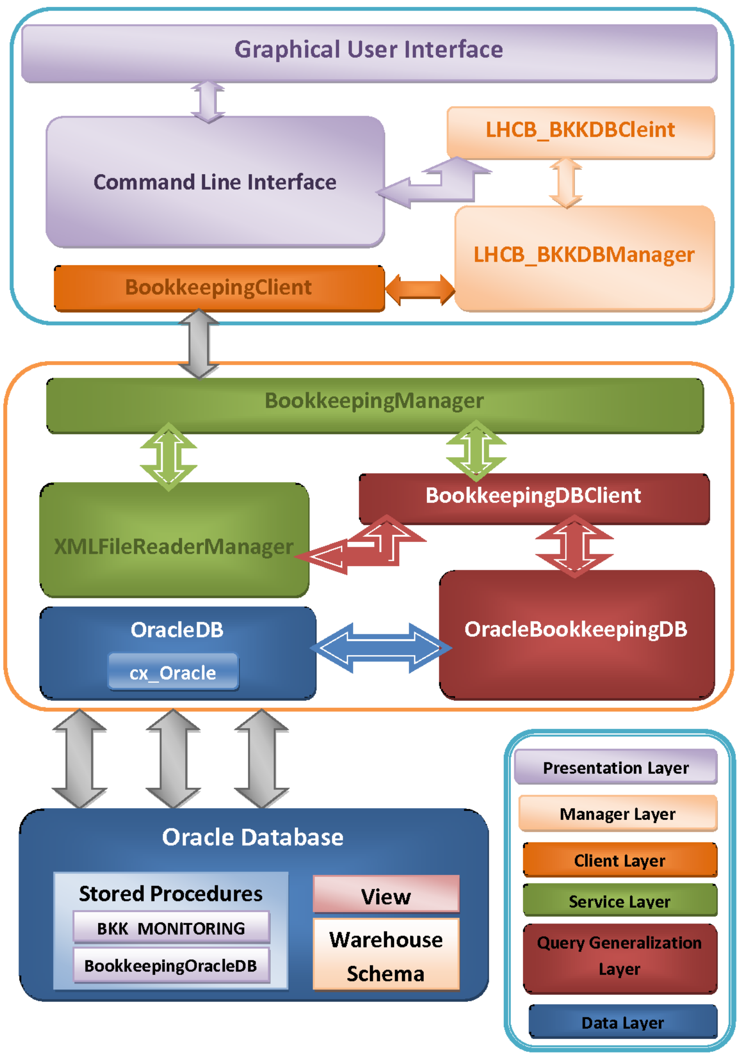


Figure 7.2.: he box on the lower right gives an overview of the different layers. The rest of the figure shows the interrelationships of the components and is colour coded to indicate which layer it belongs to.

used to centralize the logic of the system that is required to execute several SQL statements. Everything related to the database belongs to this layer such as Oracle specific subroutines: *Stored Procedures* or *User Defined Functions*, *Triggers* etc.

7.4.1. The LHCb dataset

According to the origin of the data, which can be real or simulated, it is necessary to store additional metadata information such as data taking conditions or simulation conditions, different application(s) etc. Taking into account the metadata information in the database, the following relations have to be stored in a database table: Configuration name and version, Simulation or data taking conditions, Steps, Processing pass, Event type or stream, Production or run number, File types, Data quality, Jobs and Files. These relations will be presented in the next sections.

Configuration name and Configuration version

The **Configuration name and version** describes the data type which are produced by different activities. They are used to distinguish the real and simulated data.

The **Configuration Name** of the simulated data is *MC*. As the name suggests all the simulated data is grouped under *MC*. The **Configuration Version** is the data simulation year.

Data taking or Simulation condition

The **Data taking conditions** is used to determine the conditions that describe the detector status when the real data was taken. It defines a set of runs which have been taken with the same conditions.

The **Simulation conditions** describe the simulated (Monte Carlo) data. The simulation conditions are equivalent to the data taking conditions for simulated data.

Event type or Stream

The **event type** describes the way the data samples were produced and classify the decay channels[227].

The **stream** describes the way the real data were produced. It contains the events which are classified according to the physics triggers which fired.

Steps

Steps is an abstraction of an application and its configurations which can be executed during data processing. A step has a **Step Name** which describes the step, and it has an associated application together with various option files which were introduced in Section 4.2. The **Application Name** and **Application Version** contain the name and version of the applications which are used for the data processing phases. In addition each application has **Option Files** which are used to describe the Event type, Simulation Condition etc. The step also contains information about tags which inform the user about the detector description: **DDDb**, **CondDb**. The steps can use additional packages which are called **Extra Packages**.

Processing pass

The **Processing pass** describes the data processing phases which were introduced in section 4.3 for real and simulated data. The Processing Pass is a group of different Steps which are compatible. The Steps of a Processing pass form a hierarchical structure. For example the /Real Data/Reco10, /Real Data/Reco10/Stripping11, /Real Data/Reco10/Stripping12, /Sim01/Trigger0xffff/Reco10.

Production or run number

The **production number** is used to identify a dataset which was produced by the same processing phases (processing pass). It groups all the jobs which have the same simulation conditions. The **run number** is used to identify a run. It groups the data which have been taken with the same data taking conditions, during a short data-taking period (typically an hour).

Jobs

In order to analyse data in the Grid using Ganga/Dirac, we have to define a task, or in other words, a job. From the LHCb Bookkeeping System point of view the job can have two origins depending on whether it is real or simulated data. We decided to use only one classification of a job whether the origin is the LHCb detector or the Grid.

Both of them have associated parameters such as Job Start, Job End, CPU time, Memory. If the job origin is the LHCb detector, then in addition, it contains parameters which are needed for real data such as run number, Luminosity, etc.

Filetypes

The jobs have output files which have various file types. The **filetype** describes the format and type of the files.

Files

Each job produces **Files** which have associated parameters such as File Name, Size, EventType, Luminosity, etc.

7.4.2. Bookkeeping Metadata catalogue

The Bookkeeping Metadata catalogue is an abstraction layer on top of the Oracle database. It consists of a module which allows access to the Oracle database and provides the functionalities which is required to manipulate the data.

The database schema

In order improve performance we decided to split the database schema into two parts: **Warehouse schema** and **Views**. Figure 7.3 shows both schemas which are used to store the LHCb datasets.

The Warehouse schema consists of various tables which are up to date, which means if an operation is performed on the Bookkeeping Metadata catalogue, the result is immediately available in these tables. The tables are *partitioned* and each table contains indexes, which are *partitioned* and *compressed*.

In order to minimize the load of the database and maximise the response time we use *Views* which is a representation of the SQL statement. They are part of the database kernel and are stored in memory.

Oracle provides different types of Views. The Views are useful, because the queries are stored in the memory and it hides the complexity of the query. However, if we have a very complex SQL statement, they may have a bad performance, because more than two big tables can be joined and joining big tables may take time. The *Materialized View* is used to improve performance of the standard views. The difference between the standard and the Materialized View is the Materialized View stores the SQL statement result on disk and, as soon as it is created, all the tables of a complex query will be joined. Since all of the table joins have been done, and the result are kept in the disk, running the SQL statement using the Materialized View will be far faster than the standard view. The Materialized View has the disadvantages that the data may become out of date, because new data may have entered the database or been deleted etc. In order to have up-to-date data, we have to periodically refresh the Views [230, 231]. As we only have two Views we did not concentrate on optimizing the refresh of these Views. Our Materialized View refresh is performed in parallel and takes around 15 minutes, which is expected according to the size of the tables (the three biggest tables have 160, 80 and 30 million rows). The refresh of the *prodview* and the *prodrunview* Materialize Views is performed every 30 minutes and they are used to extract relevant information from the Warehouse schema which is used by the users. The Materialized Views contain only information which will not change very often. Consequently, the tables are used together with these two Materialized Views to perform user requests.

Different SQL Stored Procedures and Functions are implemented and used, because they are faster than the traditional SQL statements. They are pre-compiled and remain in the memory after first being loaded into the memory.

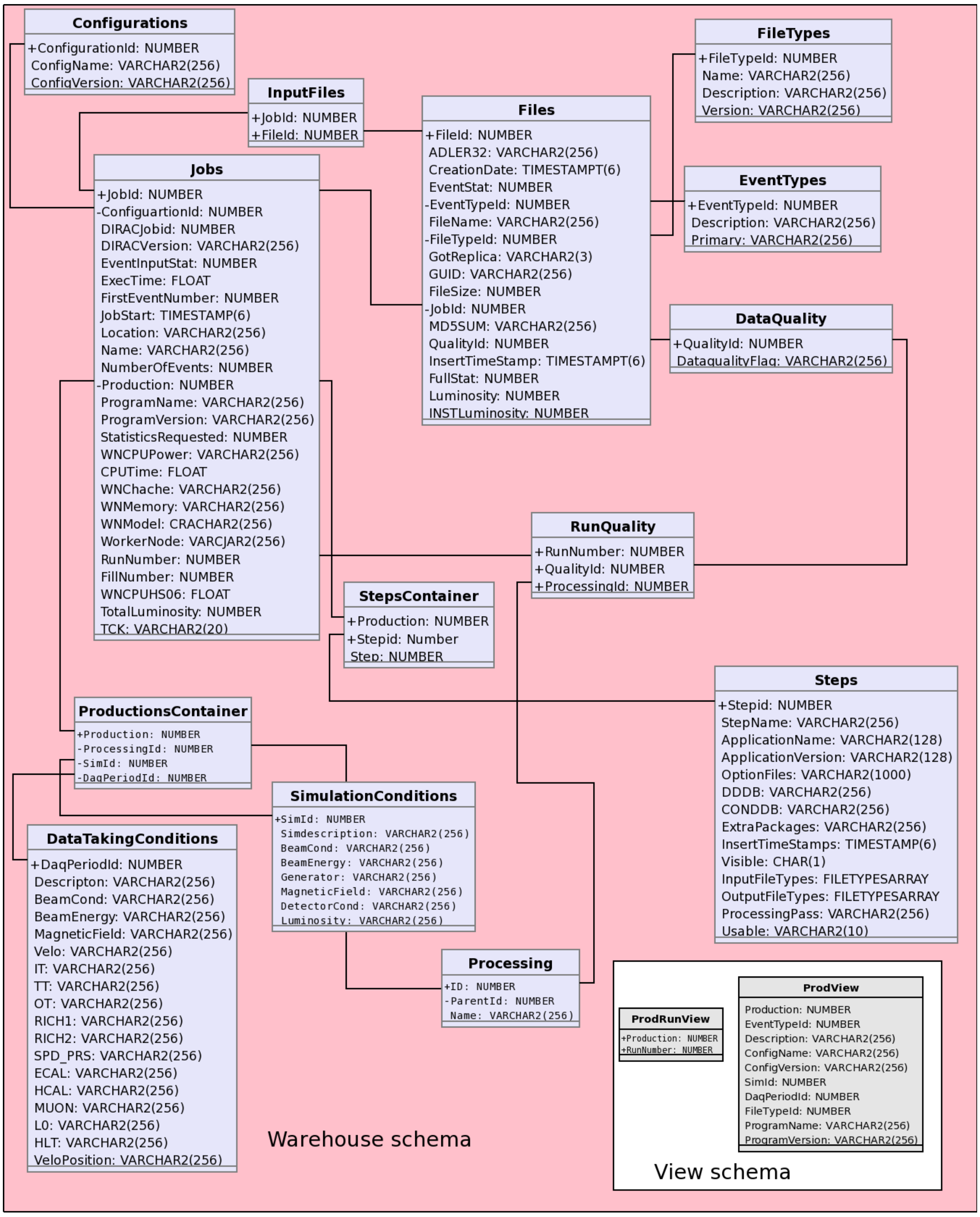


Figure 7.3.: The Bookkeeping Metadata catalogue schema.

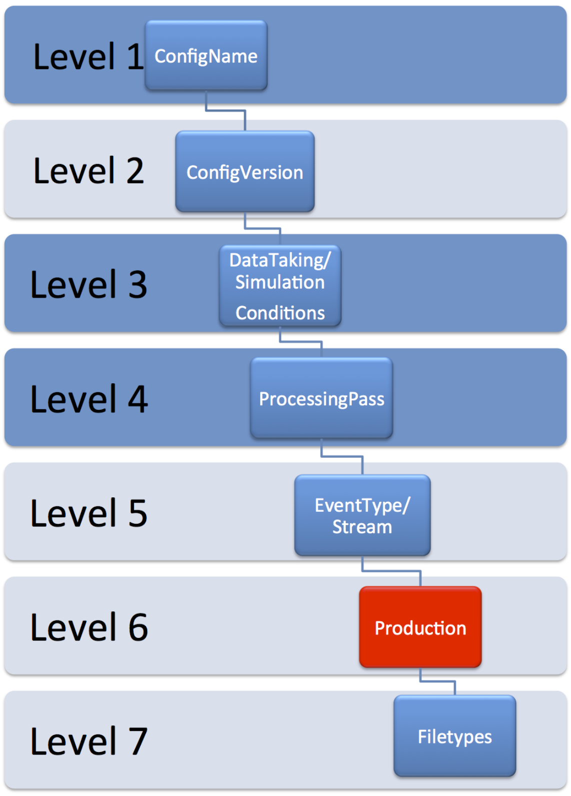


Figure 7.4.: Bookkeeping Tree; The color of the boxes indicate the typical use-case. The levels which are marked dark blue are used very often. The level 6 marked red is only used in specific cases.

Data categorization

In order to improve system performance we decided to categorize the data. Seven categories are used: Configuration Name, Configuration version, Event type/ Stream, Simulation/ Data taking conditions, Processing Pass, Filetypes. Each category forms a hierarchical structure which we call a **Bookkeeping Tree**. Figure 7.4 shows one Bookkeeping Tree. Each level of the Bookkeeping Tree corresponds to a data category. The first and second level of the Bookkeeping Tree is always Configuration Name and Configuration Version while the other levels can be changed. In addition we can set the visibility of the level(s). For example: Most of the time we have a lot of productions which correspond to the same dataset. From the user point of view the production is not really interesting. Consequently, the users can hide (set invisible) the production level (which is marked as red in the Figure 7.4). According to the users requirements we implemented four types of the Bookkeeping Tree, corresponding to the most common ways a user would wish to browse the data.

1. ConfigName, ConfigVersion, DataTaking/Simulation conditions, Processing Pass, Event Type, Production, File Types
2. ConfigName, ConfigVersion, Event Type, DataTaking/Simulation conditions, Processing Pass, Production, File Types
3. Production, Event Type, File Type
4. Run Number, Processing Pass, Stream, File Type

The most common queries are the 1 and 2 in terms of use. The last two queries are used by the experts (production, managers, shifters, etc.) for monitoring the production operations. Each level of the Bookkeeping Tree is associated with the relational algebra operations which are introduced below. These operations translated into SQL queries (more details in Section 7.5) and are used to retrieve information from the database. The result of a query only contains n-tuples whose values satisfy a given S condition, where S is a set of values selected from the result of the queries executed in each level of the Bookkeeping Tree (S = $\{s1, s2, s3_1, s3_2, s4, s5, s6, s7\}$). The relational algebra operations are the following:

- Level 1:
  $R1 = \prod_{ConfigName}(prodview)$
  $s1 = t[ConfigName]$ where t is an n-tuple in $R1$
- Level 2:
  $R2 = \prod_{ConfigVersion}(\sigma_{ConfigName=s1}(prodview))$
  $s2 = t[ConfigVersion]$, where t is an n-tuple in $R2$
- Level 3:
  $R3_1 = \prod_{simdesc}(\sigma_{ConfigName=s1 \wedge ConfigVersion=s2}(prodview) \bowtie_{simid=simid} (simulationConditions))$
  $R3_2 = \prod_{description}(\sigma_{ConfigName=s1 \wedge ConfigVersion=s2}(prodview) \bowtie_{daqperiodid=daqperiodid} (datatakingconditions))$

  $R3 = R3_1 \cup R3_2$
  $s3_1 = t[simdesc]$ or $s3_2 = t[description]$, where t is an n-tuple in $R3$
- Level 4:
  $R4_1 = \prod_{name}((\sigma_{ConfigName=s1 \wedge ConfigVersion=s2 \wedge simid=s3_1}(prodview) \bowtie_{production=production} (productionscontainer)) \bowtie_{processingid=id} (processing))$
  $R4_2 = \prod_{name}((\sigma_{ConfigName=s1 \wedge ConfigVersion=s2 \wedge daqperiodid=s3_2}(prodview) \bowtie_{production=production} (productionscontainer)) \bowtie_{processingid=id} (processing))$

  $R4 = R4_1 \cup R4_2$
  $s4 = t[name]$, where t is an n-tuple in $R4$
- Level 5:
  $R5_1 = \prod_{EventTypeId}((\sigma_{ConfigName=s1 \wedge ConfigVersion=s2 \wedge simid=s3_1}(prodview) \bowtie_{production=production} (productionscontainer)) \bowtie_{processingid=id} (\sigma_{name=s4}(processing)))$

$R5_2 = \prod_{EventTypeId}((\sigma_{ConfigName=s1 \wedge ConfigVersion=s2 \wedge daqperiodid=s3_2}(prodview) \bowtie_{production=production}$
$(productionscontainer)) \bowtie_{processingid=id} (\sigma_{name=s4}(processing)))$

$R5 = R5_1 \cup R5_2$
$s5 = t[EventTypeId]$, where t is an n-tuple in $R5$

- Level 6:
  $R6_1 = \prod_{Production}((\sigma_{ConfigName=s1 \wedge ConfigVersion=s2 \wedge simid=s3_1 \wedge eventtypeid=s5}(prodview)$
  $\bowtie_{production=production} (productionscontainer))$
  $\bowtie_{processingid=id} (\sigma_{name=s4}(processing)))$
  $R6_2 = \prod_{Production}((\sigma_{ConfigName=s1 \wedge ConfigVersion=s2 \wedge daqperiodid=s3_2 \wedge eventtypeid=s5}(prodview)$
  $\bowtie_{production=production} (productionscontainer))$
  $\bowtie_{processingid=id} (\sigma_{name=s4}(processing)))$

  $R6 = R6_1 \cup R6_2$
  $s6 = t[Production]$, where t is an n-tuple in $R6$

- Level 7:
  $R7_1 = \prod_{Name}(\sigma_{filetypeid}((\sigma_{ConfigName=s1 \wedge ConfigVersion=s2 \wedge simid=s3_1 \wedge eventtypeid=s5}$
  $\wedge production = s6(prodview)$
  $\bowtie_{production=production} (productionscontainer))$
  $\bowtie_{processingid=id} (\sigma_{name=s4}(processing)))$
  $\bowtie_{filetypeid=filetypeid} (filetypes))$
  $R7_2 = \prod_{Name}(\sigma_{filetypeid}((\sigma_{ConfigName=s1 \wedge ConfigVersion=s2 \wedge daqperiodid=s3_1 \wedge eventtypeid=s5}$
  $\wedge production = s6(prodview)$
  $\bowtie_{production=production} (productionscontainer))$
  $\bowtie_{processingid=id} (\sigma_{name=s4}(processing)))$
  $\bowtie_{filetypeid=filetypeid} (filetypes))$

  $R7 = R7_1 \cup R7_2$
  $s7 = t[Name]$, where t is an n-tuple in $R7$

The Oracle interface design and implementation

We used the cx_Oracle module to implement an interface on top of the Oracle database in order to execute SQL commands, Stored Procedures and Functions and parsing their result. This interface is called **OracleDB** and it is designed to manage thousands of connections in an optimized fashion. It allows the setting of a limit for concurrent connections. In addition, we keep the connections in an internal connection queue and as soon as a new request arrives, one connection from this queue is established. In this way we save time and resources. Another advantage of this interface is to hide the complexity of the database access.

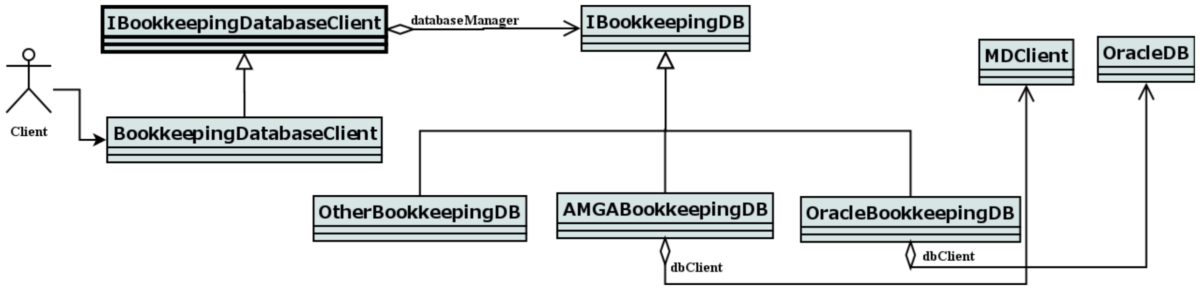


Figure 7.5.: Data and Query generation Layer.

7.5. Query generalization layer

The **Query Generation Layer** is used to build a query using different conditions as specified by the users. It plays a vital role in our system because it makes the relation between the application and the database, generates the queries, calls the Database Layer using these generated queries, parses the result and returns the result to the next layer. Together with the Data Layer it is responsible for the database specific operations. Figure 7.5 shows the detailed UML diagram of the Query Generalization layer.

Two modules provide access to the database:*MDClient* and *OracleDB*. The MDClient is a python API to the AMGA based Oracle database. The OracleDB has been presented in Section 7.4.2. *IBookkeepingDB* is an abstract interface which declares all the abstract methods which have to be implemented by its subclasses. We have implemented the *AMGABookkeepingDB* class because the old Bookkeeping System was based on AMGA, but in parallel the *OracleBookkeepingDB* class was developed. We designed an interface which allows the integration of any types of database. However, to decouple the various implementations of the IBookkeepingDB interface, we introduced the *IBookkeepingDatabaseClient* abstract interface. The *BookkeepingDatabaseClient* implements the IBookkeepingDatabaseClient and consists of the concrete implementation of the IBookkeepingDB. This technique allows us to easily switch between various implementation of the IBookkeepingDB without changing the functionality of the system.

OracleBookkeepingDB

This implements the IBookkeepingDB interface and provides all the methods needed to create the Bookkeeping Tree introduced in section 7.4.2. Each method implements a relation using given conditions. In addition it also provides the functionalities used by different DIRAC Systems during the data processing.

Each level of the Bookkeeping Tree is a relational algebra operation which can be performed on different relations. We used the following procedure to create the select commands based on the Bookkeeping Tree definition introduced in section 7.4.2:

- select the n-tuples which corresponds to the *k*-th Level of the Bookkeeping Tree by using the $\prod$ from *Rk* relation.
- select the name of the relations (which are names of tables) from *Rk*.
- make the condition string by comparing σ from *Rk* to the given conditions.
- return the result of the *Rk* relation.

The database queries are dynamic which means we do not have a predefined query; we only have information about the way a query can be created. Consequently, we have full control on the queries, can optimize them, and in case of a problem, the problematic query can be identified very easily and fixed.

7.6. Service Layer

The **Service Layer** serves the requests to the clients using the previous layer. It provides all the functionalities that are used to manipulate the data.

The Bookkeeping Service

The Bookkeeping Service runs the BookkeepingManager module which is dedicated to serve the user requests. The BookkeepingManager is based on the DIRAC secure framework. It manages the connections to the database, selecting and inserting data using the Bookkeeping-DatabaseClient. The Bookkeeping Service exploits the advantages of the DIRAC framework such as monitoring of this service, security which is provided by the DISET, multi-threading etc.

The Bookkeeping Service has to serve different user-specific requests which are created when a job finishes in the Grid. In order to interpret these requests the Bookkeeping Service has an associated module called *XMLFileReaderManager*. This module parses and interprets the XML requests, makes various checks (the minimal information are provided) and inserts the content of the XML file into the database.

7.7. Client Layer

The **Client Layer** is used by the Managers. It connects to the Service Layer and exposes the functionalities which are provided by the Service Layer.

The Bookkeeping Client

The Bookkeeping Client exposes the Bookkeeping Service functionalities and provides access to the Bookkeeping Service. The communication between the client and service are secure provided by DISET protocol. Because of this all users on the Grid have to use this client to access the Bookkeeping Service. When the user instantiates a Bookkeeping Client then a connection is established to the Bookkeeping Service. In order to allow efficient communication between the client and service the connections are kept for a short period. That means when the client sends a request to the Bookkeeping Service, then the connection will be re-used. In this way we save the overhead of time required to establish connections.

7.8. Manager Layer

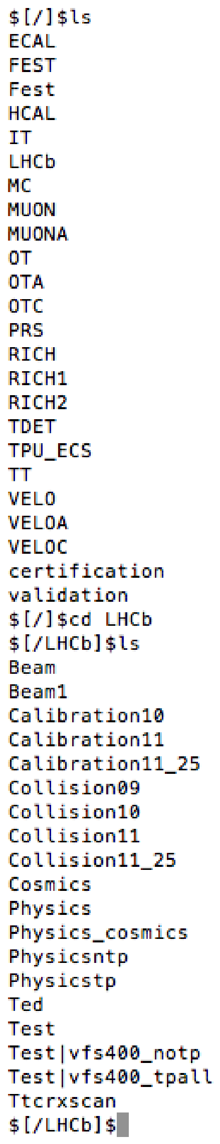
The **Manager Layer** provides the functionalities needed to present the database content as a *Virtual File system*. The representation of the data is organized into virtual directories and files. It consists of the LHCB_BKKDBManager class and the LHCB_BKKDBClient. The implementation of these classes is based on the Feicim Managers architecture which have been introduced in section 5.6. The OracleBookkeepingDB provides the information which is needed to create the Bookkeeping Tree structure while the LHCB_BKKDBManager is used to build the four Bookkeeping Tree (which were introduced in section 7.4.2). The LHCB_BKKDBClient exposes the functionalities of the LHCB_BKKDBManager and allows the users to browse through the data as a file system.

7.9. Presentation layer

The **Presentation Layer** is the highest level used by the users. It is the interaction point between the users and the database. The Presentation Layer implements a CMVC design pattern where the Manager Layer is the Model used by the Controllers. Various User Interfaces are available which are part of the Presentation layer in order to fulfill different user requests. We paid attention to design and implement User Interfaces which provide similar functionalities. To meet this requirement we developed the Command Line Interface which is the core of the User Interfaces. On top of this different User Interfaces are implemented which are discussed below.

7.9.1. Command Line Interface

Using the **Command Line Interface** (CLI) the users can browse through the database content in a similar fashion to a UNIX file system. Figure 7.6 shows the command line interface.



```
$[/]$ls
ECAL
FEST
Fest
HCAL
IT
LHCb
MC
MUON
MUONA
OT
OTA
OTC
PRS
RICH
RICH1
RICH2
TDET
TPU_ECS
TT
VELO
VELOA
VELOC
certification
validation
$[/]$cd LHCb
$[/LHCb]$ls
Beam
Beam1
Calibration10
Calibration11
Calibration11_25
Collision09
Collision10
Collision11
Collision11_25
Cosmics
Physics
Physics_cosmics
Physicsntp
Physicstp
Ted
Test
Test|vfs400_notp
Test|vfs400_tpall
Ttcrxscan
$[/LHCb]$
```

Figure 7.6.: A screen shot of the Bookkeeping Command Line showing the response to the command 'ls' - The users can use UNIX commands like ls, cd, help to browse the database content.

7.9.2. Graphical User Interface

The GUI is implemented on top of the CLI and its design is based on the Composite Model View Controller design pattern which were introduced in section 5.4.1.

The GUI consists of the following widgets: Main widget, Production Lookup, Info dialog, Processing Pass dialog, Bookmarks widget, Add Bookmarks widget, File dialog, Advanced save dialog, History dialog and LogFile widget. Using the Composite Model View Controller we built the Controllers hierarchy which is shown in Figure 7.7.

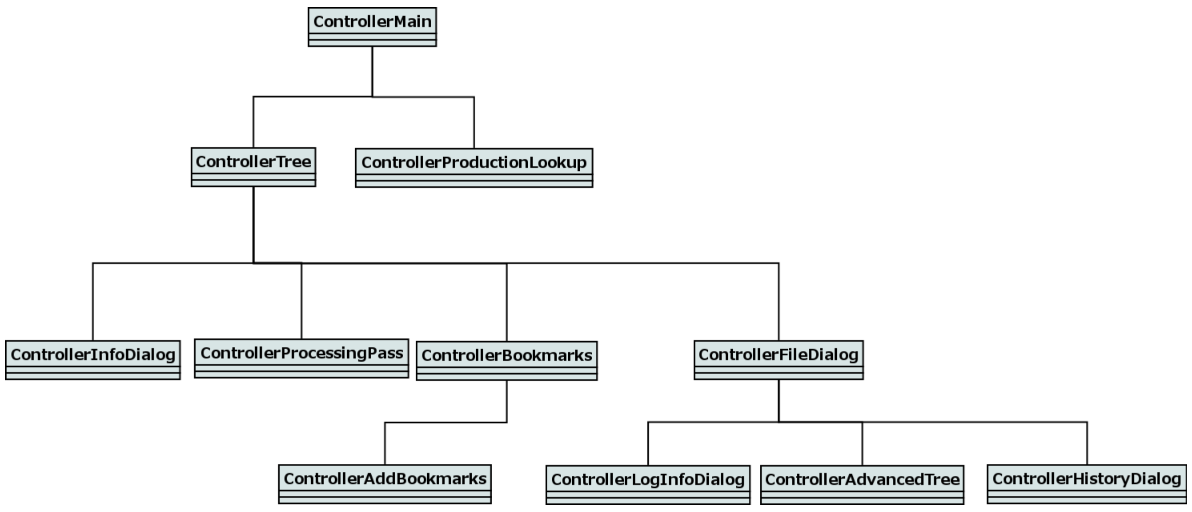


Figure 7.7.: Controllers which are used by the GUI.

Each Controller has an associated widget which is used to present the data to the users. The **ControllerMainwidget** uses the CLI in order to answer the users requests. In addition it distributes the data to its child Controllers.

Main widget

The **Main widget** allows the users to browse through the content of the database. This widget has an associated controller which is called the **ControllerTree**. The ControllerTree provides the data using Messages to its child controllers and widgets. Figure 7.8 shows the main Bookkeeping windows.

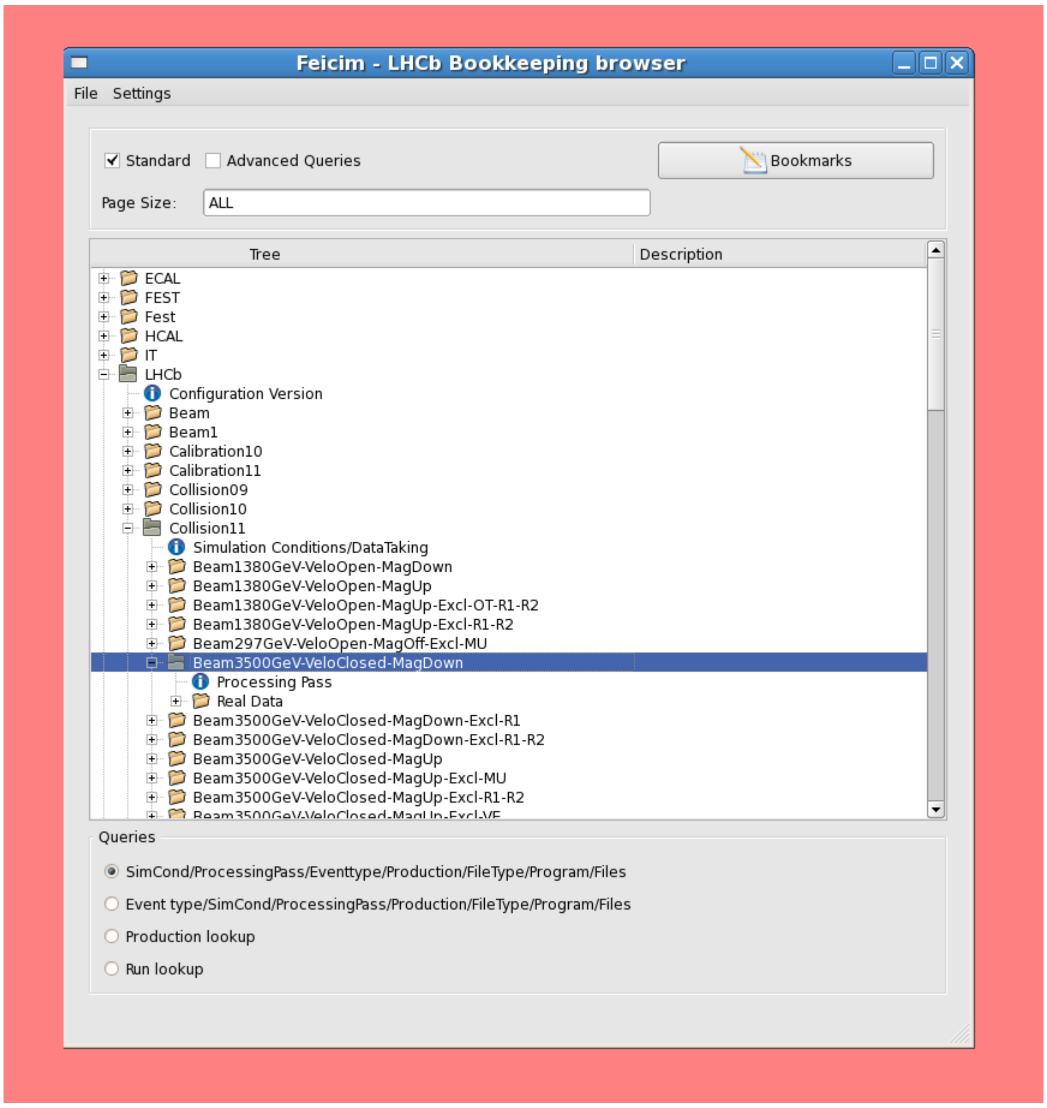
At the top of the window two check boxes can be seen, one text box and one Bookmarks button. The Check boxes allow the users to select the type of queries. The user can also limit the number of datasets in the File Dialog window entering a number into the text box. When a dataset is selected, then it can be bookmarked using the Bookmarks button.

In the middle of the picture the Bookkeeping Tree is found. The users can navigate through the folders by selecting their metadata which identify a specific dataset.

At the bottom of the picture the different types of Bookkeeping Tree (which were introduced in section 7.4.2) can be selected.

ProductionLookup

The **ProductionLookup** is a widget which shows all the available production/run numbers which are available in the Bookkeeping Metadata Catalogue. The **ControllerProduction-Lookup** associated to this widget is used to provide the data to the widget by asking the



☑ Standard ☐ Advanced Queries

Page Size: ALL

```
⊞ ECAL
⊞ FEST
⊞ Fest
⊞ HCAL
⊞ IT
⊟ LHCb
    Configuration Version
    ⊞ Beam
    ⊞ Beam1
    ⊞ Calibration10
    ⊞ Calibration11
    ⊞ Collision09
    ⊞ Collision10
    ⊟ Collision11
        Simulation Conditions/DataTaking
        ⊞ Beam1380GeV-VeloOpen-MagDown
        ⊞ Beam1380GeV-VeloOpen-MagUp
        ⊞ Beam1380GeV-VeloOpen-MagUp-Excl-OT-R1-R2
        ⊞ Beam1380GeV-VeloOpen-MagUp-Excl-R1-R2
        ⊞ Beam297GeV-VeloOpen-MagOff-Excl-MU
        ⊟ Beam3500GeV-VeloClosed-MagDown
            Processing Pass
            ⊞ Real Data
        ⊞ Beam3500GeV-VeloClosed-MagDown-Excl-R1
        ⊞ Beam3500GeV-VeloClosed-MagDown-Excl-R1-R2
        ⊞ Beam3500GeV-VeloClosed-MagUp
        ⊞ Beam3500GeV-VeloClosed-MagUp-Excl-MU
        ⊞ Beam3500GeV-VeloClosed-MagUp-Excl-R1-R2
```

[● SimCond/ProcessingPass/Eventtype/Production/FileType/Program/Files]

○ Event type/SimCond/ProcessingPass/Production/FileType/Program/Files

○ Production lookup

○ Run lookup

Figure 7.8.: Bookkeeping Main widget.

ControllerMain controller. Using this widget the users can find specific information for a given production/run and they can select files which corresponds to a given production/run.

InfoDialog

Each folder shown in Figure 7.8 may have associated metadata information which can be seen by right-clicking on the folder. The *InfoDialog* widget is used to show this additional information. To provide the metadata information the *ControllerInfoDialog* sends metadata related requests to its appropriate controller which will return the requested metadata information.

ProcessingPassDialog

The **ProcessingPassDialog** deals with the data provenance information such as data processing phases, software used to process data etc. (more information found in section 7.4.1). The **ControllerProductionLookup** provides this information to the ProcessingPassDialog. Using the information provided by this dialog the users have information about the processing of specific datasets. Sometimes it is necessary to re-process datasets which have already been processed using different algorithms. Without this information re-processing these datasets is not trivial.

BookmarksWidget

The users can save their selected datasets using the **BookmarksWidget** by right clicking on a folder in the Bookkeeping Tree. Using this widget each user can bookmark their favour datasets. The **ControllerBookmarks** is assigned to this widget and provides the bookmarks to it.

AddBookmarksWidget

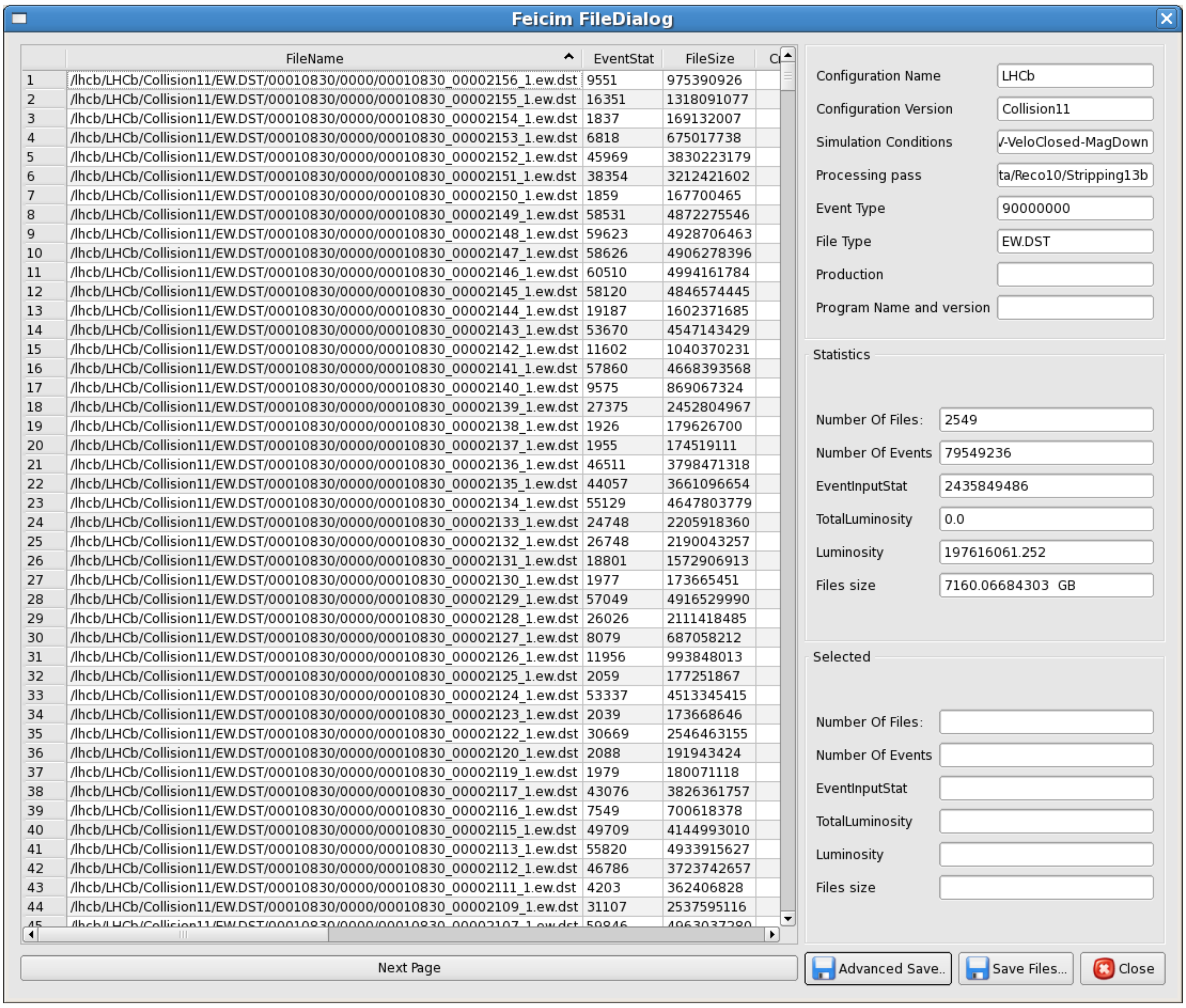
To add and delete Bookmarks the **AddBookmarkswidget** can be used which provide a user friendly presentation of the bookmarks. The **ControllerAddBookmarks** is assigned to this widget in order to delete or add bookmarks by sending Messages to its parent controller.

FileDialog

The **FileDialog** displays the LFN names and additional information from a given dataset. The **ControllerFileDialog** provides these LFN and additional information to the FileDialog widget. Figure 7.9 show an example of the FileDialog widget.

AdvancedSave

The users can save their selected dataset in different file formats, in particular to a Gaudi configuration file. In addition it hides the complexities of the data accessing using DIRAC Data Management. It allows the users to save data in the POOL[232] XML file format which contains the information required to have access to the data stored in a Grid Storage Element. The AdvancedSave widget has an associated controller which is called the **ControllerAdvancedSave**. It provides different information to the AdvancedSave widget such as Grid sites, file name etc. required to save the LFNs.



	FileName	EventStat	FileSize	C
1	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002156_1.ew.dst	9551	975390926	
2	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002155_1.ew.dst	16351	1318091077	
3	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002154_1.ew.dst	1837	169132007	
4	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002153_1.ew.dst	6818	675017738	
5	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002152_1.ew.dst	45969	3830223179	
6	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002151_1.ew.dst	38354	3212421602	
7	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002150_1.ew.dst	1859	167700465	
8	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002149_1.ew.dst	58531	4872275546	
9	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002148_1.ew.dst	59623	4928706463	
10	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002147_1.ew.dst	58626	4906278396	
11	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002146_1.ew.dst	60510	4994161784	
12	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002145_1.ew.dst	58120	4846574445	
13	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002144_1.ew.dst	19187	1602371685	
14	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002143_1.ew.dst	53670	4547143429	
15	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002142_1.ew.dst	11602	1040370231	
16	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002141_1.ew.dst	57860	4668393568	
17	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002140_1.ew.dst	9575	869067324	
18	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002139_1.ew.dst	27375	2452804967	
19	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002138_1.ew.dst	1926	179626700	
20	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002137_1.ew.dst	1955	174519111	
21	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002136_1.ew.dst	46511	3798471318	
22	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002135_1.ew.dst	44057	3661096654	
23	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002134_1.ew.dst	55129	4647803779	
24	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002133_1.ew.dst	24748	2205918360	
25	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002132_1.ew.dst	26748	2190043257	
26	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002131_1.ew.dst	18801	1572906913	
27	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002130_1.ew.dst	1977	173665451	
28	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002129_1.ew.dst	57049	4916529990	
29	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002128_1.ew.dst	26026	2111418485	
30	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002127_1.ew.dst	8079	687058212	
31	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002126_1.ew.dst	11956	993848013	
32	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002125_1.ew.dst	2059	177251867	
33	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002124_1.ew.dst	53337	4513345415	
34	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002123_1.ew.dst	2039	173668646	
35	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002122_1.ew.dst	30669	2546463155	
36	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002120_1.ew.dst	2088	191943424	
37	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002119_1.ew.dst	1979	180071118	
38	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002117_1.ew.dst	43076	3826361757	
39	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002116_1.ew.dst	7549	700618378	
40	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002115_1.ew.dst	49709	4144993010	
41	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002113_1.ew.dst	55820	4933915627	
42	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002112_1.ew.dst	46786	3723742657	
43	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002111_1.ew.dst	4203	362406828	
44	/lhcb/LHCb/Collision11/EW.DST/00010830/0000/00010830_00002109_1.ew.dst	31107	2537595116	

Configuration Name	LHCb
Configuration Version	Collision11
Simulation Conditions	/-VeloClosed-MagDown
Processing pass	ta/Reco10/Stripping13b
Event Type	90000000
File Type	EW.DST
Production	
Program Name and version	

Statistics

Number Of Files:	2549
Number Of Events	79549236
EventInputStat	2435849486
TotalLuminosity	0.0
Luminosity	197616061.252
Files size	7160.06684303 GB

Selected

Number Of Files:	
Number Of Events	
EventInputStat	
TotalLuminosity	
Luminosity	
Files size	

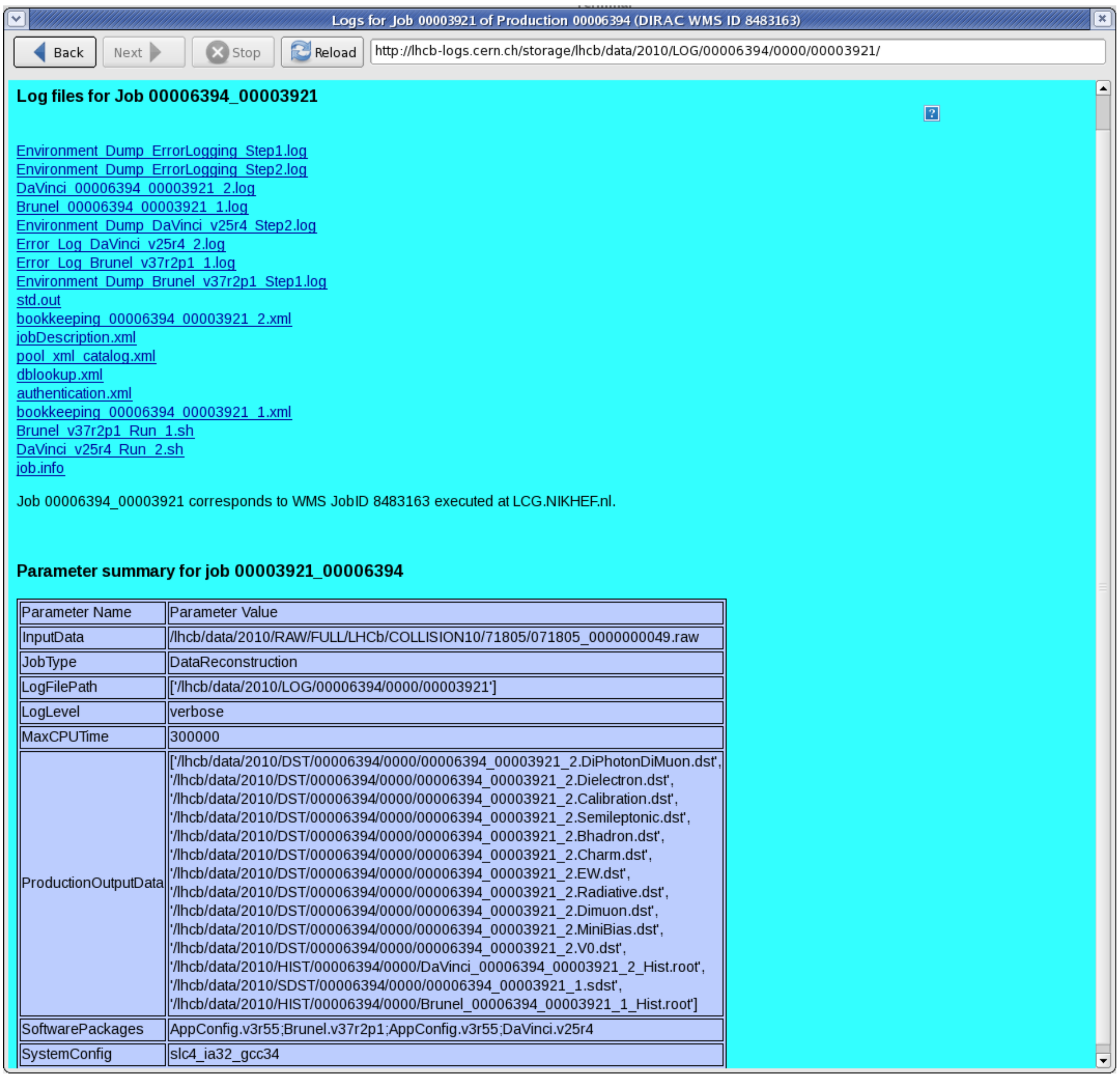
Figure 7.9.: Example view of the FileDialog widget.

HistoryDialog

Each Grid job corresponds to a step and each step create files. The HistoryDialog widget allows navigating through the chain of files in order to have a clear history of the file creation phases. The **ControllerHistoryDialog** is the controller of the HistoryDialog widget and provides the ancestors or descendants needed to navigate the files chain.

LogFileWidget

Each job in the Grid creates log files which are stored in the LHCb LogSE (Log Storage Element). The Bookkeeping Metadata catalogue stores these log file relative paths that can be used to open a log file in the LogSE using the HTTP protocol. Figure 7.10 shows the


Logs for Job 00003921 of Production 00006394 (DIRAC WMS ID 8483163)
Back
Next
Stop
Reload
http://lhcb-logs.cern.ch/storage/lhcb/data/2010/LOG/00006394/0000/00003921/

Log files for Job 00006394_00003921

Environment_Dump_ErrorLogging_Step1.log
Environment_Dump_ErrorLogging_Step2.log
DaVinci_00006394_00003921_2.log
Brunel_00006394_00003921_1.log
Environment_Dump_DaVinci_v25r4_Step2.log
Error_Log_DaVinci_v25r4_2.log
Error_Log_Brunel_v37r2p1_1.log
Environment_Dump_Brunel_v37r2p1_Step1.log
std.out
bookkeeping_00006394_00003921_2.xml
jobDescription.xml
pool_xml_catalog.xml
dblookup.xml
authentication.xml
bookkeeping_00006394_00003921_1.xml
Brunel_v37r2p1_Run_1.sh
DaVinci_v25r4_Run_2.sh
job.info

Job 00006394_00003921 corresponds to WMS JobID 8483163 executed at LCG.NIKHEF.nl.

Parameter summary for job 00003921_00006394

Parameter Name	Parameter Value
InputData	/lhcb/data/2010/RAW/FULL/LHCb/COLLISION10/71805/071805_0000000049.raw
JobType	DataReconstruction
LogFilePath	['/lhcb/data/2010/LOG/00006394/0000/00003921']
LogLevel	verbose
MaxCPUTime	300000
ProductionOutputData	['/lhcb/data/2010/DST/00006394/0000/00006394_00003921_2.DiPhotonDiMuon.dst', '/lhcb/data/2010/DST/00006394/0000/00006394_00003921_2.Dielectron.dst', '/lhcb/data/2010/DST/00006394/0000/00006394_00003921_2.Calibration.dst', '/lhcb/data/2010/DST/00006394/0000/00006394_00003921_2.Semileptonic.dst', '/lhcb/data/2010/DST/00006394/0000/00006394_00003921_2.Bhadron.dst', '/lhcb/data/2010/DST/00006394/0000/00006394_00003921_2.Charm.dst', '/lhcb/data/2010/DST/00006394/0000/00006394_00003921_2.EW.dst', '/lhcb/data/2010/DST/00006394/0000/00006394_00003921_2.Radiative.dst', '/lhcb/data/2010/DST/00006394/0000/00006394_00003921_2.Dimuon.dst', '/lhcb/data/2010/DST/00006394/0000/00006394_00003921_2.MiniBias.dst', '/lhcb/data/2010/DST/00006394/0000/00006394_00003921_2.V0.dst', '/lhcb/data/2010/HIST/00006394/0000/DaVinci_00006394_00003921_2_Hist.root', '/lhcb/data/2010/SDST/00006394/0000/00006394_00003921_1.sdst', '/lhcb/data/2010/HIST/00006394/0000/Brunel_00006394_00003921_1_Hist.root']
SoftwarePackages	AppConfig.v3r55;Brunel.v37r2p1;AppConfig.v3r55;DaVinci.v25r4
SystemConfig	slc4_ia32_gcc34

Figure 7.10.: Web browser to show the content of the LogSE interface.

LogFileWidget which deals with the log files. It is implemented like a web browser and allows the user to browse easily through the content of the LogSE. The **ControllerLogInfoDialog** is used to control the LogFileWidget and provides the data displayed by the LogFileWidget.

Instead of having one controller which could be overloaded by sending too many messages to it, we distributed the data (model) to different controllers: ControllerMain, ControllerTree, ControllerFileDialog. These three controllers become the main controllers. Consequently, the ControllerMain controller only receives messages if the ControllerTree or the ControllerFileDialog can not handle the requests. In this way we reduce the load of the Bookkeeping Metadata catalogue.

The widgets: ProductionLookup, Processing Pass dialog, Info dialog, Bookmarks widget, AddBookmarks widget and their controllers are used to *define datasets*. The main controller of these controllers is the ControllerTree. The File dialog and its child widgets and their controllers are used to *select datasets*.

7.9.3. Web Interface

The CLI interface is integrated into the DIRAC Web Portal to provide more possibilities to select datasets from the Bookkeeping Metadata catalogue. The LHCb Bookkeeping Web interface[237] can be accessed using a valid Grid Certificate which is uploaded to the web browser. It provides similar functionality to other User Interface. Figure 7.11 shows the Bookkeeping web interface.

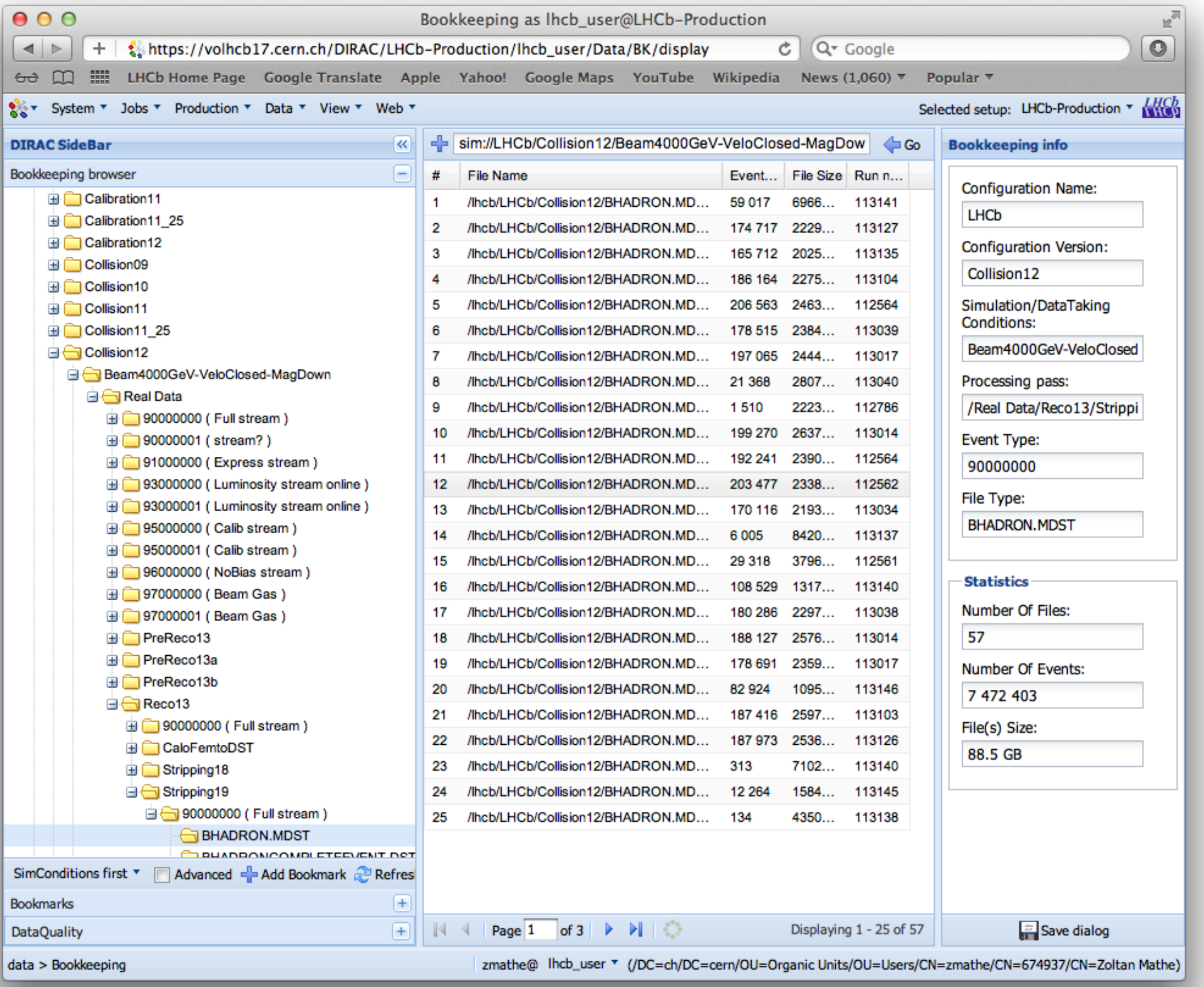
7.10. The LHCb Bookkeeping System performances

To achieve very good performance of a complex system is extremely difficult. We used profilers for performance analysis, different monitoring tools to monitor our system, and Multi-Mechanize frameworks for performance and load testing[239]. The **profiling** is a form of dynamic program analysis that measures different metrics such as the usage of memory, the number of function calls, etc, which can be used for program optimization. We used two profilers: AUTOTRACE[233] to analyse the database performance and we developed a python tool for profiling the Bookkeeping service. We used the DIRAC **monitoring** pages to monitor the Bookkeeping service and the CERN oracle monitoring page, which will be presented in Section 7.10 . Various query optimisation techniques were studied; the most important techniques are introduced in the next sections.

Profiling

Each query is executed in different ways by the RDBMS. The **execution plan** contains the steps that the Oracle database performs while executing the query and provides statistics about how much work (resources) is needed to execute the query. AUTOTRACE is a tool used for tracing Oracle database queries in order to identify their performance problems. AUTOTRACE is provided by Oracle and can be used through the SQL*Plus[238] Oracle command line utility. Figure 7.12 shows an execution plan for a given query. At the top of the figure 7.12 we can see a table which has the following columns:

- *Id* column contains the order of the steps.
- *Operation* column contains the operations (Join, Select, etc.) which will be performed on the tables.
- *Name* column contains the name of the tables which are used by the Operation.



Calibration11
Calibration11_25
Calibration12
Collision09
Collision10
Collision11
Collision11_25
Collision12
Beam4000GeV-VeloClosed-MagDown
Real Data
90000000 (Full stream)
90000001 (stream?)
91000000 (Express stream)
93000000 (Luminosity stream online)
93000001 (Luminosity stream online)
95000000 (Calib stream)
95000001 (Calib stream)
96000000 (NoBias stream)
97000000 (Beam Gas)
97000001 (Beam Gas)
PreReco13
PreReco13a
PreReco13b
Reco13
90000000 (Full stream)
CaloFemtoDST
Stripping18
Stripping19
90000000 (Full stream)
BHADRON.MDST

sim://LHCb/Collision12/Beam4000GeV-VeloClosed-MagDow

#	File Name	Event...	File Size	Run n...
1	/lhcb/LHCb/Collision12/BHADRON.MD...	59 017	6966...	113141
2	/lhcb/LHCb/Collision12/BHADRON.MD...	174 717	2229...	113127
3	/lhcb/LHCb/Collision12/BHADRON.MD...	165 712	2025...	113135
4	/lhcb/LHCb/Collision12/BHADRON.MD...	186 164	2275...	113104
5	/lhcb/LHCb/Collision12/BHADRON.MD...	206 563	2463...	112564
6	/lhcb/LHCb/Collision12/BHADRON.MD...	178 515	2384...	113039
7	/lhcb/LHCb/Collision12/BHADRON.MD...	197 065	2444...	113017
8	/lhcb/LHCb/Collision12/BHADRON.MD...	21 368	2807...	113040
9	/lhcb/LHCb/Collision12/BHADRON.MD...	1 510	2223...	112786
10	/lhcb/LHCb/Collision12/BHADRON.MD...	199 270	2637...	113014
11	/lhcb/LHCb/Collision12/BHADRON.MD...	192 241	2390...	112564
12	/lhcb/LHCb/Collision12/BHADRON.MD...	203 477	2338...	112562
13	/lhcb/LHCb/Collision12/BHADRON.MD...	170 116	2193...	113034
14	/lhcb/LHCb/Collision12/BHADRON.MD...	6 005	8420...	113137
15	/lhcb/LHCb/Collision12/BHADRON.MD...	29 318	3796...	112561
16	/lhcb/LHCb/Collision12/BHADRON.MD...	108 529	1317...	113140
17	/lhcb/LHCb/Collision12/BHADRON.MD...	180 286	2297...	113038
18	/lhcb/LHCb/Collision12/BHADRON.MD...	188 127	2576...	113014
19	/lhcb/LHCb/Collision12/BHADRON.MD...	178 691	2359...	113017
20	/lhcb/LHCb/Collision12/BHADRON.MD...	82 924	1095...	113146
21	/lhcb/LHCb/Collision12/BHADRON.MD...	187 416	2597...	113103
22	/lhcb/LHCb/Collision12/BHADRON.MD...	187 973	2536...	113126
23	/lhcb/LHCb/Collision12/BHADRON.MD...	313	7102...	113140
24	/lhcb/LHCb/Collision12/BHADRON.MD...	12 264	1584...	113145
25	/lhcb/LHCb/Collision12/BHADRON.MD...	134	4350...	113138

Page 1 of 3 Displaying 1 - 25 of 57

Configuration Name: LHCb
Configuration Version: Collision12
Simulation/DataTaking Conditions: Beam4000GeV-VeloClosed
Processing pass: /Real Data/Reco13/Strippi
Event Type: 90000000
File Type: BHADRON.MDST

Statistics
Number Of Files: 57
Number Of Events: 7 472 403
File(s) Size: 88.5 GB

zmathe@ lhcb_user (/DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=zmathe/CN=674937/CN=Zoltan Mathe)

Figure 7.11.: Feicim data browser GUI.

- *Rows* column contains the number of rows which will be processed by a step.
- *Bytes* column contains the size of the data which will read by a step.
- *Cost (%CPU)* columns contains the cost of an operation.
- *Time* columns contains the estimation of the time needed to execute a step.
- *Pstart and Pstop* (partition start and partition stop) columns contains the beginning and the end values of the partitions being accessed by each operations.

In the middle of figure 7.12 we can see the *Predicate Information* (access and filter conditions) which tell how we accessed a table and which filter condition will be performed. At the bottom of the Figure 7.12 we can see *Statistics*:

- *recursive calls* The number of internal Oracle calls during the execution of the query.
- *db block gets* The number of database blocks read.
- *consistent gets* The number of consistent reads from buffer.
- *physical reads* The number of physical reads from disk.
- *redo size* The number of redo entries.
- *bytes sent via SQL*Net to client* The number of bytes sent across the network from server to client.
- *bytes received via SQL*Net from client* The number of bytes received across the network from client to server.
- *SQL*Net roundtrips to/from client* The number of exchanges between the client and server. If this number is high, we have to customize the Oracle array size according to the network connection.
- *sorts (memory)* The number of data sorts using memory.
- *sorts (disk)* The number of data sorts using disk.
- *rows processed* The number of rows processed by the query.

We noted in particular the following parameters: Name, Rows, Bytes. In addition we checked the db block gets, consistent gets, physical reads parameters and row processed. The *SQL*Net round-strips to/from client* tells how many connections were used to return the data.

We implemented a python tool for profiling the Bookkeeping service. The Bookkeeping service logs the request from its clients, the start and end time stamps of a request in addition to the time which was spent to serve the request. It logs where the requests were made (the IP address of the client) and the user nick name retrieved from the user certificate (proxy). The python tool collects the information from the Bookkeeping service log. It creates different plots using the ROOT framework[242] to collect the statistics.

Monitoring

We split the monitoring tools into two categories according to their usage: monitoring the Bookkeeping Service and monitoring the Oracle database.

One of the basic services of the DIRAC framework is the DIRAC Monitoring service which provides the real status of the DIRAC services[184]. The DIRAC web portal displays the monitoring information. The DIRAC Monitoring Service is used to monitor the Bookkeeping service[234].

The Oracle Enterprise Manager is used for Oracle database level monitoring[235]. We used the PhyDB portal for the Oracle service monitoring provided by the CERN physics database service team[236].

Performance and load testing

To understand the behaviour of the Bookkeeping Service under load we used the Multi-Mechanise open source framework for API performance and load testing. During the performance testing we can identify the weakness of our system and we can check how the system meets the stated performance criteria. We used this tool to measure different parameters such as response time, throughput etc. which are introduced in Section 7.10.1.

Techniques used for query optimization

One of the most important functional requirements of the Relational Database Management System (RDBMS) is to execute queries in a reliable time. Various commercial or scientific applications have to manage very big tables which can contain billions of rows that require fast results. The LHCb detector and Monte Carlo simulation produce huge amount of data and its associated metadata are stored in large database tables (one table contains more than 100 million rows). Accessing these metadata information must be fast. The users cannot wait one or two minutes to get the results. That requires an understanding and study of the query optimization procedures. A given query can have many execution plans in the RDBMS and give an equivalent result. Only the time needed to execute this query is different. We use AUTOTRACE to generate the query execution plan and also to display statistics. In addition we made the query tree understand how the tables are joined and how the Oracle query optimizer should join the tables. We tried to decouple the queries and execute them sequentially. Instead of running a big query we split it into two or more queries. Each query reduces the number of rows in the tables. When the last query is performed the number of rows in the tables is already reduced. Consequently, the query will be very fast. We attempt to avoid the usage of Natural joins.

There are many queries in the LHCb Bookkeeping System and since we have optimized each in the same way we present the following query optimization as an example. For a given simulation condition, [Configuration Name and Version, Event Type, File type and processing pass] the query has to return the number of files, the size of the files and the total number of events. The following tables are used: prodview (or we may call it bview), configurations, files, simulationconditions, productionscontainer, processing and filetypes which were introduced in Section 7.4.2.

We divided the query into 3 queries which will be executed sequentially:

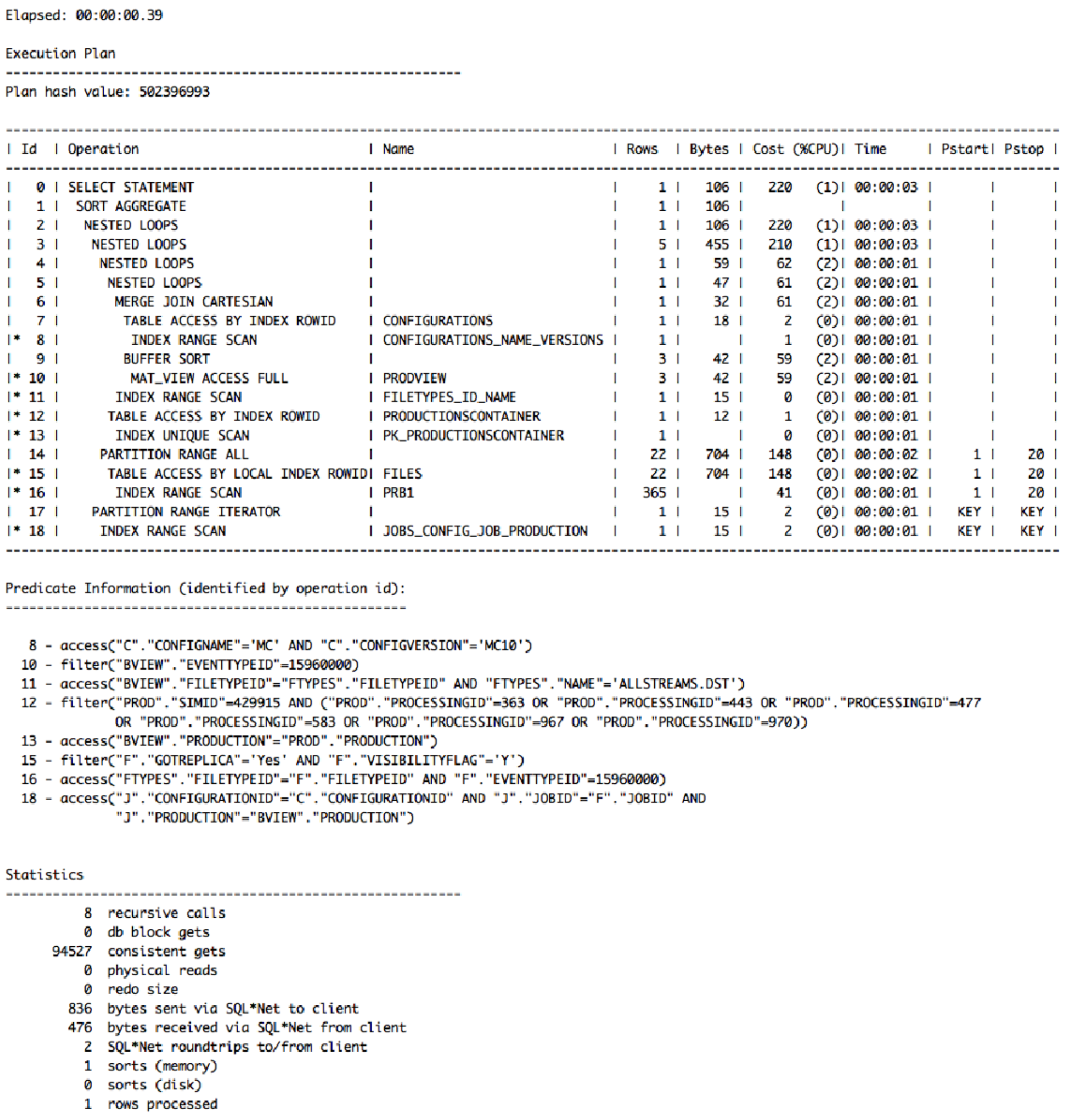
```
1 SELECT simid
FROM simulationconditions
WHERE simdescription = 'Beam3500GeV-Oct2010-MagDown-Nu2,5';
```

```
2 SELECT distinct prod.processingid
FROM  productionscontainer prod
WHERE  prod.processingid in
 ( SELECT v.id
   FROM
    ( SELECT distinct SYS_CONNECT_BY_PATH(name, '/') Path, id ID
        FROM processing v
        START WITH id in
          (SELECT distinct id
            FROM processing
            WHERE name='Sim01')
        CONNECT BY NOCYCLE PRIOR  id=parentid) v
        WHERE
        v.path='/Sim01/Trig0x002e002aFlagged/Reco08/Stripping12Flagged') and
prod.simid=429915 and
prod.simid is not null;
```

```
3  SELECT /*+ NOPARALLEL(bview) */ count(*), SUM(f.EventStat),
                SUM(f.FILESIZE), SUM(f.luminosity), SUM(f.instLuminosity)
  FROM  files f, jobs j, productionscontainer prod, configurations c,
           filetypes ftypes, prodview bview
  WHERE j.jobid=f.jobid and
                ftypes.filetypeid=f.filetypeid and
                f.gotreplica='Yes' and
                f.visibilityflag='Y' and
                ftypes.filetypeid=f.filetypeid and
                j.configurationid=c.configurationid  and
                c.configname='MC'  and
                c.configversion='MC10'  and
                prod.simid=429915 and
                prod.simid is not null  and
                j.production=bview.production and
                bview.production=prod.production and
                bview.eventtypeid=15960000 and
                f.eventtypeid=bview.eventtypeid  and
                ftypes.name='ALLSTREAMS.DST' and
                bview.filetypeid=ftypes.filetypeid and
                prod.processingid in (477,967,583,970,443,363);
```

The first and second query have an average execution time of 0.08 second, while the third query takes much longer (around 0.39 second). In order to optimize the query we used AUTOTRACE and we generated the execution plan shown in Figure 7.12.

When the execution time is not acceptable (ie. it takes more than 20 second), we compare the Oracle result to our query tree (see Figure 7.13) and try to understand the cause of the problem. As we can see in figure 7.12 we have a Cartesian product called MERGE JOIN CARTESIAN. According to the query tree the configurations and prodview tables are joined and the result is this Cartesian product. But if we analyse the problem deeper, we find another problem. We have to join the *filetypes* and *files* tables. If this join happens before the join of the jobs, files or *bview* tables, it will create a Cartesian product again. The solution is to change this query. We



```
Elapsed: 00:00:00.39

Execution Plan
----------------------------------------------------------
Plan hash value: 502396993

--------------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                             | Name                          | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
--------------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                      |                              |     1 |   106 |   220   (1)| 00:00:03 |       |       |
|   1 |  SORT AGGREGATE                       |                              |     1 |   106 |            |          |       |       |
|   2 |   NESTED LOOPS                        |                              |     1 |   106 |   220   (1)| 00:00:03 |       |       |
|   3 |    NESTED LOOPS                       |                              |     5 |   455 |   210   (1)| 00:00:03 |       |       |
|   4 |     NESTED LOOPS                      |                              |     1 |    59 |    62   (2)| 00:00:01 |       |       |
|   5 |      NESTED LOOPS                     |                              |     1 |    47 |    61   (2)| 00:00:01 |       |       |
|   6 |       MERGE JOIN CARTESIAN            |                              |     1 |    32 |    61   (2)| 00:00:01 |       |       |
|   7 |        TABLE ACCESS BY INDEX ROWID    | CONFIGURATIONS               |     1 |    18 |     2   (0)| 00:00:01 |       |       |
|*  8 |         INDEX RANGE SCAN              | CONFIGURATIONS_NAME_VERSIONS |     1 |       |     1   (0)| 00:00:01 |       |       |
|   9 |        BUFFER SORT                    |                              |     3 |    42 |    59   (2)| 00:00:01 |       |       |
|* 10 |         MAT_VIEW ACCESS FULL          | PRODVIEW                     |     3 |    42 |    59   (2)| 00:00:01 |       |       |
|* 11 |       INDEX RANGE SCAN                | FILETYPES_ID_NAME            |     1 |    15 |     0   (0)| 00:00:01 |       |       |
|* 12 |      TABLE ACCESS BY INDEX ROWID      | PRODUCTIONSCONTAINER         |     1 |    12 |     1   (0)| 00:00:01 |       |       |
|* 13 |       INDEX UNIQUE SCAN               | PK_PRODUCTIONSCONTAINER      |     1 |       |     0   (0)| 00:00:01 |       |       |
|  14 |     PARTITION RANGE ALL               |                              |    22 |   704 |   148   (0)| 00:00:02 |     1 |    20 |
|* 15 |      TABLE ACCESS BY LOCAL INDEX ROWID| FILES                        |    22 |   704 |   148   (0)| 00:00:02 |     1 |    20 |
|* 16 |       INDEX RANGE SCAN                | PRB1                         |   365 |       |    41   (0)| 00:00:01 |     1 |    20 |
|  17 |    PARTITION RANGE ITERATOR           |                              |     1 |    15 |     2   (0)| 00:00:01 |   KEY |   KEY |
|* 18 |     INDEX RANGE SCAN                  | JOBS_CONFIG_JOB_PRODUCTION   |     1 |    15 |     2   (0)| 00:00:01 |   KEY |   KEY |
--------------------------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   8 - access("C"."CONFIGNAME"='MC' AND "C"."CONFIGVERSION"='MC10')
  10 - filter("BVIEW"."EVENTTYPEID"=15960000)
  11 - access("BVIEW"."FILETYPEID"="FTYPES"."FILETYPEID" AND "FTYPES"."NAME"='ALLSTREAMS.DST')
  12 - filter("PROD"."SIMID"=429915 AND ("PROD"."PROCESSINGID"=363 OR "PROD"."PROCESSINGID"=443 OR "PROD"."PROCESSINGID"=477
              OR "PROD"."PROCESSINGID"=583 OR "PROD"."PROCESSINGID"=967 OR "PROD"."PROCESSINGID"=970))
  13 - access("BVIEW"."PRODUCTION"="PROD"."PRODUCTION")
  15 - filter("F"."GOTREPLICA"='Yes' AND "F"."VISIBILITYFLAG"='Y')
  16 - access("FTYPES"."FILETYPEID"="F"."FILETYPEID" AND "F"."EVENTTYPEID"=15960000)
  18 - access("J"."CONFIGURATIONID"="C"."CONFIGURATIONID" AND "J"."JOBID"="F"."JOBID" AND
              "J"."PRODUCTION"="BVIEW"."PRODUCTION")


Statistics
----------------------------------------------------------
          8  recursive calls
          0  db block gets
      94527  consistent gets
          0  physical reads
          0  redo size
        836  bytes sent via SQL*Net to client
        476  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Figure 7.12.: Execution plan for a given query - At the top of the picture we can see the time required to execute the query 3. The join order of the tables are shown in the middle of the picture. Step 6 produces a CARTESIAN product which result is the high number of consistent gets as shown at the bottom of the picture.

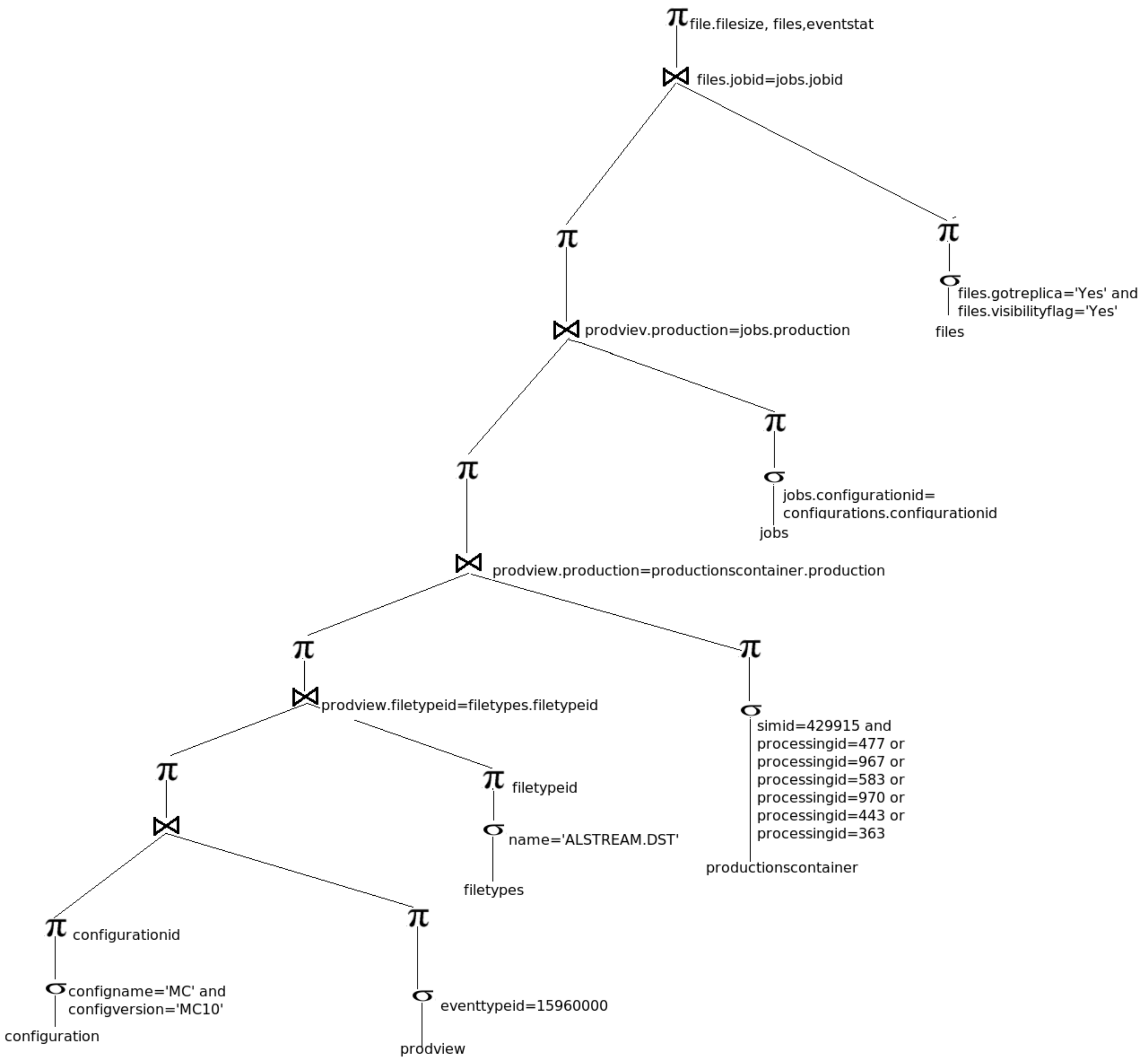

π file.filesize, files,eventstat
⋈ files.jobid=jobs.jobid
π
σ files.gotreplica='Yes' and files.visibilityflag='Yes'
files
π
⋈ prodviev.production=jobs.production
π
σ jobs.configurationid= configurations.configurationid
jobs
π
⋈ prodview.production=productionscontainer.production
π
σ simid=429915 and processingid=477 or processingid=967 or processingid=583 or processingid=970 or processingid=443 or processingid=363
productionscontainer
π
⋈ prodview.filetypeid=filetypes.filetypeid
π filetypeid
σ name='ALSTREAM.DST'
filetypes
π
⋈
π configurationid
σ configname='MC' and configversion='MC10'
configuration
π
σ eventtypeid=15960000
prodview

Figure 7.13.: Query tree for a given query - This figure shows the equivalent query tree to the execution plan 7.12. At the bottom of the picture you can see the configuration and prodview tables. The result of the join of these table is a CARTESIAN product.

kept the first query and we made a new query by combining queries 2 and 3. The optimized query is the following:

```
SELECT count(*), SUM(f.EventStat), SUM(f.FILESIZE), SUM(f.luminosity),
  SUM(f.instLuminosity) FROM jobs j, files f WHERE
j.configurationid=(select c.configurationid from configurations c  where
c.configname='MC' and
c.configversion='MC10') and
j.production IN
(SELECT /*+ NOPARALLEL(bview) */ bview.production FROM productionscontainer prod,
   prodview bview, filetypes ftypes WHERE
   prod.processingid IN (SELECT v.id FROM
   (SELECT distinct SYS_CONNECT_BY_PATH(name, '/') Path, id ID
  FROM processing v START WITH id in (SELECT distinct id from processing
  WHERE name='Sim01') CONNECT BY NOCYCLE PRIOR  id=parentid) v
           WHERE v.path='/Sim01/Trig0x002e002aFlagged/Reco08/Stripping12Flagged') and
                  prod.simid=429915 and prod.simid is not null and
                  bview.production=prod.production and
                  bview.eventtypeid=15960000 and
                  bview.filetypeid=ftypes.filetypeid and
                  ftypes.name='ALLSTREAMS.DST'
              )and
j.jobid=f.jobid and
f.gotreplica='Yes' and
f.visibilityflag='Y' and
f.eventtypeid= 15960000 and
f.filetypeid in (SELECT ft.filetypeid FROM filetypes ft WHERE ft.name='ALLSTREAMS.DST');
```

After this change the new query takes 0.08 seconds as shown on top of figure 7.14. In addition the number of consistent reads from the database buffer (consistent gets) is decreased from 94,527 to 20,111, and the CARTESIAN product no longer exists.

7.10.1. Response time and Throughput

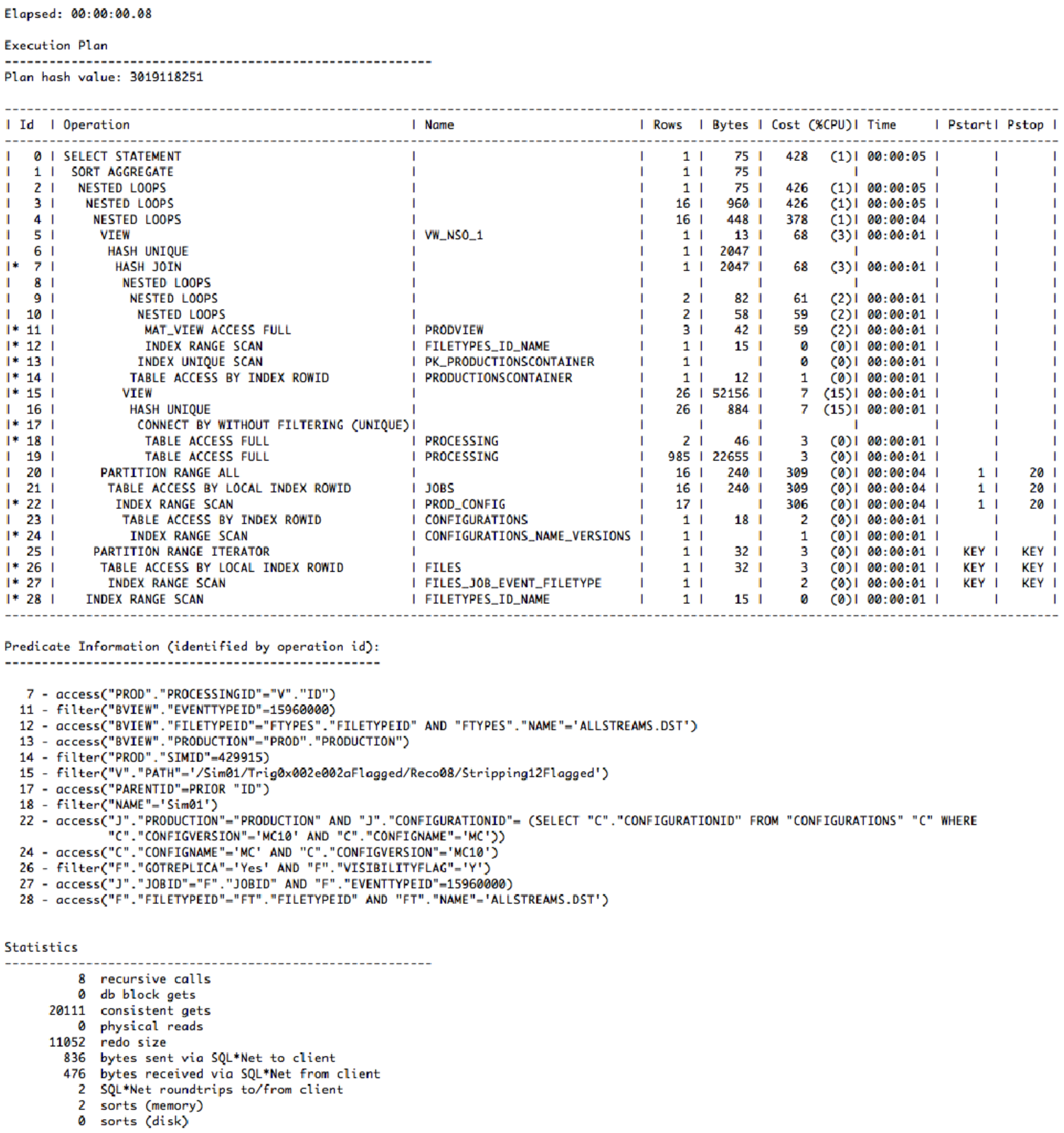
As the LHCb Bookkeeping system has to serve thousands of queries per minute, the performance and capability of the system must be measured. We distinguish between the **Response time** and **Throughput**.

The Response time is the time spent during the execution of a task. For example, it is the time required for the system to show the contents of a folder after the user clicks on it. The Response time consists of the Latency and Processing time. **Latency** is the amount of time delay experienced in a system. This delay can be influenced by the network, high workload, disk or memory etc. The **Processing Time** is the amount of time required by a system to process a given request. In depends on the Latency and Processing Time:

$$Latency + ProcessingTime = ResponseTime \tag{7.1}$$

Throughput is the number of executions in a specified time interval. For example during one second how many times the users can open a folder in the Bookkeeping Tree.

The LHCb Bookkeeping system is widely used by the users and Production System. The Response time is more relevant for the users but the throughput is more important for the



```
Elapsed: 00:00:00.08

Execution Plan
----------------------------------------------------------
Plan hash value: 3019118251

------------------------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                                    | Name                          | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
------------------------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                             |                               |     1 |    75 |   428   (1)| 00:00:05 |       |       |
|   1 |  SORT AGGREGATE                              |                               |     1 |    75 |            |          |       |       |
|   2 |   NESTED LOOPS                               |                               |     1 |    75 |   426   (1)| 00:00:05 |       |       |
|   3 |    NESTED LOOPS                              |                               |    16 |   960 |   426   (1)| 00:00:05 |       |       |
|   4 |     NESTED LOOPS                             |                               |    16 |   448 |   378   (1)| 00:00:04 |       |       |
|   5 |      VIEW                                    | VW_NSO_1                      |     1 |    13 |    68   (3)| 00:00:01 |       |       |
|   6 |       HASH UNIQUE                            |                               |     1 |  2047 |            |          |       |       |
|*  7 |        HASH JOIN                             |                               |     1 |  2047 |    68   (3)| 00:00:01 |       |       |
|   8 |         NESTED LOOPS                         |                               |       |       |            |          |       |       |
|   9 |          NESTED LOOPS                        |                               |     2 |    82 |    61   (2)| 00:00:01 |       |       |
|  10 |           NESTED LOOPS                       |                               |     2 |    58 |    59   (2)| 00:00:01 |       |       |
|* 11 |            MAT_VIEW ACCESS FULL              | PRODVIEW                      |     3 |    42 |    59   (2)| 00:00:01 |       |       |
|* 12 |            INDEX RANGE SCAN                  | FILETYPES_ID_NAME             |     1 |    15 |     0   (0)| 00:00:01 |       |       |
|* 13 |           INDEX UNIQUE SCAN                  | PK_PRODUCTIONSCONTAINER       |     1 |       |     0   (0)| 00:00:01 |       |       |
|* 14 |          TABLE ACCESS BY INDEX ROWID         | PRODUCTIONSCONTAINER          |     1 |    12 |     1   (0)| 00:00:01 |       |       |
|* 15 |         VIEW                                 |                               |    26 | 52156 |     7  (15)| 00:00:01 |       |       |
|  16 |          HASH UNIQUE                         |                               |    26 |   884 |     7  (15)| 00:00:01 |       |       |
|* 17 |           CONNECT BY WITHOUT FILTERING (UNIQUE)|                             |       |       |            |          |       |       |
|* 18 |            TABLE ACCESS FULL                 | PROCESSING                    |     2 |    46 |     3   (0)| 00:00:01 |       |       |
|  19 |            TABLE ACCESS FULL                 | PROCESSING                    |   985 | 22655 |     3   (0)| 00:00:01 |       |       |
|  20 |      PARTITION RANGE ALL                     |                               |    16 |   240 |   309   (0)| 00:00:04 |     1 |    20 |
|  21 |       TABLE ACCESS BY LOCAL INDEX ROWID      | JOBS                          |    16 |   240 |   309   (0)| 00:00:04 |     1 |    20 |
|* 22 |        INDEX RANGE SCAN                      | PROD_CONFIG                   |    17 |       |   306   (0)| 00:00:04 |     1 |    20 |
|  23 |         TABLE ACCESS BY INDEX ROWID          | CONFIGURATIONS                |     1 |    18 |     2   (0)| 00:00:01 |       |       |
|* 24 |          INDEX RANGE SCAN                    | CONFIGURATIONS_NAME_VERSIONS  |     1 |       |     1   (0)| 00:00:01 |       |       |
|  25 |     PARTITION RANGE ITERATOR                 |                               |     1 |    32 |     3   (0)| 00:00:01 |   KEY |   KEY |
|* 26 |      TABLE ACCESS BY LOCAL INDEX ROWID       | FILES                         |     1 |    32 |     3   (0)| 00:00:01 |   KEY |   KEY |
|* 27 |       INDEX RANGE SCAN                       | FILES_JOB_EVENT_FILETYPE      |     1 |       |     2   (0)| 00:00:01 |   KEY |   KEY |
|* 28 |    INDEX RANGE SCAN                          | FILETYPES_ID_NAME             |     1 |    15 |     0   (0)| 00:00:01 |       |       |
------------------------------------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   7 - access("PROD"."PROCESSINGID"="V"."ID")
  11 - filter("BVIEW"."EVENTTYPEID"=15960000)
  12 - access("BVIEW"."FILETYPEID"="FTYPES"."FILETYPEID" AND "FTYPES"."NAME"='ALLSTREAMS.DST')
  13 - access("BVIEW"."PRODUCTION"="PROD"."PRODUCTION")
  14 - filter("PROD"."SIMID"=429915)
  15 - filter("V"."PATH"='/Sim01/Trig0x002e002aFlagged/Reco08/Stripping12Flagged')
  17 - access("PARENTID"=PRIOR "ID")
  18 - filter("NAME"='Sim01')
  22 - access("J"."PRODUCTION"="PRODUCTION" AND "J"."CONFIGURATIONID"= (SELECT "C"."CONFIGURATIONID" FROM "CONFIGURATIONS" "C" WHERE
              "C"."CONFIGVERSION"='MC10' AND "C"."CONFIGNAME"='MC'))
  24 - access("C"."CONFIGNAME"='MC' AND "C"."CONFIGVERSION"='MC10')
  26 - filter("F"."GOTREPLICA"='Yes' AND "F"."VISIBILITYFLAG"='Y')
  27 - access("J"."JOBID"="F"."JOBID" AND "F"."EVENTTYPEID"=15960000)
  28 - access("F"."FILETYPEID"="FT"."FILETYPEID" AND "FT"."NAME"='ALLSTREAMS.DST')


Statistics
----------------------------------------------------------
          8  recursive calls
          0  db block gets
      20111  consistent gets
          0  physical reads
      11052  redo size
        836  bytes sent via SQL*Net to client
        476  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
```

Figure 7.14.: Execution plan for a give query - Figure shows the execution plan of the query 4. The elapsed time is lower than the previous query execution plan. In addition, the CARTESIAN product is not longer exists and the number of consistent gets are reduced.

Table Name	Number of rows
files	112,081,699
jobs	50,046,917
productionscontainer	19,511
prodview	17,876
processing	999
configurations	202
filetypes	99

Table 7.5.: The tables are used by the queries. The 'Table Name' column contains the name of the tables, the 'Number of rows' column contains the number of rows in each table.

Production system. To measure the Response time and the Throughput we used the **Multi-Mechanize** open source framework and we ran simultaneously different tasks executed by the Bookkeeping Service.

7.10.2. Testing Oracle 11g performances using Multi-Mechanize

We used 261 different datasets (261 different queries) to measure the response time and throughput, because only one dataset might not be typical of the system behaviour. Our requirement is the maximum response time must be less than 20 second and the throughput must be more than 10 transaction per second, because the users cannot wait minutes to get the answer for a given request. We used these 261 different datasets randomly in order to avoid the database cache. The queries were performed on seven tables which are shown in Table 7.5.

The database configuration was the following: 2 node cluster (itrac1011, itrac1021). Each node had 4 x QuadCore Intel E5630 @ 2.53GHz CPU, 48 GB of RAM and 36 SATA 7200 RPM 2 TB disks configured with Oracle ASM as RAID10[240, 241].

We used two Bookkeeping servers in parallel which ran a dedicated machine with the following configuration: 2 x QuadCore Intel L5420 @ 2.50 GHz CPU, 16 GB of RAM. The Bookkeeping clients (test script) ran in a 4 x QuadCore Intel L5520 @ 2.27 GHz CPU, 48 GB RAM machine.

We define the following terms:

- **transactions** the number of the executed queries during a specified period.
- **run time** the period during which our test script is executing.

- **time-series interval** (or interval data): time series interval for the results analysis. It is a number which gives the length of the intervals in second.
- **threads**: number of treads which will be executed parallel (or number of virtual users).
- **rampup**: is a number in seconds (60 second is used in our experiments). It is the time required to start all threads.
- **throughput** the number of transactions during a specified period.
- **response time** is the time spent during the execution of tasks. It include the client connection to the server, and database execution time. The following metrics are used: *average*, *min*, *max*, *stdev* and *percentiles* (80th, 90th and 95th).

We implemented a script called **query.py** which chooses a dataset randomly and makes a request to the Bookkeeping system with this dataset. The script was executed in parallel to query the Bookkeeping system and continuously by a certain number of threads (virtual user) using the Multi- Mechanise framework.

100 users

We run simultaneously 100 threads (which equals 100 users) by executing **query.py** continuously during 7200 seconds and calculated the response time, and transactions as a function of time. The results are presented in Figures 7.15, 7.16 and 7.17. Figure 7.15 shows the average, 80 and 90 percentile response time. Figure 7.16 shows the scatter plot of the response time in a given period. The throughput of the system shown in Figure 7.17 is roughly 25 queries per second.

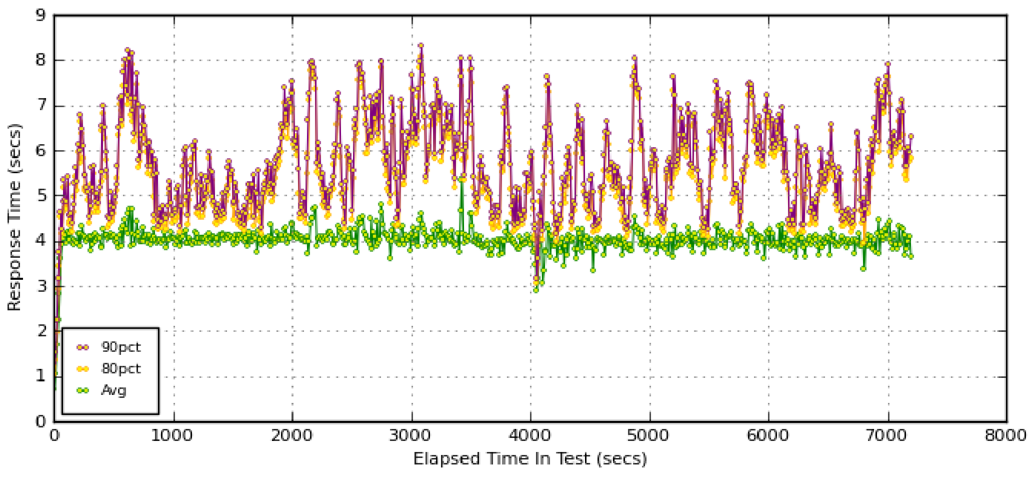


Figure 7.15.: The average and the percentiles(80th, 90th) response time plot for the queries for Experiment 1

Taking into account the results of this experiment the system was not overloaded. In addition the response time and the throughput were stable during the test.

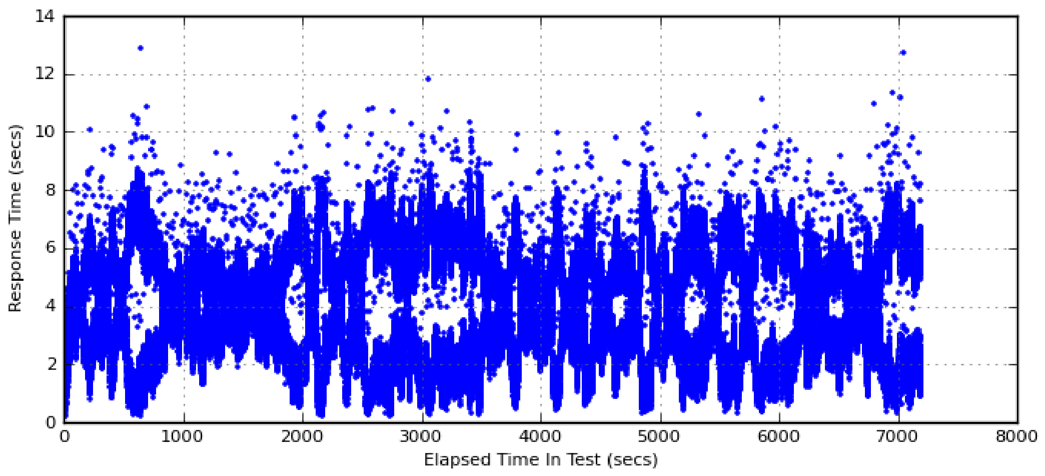


Figure 7.16.: Response time of the queries for 100 users.

In Table 7.6 we summarize the experiment results giving the minimum and maximum response time, the average and the percentiles (80th, 90th, 95th) the minimum (min), the maximum (max) and the standard deviation (stdev).

The maximum response time is 12.852 shown in Table 7.6. The LHCb Bookkeeping System performed very well compared to our requirement that the maximum response time be less than 20 seconds.

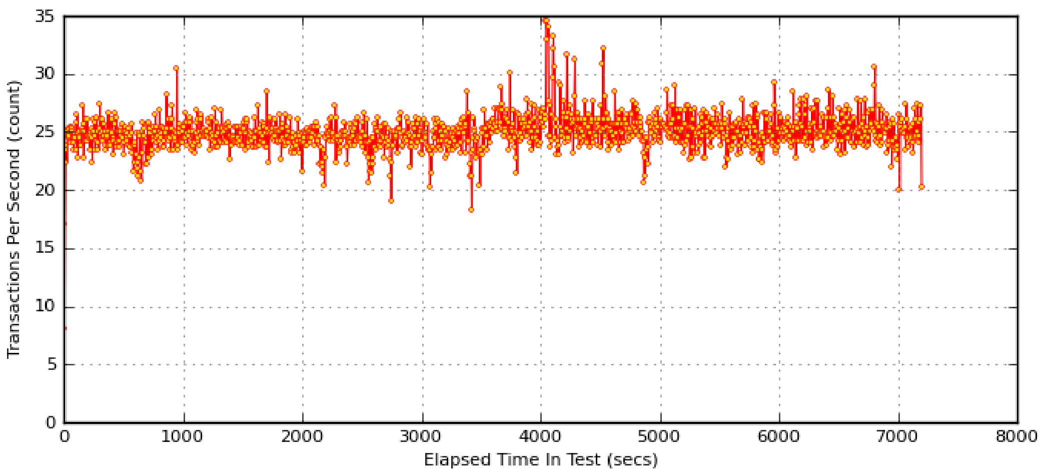


Figure 7.17.: Executed queries per second for 100 users.

200 users

In order to load the system, we increased the number of users (threads) to 200. The results of the test are found in Figure 7.18, 7.19 and 7.20 and the summary found in Table 7.6.

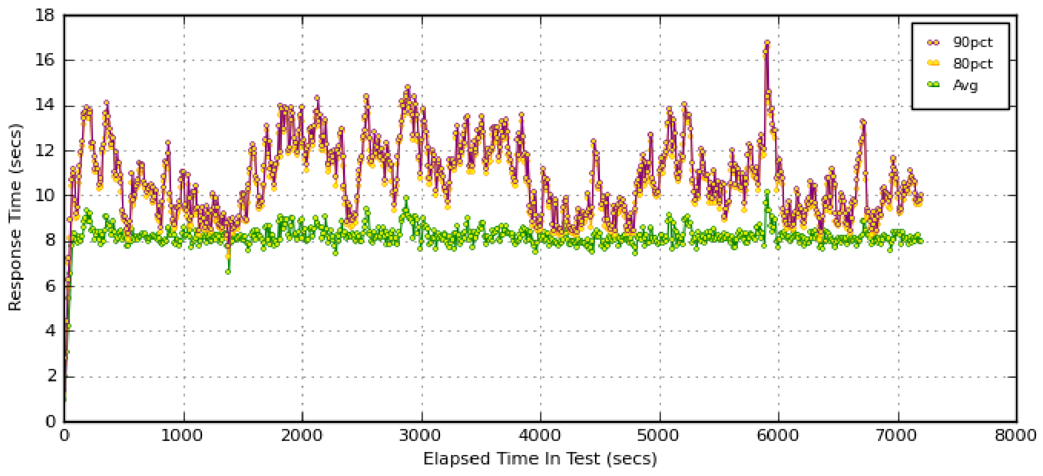


Figure 7.18.: The average and the percentiles(80th, 90th) response time plot for the queries for 200 users.

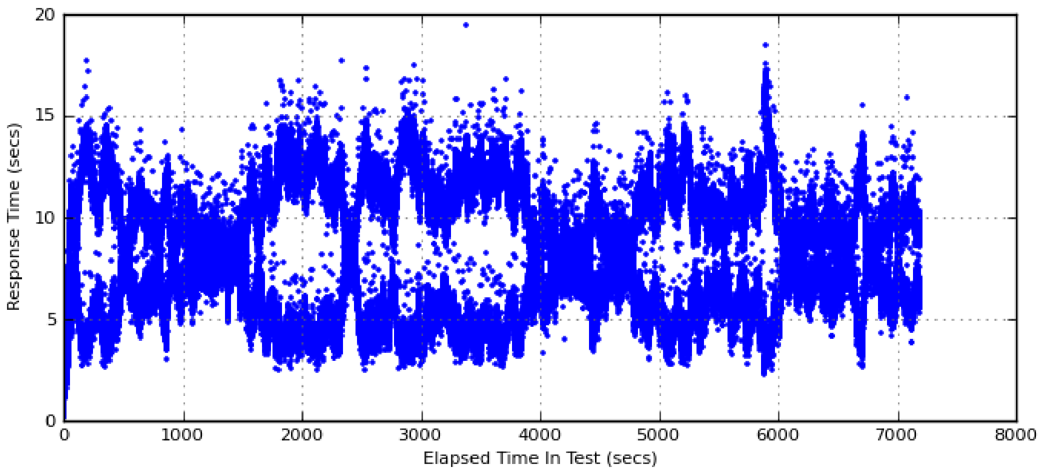


Figure 7.19.: Response time of the queries for 200 users.

By increasing the number of threads the response time is increased while the throughput is not changed. The average response time shown in Figure 7.18 roughly doubled compared to the previous experiment (100 users). Figure 7.20 shows the throughput of the system. The number of executed queries shown in Table 7.6 is lower than the result of the 100 users, which is a result of the high workload. The maximum response time shown in Table 7.6 is 19.4 second which is high but it is less than 20 seconds.

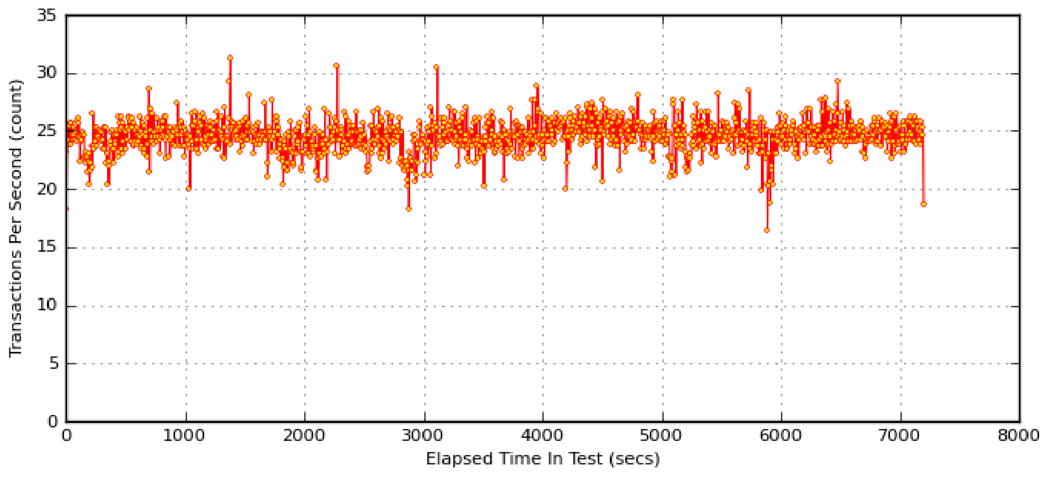


Figure 7.20.: Executed queries per second for 200 users.

500 users

In an attempt to see how the system behaves under very high load we increased the number of users (threads) to 500. Figure 7.21, 7.22 and 7.23 show the results of the test. A summary of these results is found in Table 7.6

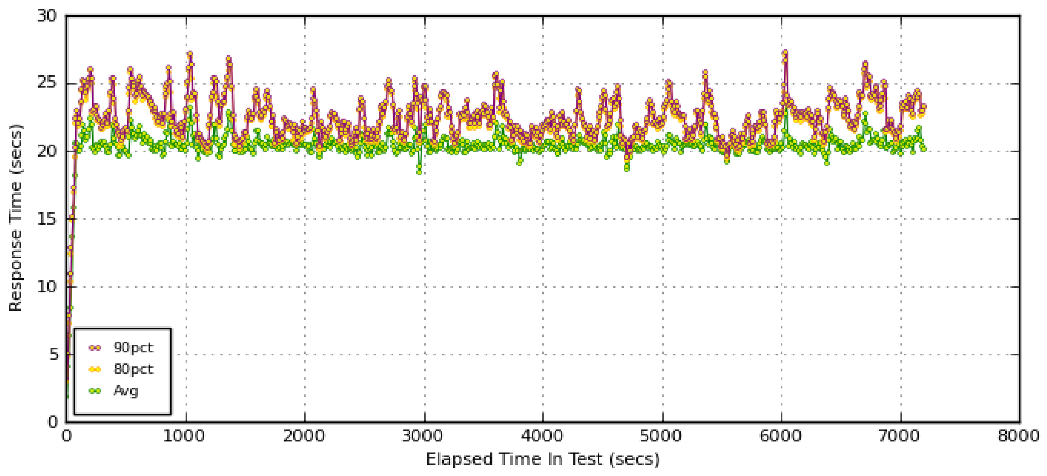


Figure 7.21.: The average and the percentiles(80th, 90th) response time plot for the queries for 500 users.

The average response time shown in Figure 7.21 is roughly five times bigger than first experiment (100 users). The difference of the the average response time and the percentiles (80th, 90th) is small according to this figure, which means the response time of the queries is roughly same. Figure 7.22 shows the response time for a given period. The throughput shown

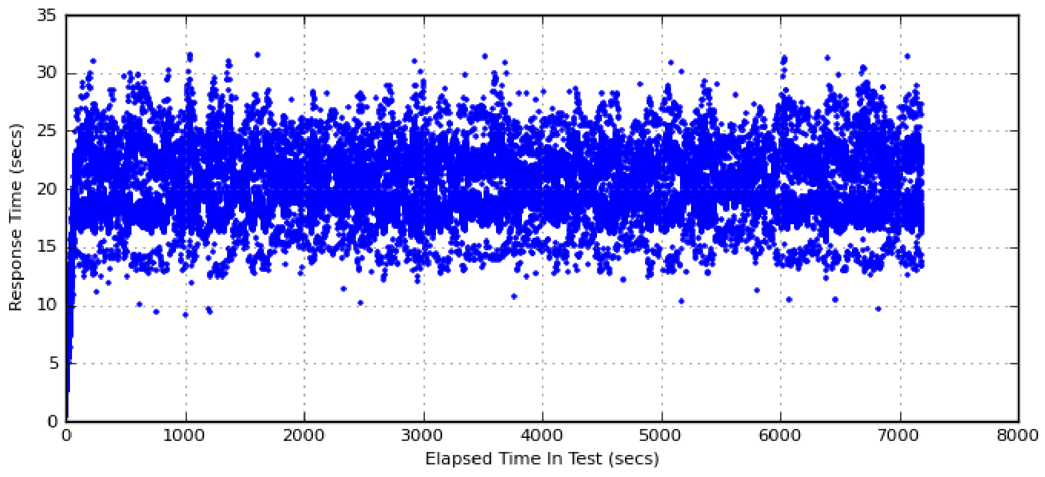


Figure 7.22.: Response time of the queries for 500 users.

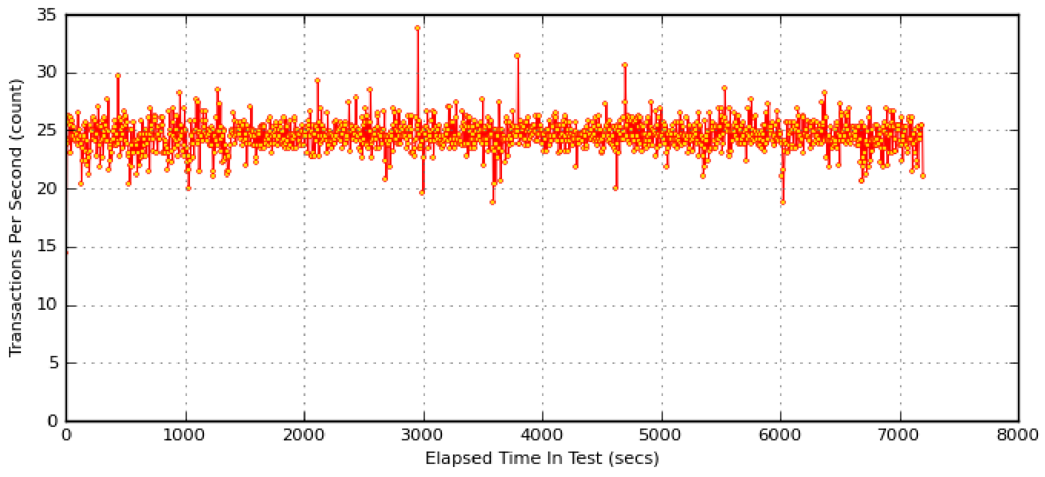


Figure 7.23.: Executed queries per second for 500 users.

in Figure 7.23 is not changed compared to the 100 and 200 users experiments. The maximum response time shown in Table 7.6 is 31.5 seconds which is more than 20 seconds.

In order to understand where that time is spent we calculated the average Latency. The average response time is 20.3 seconds shown in Table 7.6. The average processing time is 0.22 seconds as shown in figure 7.24, which was measured using a python profiler tool introduced in section 7.10, which analysed the log files of the system. The x-axis is the time required to process certain queries, and the y-axis is the number of queries which are executed within a certain time. According to the picture most of the requests take less than 0.5 second. The huge peak at 0.3 seconds is related to the high number of queries that required this time to be finished. The processing time of the Bookkeeping servers also includes the response time of the Oracle database, because the information which is served by the Bookkeeping servers are retrieved

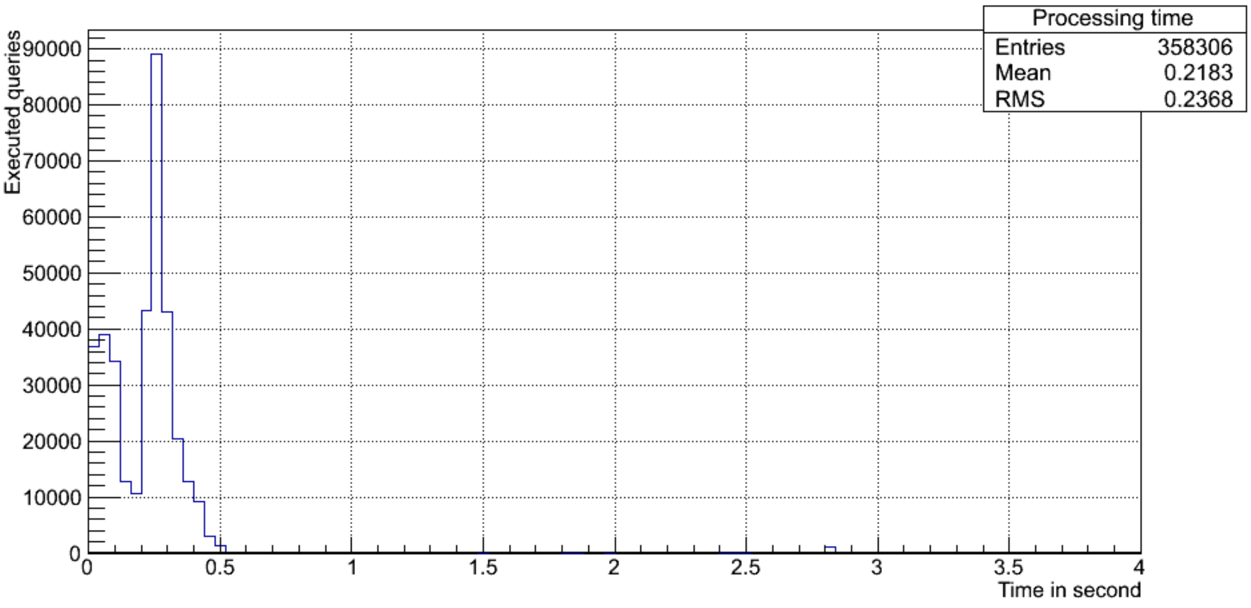


Figure 7.24.: The number of queries which required a certain time to be processed.

Number of users (threads)	100	200	500	1000
Time for Test Run	7,200	7,200	7,200	7,200
Number of transactions	179,000	176,000	177,000	222,000
Minimum response time	0.172	0.113	0.333	0.33
Average Response time	4.00	8.135	20.288	33.0
80pct response time	5.621	10.883	22.410	34.0
90pct response time	6.319	11.950	23.476	35.0
95pct response time	6.883	12.775	24.265	35.9
Maximum response time	12.852	19.431	31.470	47.5
Standard deviation of response time	1.730	2.809	2.565	2.269

Table 7.6.: Summary of the four experiment results

from the database. Consequently, although the database performed very well, the average processing time is 20.3 seconds. The average Latency, calculated using Formula 7.1, is 20.05 seconds. During this time the queries were waiting to be executed by the Bookkeeping system which was under heavy load. To decrease the Latency of the Bookkeeping system another Bookkeeping service would be required in order to share the load.

1000 users

We increased the number of users (threads) to 1000 in order to see how the system behaves beyond 500 users. We expected the average response time will be roughly 40 seconds instead of 33 seconds as shown in Figure 7.25. This difference can be attributed to the network traffic which was lower than the previous experiment (500 users) or the other resources were not busy when this experiment was performed. The scattered plots of the response is shown in Figure 7.26. The response time is less than 50 second and beyond our requirement according to this figure.

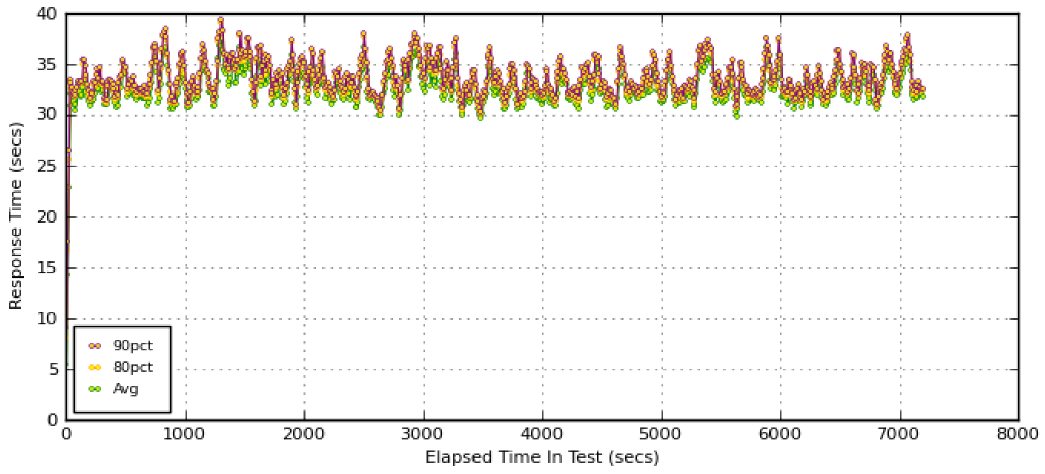


Figure 7.25.: The average and the percentiles(80th, 90th) response time plot for the queries for 1000 users.

Figure 7.27 shows the throughput of the system which is higher than the previous experiments that can be affected to the lower response time.

If the system will be used more than 500 users, It will be required to use another two load balanced services according to the results.

Summary

The performance of the LHCb Bookkeeping System was evaluated using four different tests which simulate the behaviour of the system under high work load. Table 7.7 gives a summary of the tests. According to the results of the experiments the response time increased linearly with the number of threads (virtual users) as shown in figure 7.28. The colors of the curve indicate the number of users. The response time increases linearly until reach the 600 seconds time limit of the Bookkeeping service. This means the queries must be served within 600 seconds, otherwise they will fail due to time out. The throughput remained at roughly 25 queries per second except the last experiment as shown in figures 7.17, 7.20, 7.23 and 7.27. Figure 7.29 shows the number of queries executed by 100, 200, 500 and 1000 users. According

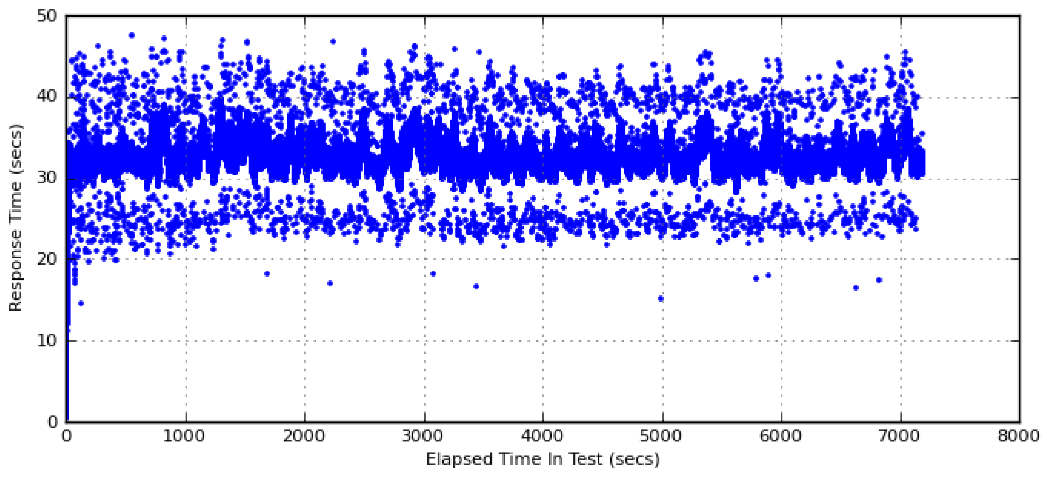


Figure 7.26.: Response time of the queries for 1000 users.

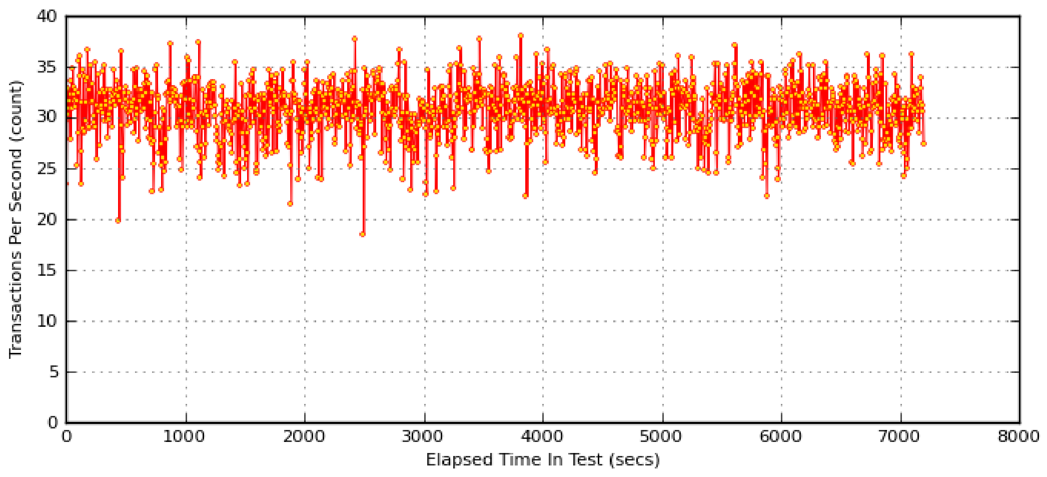
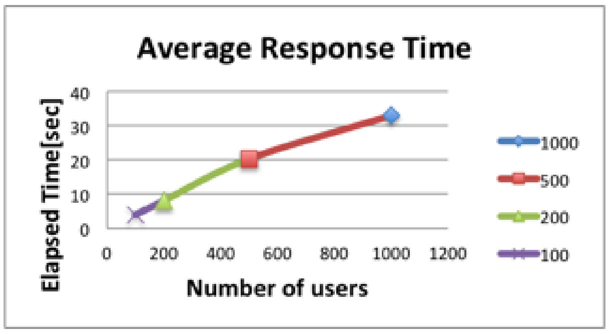


Figure 7.27.: Executed queries per second for 1000 users.



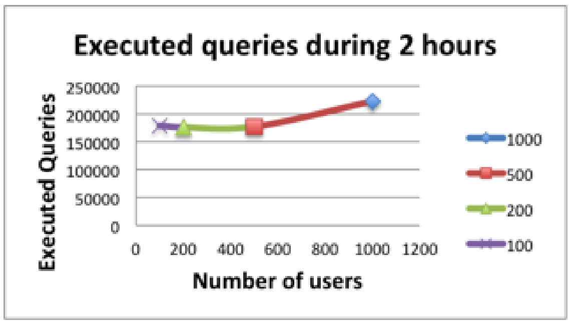


Figure 7.28.: The average response time of the system under heavy load: 100, 200, 500 and 1000 users used the system.

Figure 7.29.: The number of executed queries during 2 hours made by 100, 200, 500 and 1000 users.

to this figure the throughput was stable when the system was used less than 500 users. The throughput was increased linearly by increasing the number of users.

Users	Average Response Time(second)	Average Latency(second)	Throughput
100	4.0	3.7	25
200	8.1	7.8	25
500	20.3	20.0	25
1000	33.00	32.8	32

Table 7.7.: Summary of the Average Response time, Latency and Throughput of the three experiments when two Bookkeeping servers are used; Latency is calculated using 7.1 formula and we assumed the Processing time is roughly 0.22 measured when the system was used by 500 users.

The current state is adequate for LHCb. If the Bookkeeping service were used by 500 users simultaneously, the LHCb Bookkeeping System will not scale. The solution of this problem would be a load-balanced Bookkeeping service that will increase the throughput and decrease the response time.

7.11. Summary

When the datasets are distributed to the the Grid and not centrally managed, scalable and robust metadata management tool is essential. We re-designed the LHCb Bookkeeping system and we introduced a new concept to store and visualise the metadata information. The design of the LHCb Bookkeeping System is based on Feicim architecture. The Feicim framework is used to implement the Graphical User Interface which present the database content in a hierarchical format. It allows the users to browse the database as a Virtual File System. The users can select their datasets for analysis and it also allows to define the access protocol to the datasets. The LHCb Bookkeeping System is more than a metadata catalogue, because it is also used by other DIRAC systems for data processing such as Production Management System, Transformation system, etc. It is widely used by the LHCb experiment. Because of this the performance of the system is very important. We studied different query optimization techniques. In addition we implemented or re-used different tools for monitoring and profiling the system. We executed different tests in order to evaluate the limits of the system. According to the tests the system behaved very well.

8. Conclusion

The ability to browse algorithms, graphically combine algorithms and submit them to run on data on the Grid, is desirable. The architecture of Feicim allows the existing functionalities to be extended. A new component could be implemented which allows the users to define work flow modules and execute them in the Grid. As Feicim is based on loosely coupled components this component could be implemented using the JGraph[246] drawing and graph visualization library. The modules of the work flow consist of different algorithms which will be executed by physics applications such as Davinci or Brunel and each algorithm has different configuration requirements. To connect these components the COmputational MODule Integrator (COMODI)[247] components based framework could be used utilising a component repository[248]. This repository can be used to store the components which can be accessed by different LHCb users. DIRAC/Ganga would then be used to execute the algorithm chain defined in the work flow.

Feicim could be used by other LHC experiments. Due to the design of Feicim each component can be used as a standalone application. Feicim can be reused as a high level GUI on top of various databases. In addition it can be a layer on top of different distributed analysis applications. As Feicim is capable of discovering file content it can be re-used to browse inside an object file which is not produced by the LHCb experiment. Various algorithms can be integrated into Feicim that can be used for Data mining. The tree like format data representation can be applied to various scientific communities with simple modifications of Feicim. The Feicim Tree Traversal Algorithm also can be used to visit different tree data structures. The CMVC can be reused by other applications which have to control a complex GUI. The easy to use plot generation as well as the job submission module can also be re-used. The Feicim framework support loosely coupled components which means it can be used as a framework for a new distributed analysis tool in other scientific communities.

A. Appendix

A.1. Relational Algebra

In this section we introduce the family of relational algebra operations usually associated with the Relational Data Model.

Relational Algebra Operations.

We distinguish two groups of operations in relational algebra, which are the following:

- set operation which are unary operations; the unary operations operate on single relation. We consider two such operations: select and project operations.
- database operations which operate on two tables and are binary operation. We consider Union, Set Difference, Cartesian product and Join.

The Select, Project and Join operations are developed specifically for relational databases. The next sub sections provide a description of these operations.

Selection operation

The Selection operation works on a single relation R and defines a relation that contains only those tuples of R that satisfy a selection condition (predicate)[224]. If we think the relation is a table, the select operations select some of the rows on the table. The selection operation is written as:

$\sigma_F(R)$, where σ denote the SELECT operator, F refer to a selection condition (predicate) which is a Boolean expression specified on the attributes of relation R.

For example: Table A.2 contains the result of a selection operation which is performed in the relation R shown in Table A.1.

A	B	C	D
a	b	c	d
d	g	h	f
f	g	c	a

Table A.1.: Relation R

A	B	C	D
d	g	h	f
f	g	c	a

Table A.2.: $\sigma_{B=g}(R)$

Projection operation

The projection works on single relation R and defines a relation that contains the values of specified attributes and eliminates duplicates[225]. If a relation is a table, the Select operation selects some of the rows from the table and discards other rows, while the Project operation selects certain columns from the table and discards other columns. The Projection operation is written as:

$\prod_{A_1,A_2,A_3,\dots,A_n}(R)$, where $A_i, j = i, n$ are attribute names of the R relation.

For example: Table A.3 contains the result of a projection which is performed on the R (see Table A.1) relation.

B	D
a	d
d	f
f	a

Table A.3.: $\prod_{A,D}(R)$

Union

The Union (denoted as $\cup$) of a given R and S relations (denoted $R \cup S$) is a set of tuples which are in R or S or both[225].

The Union operator can be applied when the 'arity' of two relations R and S are the same. That means all tuples in the result contain the same number of attributes and all attributes are defined from the same domain. When the Union operation is performed the attributes for the operand relations R and S are ignored. Consequently, the attribute names of the result relation can be given arbitrarily.

Table A.5 gives a simple example the union of R (see Table A.1) and S (see Table A.4) relations.

E	F	G	H
e	f	c	d
f	g	c	a

Table A.4.: S relation

a	b	c	d
d	g	h	f
f	g	c	a
e	f	c	d

Table A.5.: $R \cup S$

Set difference

The difference of R and S relations is the set of tuples which in R but not in S. The Set difference is denoted as $R - S$ and requires both sets have the same arity. When this operation is performed the attributes for the operand relations R and S are ignored. Consequently, the attribute names of the result relation can be given arbitrarily.

For example: Table A.6 contains the result of the difference of R (see Table A.1) and S (A.4) relations.

a	b	c	d
d	g	h	f

Table A.6.: $R - S$ relation

Cartesian Product

The Cartesian product (Cross product or Cross Join) of two sets R and S, denoted $R \times S$ defines a relation that is a concatenation of every tuple in relation R with every tuple in relation S [224, 226, 228].

If R relation has n_R tuples and S has n_S tuples, $R \times S$ will have $n_R * n_2$ tuples.

For example: Table A.7 contains the result of a Cartesian product which is performed on the R (see Table A.1) and S (see Table A.16) relations.

Join Operations

The Join operation combines two R and S relations to form a new Q relation. If we consider these relations as tables, the tables should be joined based on a common column. The Join operation has the following types:

A	B	C	D	E	F	G	H
a	b	c	d	e	f	c	d
a	b	c	d	f	g	c	a
d	g	h	f	e	f	c	d
d	g	h	f	f	g	c	a
f	g	c	a	e	f	c	d
f	g	c	a	f	g	c	a

Table A.7.: $R \times S$ relation

1. Theta Join
2. Equi Join
3. Natural Join
4. Semi Join
5. Outer Join

Theta Join

The most general Join operation is the Theta Join. The Theta Join is defined as the result of performing a selection operation using comparison operator Θ on the Cartesian product. The Theta Join of R and S on attributes i and j, is written $R \underset{i\Theta j}{\bowtie} S$, where Θ is an arithmetic comparison operator such as $<, =, <=, >, >=, \neq$ [226]. In the join, only combinations of tuples which satisfy the join condition appear in the result. For example: Given R and S relations which are in Table A.8 and Table A.9, the result of $R \underset{A<E}{\bowtie} S$ is shown in Table A.10.

A	B	C	D
1	2	3	4
5	6	7	8
9	10	11	12

Table A.8.: relation R

E	F	G
2	5	6
10	9	2

Table A.9.: relation S

A	B	C	D	E	F	G
1	2	3	4	2	5	6
1	2	3	4	10	9	2
5	6	7	8	10	9	2
9	10	11	12	10	9	2

Table A.10.: $R \underset{A<E}{\bowtie} S$

Equi Join and Natural Join

A Join, where only the = comparison operator is used, is called an **Equi Join**. The result(s) of the Equi Join, which is denoted as relation E, always have one or more pairs of attributes that have identical values in every tuple. For example, consider the relations S shown in Table A.11 and R shown in Table A.12, the value of the attributes A and D are identical in every tuple of E relation, because of the equality join condition specified on these two attributes.

A	B	C
a	c	b
e	c	b
d	b	a
c	f	g

Table A.11.: R relation

B	C	D
c	b	a
c	b	d
a	f	b

Table A.12.: S relation

A	B	C	B	C	D
a	c	b	c	b	a
d	b	a	c	b	d

Table A.13.: E = $R \propto_{R.A=S.D} S$

The result of Equi Join always contains one or more pairs of attributes that have identical values in every tuple. The **Natural Join** was created to get rid of the second attribute in an Equi Join condition.

Formally, if $A = A_1, A_2, .., A_k$ are attribute names used for R and S, we have $R \bowtie S = \prod_A(\sigma_C(R \times S))$, where the selection σ_C checks equality of all common attributes while the projection eliminates the duplicate common attributes[226].

A	R.B	R.C	S.B	S.C	D
a	c	b	c	b	a
a	c	b	c	b	d
e	c	b	c	b	a
e	c	b	c	b	d

Table A.14.: $\sigma_{R.B=S.B \wedge R.C=S.C}(R \times S)$

A	B	C	D
a	c	b	a
a	c	b	d
e	c	b	a
e	c	b	d

Table A.15.: $\prod_{A,R.B,R.C,D}$ $(\sigma_{R.B=S.B \wedge R.C=S.C}\ (R \times S))$

The following example describes all the steps which have to perform on R (see Table A.11) and S (see Table A.12) relations in order to compute the $R \bowtie S$:

1. Compute $R \times S$ (Table A.16)
2. Using the result of the previous step, compute the selection written as $\sigma_{R.B=S.B \wedge R.C=S.C}(R \times S)$. Table A.14 contains the result of this selection..
3. For each attribute names from the previous result, project out the column S.A, and rename the remaining columns from R.A to A. Table A.15 contains the result of the Natural join.

A	R.B	R.C	S.B	S.C	D
a	c	b	c	b	a
a	c	b	c	b	d
a	c	b	a	f	b
e	c	b	c	b	a
e	c	b	c	b	d
e	c	b	a	f	b
d	b	a	c	b	a
d	b	a	c	b	d
d	b	a	a	f	b
c	f	g	c	b	a
c	f	g	c	b	d
c	f	g	a	f	b

Table A.16.: $S \times R$

Semi join

The **Semi join** of the relation R by relation S is the projection onto the attributes of R of the Natural join of R and S[226]. Formally, the Semi join is written: $R \propto S = \prod_R (R \bowtie S)$.

For example: Consider R (see table A.11) and S (see table A.12) relations. The Semi Join is the projection of the A.15 relation attributes onto A, B, C is shown in the Table A.17. If we use the same relations (R and S), and we change the relations $S \propto R$, the result of the relation is the projection of the A.15 relation attributes onto B, C, D shown in Table A.18.

When we use the Semi join or Natural join then the attributes of the relation become crucial. If the relations R and S contain two many tuples, it will be expensive to calculate the result of the $R \propto S$ or $R \bowtie S$.

A	B	C
a	c	b
e	c	b

Table A.17.: $R \propto S$

B	C	D
c	b	a
c	b	d

Table A.18.: $S \propto R$

Outer join

The Outer join combines two relations R and S by keeping all the tuples in R, or all those in S, or all those in both relations without checking whether or not they have matching tuples in the other relation.

It has tree types:

- *Right Outer join* operates on two relations R and S. The result of the join includes the tuples from S that do not have matching values in the common column of R. For example, consider the relations R shown in Table A.19 and S shown in Table A.20. The Right Outer join is performed on the C attribute in both relations. Table A.21 contains the result $R \mathbin{⟖}_{R.C=S.C} S$. The result contains all tuples from the right relation (in our case is the S relation). If there are any unmatched values, a NULL value id returned.
- *Left Outer join* operates on two relations R and S. The result of the join includes the tuples from R that do not have matching values in the common column of S. For example, consider the relations R shown in Table A.19 and S shown in Table A.20. The Left Outer join is performed on the C attribute in both relations. Table A.22 contains the result $R \mathbin{⟕}_{R.C=S.C} S$. The result contains all the tuples from the left relation (in our case R)
- *Full Outer Join* operates on two relations R and S.The result include the tuples from R that do not have matching value in the common columns of S and the tuples from S that do not have matching value in the common columns of R. For example, consider the relations R shown in Table A.19 and S shown in Table A.20. The Full Outer join is performed on the C attribute in both relations. Table A.23 contains the result $R \mathbin{⟗}_{R.C=S.C} S$. The result contains all tuples from both the R and the S relations.

A	B	C
a	c	b
e	f	g
c	h	i
g	b	a

Table A.19.: R relation

C	D
l	f
a	g
b	h
e	j
c	k
g	l

Table A.20.: S relation

A	B	R.C	S.C	D
NULL	NULL	NULL	l	f
g	b	a	a	g
a	c	b	b	h
NULL	NULL	NULL	e	j
NULL	NULL	NULL	c	k
e	f	g	g	l

Table A.21.: $R ⟖_{R.C=S.C} S$

A	B	R.C	S.C	D
a	c	b	b	h
e	f	g	g	l
c	h	i	NULL	NULL
g	b	a	a	g

Table A.22.: $R ⟕_{R.C=S.C} S$

A	B	R.C	S.C	D
a	c	b	b	h
e	f	g	g	l
c	h	i	NULL	NULL
g	b	a	a	g
NULL	NULL	NULL	l	f
NULL	NULL	NULL	e	j
NULL	NULL	NULL	c	k

Table A.23.: $R ⟗_{R.C=S.C} S$

Bibliography

[1] W. N. Cottingham and D.A. Greenwood, *An Introduction to the Standard Model of Particle Physics, Second Edition*, Cambridge University Press 2007.

[2] LHCb Collaboration et al, *The LHCb Detector at the LHC*, JINST 3 S08005, 2008.

[3] *The four main LHC experiments*, `http://cdsweb.cern.ch/record/40525`

[4] LHCb Collaboration, *LHCb : Technical Proposal*, CERN-LHCC-98-004, 1998. - 170 p.

[5] LHCb Collaboration, *LHCb VELO Technical Design Report*, CERN/LHCC-01-011, 2001.

[6] J.A. Hernando, *The LHCb Trigger*, ACACTA PHYSICA POLONICA, 2007.

[7] R. Antunes Nobrega et al. (LHCb Collaboration), *LHCb Reoptimized Detector Design and Performance*, CERN/LHCC 2003-030 (2003)

[8] J. J. van Hunen , *The LHCb tracking system*, CERN-LHCb-2006-027

[9] LHCb Collaboration, *LHCb Inner Tracker Technical Design Report*, CERN/LHCC 2002-029,2002.

[10] LHCb Collaboration, *LHCb Outer Tracker Technical Design Report*, CERN/LHCC 2001-024, 2001.

[11] LHCb Collaboration, *LHCb RICH Technical Design Report*, CERN/LHCC-00-037, 2000.

[12] LHCb Collaboration, *LHCb Calorimeter Technical Design Report*, CERN/LHCC-00-037, 2000.

[13] LHCb Collaboration, *LHCb Muon Technical Design Report*, CERN/LHCC-01-010, 2001.

[14] LHCb Collaboration, *LHCb Trigger Technical Design Report*, CERN/LHCC-03-031, 2003.

[15] LHCb Collaboration, *LHCb Online System Technical Design Report*, CERN/LHCC-01-040, 2001.

[16] The Metacomputer: One from Many, `http://archive.ncsa.illinois.edu/Cyberia/MetaComp/MetaHistory.html`, NCSA,1995

[17] I. Foster, J.Geisler, J. Nickless, W. Smith and S. Tuecke, *Software Infrastructure for the I-WAY High Performance Distributed Computing Experiment*, IEEE Symposium on High Performance Distributed Computing,1997

[18] I. Foster and C. Kesselman, *Globus: a Metacomputing infrastructure Toolkit*, International Journal of Supercomputer Applications, 1997

[19] *Globus Toolkit*, `http://globus.org/`

[20] I. Foster and C. Kesselman *The Grid : Blueprint for a New Computing Infrastructure*, Morgan Kaufmannm publishers,1998

[21] Gentzsch, W. *Response to Ian Foster's "What is the Grid?*, GridToday (2002)

[22] *Enterprise Grid Alliance*, `http://xml.coverpages.org/ni2004-04-20-a.html`

[23] *Open Grid Forum*, `http://www.ogf.org/`

[24] I. Foster, C. Kesselman and S. Tuecke *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*, Global Grid Forum, 2002

[25] *History of Grid*, `http://www.it-tude.com/historyofgrid.html`

[26] M. Gudgin, M. Hadley,N. Mendelsohn, J.J. Moreau, H.F. Nielsen, A. Karmarkar and Y. Lafon *SOAP Version 1.2, W3C Recommendation*, 2007

[27] E. Christensen, F. Curbera, G. Meredith and S. Weerawarana *Web Services Description Language (WSDL) 1.1*, W3C Note, 15 March 2001, http://www.w3.org/TR/wsdl, 15 March 2001

[28] J. Moreau, A. Ryman, R. Chinnici and S. Weerawarana *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, http://www.w3.org/TR/wsdl20, 26 June 2007

[29] Whitepaper, K. Cza jkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke, *From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring and Evolution, Whitepaper*, 2004

[30] M. Gudgin, M. Hadley,N. Mendelsohn, J.J. Moreau, H.F. Nielsen, A. Karmarkar and Y. Lafon *SOAP Version 1.2, W3C Recommendation*, 2007

[31] Q. Guan *Grid-enabled Urban-CA GIS*

[32] L. Vaquero, et al, *A break in the clouds: towards a cloud definition*, ACM SIGCOMM Computer Communication Review, 39 (2009), 137150

[33] M. Stevens *What Cloud Computing Means to You: Effi ciency, Flexibility, Cost Savings*, 2009

[34] A. Agarwal et al, *Deploying HEP Applications Using Xen and Globus Virtual Workspaces* CHEP08, 2008

[35] *The TeraGrid homepage*, `www.teragrid.org/`

[36] *The Grid-Ireland homepage*, `http://www.grid.ie/`

[37] *The Open science Grid homepage*, `http://www.opensciencegrid.org/`

[38] *The DataGrid project*, `http://eu-datagrid.web.cern.ch/eu-datagrid/`

[39] *Architecture*, `http://en.wikipedia.org/wiki/Systems_architecture`

[40] I. Foster, C. Kesselman and S. Tuecke, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*, International Journal of Supercomputer Applications, 2009

[41] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, *SETI@home: An Experiment in Public-Resource Computing. Communications of the ACM*, 2002

[42] D. P. Anderson, *BOINC: A System for Public-Resource Computing and Storage*, 5th International Workshop on Grid Computing, 2004

[43] Dominique A. Heger, *An Introduction to Grid Technology Vision, Architecture, & Terminology*, Fortuitous Technology, 2006

[44] I. Foster, C. Kesselman and S. Tuecke, *The Anatomy of the Grid*, Intl. J. Supercomputer Applications, 2001

[45] *Grid architecture*, `http://lcg.web.cern.ch/LCG/public/components.htm`

[46] T. Singha , G. Kumarb *Emerging Trends in Networking Environment*, Challenges & Opportunities in Information Technology, 2007

[47] K. Czajkowskiy, S.Fitzgeraldz, I. Fosterx, C. Kesselman, *Grid Information Services for Distributed Resource Sharing*, Proc. 10th IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), IEEE Press, 2002

[48] *OGF GLUE Working Group*, `http://www.ggf.org`

[49] S. Andreozzi et al, *GLUE Specification v. 2.0*, Open Grid Forum, 2009

[50] S. Tucke et al, *Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile*, Internet Engineering Task Force RFC 3820, 2004

[51] *Load Sharing Facility (LSF)*, `www.platform.com/workload-management/high-performance-computing/lp`

[52] *Portable batch system (PBS)*, `http://hpc.sissa.it/pbs/pbs.html`

[53] *Condor*, `http://www.cs.wisc.edu/condor`

[54] A. Sim, A. Shoshani et al, *The Storage Resource Manager Interface Specification Version 2.2*, Open Grid Forum document, 25 September 2009

[55] *BBFTP*, `http://doc.in2p3.fr/bbftp/index.html`

[56] *File Transfer Protocol(FTP)*, `http://en.wikipedia.org/wiki/File_Transfer_Protocol`

[57] I. Mandrichenko, W. Allcock and T.Perelmutov, *GridFTP v2 Protocol Description*, Open Grid Forum document, May 4, 2005

[58] *Grid File System(GFS)*, `https://forge.gridforum.org/projects/gfs-wg`

[59] A. Jagatheesan, *The GGF Grid File System Architecture Workbook*, GGF Grid File System Working Group, April, 2005

[60] *EGEE*, `http://www.eu-egee.org`

[61] *EGI*, `http://www.egi.eu/`

[62] *SOA*, `http://en.wikipedia.org/wiki/Service-oriented_architecture`

[63] *BDII*, `https://twiki.cern.ch/twiki/bin/view/EGEE/InformationSystemOverview`

[64] *R-GMA*, `http://www.r-gma.org/fivemins.html`

[65] *The gridmap file*, `http://gdp.globus.org/gt3-tutorial/multiplehtml/ch15s01.html`

[66] P. Andreettoo et al, *The gLite Workload Management System*, CHEP07, 2007

[67] M. Livny et al, *Distributed Policy Management and Comprehension with Classified Advertisements*, University of Wisconsin (Madison), Technical Report, 2003

[68] C. Marco et al, *The gLite Workload Management System*, CHEP09, 2009

[69] C. Aiftimiei et al, *Job Submission and Management Through Web Service: the experience witht he CREAM service*, CHEP07, 2007

[70] *CEMON home page*, `http://grid.pd.infn.it/cemon`

[71] G. Lo Presti et al, *CASTOR: A Distributed Storage Resource Facility for High Performance Data Processing at CERN*,MSST, 24th IEEE Conferance, 2007

[72] P. Fuhrmann et al, *dCache, storage system for the future*, Europar 2006, Dresden

[73] A. Lana et al, *DPM Status and Next Steps*, CHEP07, 2007

[74] *Disk Pool Manager(DPM)*, `https://svnweb.cern.ch/trac/lcgdm/wiki/Dpm`

[75] *gridFTP*, `https://it-dep-fio-ds.web.cern.ch/it-dep-fio-ds/Documentation/gridftp.asp`

[76] *RFIO*, `http://hikwww2.fzk.de/hik/orga/ges/infiniband/rfioib.html`

[77] *XRootD*, `http://xrootd.slac.stanford.edu/`

[78] *Network File System(NFS)*, `http://tools.ietf.org/html/rfc5661`

[79] R. Mollon et al, *GFAL and LCG-Util*, CHEP07, 2007

[80] *DJRA1.1 Architecture and Planning*, `https://edms.cern.ch/document/476451`

[81] S. Lemaitre et al, *Recent Developments in LFC*, CHEP07, 2007

[82] *Arda Metadata Catalogue(AMGA)*, `http://amga.web.cern.ch/amga/`

[83] *gLite User Guide*, `https://edms.cern.ch/file/722398/gLite-3-UserGuide.html`

[84] B. Koblitz et al, *The AMGA Metadata Service*, Grid Computing, Spinger Science, 2007

[85] *UNOSAT project page*, `http://unosat.web.cern.ch/unosat/`

[86] *Worldwide LHC Computing Grid*, `http://lcg.web.cern.ch/LCG/public/overview.htm`

[87] *The LCG home page*, `http://lcg.web.cern.ch/lcg/mou.htm`

[88] WLCG Accounting Summary, `https://espace.cern.ch/WLCG-document-repository/Accounting/Tier-1/2010/december-10`, December 2010

[89] WLCG Accounting Summary `https://espace.cern.ch/WLCG-document-repository/Accounting/Tier-1/2011/september-11`, September 2011

[90] I. Foster, *Globus Toolkit Version 4: Software for Service-Oriented Systems*, IFIP International Conference on Network and Parallel Computing, 2005

[91] *Globus Tolkit*, `http://www.globus.org/`

[92] R. J. Wilson, *The European DataGrid project*, 2001

[93] *Grid Forum*, `http://www.gridforum.org/News/news.php?id=132`

[94] *Cloud computing*, `http://cloudcomputing.sys-con.com/node/939230`

[95] *Open scieence grid web site*, `http://www.opensciencegrid.org/`

[96] *Nordic Data Grid Facility web site*, `http://www.ndgf.org/`

[97] Science Daily, *World's Biggest Computing Grid Launched*, Oct. 3, 2008

[98] M. Ellisman, S. Peltier, *Medical Data Federation: The Biomedical Informatics Research Network; Chapter 8; The GRID2 Blueprint for a New Computing Infrastructure,* 2009

[99] A. Rajasekar et al, *Storage Resource Broker Managing Distributed Data in a Grid* , Computer Society of India Journal, 2003

[100] D. B. Keator et al, *Derived Data Storage and Exchange Workflow for Large-Scale Neuroimaging Analyses on the BIRN Grid* , Front Neuroinformatics, 2009

[101] *XML-Based Clinical Experiment Data Exchange Schema*, http://www.xcede.org/XCEDE.html

[102] *Storage Resource Broker*, http://en.wikipedia.org/wiki/Storage_Resource_Broker

[103] *Monogo DB*, http://www.mongodb.org

[104] *Appache couchdb homepage*, http://couchdb.apache.org/index.html

[105] A. S. Szalay, J. Gray, *Scientific Data Federation:The World-Wide Telescope; Chapter 7; The GRID2 Blueprint for a New Computing Infrastructure,* 2009

[106] R. J. Hanisch et al, *Resource Metadata for the Virtual Observatory,* Astronomical Data Analysis Software and Systems XIII, ASP Conference Series, 2004

[107] *NASA homepage*, http://www.nasa.gov/

[108] *MAS homepage*, http://archive.stsci.edu/index.html

[109] *Science Telescope Science Institute Homepage*, textttthttp://www.stsci.edu/portal/

[110] *AstroGrid homepage*, http://www.astrogrid.org/

[111] *Open SkyQuery homepage*, http://openskyquery.net/

[112] *SQL language*, http://en.wikipedia.org/wiki/SQL

[113] J. Austin et al, *Distributed Aircraft Engine Diagnostic; Chapter 5; The GRID2 Blueprint for a New Computing Infrastructure,* 2009

[114] *Data Mining*, http://en.wikipedia.org/wiki/Data_mining

[115] B. Liang, M. Jessop, J. Austin, *A Grid Enabled Visual Tool for Time Series Pattern Matching,* DAME team, Advanced Computer Architectures Group, 2004

[116] R. Davis et al, *Pattern Matching in DAME using Grid Enabled AURA Technology,* DAME team, Advanced Computer Architectures Group, 2003

[117] T. Jackson et al, *Delivering a Grid enab;ed Distributed Aircraft Maintenance Environment (DAME),* DAME team, Advanced Computer Architectures Group

[118] M. Aderholz et al, *Models of Networked Analysis at Regional Centres for LHC Experiments(MONARC),* Phase 2 Report CERN/LCB 2000-001, 2000

[119] LHCb Collaboration, *LHCb computing: Technical Design Report*, CERN/LHCC-05-119, 2005

[120] ALICE Collaboration, *ALICE computing: Technical Design Report*, CERN-LHCC-2005-018, 2005

[121] P. Saiz et al, *AliEn - ALICE environment on the GRID*, Nuclear Instrumemts and Methods, 2003

[122] ATLAS Collaboration, *ATLAS computing: Technical Design Report*, CERN-LHCC-2005-022, 2005

[123] CMS Collaboration, *CMS computing: Technical Design Report*, CERN-LHCC-2005-023, 2005

[124] Kaushik et al, *Panda: Production and Distributed Analysis System for ATLAS*, CHEP06, 2006

[125] W. Bacchi et al, *Evolution of BOSS, a tool for job submission and tracking*, CHEP06, 2006

[126] D. Spiga, *CMS workload management*, Nuclear Physics B, 2007

[127] D. Spiga et al, *CRAB: the CMS distributed analysis tool development and design*, Nuclear Physics B, 2008

[128] *Parallel ROOT Facility*, `http://root.cern.ch/drupal/content/proof`

[129] *MONALISA Repository for ALICE*, `http://alimonitor.cern.ch/reports/`

[130] D. L. Adams et al, *DIAL: Distributed Interactive Analysis of Large Datasets*, CHEP06, 2006

[131] T. Maeno, *PanDA: Distributed Production and Distributed Analysis System for ATLAS*, CHEP07, 2007

[132] G. Negri et al, *Distributed Computing in ATLAS*, Porcessing of Sience conferance, 2008

[133] J. Elmsheuser et al, *Distributed Analysis in ATLAS using GANGA*, CHEP09, 2009

[134] D. Spiga, *CMS workload management*, 2007

[135] CMS Collaboration, *Distributed Analysis in CMS*, J Grid Computing, 2010

[136] D. Evans et al, *The CMS Monte Carlo Production System: Development and Design*, Nuclear Physics B, 2008

[137] S. Bagnasco et al, *AliEn: ALICE Environment on the GRID*, CHEP08, 2008

[138] P. Buncic et al,*The architecture of the AliEn system*, CHEP04, 2004

[139] S. Albrand et al, *ATLAS metadata interface(AMI) and ATLAS metadaat catalogs*, CHEP04, 2004

[140] J. Cranshaw et al, *Integration of the ATLAS Tag Database with Data Management and Analysis Components*, CHEP07, 2007

[141] S. Albrand et al, *The ATLAS metadata interface*, CHEP09, 2009

[142] S. Albrand et al, *The ATLAS metadata interface*, CHEP07, 2007

[143] A. Afaq et al, *The CMS Dataset Bookkeeping Service*, CHEP07, 2008

[144] *Java servlet*, `http://en.wikipedia.org/wiki/Java_Servlet`

[145] V. Kuznetsov, D. Riley, *The CMS DBS Query Language*, CHEP09, 2010

[146] A. Dolgert, et al, *A multi-dimensional view on information retrieval of CMS data*, CHEP07, 2008

[147] V. Kuznetsov, et al, *The CMS Data Aggregation System*, ICCS 2010, 2010

[148] LHCb software architecture group., *GAUDI LHCb Data Processing Applications Framework*, Architecture Design Document LHCb 98-064, CERN, 1998.

[149] *GLAST experiment*, `http://fermi.gsfc.nasa.gov/`

[150] E. Radicioni, *Results from the HARP Experiment*, TAUP 2007.

[151] *DayaBay experiment*, `http://dayabay.bnl.gov/`

[152] *MINERvA experiment*, `http://minerva.fnal.gov/`

[153] G. Barrand et al, *GAUDI - A Software Architecture and Framework for building HEP Data Processing Applications*, Computer Physics Communications, Volume 140, Issue 1-2, p. 45-55.

[154] M. Cattaneo et al, *The new LHCb Event Data Model*, LHCb-2001-142,2001.

[155] S. Miglioranzi et al, *The LHCb Simulation Application, Gauss: Design,Evolution and Experience*, CHEP10, CERN-LHCb-PROC-2011-006

[156] I. Belyaev et al, *Simulation Application for the LHCb Experiment*, CHEP03, 2003

[157] *Gauss*, `http://lhcb-release-area.web.cern.ch/LHCb-release-area/DOC/gauss/`

[158] *Pythia6*, `http://projects.hepforge.org/pythia6/`

[159] *Pythia8*, `http://home.thep.lu.se/ torbjorn/pythiaaux/present.html`

[160] *Hijing*, `http://www-nsdth.lbl.gov/ xnwang/hijing/`

[161] *EvtGen*, `http://lhcb-release-area.web.cern.ch/LHCb-release-area/DOC/gauss/generator/evtgen.php`

[162] S. Agostinelli et al, *Geant4 - A Simulation Toolkit*, Nuclear Instruments and Methods A, 2003

[163] *Boole*, `http://lhcb-release-area.web.cern.ch/LHCb-release-area/DOC/boole/`

[164] R. Graciani, *LHCb Computing Resources:2011 reassessment, 2012 request and 2013 forecast*, LHCb Public Note, 2011

[165] *EGI accounting portal*, `https://www4.egee.cesga.es/accounting/reports.html`

[166] S. Paterson et al, *Performance of combined production and analysis WMS in DIRAC*, CHEP09

[167] A. Tsaregorodtsev et al, *DIRAC: A Community Grid Solution*, CHEP07

[168] A. Tsaregorodtsev et al, *DIRAC, Distributed Infrastructure with Remote Agent Control*, CHEP03

[169] P. Buncic et al, *Architectural Roadmap Towards Distributed Analysis*, Technical report, CERN-LCG-2003-033, 2003

[170] J. Closier et al, *Results of the LHCb Data Challenge 2004*, CHEP04

[171] A. Tsaregorodtsev et al, *DIRAC: Workload Management System*, CHEP04

[172] J. P. Baud et al, *DIRAC Review Report*, LHCb -2006-04, 2006

[173] R. Nandakumar et al, *The LHCb Computing Data Challenge DC06*, CHEP07

[174] J. Closier, S.K. Paterson *Performance of Combined Production And Analysis WMS in DIRAC*, CHEP09

[175] *CREATIS*, `http://www.creatis.insa-lyon.fr/site`

[176] *Virtual Imaging Platform*, `http://www.creatis.insa-lyon.fr/vip/node/2`

[177] *International Linear Collider*, `http://www.linearcollider.org/about/The-people/The-ILC-community`

[178] *Linear Collider Detector*, `http://lcd.web.cern.ch/LCD`

[179] J. Belle II collaboration, *Belle II technical design report*, KEK report 2010

[180] *Gisela project*, `http://www.gisela-grid.eu`

[181] *Cherenkov Telescope Array project*, `http://www.cta-observatory.org`

[182] *BES collaboration*, `http://bes.ihep.ac.cn`

[183] *SuperB collaboration*, `http://web.infn.it/superb`

[184] R. Graciani Diaz, *LHCb: DIRAC Framework for Distributed Computing*, CHEP07

[185] R. Graciani Diaz et al, *Belle-DIRAC Setup for Using Amazon Elastic Compute Cloud*, Volume 9,Journal of Grid Computing, 2011

[186] A.C. Smith et al, *DIRAC: Reliable Data Management for LHCb*, CHEP07

[187] A. Tsaregorodtsev et al, *DIRAC3 - the new generation of the LHCb grid software*, CHEP09

[188] Proposed Ganga work plan in context of GridPP, *LHCb/ATLAS GANGA/Gaudi*

Project, 2001

[189] K. Harrison et al, *GANGA: a user-Grid interface for Atlas and LHCb*, CHEP03

[190] A. Soroko et al, *The GANGA user interface for physics analysis on distributed resources*, CHEP04

[191] K. Harrison et al, *Ganga user interface for job denition and management*, Fourth UK e-Science All-Hands Meeting, 2005

[192] K. Harrison et al, *Ganga: a Grid User Interface*, CHEP06

[193] F. Brochu et al, *Ganga: a tool for computational-task management and easy access to Grid resources*, CHEP09

[194] B.C. Neumann et al, *Kerberos: an authentication service for computer networks*, IEEE, 1994

[195] J.H. Morris et al, *Andrew: a distributed personal computing environment, Commun*, ACM 1986

[196] T. Reenskaug, *The Model-View-Controller (MVC) Its Past and Present*, University of Oslo, 2003

[197] T. E. B. Hudzia, et al, *A Java based architecture of p2p-grid middleware* The 2006 International Conference on Parallel and Distributed Processing Techniques and Applications, 2006.

[198] Zs. I. Lazar. et a, *Feicim: A browser for data and algorithms*, CHEP07

[199] *Formal grammar*, `http://en.wikipedia.org/wiki/Formal_grammar`

[200] *Design patterns*, `http://en.wikipedia.org/wiki/Design_pattern_(computer_science)`

[201] *Creational Design patterns*, `http://en.wikipedia.org/wiki/Creational_pattern`

[202] *Factory* , `http://en.wikipedia.org/wiki/Factory_method_pattern`

[203] *Builder* , `http://en.wikipedia.org/wiki/Builder_pattern`

[204] E. Gamma et al, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[205] *Composite* ,`http://en.wikipedia.org/wiki/Composite_pattern`

[206] *Template Method* , `http://en.wikipedia.org/wiki/Template_method_pattern`

[207] *Chain of Responsibility* , `http://www.oodesign.com/chain-of-responsibility-pattern.html`

[208] *Chain of Responsibility* , `http://en.wikipedia.org/wiki/Chain_of_responsibility_patterna`

[209] *Qt-cross-platform application and UI framework* , `http://qt.nokia.com/`

[210] *Java Swing* , `http://en.wikipedia.org/wiki/Swing_(Java)`

[211] *MFC* , `http://en.wikipedia.org/wiki/Microsoft_Foundation_Class_Library`

[212] *GTK* , `http://www.gtk.org/`

[213] *Spring framework* , `http://www.springsource.org/`

[214] *django* , `https://www.djangoproject.com/`

[215] *Pylons* , `http://pylonsproject.org/`

[216] *ASP.NET MVC home page* , `http://www.asp.net/mvc`

[217] *Struts* , `http://struts.apache.org/`

[218] *Oracle Application Framework* , `http://en.wikipedia.org/wiki/Oracle_Application_Framework`

[219] F. Buschmann. et al, *Pattern-orineted software architecture, A System of Patterns*, John Wiley Sons, 1996

[220] A. Karagkasidis, *Developing GUI Applications:Architectural Patterns Revisited*, EuroPLoP, 2008

[221] J. Cai, et al, *HMVC: The layered pattern for developing strong client tiers,This hierarchical model eases the development of a Java-based client tier*, Java World, 2000

[222] *Extensible Markup Language* , `http://en.wikipedia.org/wiki/XML`

[223] *Bookkeeping working group* , `http://lhcb-comp.web.cern.ch/lhcb-comp/bookkeeping/project.htm`

[224] R. Elmasri, S. B. Navathe, *Fundamentals of Database Systems* , Fourth Edition, Addison Wesley, 2004

[225] S. Sumathi, S. Esakkirajan *Fundamentals of Relational Database Management Systems* , Fourth Edition, Springer, 2007

[226] J.D. Ullman, *Principle of Database and knowledge-base System*, Volume 1, Computer Science Press, 1988

[227] G. Corti et al, *Monte Carlo Event Type Definition Rules*, LHCb Internal Note, Jan 11, 2007

[228] *Certesian product* , `http://en.wikipedia.org/wiki/Cartesian_product`

[229] *SQL* , `http://en.wikipedia.org/wiki/SQL`

[230] L. Hoobs, *Oracle Materialized Views Query Rewrite*, Oracle Corporation, Computer

Science Press, 2005

[231] N. Folkert, *Optimizing Refresh of a Set of Materialized Views*, Oracle Corporation

[232] *POOL project homepage* ,`http://lcgapp.cern.ch/project/persist`

[233] B. Bryla and K. Loney, *Oracle Database 11g DBA Handbook*, Oracle Press, Mc Graw Hill company, DOI: 10.1036/0071496637

[234] *DIRAC Monitoring* , `https://lhcbweb.pic.es/DIRAC/LHCb-Production/lhcb_user/ systems/activitiesMonitoring/ systemPlots? componentName= Bookkeeping/BookkeepingManager`

[235] M Girone, *CERN Database Services for the LHC Computing Grid*, CHEP07, 2007

[236] *Physics Databases Services portal* , `https://phydb.web.cern.ch/phydb/`

[237] *LHCb Bookkeeping Web User Intefcae* , `https://lhcbweb.pic.es/DIRAC/LHCb-Production/lhcb_user/Data/BK/display`

[238] *SQL Plus* , `http://en.wikipedia.org/wiki/SQL*Plus`

[239] *Multi Mechanize* , `http://code.google.com/p/multi-mechanize/`

[240] *Oracle Automatic Storage Management* ,`http://www.oracle-base.com/articles/10g/AutomaticStorageManagement10g.php`

[241] *Redundant Array of Independent Disks* , `http://en.wikipedia.org/wiki/RAID`

[242] R. Brun and F. Rademakers, *ROOT - An object oriented analysis framework.*, Nuc.Inst.Meth. in Phys. Res A, 389 (1997) 81.

[243] D.E. Knuth, *Third edition:The art of computer programming.*, Addison-Wesley, 1997

[244] *Python inspect module*, `http://docs.python.org/library/inspect.html`

[245] *Root qt integration project*, `http://root.bnl.gov/QtRoot/How2Install4Unix.html`

[246] *JGraph home page*, `http://www.jgraph.com`

[247] *COMODI home page*, `http://comodi.phys.ubbcluj.ro/main.htm`

[248] Zs. I. Lazar, L. I Kovacs, Z. Mathe, *COMODI: Architecture for a Component-based Scientific Computing System*, PARA06, UmeAA, Sweden