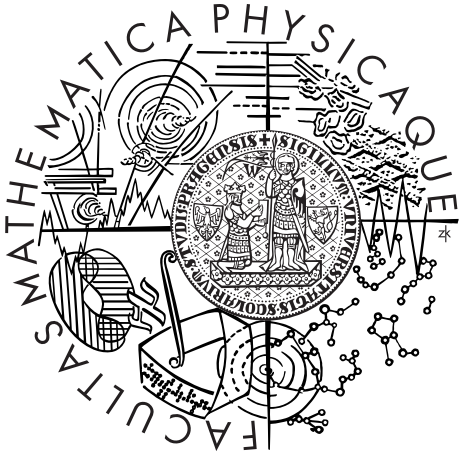
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Martin Adam

Optimization of Processing of Data Files in System DIRAC

Department of Software Engineering

Supervisor of the bachelor thesis: doc. RNDr. Irena Holubová, Ph.D.
Study programme: Computer Science
Specialization: Administration of Computer Systems

Prague 2015

I would like to thank my supervisor, doc. RNDr. Irena Holubová, Ph.D., for her patience and advice when writing this thesis.

I would also want to thank Dr. Andrei Tsaregorodtsev, the DIRAC product manager for finding me a suitable task I could work on during my thesis and for continuous encouragement and help during my work.

Further, I thank RNDr. Jiří Chudoba, Ph.D. for providing me with a suitable testing environment in the computing center of the Institute of Physics of the Czech Academy of Sciences, and RNDr. Dagmar Adamová, CSc. for consultations concerning Grip issues.

Finally my thanks belong to my parents for helping me to enhance the text of this thesis and last but not least I thank my girlfriend Annie for her great patience and support.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Optimalizace zpracování dat v systému DIRAC

Autor: Martin Adam

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: doc. RNDr. Irena Holubová, Ph.D.

Abstrakt: Systém DIRAC je softwarový framework poskytující kompletní řešení pro jednu nebo více uživatelských komunit, které potřebují zajistit přístup k distribuovaným výpočetním zdrojům. V této práci je rozšířen DIRAC File Catalog (DFC) o modul DatasetManager, přidávající funkcionalitu datasetů definovaných dotazem nad metadaty. K vylepšení práce s dotazy v kódu systému je vyvinuta nová třída MetaQuery, která shlukuje obslužné metody a přidává normalizaci a optimalizaci dotazu na vstupu. Jazyk vyjadřující dotazy byl také rozšířen přidáním možnosti používat logické spojky a závorky.

Druhá část práce se zabývá testováním hypotézy, že použití NoSQL databáze jako back-end pro metadatovou část DFC by přineslo vylepšení výkonu vyhledávání. Několik NoSQL databází je otestováno na datech podobných produkčním datům používaných systémem DIRAC. Nejvýkonější z testovaných databází je pak připojena k DFC použitím nového specializovaného rozhraní.

Klíčová slova: Systém DIRAC, NoSQL databáze, efektivní zpracování datových souborů, dotazování nad metadaty

Title: Optimization of Processing of Data Files in System DIRAC

Author: Martin Adam

Department: Department of Software Engineering

Supervisor: doc. RNDr. Irena Holubová, Ph.D.

Abstract: DIRAC is a software framework for distributed computing providing a complete solution to one (or more) user community requiring access to distributed resources. In this thesis the DIRAC File Catalog (DFC) is extended by adding a DatasetManager module, thus adding support for datasets based on metadata queries. To improve the metaquery handling in the code, a new class MetaQuery was implemented that bundles the handling methods and adds normalization and optimization of the user input. The metaquery language was extended enabling logical operators and parenthesis.

In the second part of the thesis the hypothesis that connecting the metadata part of the DIRAC File Catalog to a NoSQL database could improve metaquery performance is evaluated. Several databases are tested and the best performing one is then connected via an interface module to the DFC.

Keywords: System DIRAC, NoSQL databases, efficient processing of data files, metadata querying

Contents

Introduction

The Worldwide LHC Computing Grid

In the past decade the scientific research collaborations, as well as the commercial world, have witnessed the start of a period of explosive data growth. The expressions like "Data Tsunami" or "Big Data" became widespread and entered a standard terminology. The scientific collaborations developed a world leading expertise in building and operating very large scale infrastructures for handling unprecedented and ever growing volumes of data to be processed and analyzed [1]. One of the most important scientific communities, for which these abilities became essential, is the High Energy Physics (HEP) community of the experiments at the Large Hadron Collider (LHC) at CERN[1].

Physicists collide particles accelerated to enormous energies to get at very short scales, which enables them to reveal at the microscopic level the basic ingredients of the matter and their interactions. On the edge of our millennium their ultimate instrument is the LHC, colliding protons or atomic nuclei. In each collision debris of tens of thousands of new particles are created which have to be registered in huge detectors and carefully analyzed for signs of new physical phenomena. The collisions take part at four interaction areas, where the detectors (ATLAS, ALICE, CMS and LHCb) were built. Each experiment collects huge volumes of data that have to be stored, remaining accessible to many groups of physicist scattered all around the globe.

The volumes of collected data are larger than any single computing center within the LHC collaboration could handle, so the concept of distributed data management was conceived. In 2001 the CERN Council approved the start of an international collaborative project that consists of a grid-based computer network infrastructure, the Worldwide LHC Computing Grid (WLCG) [2].

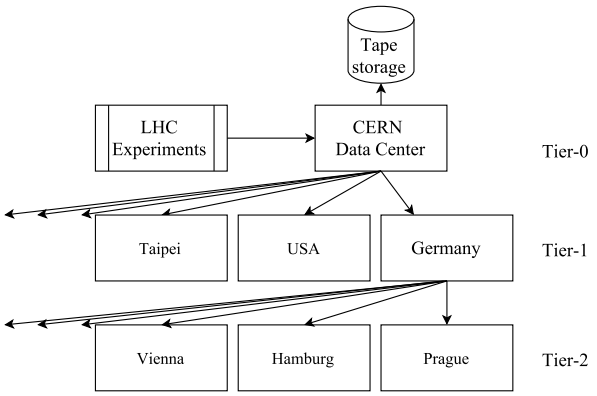


Figure 1: The WLCG Tier-1 centers with CERN Tier-0 in the middle

The WLCG has a hierarchical architecture, where participating sites are categorized – according to the resources and services they provide – into four importance levels called Tiers. Each Tier is represented by a single or distributed computing and storage cluster and yields a specific set of services. The largest center, CERN data center or Tier-0, provides the permanent storage of experimental data and makes the data available for the WLCG processing. Although it contributes less than 20% of the

[1]European Organization for Nuclear Research (name derived from Conseil Européen pour la Recherche Nucléaire) – European research organization that operates the largest particle physics laboratory in the world.

WLCG computing capacity, the role of CERN is unique in keeping one copy of the data from all experiments and for performing the first pass of the data reconstruction. When LHC is not running, Tier-0 provides resources for re-processing of the raw experimental data and eventually for simulation campaigns.

Another copy is passed to one of the Tier-1 centers. Tier-1s are huge computing centers located in Europe, Canada, USA and Taipei. They provide non-stop support for the Grid, store a share of raw data, perform reprocessing and store its output. They are connected to CERN with dedicated high-bandwidth optical-fiber links. Then there are more than 160 Tier-2 centers all around the world. Their role is mainly to run simulation campaigns and end-user analysis. Tier-3 centers are small local computing clusters at universities or research institutes and even individual PCs [3].

Grid Middleware

The operation and functionality of WLCG, as well as other Grid systems, is enabled by specific software packages and protocols, so-called Grid middleware. It manages the basic domains of the Grid functions: job management, data management, security and information services [4]. The term middleware reflects the specific role of this software system: it is a layer between the application area for solving user tasks and the resource area consisting of basic fabric and connectivity layer.

The vast variety of requirements and needs of the user communities from the four LHC experiments is impossible to meet with only one set of middleware components. Consequently, each experiment user group started developing its own set of tools. For example AliEn is a middleware solution made by the ALICE experiment collaboration and DIRAC was developed by the LHCb collaboration. Along with some packages from the WLCG-middleware they include some additional specific packages and provide complete framework for data processing according to the individual experiments' computing models.

DIRAC

The DIRAC[2] middleware was developed to satisfy the needs of not only the LHCb collaboration developing it, but also to enable other smaller experiments to use it as their middleware solution. This was achieved by focusing on modular architecture that enables adding new features or modifying the systems behavior according to individual experiment's needs.

DIRAC is constructed from loosely coupled systems where each system manages one part of its functionality. This thesis focuses on the DIRAC File Catalog, which is a part of the Data Management system. This particular system is responsible for data managing tasks across a wide range of distributed storage elements. It also enables users to quickly find and use their files. It is accomplished by maintaining a directory structure with a similar interface as the UNIX shell and enabling users to define their own metadata and use them to search for files.

[2]The Distributed Infrastructure with Remote Agent Control

Goals of the Thesis

The first task of this thesis is to upgrade the DIRAC File Catalog (DFC) by:

- adding a new module for dataset support, enabling users to bundle their files based on a metadata search (a metaquery) into a single object,
- implementing a class to encapsulate all the methods handling metaquery and to extend its functionality by adding normalization and optimization procedures.

The second task is to test the hypothesis that storing the user defined metadata in a suitable NoSQL database would improve metaquery performance. If the tests prove that hypothesis, the task is to extend the code of DIRAC to incorporate the database in the File Catalog making a prototype, that can be then evaluated by the DIRAC collaboration.

Structure of the Thesis

In Chapter 1 the DIRAC middleware will be introduced with the focus on the data management part and the file catalog. Afterwards DIRAC will be compared to two other middleware solutions in Chapter 2, more specifically ATLAS Distributed Data Management system and ALICE Environment framework.

In the next two chapters our contribution to DIRACs code will be presented. The Chapter 3 is about the new MetaQuery class and the Chapter 4 deals with the Dataset Manager.

In the last part several NoSQL databases will be tested in order to select the one that is the best option for storing file metadata (Chapter 5) and in the Chapter 6 a module created for integrating that database is described.

Finally, the Chapter 7 provides user documentation to all the commands used to interact with the CLI[3] of the DFC used to control any of the parts that were changed by this project. The last Chapter summarizes evaluation of the test results and conclusions.

[3]Command Line Interface

1. DIRAC System

The LHCb Collaboration [6] is running one of the four large experiments at the LHC particle collider at CERN, Geneva. The amount of data produced by the experiment annually is so large that it requires development of a specialized system for the data reconstruction, simulation and analysis. The DIRAC project of the LHCb Collaboration was started to provide such a system.[7] The developers were aiming to create a easy to run system, which would be able to seamlessly utilize the various heterogeneous computing resources available to the LHCb Collaboration, that can be run by only one production manager.

The DIRAC software architecture is based on a set of distributed, collaborating services. Designed to have a light implementation, DIRAC is easy to deploy, configure and maintain on a variety of platforms. Following the paradigm of a Services Oriented Architecture (SOA) [8], DIRAC is lightweight, robust and scalable. One of the primary goals was to support various virtual organization with their specific needs: it supports well isolated plugable modules, where the organizations specific features can be located. It allows to construct grids of up to several thousands processors by integrating diverse resources within its integrated Workload Management System. The DIRAC Data Management components provide access to standard grid storage systems. The File Catalog options include the LCG File Catalog (LFC) as well as a simple DIRAC File Catalog discussed later.

1.1 DIRAC Architecture

DIRAC components can be grouped in to four categories:

Resources

Because every grid middleware has to deal with a large number of different technologies, it needs its own interfaces for each of them. To this end DIRAC has a class of components called Resources, which provides access to the available computing and storage facilities. Computing resources include individual PCs, computer farms, cloud resources and computing grids. Storage resources include storage elements with SRM interface [9] and most of the popular data access protocols (gridftp, (s)ftp, http,...) are integrated as well[1].

[1]DIRAC does not provide its own complex storage element, it includes however a Storage Element service which yields access to disk storage managed by a POSIX compliant file system. This is sufficient for small deployments and development purposes

Services

The DIRAC system is built around a set of loosely coupled services that keep the system status and help to carry out workload and data management tasks. The Services are passive components which are only reacting to the requests of their clients, possibly contacting other services in order to accomplish their tasks. Each service has typically a MySQL [15] database backend to store the state information. The services accept incoming connections from various clients. These can be user interfaces, agents or running jobs.

Agents

Agents are light and easy-to-deploy software components built around a unified framework. They are active components, usually running close to the corresponding services, watching for changes in the service states and reacting accordingly by initiating actions like job submission or result retrieval.

Interfaces

The DIRAC functionality is exposed to the system developers and to the users in a variety of ways:

- the DIRAC programming language is python, so the programming interfaces (APIs) are available in this language,
- for users the DIRAC functionality is available through a command line interface. Some subsystems have specialized shells to work with,
- DIRAC also provides a web interface suitable for monitoring and managing the system behavior.

1.2 DIRAC Framework

DIRAC's logic is built on top of basic services that provide transversal functionality to DIRAC Systems [11]. The set of basic services that form the core framework for DIRAC are:

- **DISET – secure communication protocol** is the DIRAC secure transport layer. DISET was created by enhancing the XML-RPC protocol with a secure layer with GSI[2] compliant authentication mechanism [12]. It takes care of all the communication needs between DIRAC components. When secure connections are not needed communication is done using plain TCP/IP.
- **Configuration System** is built to provide static configuration parameters to all the distributed DIRAC components. Being able to distribute the configuration is critical. When the configuration data is not available, the systems cannot work. For maximum reliability it is organized as a single master service, where all the parameter updates are done, and multiple read-only slave services, which are distributed geographically.

[2]Grid Security Infrastructure

- **Web Portal** is the standard way to present information to visitors. It provides authentication based on user grid credentials and user groups: ordinary users can see their jobs and have minimal interaction with them, administrators can change the global configuration in a graphical interface. All the monitoring and control tools of a DIRAC system are exported through the Web portal, which makes them uniform for users working in different environments and on different platforms.

- **Logging System** is a uniform logging facility for all the DIRAC components.

- **Monitoring System** collects activity reports from all the DIRAC services and some agents. Together with the Logging Service, it provides a complete view of the status of the system for the managers.

Other widely used systems include the Workload Management System, able to manage simultaneously computing tasks for a given user community [13], Request Management System [14] managing simple operations, that are performed asynchronously on behalf of users, and others.

1.3 DIRAC Data Management System

The DIRAC middleware solution covers the whole range of tools to handle data management tasks. The low level tools include many clients for various types of storage. In addition to providing clients of all the common storage systems, DIRAC also implements its own simple Storage Element. Both technologies are exposed to the users by a uniform API. Many experiments use the LCG File Catalog (LFC) through a client API which connects it to the DIRAC system, others are using the DIRACs own File Catalog[3]. The Replica Manager class encapsulates all the basic file management operations: uploading, replication, registration. It masks the diversities of different storage systems and can handle several file catalogs at the same time.

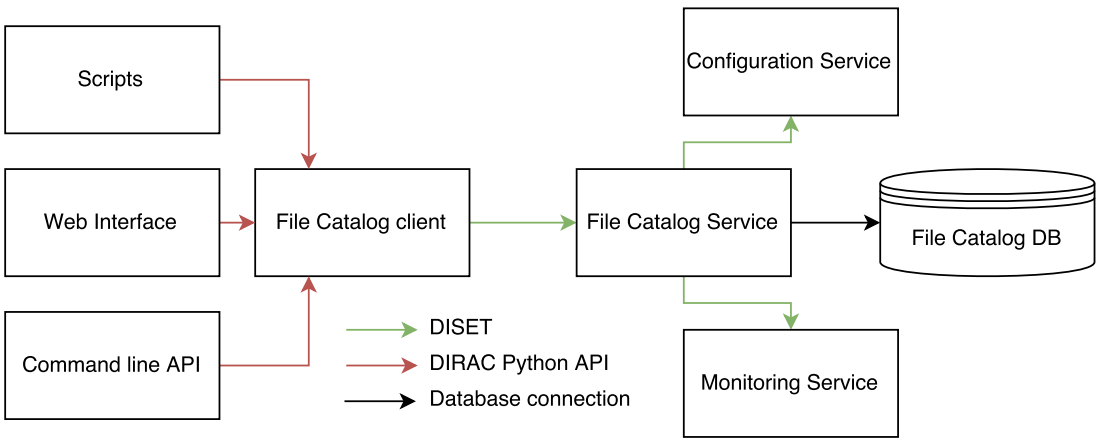


Figure 1.1: DIRAC File Catalog built inside the DISET framework

[3]LHCb, the experiment developing DIRAC, uses a third party file catalog.

The higher level Data Management components include an automatic Data Distribution System and a Data Integrity Checking System. The Data Distribution System allows the definition of destination storages for all the types of data used by a VO[4] before the data actually becomes available. Once the first replicas of the data to be distributed are registered in the File Catalog, the replication requests are formed and sent for execution automatically. The Data Integrity Checking System is checking the integrity of data stored and registered in various storage systems and file catalogs.

1.4 DIRAC File Catalog

Any middleware solution covering the problem of Data Management must have a file catalog enabling the user to find the files he wants to work with, and a replica catalog keeping tracks of file replication.The DIRAC couples these two services into one, introducing the DIRAC File Catalog (DFC) [16]. Implemented using the DISET framework (see Figure 1.1), the access to DFC is done with an efficient secure protocol compliant with the grid security standards. The protocol allows also to define access rules for various DFC functionalities based on the user roles. The clients are getting the service access details from the common Configuration Service. The DFC behavior is monitored using the standard DIRAC Monitoring System and the errors in the service functioning are reported to the Logging System. A MySQL database is used to store all the DFC contents (for the scheme of the database, see Figure 1.3).

Following the plug-able module schema, there are many file catalog modules altering its behavior. For example the data files in DFC are stored in a hierarchical directory structure and its implementation depends on the current module plugged in the `dtree` slot in the `FileCatalogDB` class instance inside the `FileCatalogHandler` (see Figure 1.2).

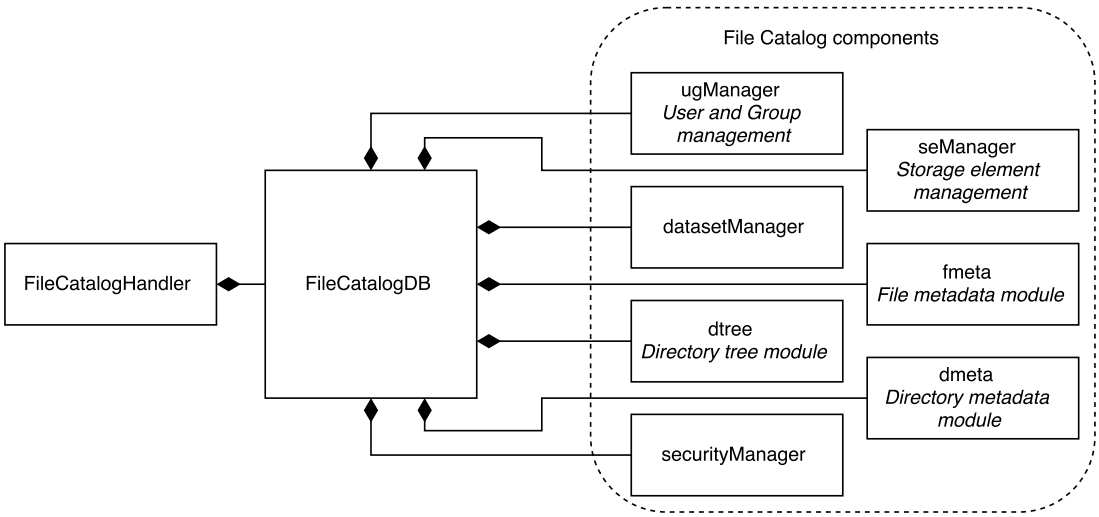


Figure 1.2: Class diagram inside the File Catalog service. The clients requests are arriving at the FileCatalogHandler class, the FileCatalogDB class then issues requests for MySQL database, sometimes using one or more of its' modules

[4]Virtual Organization

1.4.1 DIRAC Replica Catalog

When managing distributed storage systems, one of the major problems is how to translate the logical file name (LFN) into a URL of the file replicas, which in the DIRAC system is a task for the Replica Catalog. The replica information can be stored in 2 different ways. In the first case, the full Physical File Names (PFN) are stored as complete access URLs. In the second case, the DFC exploits a convention that the PFNs are containing the corresponding LFNs as its trailing part. This helps to establish correspondence between the physical files and entries in the catalogs when checking the data integrity. There is also no need to store the full PFN in the catalog, when the information about the wanted storage element (SE) is available. In such a case the PFN can be constructed on the fly and when the SE description changes, e.g. its access point, it is enough to change the SE description in the DIRAC Configuration System to continue receiving correct PFNs. Support for managing various user and group usage quotas is also implemented. Another feature supported by the Catalog based on the experience in the HEP experiments is to store and provide information about ancestor-descendant relations between files.

1.4.2 Metadata Catalog

Each virtual organization has varying needs in terms of file metadata, DIRAC takes this chance and attracts more users by enabling them to define their own. The Metadata Catalog keeps track of these arbitrary user metadata associated with his files and directories as key-value pairs. The catalog administrators can declare certain keys. In the current implementation whenever an index is being declared, another table in the MySQL database is created to store the values of the respective metadata. The indexed metadata can be used in file lookup operations. The value types include numbers, strings or time stamps. To decrease the amount of data in the database, and to simplify the administration, the metadata values are inherited in the directory structure. So when a metadata is associated with a directory, every file in the sub-tree with the root in this directory has the metadata set as well. The directory hierarchy is common for both the Replica Catalog and the Metadata Catalog.

1.4.3 DFC Interfaces

The DFC is providing several user interfaces. The command
`dirac-dms-filecatalog-cli` launches a shell connected to the file catalog the user is supposed to interact with, based on his or her user group and VO. Its functionality is similar to a classic UNIX shell, with commands like `ls, cd, chmod,...`, with added file catalog specific commands for handling metadata, replication, storage element usage reports.

The DFC Python API is a programming interface that allows creating any custom commands or even small applications working on the DFC data.

The only graphical interface is provided by the Web portal. It features the DFC file browser with a look and feel similar to desktop applications.

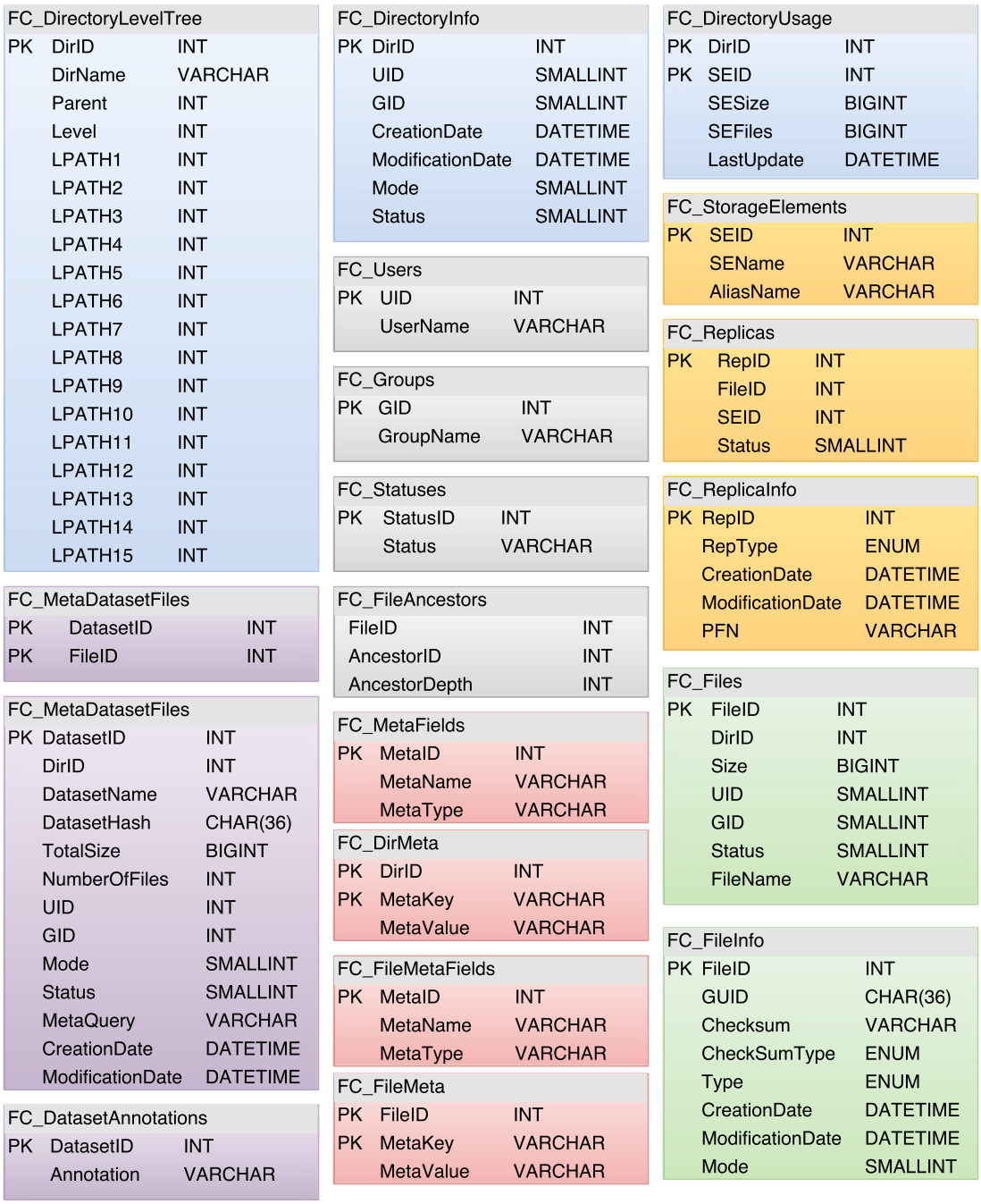


Figure 1.3: UML diagram of the tables in the current MySQL implementation of the DIRAC File Catalogs DB. The color scheme is chosen to make the orientation in the diagram easier: the grey tables are "the core" of the File Catalog, blue are directory related, green are file related, purple are for the dataset manager, yellow are the Replica Catalog and red are the Metadata Catalog (in the initial state. When the user starts creating indexes, for each index there is going to be a new table). Note that there are no relations between the tables, this is because no foreign keys are declared, all the table relationships are handled in DIRACs code.

2. Related Works

Currently there are several other middleware projects similar to DIRAC. The DFC was developed by the LHCb experiment collaboration which, however, has similar needs as other LHC experiments. In particular the ATLAS Distributed Data Management system and the AliEn file catalog are the closest comparable solutions.

2.1 ATLAS DDM

The ATLAS experiment at LHC is a general-purpose particle detector designed to investigate physics at the energy frontier. The output data rate, already reduced by the online trigger is 200-400Hz. Even with this reduction ATLAS records a huge amount of data – more than 10PB of raw collision data have been taken since the LHC started running in November 2009. After the first pass of processing was done at CERN, the data is registered in the ATLAS Distributed Data Management system (DDM) [17].

Although a basic unit of data in the ATLAS DDM is a file, the basic operation unit is a dataset: they may be transferred to the Grid sites, whereas single files may not. There is also one more layer of aggregation called the container, which is a collection of datasets. Datasets may overlap and in practice they do so in a hierarchical manner: production will use small datasets, referring to a few jobs processed at a site. These datasets are then aggregated into the main dataset that refers to a particular task in the ATLAS production system. These main datasets then may be added to a container, where the output of several similar tasks is gathered. Datasets may be open (new files can be added to them), closed (no new files may be added, but can have versions which differ in their content) and frozen (no new files may be added and no new versions created).

The responsibilities of the ATLAS DDM cover data registration, transfers between sites, file deletion, dataset consistency ensuring (manage file loses on sites), enforcing ATLAS Computing model policies and monitoring. The current implementation is called Rucio (the previous one was DQ2).

2.1.1 Rucio

In the current implementation files, datasets and containers follow an identical naming scheme which is composed of the scope and a name, called a data identifier [18]. Scopes are used to separate production and individual users. Metadata associated with a data identifier is represented using attributes (e.g. for a file its availability, for a dataset whether it is open, closed or frozen,...) which are represented as key-value pairs. The set of available attributes is restricted. Some metadata attributes can be set by a user, e.g. physics attributes (run number). Metadata not set by users include system attributes (size, creation time,...). For datasets and containers it is possible that the value of metadata is a function of the metadata of its constituents, e.g. the total size is the sum of the sizes of the constituents.

The Rucio Storage Element (RSE) is a repository for physical files. It is the smallest unit of storage space addressable within Rucio. It has a unique identifier and a set of properties (e.g. supported protocols, storage type, physical space,...). Physical files stored on RSEs are identified by their Physical File Name (PFN). The PFN is a fully qualified path identifying a replica of a file. The mapping between the file identifier and the PFN is a function of the identifier, RSE and protocol. Replica management in Rucio is based on replication rules defined on data identifier sets. A replication rule is owned by an account and defines the minimum number of replicas to be available on a list of RSEs.

2.2 AliEn

AliEn (ALICE Environment) [19] is a Grid framework developed by the ALICE Collaboration and used in production for more than 10 years.

The AliEn File Catalog is one of the key components of the AliEn system. It provides the mapping between Logical File Names (LFNs) visible to the end user and one or more Physical File Names (PFNs) which describe the physical location of the file by identifying the name of a storage element and the path to the local file. The LFNs are manipulated by users, one LFN can point to more PFNs. To prevent duplicate entries, each LFN is associated with a Globally Unique IDentifier (GUID).

The interface of the catalog is a hierarchical structure of directories and files, similar to the UNIX file system. The directories and files in the File Catalog have UNIX-like privileges.

It also extends the familiar file system paradigm to include information about running processes in the system (in analogy to the `/proc` directory in Linux systems). Each job inserted into AliEn Task Queue gets a unique id and a corresponding `/proc/id` directory, where it can register temporary files, standard input and output as well as all job products [20].

The File Catalog is designed to allow each directory node in the hierarchy to be supported by different database engines, possibly running on different hosts and even having different internal table structures optimized for a particular directory branch.

From the user point of view, the catalog behaves as a single entity. Internally it is divided between the LFN and GUID catalogs that are kept in separate databases. Every LFN database has an index table used to locate the files and a table that contains all the information about the concrete LFN (owner, group, size,...).

In addition, there can be user-defined tables containing metadata. AliEn supports file collections. A file collection is a user-defined list of entries that can be either other collections, LFNs or GUIDs. The collection mimics a file in the file catalog, its size is the sum of sizes of all its components. If a command is executed over a collection, it will be executed on each entry of the collection (e.g. if a user issues a command to replicate a collection, all the files of that collection will be replicated).

2.3 Comparison

The ATLAS DDM is made to be used by the ATLAS collaboration only, so it does not have to cover all the requests from other experiments, e.g. user-defined metadata. The DDM also introduces more layers of data structures above files: there are datasets, nested datasets and containers. Furthermore, there is no structure in the names of datasets and containers, they have just unique identifiers.

AliEn is opened for other virtual organizations, however there are only 2 other virtual organizations using AliEn services, while there are more than 15 ones using DIRAC. Since the administration of the whole middleware solution requires high effort, a proposition was made to integrate smaller independent DIRAC installations into a single service to optimize operational costs. Thus the France-Grilles DIRAC service, after an initial pre-production period of 6 months, was put into production in May 2012 [21]. This enabled small user groups, not capable of installing, configuring and maintaining such complex systems as DIRAC, to use it without the need to understand and operate the complex software and configurations. From the data management point of view, the structure is similar: there are files and datasets in a directory structured space, user can define new metadata.

Unlike DIRAC, AliEn uses the directory structure not solely for keeping the information about the files, but also introduces the `/proc` directory for temporary job files and outputs.

3. The Metaquery

When working with metadata in the DFC's metadata catalog, one of the most important operations is to find files by giving a specific set of conditions on their metadata. Each condition concerning one metafield is called term and is in the form of `<metafield> <operator> <value>`, where

- metafield is the name of the metadata field,
- operator is one of the classical comparison operators: `=`,`!=`,`<`,`<=`, `>`,`>=`,
- value is the value specifying the condition.

In the current production implementation the terms are coupled together by writing them in a space separated list, which translates into a conjunction with each space having the same meaning as the logical operator `AND`.

As the DFC grows bigger and more elaborate, the need for new classes emerges. Before the start of this project, the information about the metaquery was stored in a python dictionary data structure and every operation upon it was done in the code of the particular procedure. The handling was done similarly in several places so a decision was made to create a Metaquery class to encapsulate all the handling methods.

There was also a need for extending the metaquery language by adding the logical operators `OR` and `NOT` and allowing brackets. Another task was set when the demand for the user input to be optimized and normalized was raised. This meant that if two users inserted two equivalent metaqueries but wrote them down differently (e.g. `A=2 AND (B>3 OR C!=4)` vs. `A=2 AND B>3 OR A=2 AND C!=4`), internally they would be stored in the same way. This is opening an option for the metaquery to be hashed so that all the equivalent metaqueries would have the same hash and thus it could be used as a database index etc. The task of studying all the possible utilizations of the metaquery hash is not the aim of this project but should be engaged in the future.

3.1 Metaquery Theory

When evaluating the truth-value of a formula on a given substitution of its free variables, the form that comes in mind is the disjunctive normal form (DNF). A logical formula is in DNF if it is a disjunction of at least one conjunction of at least one literal. In our case literals are the terms specifying conditions on metadata and the variable substitution is the trueness of the term when a specific value of metadata is substituted for the metafield name. The whole formula is then true when at least one of the disjuncts evaluates as true, which makes evaluating the trueness of the formula on a given set of metadata values rather easy.

To be able to use this form, an efficient practical implementation had to be discussed, having in mind that backwards compatibility with the old metaquery representation is mandatory. The decision was made to represent the formula in DNF using a list of dictionaries, where each dictionary represents one conjunct of the outer disjunction. This is similar to the status before this project when only one dictionary was used to represent a conjunction of terms.

3.2 Metaquery Implementation

The Metaquery class contains

- the information about the metaquery itself written in a form of list,
- information about the types of all the metafields,
- handling and evaluating methods and functions.

Serialization of the class is done by exporting the list containing information about the formula. When a handling method has to be used somewhere in the code, a new instance of the class is created using this list. This helps to limit the network traffic to a minimum.

3.2.1 Query Input

The class has a constructor taking 2 parameters: `queryList` - the list representation of the metaquery and `typeDict` - the dictionary of metafield types. When a new metaquery is to be created from a user input, an instance of the Metaquery class is created without any information about the query itself, but ideally using a valid `typeDict`. Then the query string is parsed, tokenized and fed into the `setMetaQuery` method.

When the user inputs a query it first has to be tokenized. To enable the user to input the query without caring about whitespace distribution, the tokenization is done character by character. The user can also use the old input method (query string is in form `<metafield> <operator> <value> [<metafield> <operator> <value>]*`), the output will be then one conjunction as expected (AND operators are inserted in the correct places). However when the user uses at least one logical operator, no additional insertions are made and the rest of the input has to be a well formed formula.

Once the query string is tokenized, it has to be converted into DNF (see Algorithm 1) and optimized to gain the metaquery properties. Every formula can be transferred to DNF and method `setMetaQuery` takes care of it and also calls a private method for optimization.

The normalization algorithm follows the proof in a sense that when it traverses the input left to right, recursively descending into every sub-formula, it first removes the negations[1] and then takes care of the elements themselves, distributing the disjunction in the correct manner. The function takes one token of the query at a time, creates terms and connects them into inner conjunctions. These are then redistributed to form the outer disjunction converting the input into DNF. While adding terms into the conjunction and redistributing them following the distributivity rules of boolean algebra the elements of newly created conjunctions are checked for duplicities and colliding terms. The user is alerted when any are discovered. For instance when the user input would lead to a situation where `meta1 = 2 AND meta1 = 3`, the algorithm detects this, fails and alerts the user[2].

[1] Removing negation depends on the position of the NOT operator. When it is in front of the parenthesis surrounding a sub-formula it involves switching the meaning of the logical operators in it. When the NOT operator is in front of a term the comparison operators have to change (e.g. $< \rightarrow \geq$).

[2] This is a feature requested by the DIRAC community

Algorithm 1 Convert the tokenized input string into a query in DNF

```
 1: procedure ToDnf(inputList)
 2:     termTmp = [ ]; last = [ ]
 3:     for all atoms in inputList do
 4:         if atom is '(' then
 5:             Find correct ')'
 6:             last ← ToDNF(sub-list between parentheses)
 7:         else if atom is logical operator then
 8:             if termTmp is not empty then
 9:                 newTerm ← Parse(termTmp)
10:                 termTmp = [ ]
11:             else if negGlobal is set then
12:                 Switch atom meaning
13:             end if
14:             if atom is AND then
15:                 add last or newTerm in conjunction buffer
16:             else if atom is NOT then
17:                 switch negGlobal flag
18:             else                                    ▷ atom is OR
19:                 add term or buffered conjunction to output
20:             end if
21:         else
22:             add atom to termTmp
23:         end if
24:     end for
25: end procedure
```

The query in DNF is then passed to a method that goes through the disjuncts, checks them for common metafields and whether they are both relevant for the meaning of the query. If not, the irrelevant parts are removed.

3.2.2 Query Evaluation

To provide a correct input is a major challenge of the Metaquery class, although the main reason the file catalog features metaqueries is evidently the search for files. For files already stored in databases the DIRAC project tries to leave as much work as possible on the database, so the only thing to solve in this area is the interface between the database and the file catalog (the conversion between the Metaquery class and the databases query language). This problem is tackled in the Chapter 5.

Applying the metaquery to a particular set of metadata (for instance when a new file is uploaded to the file catalog) is handled by the Metaquery class itself, although there has been a discussion whether it is possible to let the database engine handle this situation. The conclusion was that applying an ordinary query would not be a big enough problem to justify initiation of network communication between the catalog server and the database server (which is the current practice of deployment). Instead a dictionary, where metafields are keys indexing their values, is passed to the method `applyQuery()` on an initialized Metaquery class.

Evaluating whether a concrete set of metadata satisfies the metaquery is made as simple as possible by using the DNF, where each element of the disjunction is tested (all the conditions in the element must be satisfied by the set of metadata). If at least one passes, the query is satisfied.

3.2.3 Output of the Query

For the development of the function providing the output of the Metaquery class, the need for a consistent result of equivalent metaqueries was the main goal. A disjunction is commutative so there has to be an artificial rule forcing the disjuncts to appear always in the same order. Internally the elements of the disjunction are represented as dictionaries, so sorting them according to their keys (the metafields) is the obvious choice. However, theoretically there could be two disjuncts with exactly the same metafields, so the values also had to be considered. In the case of identical metafields the optimization function guarantees that their values are going to be different.

4. The DatasetManager

As mentioned before, the DIRAC software architecture is based on a set of distributed, collaborating services, following the paradigm of a Service Oriented Architecture (SOA). The services themselves consist of modules. DatasetManager is one of the modules used by the File Catalog, where the other include e.g. FileMetadata module, DirectoryTree module etc.

The dataset is a set of files grouped together based on their common metadata. To accomplish that, each dataset is associated with a metaquery and all the files that it contains are satisfying it. Moreover there are two types of datasets: a dynamic dataset and a frozen (static) one. A dynamic dataset is exactly what is described above: a set of all the files that satisfy a given metaquery. The dynamic dataset changes every time a file satisfying its metaquery is added to the file catalog. A frozen dataset is a snapshot of a dynamic dataset. The user can freeze a dataset – the files of which he is using – in order to maintain the set of files for further evaluation or cross-checking the results of his work with other users over the same set of files. When a frozen dataset is later deleted, the user is asked whether the files, that are exclusively contained in this dataset should be deleted.

4.1 Previous Status

Before the start of this project, a prototype of the DatasetManager module was already in place. It complied with the demands requiring, what the structure of the database should look like and what is the expected behavior of the datasets, but many crucial features (like e.g. deleting files when a frozen dataset is deleted, dataset replication and overlapping and other) and properties (identifying the dataset based on its' location in the directory tree) were missing.

4.2 Implementation Details

The DatasetManager is one of the modules in the file catalog. The class offers multiple methods handling the datasets and exposing them to other modules. The state of the datasets is saved in a database, the class itself holds only an instance of the database interface object, so restarting the FileCatalog service is done without any problems. The module handles mainly basic CRUD operations with some additions.

4.2.1 Releasing and Freezing a Dataset

When a dataset is created, an entry is inserted to the database. It contains the dataset name, id, metaquery, status, id of the directory it is located in, UID and GID of the user who created it and some other properties. When a dataset is frozen, the query is evaluated and all the files, which the dataset contains, are associated with it (the records are inserted in the table `FC_MetaDatasetFiles`). Releasing the dataset only deletes these records and updates the cached properties.

When a dataset is deleted, its record in the database is removed. Moreover when the deleted dataset is frozen it is requested it would be deleted with the files it contains not including files contained by another frozen dataset. This is done using one select

```
SELECT FileID
FROM FC_MetaDatasetFiles
WHERE FileID IN (
        SELECT FileID
        FROM FC_MetaDatasetFiles
        WHERE DatasetID=X
)
GROUP BY FileID
HAVING Count(*)=1;
```

where in place of the `X` symbol the id of the deleted dataset is inserted. The user is then asked, whether he really wants delete those files from the file catalog. When he agrees, a request to the DIRACs Request Management System (RMS) is created. This system ensures that all the files are deleted from all the storage elements as well as from the file catalog database, which prevents memory leaks. The RMS is also used when dataset replication is executed.

4.2.2 Dataset Overlapping

Another important feature not included in the previous implementation was the determination of dataset overlapping. In the first step the metaqueries of the two datasets are combined into a conjunction and the check proceeds only if the conjunction is a valid metaquery. The procedure of the second step depends on the statuses of the datasets. When the two datasets in question are dynamic, the files satisfying the conjunction of their metaqueries are their overlap. When comparing a dynamic and a frozen dataset, the files satisfying the conjunction are compared to the list of the frozen datasets files. Finally, when two frozen datasets are checked, the sets of their files are simply compared and if the intersection is empty they do not overlap.

5. Choosing the NoSQL Database

One of the main goals of this project was to test whether connecting the file catalog, more specifically its metadata part, to a NoSQL database would improve the feedback speed of the service thus making it more pleasant to use or make it easier to implement and maintain. The new database had to satisfy the following conditions in order to be connectable to DIRAC and deployable in the computing centers

- a freeware version has to be available for DIRAC,
- the database has to have a python interface or client so it would be easy to incorporate into the DIRACs code.

The characteristics of the data itself add some restrictions. The database should be optimized for search speed. When combined with the current metadata implementation, we get two different types of data retrieval. For directories the database has to be able to get all the metadata associated with a directory identified by an ID. On the contrary, the files are fetched based on the terms in the metaquery so all the querying techniques the metaquery could use have to be possible, including range queries.

5.1 Apache Cassandra

Apache Cassandra[1] is a distributed database for managing large amounts of structured data across many commodity servers, while providing highly available service and no single point of failure [23]. Cassandra's data model is a partitioned row store, rows are organized into tables. Each row has an altering number of columns identified by a unique ID, which are essentially a key-value pair. Adding that Cassandra features its own query language CQL (Cassandra Query Language) which resembles the standard SQL used by all relational databases, it seems to be a perfect candidate for the metadata catalog.

5.1.1 Developing a Data Model for Cassandra

Although CQL makes it look like a relational database, the developer has to keep in mind that Cassandra is different. Our first scheme used the altering number of columns and organized the data in tables, where each file or directory had its own row with metanames as column names and values as column values (see Table 5.1). Secondary indices were created over column values so that the query would be possible. Although CQLs' similarity with SQL would suggest otherwise, it turned out that this kind of indexing does not support range queries so this data-model had to be abandoned.

After understanding more thoroughly how Cassandra works, another data-model was introduced utilizing most of Cassandra specific features. The key properties Cassandra guaranties are that rows are not divided across multiple

[1] `http://cassandra.apache.org/`

1	Meta1	Meta2	Meta3	
	123	'alpha'	0	
2	Meta1	Meta2	Meta3	Meta4
	4000	'delta'	0	'beta'
3	Meta1	Meta3		
	200	1		

Table 5.1: The first data model using Cassandra. The FileID is in the first column (row key), the next columns are metanames and their values.

Meta1	123	200	4000
	{1}	{3}	{2}
Meta2	'alpha'	'delta'	
	{1}	{2}	
Meta3	0	1	
	{1,2}	{3}	
Meta4	'beta'		
	{2}		

Table 5.2: The second Cassandra scheme. FileIDs are now in sets in column values, metanames are row keys and metavalues are columns names.

nodes in the cluster and column keys inside the row are sorted. Based on these two characteristics a functioning data model was created. For directories row IDs are mapped on directory IDs and the model the same as the previous one. Retrieving a row with a given row ID is one of the supported operations. For files each metaname has its row, column names are meta values and column values are sets of fileIDs of files having this metaname and value associated with them (see Table 5.2).

The rows are grouped in tables by value type, because the algorithm used to sort the column names is unified per table. There is also an index over fileID sets, so that retrieving all metadata for one specific file is possible. However the scheme is not optimized for this operation, because this is done only for the users information and when setting new metadata.

Listing 1 Data structure described using CQL

```
CREATE TABLE file_int (
    metaname text,
    value int,
    fileid set<int>,
    PRIMARY KEY (metaname, value)
);
```

In CQL this structure looks like a table with three columns and a compound primary key (see Listing 1). This brings the main disadvantage of this approach: meta names can be queried only one at a time and the result has to be then finalized in the DIRACs code. The expected structure of data could suggest that the number of files satisfying only a part of the metaquery can be large, the idea of using Cassandra as the File Catalog database was abandoned, because fetching multiple times a large number of files from the database and then doing an intersection in the python code is not optimal.

Listing 2 Metadata stored in a document database. This is a basic example, there can be many more fields in the JSON structure, but it will always remain a simple structure.

```
{
        'id'        : id
        'metaInt1'  : 1,
        'metaStr1'  : 'qwerty',
        'metaInt3'  : 123
}
```

5.2 Document Databases

A document-oriented database replaces the concept of a *row* from the world of relational databases with a more dynamic and versatile *document*. By allowing arrays and embedded documents the document-oriented databases provide denormalization of the data and complex relationships in a single document. Also there are no predefined schemes, which is essential for our project, because the number of associated metadata varies between files. The metadata are stored in simple a JSON structure with names being metanames (see Listing 2).

Document databases were not the first choice during developing this project, because the idea of storing metadata in a JSON structure and then building indices above the properties is not as familiar as Cassandras columns. But it turned out to be even easier to use.

5.2.1 MongoDB

MongoDB is an open-source document-oriented database storing JSON files. Currently, in November 2015, MongoDB is the fourth most popular type of database management system, and the most popular for document stores [24]. In MongoDB the document is the basic unit, documents are grouped into collections. A single instance of MongoDB can host multiple databases, each grouping together multiple collections. In collections documents are identified using a special field `_id` which has to be unique within a collection [25]. This projects maps the file or directory ids from the file catalog to this id field.

5.2.1.1 Using MongoDB

On Scientific Linux it can be installed from a package so the initial set up is rather easy. MongoDB comes with its own shell based on JavaScript, which the administrator can use to query and update data as well as perform administrative operations[2]. The client is rather easy-to-use and its commands are very similar to those used by the python library. The mongo package also contains several tools for monitoring the database including the `mongotop`[3] and `mongostat`[4], and several other helping the administrator with e.g. dumping and restoring the database.

[2] `https://docs.mongodb.org/getting-started/shell/client/`
[3] `https://docs.mongodb.org/manual/reference/program/mongotop/`
[4] `https://docs.mongodb.org/manual/reference/program/mongostat/`

For further debugging the log file provides a sufficient amount of information, when its verbosity is turned up to at least 3. Overall the database is easy to use and administer thanks to many well documented utilities.

There are two types of configuration files supported by MongoDB: the older one in form of `<setting> = <value>` and the newer using YAML format [26]. The database comes with the older one (although it is not developed since version 2.4), but most of the configuration documentation is in the newer one (and some of the new features can be only set in YAML format) so the administrator should convert it manually before using the database since it could ease the usage and optional re-configuration later. The major drawback when using this database is that every property has to be indexed manually, which not only takes time, but also consumes disk space.

5.2.2 Elasticsearch

Elasticsearch (ES)[5] is a real-time distributed open-source analytics and search engine [27] built on top of Apache Lucene[6]. Unlike very complex Lucene, ES features a simple RESTful API that makes it easy to use. Moreover it can be used not only for full-text search, but also for real-time analytics or, which is important for this project, as a distributed document store, where *every* field is indexed and searchable. Its python client[7] provides a wrapper for the RESTful API as well as some useful additional commands called helpers (e.g. for bulk loading data, the command `helpers.bulk(es, dataToLoad(size))` was used).

5.2.2.1 Using ES

Like MongoDB, Elasticsearch comes in a package. The installation set reasonable defaults (the only thing that was changed on the testing server was the data directory and listening on the universal IP address had to be enabled, because the default is only local host). The configuration file is in YAML format. There is no need for any initial data structure, the first data is simply inserted using the correct URL[8]. In ES, a document belongs to a type and types are contained by indices (hence the `index` and `doc_type` in the command). An Elasticsearch cluster can contain multiple indices, which can contain multiple types (there is a rough parallel with the world of relational databases, where indices would be databases and types would correspond to tables). Unlike the relational databases ES can create the index and type with the first inserted document using its default configuration.

5.3 Database Tests

For testing, a single IBM iDataPlex dx340 server was used, equipped with 2x Intel Xeon 5440 with 4 cores, 16GB of RAM and 300GB SAS hard disk, where all the

[5] `https://www.elastic.co/products/elasticsearch`

[6] `https://lucene.apache.org/`

[7] `http://elasticsearch-py.readthedocs.org/en/master/`

[8] `http://host:port/index/doc_type/doc_id` is the URL the RESTful API uses, when using the python library `index, doc_typedoc_id` are specified using the function arguments

database files were stored. After several smaller datasets, the main testing data was generated trying to mimic the DIRAC production data structure. There is expected over 10 million files with approximately 20 different meta names. The generated set had $(10000000-1)$ files with 1 to 998 metanames associated with them[9], which is more than the production data would have, but gave the testing more data to encumber the databases. The metrics names were `test[Int|Str]NNN`, where `NNN` stands for a number. The integer values were all picked from the interval $[1; 1000]$ and the string values were one of the words from the NATO phonetic alphabet [28], which lead to easier composition of testing queries. Integer fields represent continuous data types and string fields represent discreet data types.

The data was stored in two CSV files with lines `id,metaname,value` and the sizes of 53 and 59 GB.

5.3.1 Loading the Big Data

Both databases feature bulk loading functionality. Although the user will not probably use it, in this project, which involved testing the databases on a large data, the loading of them was one of the challenges.

ES python interface features a bulk load function (the default data chunk size is 500 records). Unfortunately it crashes on a TimeoutException from time to time, making the loading by a simple script a rather long procedure. We have not been able to discover the cause of the exception.

MongoDB features a command `mongoimport` which can read a file of JSON objects and load them all. Its pace on the test data was approximately $150 - 200$ entries per second. Unfortunately the script froze from time to time so loading the whole dataset took again several days. Once the data was loaded, it could be backed up using the utility `mongodump` and then again loaded using the command `mongorestore`. These two utilities ran without problems and were used when changing the storage engine.

5.3.2 Testing Query Performance on a Predefined Sample

The task of the database in this deployment will be to accept a query, execute the search and return all the ids of documents[10] satisfying the query. To measure the results as precisely as possible, a list of queries was generated. When testing one database, all other database services were stopped so that they did not interfere with the test. As we mentioned, to effectively search for data in MongoDB the document properties have to be indexed manually. Indices were made on several integer fields (testInt1-8) and three string fields (testStr1,testStr2,testStr3). The time cost of building[11] the indices and their space requirements are listed in Table 5.3.

ES does not have a similar limitation, however the queries were kept the same so that the comparison is as correct as possible.

[9] The generator chose a randomly 1 to 499 integer metrics and the same number of string ones.

[10] Document ids are mapped on file ids, so this procedure is returning the ids of files that satisfy the query

[11] for the default storage engine

Field	Time	Disk Space (MMAP)	Disk Space (WT)
testInt1	00:20:50	211 390	46 785
testInt2	00:20:00	211 407	46 789
testInt3	00:19:59	211 390	46 785
testInt4	00:20:12	211 415	46 789
testInt5	00:19:58	211 390	46 785
testStr1	00:20:06	203 043	44 351
testStr2	00:20:13	203 026	44 351
testStr3	00:20:51	203 043	44 356
testInt6	00:21:03	211 407	46 789
testInt7	00:19:57	211 399	46 785
testInt8	00:19:58	211 390	46 785

Table 5.3: The time and storage requirements for indices used in MongoDB. The indices were built while using the MMAP storage engine. Sizes are in MB.

The queries were generated randomly combining the integer and string properties. The number of hits was not considered while generating the queries. All the queries used can be found in Appendix B.

The program testing the performance is a python script running on a personal computer in the same LAN as the database machine (similarly to the expected deployment of the service). The measured time is the interval between the moment the query was submitted, to the moment the results were extracted to a python list. Neither the preparation of the query, nor the printing of the results were included in the final time.

For ES the cache of the index was flushed after every query to keep the results consistent (although as Figure 5.1 suggests, flushing the cache does not make a notable difference for bigger data volumes).

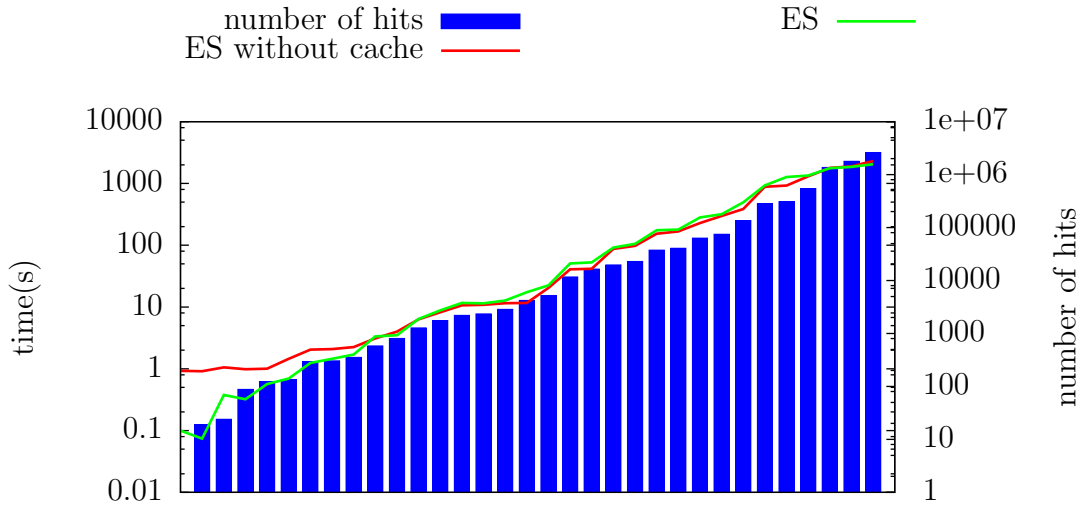


Figure 5.1: Query times for Elasticsearch comparing performance with and without dumping depending on the number of hits (logarithmic scale)

Storage Engine	Total Database Size	Total Index Size
MMAPv1	136 161	2 579
WiredTiger with zlib comp.	22 788	615
WiredTiger with default comp.	42 249	616

Table 5.4: Disk usage comparison of MongoDBs storage engines. Note the size of the indices does not depend on the compression used. All sizes are in MB.

The original MongoDB instance was using the default storage engine used by versions up to 3.0[12]. There is also a new storage engine WiredTiger[13] available and their performances were compared. Moreover the new engine brings the option of choosing the compression strategy. The comparison of disk space usage can be seen in Table 5.4 and query performance in Figure 5.2.

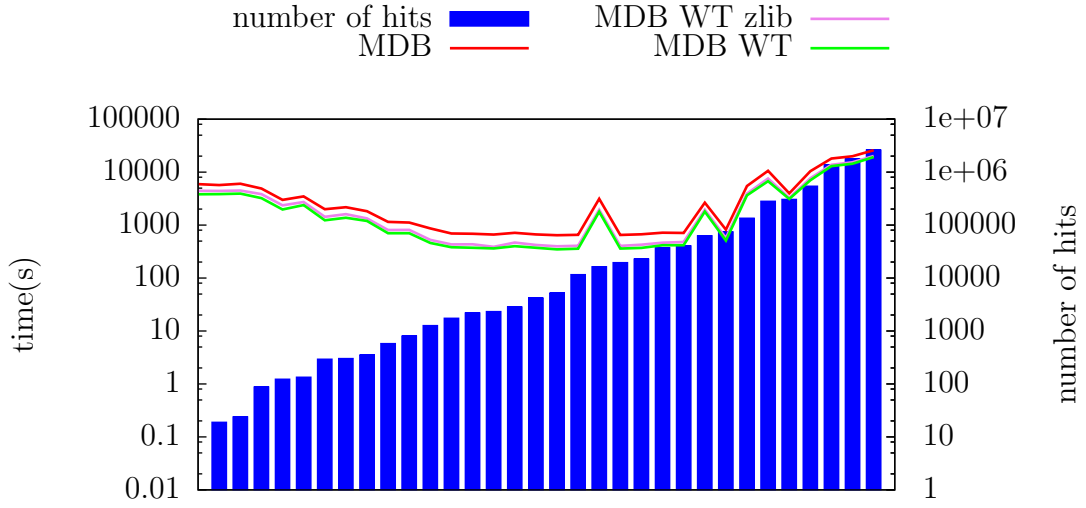


Figure 5.2: Query times for MongoDB comparing performance with different storage engines and compression options: the old MMAPv1 (MDB), the newly available WiredTiger (MDB WT) and WiredTiger with extra compression (MDB WT zlib). Logarithmic scale.

We can conclude as follows: if the administrator does not have a critically low amount of disk space available, MongoDB works best with the new WiredTiger storage engine with the default compression. In Figure 5.3 we can see the comparison of performance on the sample queries between the WiredTiger configuration of MongoDB and Elasticsearch.

The queries on MongoDB are taking much more time than on Elasticsearch. MongoDB provides a tool for explaining query execution. This becomes very useful when trying to investigate efficiency problems like this one. In the listing in Appendix C, one can see the output of the command for one of the sample queries. The query planner tries all the indices associated with one of the queried properties, then selects one and based on just this one performs the search. The

[12]MMAPv1 Storage Engine based on memory mapped files
[13] `https://docs.mongodb.org/manual/core/wiredtiger/`

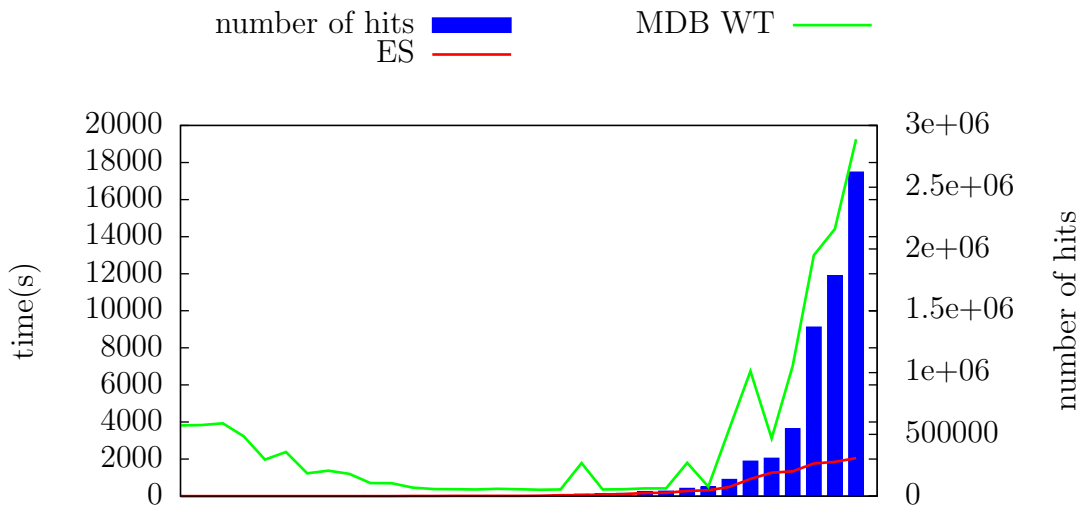


Figure 5.3: Comparison between ES and the best performing MongoDB (MDB WT) configuration.

search results are then filtered so that the returned fields satisfy the other conditions of the input query. To test the best possible performance, special compound indices were created for each query and the performance was tested using these indices.

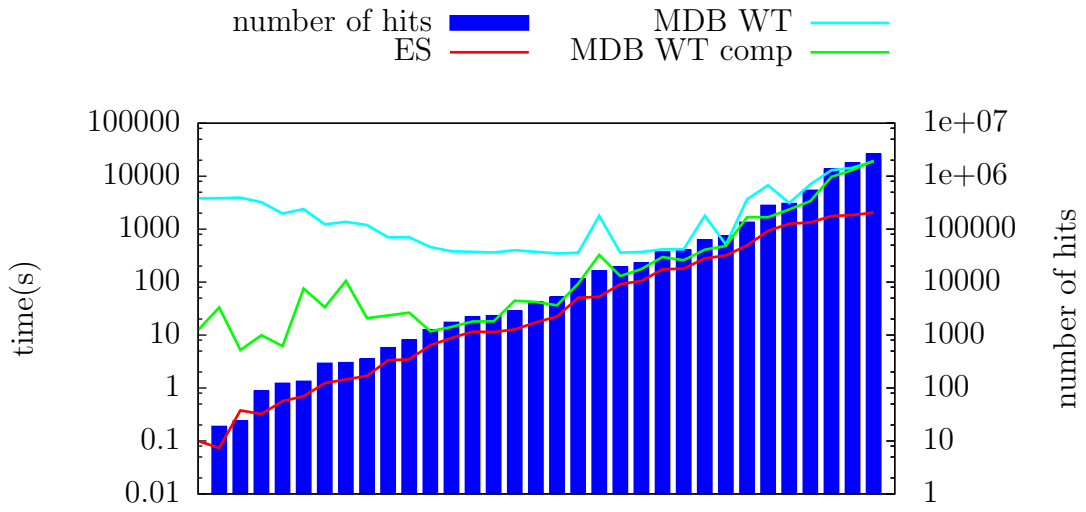


Figure 5.4: Comparison between MongoDB with indices created especially for the tested queries (MDB comp) and ES. To show the improvement the MongoDB performance without the compound indices is graphed as well (MDB WT)

Creating an index for each query is not the preferred usage pattern in this case. Also Figure 5.4 clearly states that even with these specialized indices MongoDB shows worse performance than Elasticsearch, although there is a great improvement when compared to the version without the new indices.

5.3.3 The Comparison Between NoSQL and the Current Deployment

As we mentioned, currently the database back-end to all the services in DIRAC, where a database is needed including the File Catalog and its metadata part, is relying on MySQL. The table scheme of the metadata part of the DFC can be seen in Figure 5.5.

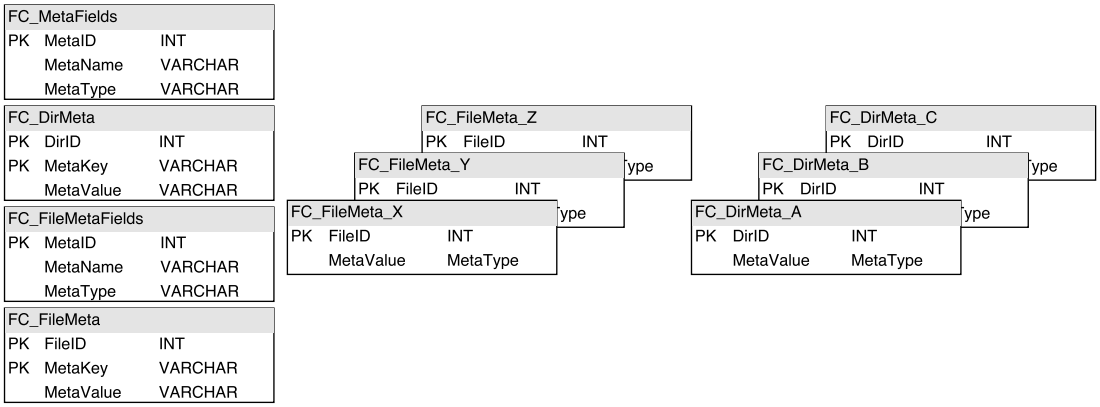


Figure 5.5: The File Catalog part of the database scheme. The tables on the left are created implicitly with the rest of the scheme. They are used for keeping the track of the types of metafields and un-indexed metadata values. For each new indexed field a new table is created, in this scheme there are 3 indexed file metafields (X, Y, Z) and three indexed directory metafields (A,B,C).

In order to see if it is worth to start using a NoSQL database, in this project we compared the current deployment with the database that performed the best in the tests we completed. When trying to test the MySQL database against the same queries by only loading the relevant data (creating tables only for 8 integer typed metadata and 3 string ones), the performance was much better than all the NoSQL databases. This was due to the fact, that the MySQL engine loaded the whole dataset in memory and did not have to use the disk when executing the query. This however is not the expected behavior, because the database engine will most likely not serve only the DFC metadata catalog, so there will not be enough memory space to keep all the metadata in cache. To test how the database could behave in production, the command `flush tables;` was used after every query to dump the cached tables. Also the whole dataset was inserted into the database. This also let us to compare the size of data indexed by MySQL and by NoSQL (Table 5.5).

To test the databases more thoroughly a new set of queries was created. These queries involve more than just the small set of metafields indexed by the MongoDB. MySQL on this set proved itself better on queries consulting a lesser number of metafields and a larger number of hits. Performance of Elasticsearch seems to correspond to the number of hits while the complexity of the query does not seem to have an effect on its speed (results are provided in Figure 5.6).

In all the tests above Elasticsearch and its indices were left with the default settings. Since ES is primarily a distributed store, the default setting assumes that there will be more than just one server in the cluster. This means that

Database	Total database size
MongoDB	42 249
Elasticsearch	59 715
csv files	113 384
MySQL	185 135

Table 5.5: The comparison of disk usage for all databases. For an illustration the size of the input CSV files is shown. All sizes are in MB.

for every read several nodes have to be contacted and even though the testing deployment consists of only one server, the index cut the data into 5 shards[14] (default setting). The volume of data used in this project is not as large to need a whole cluster to be stored so it was re-indexed into an index with no replicas and only one primary shard, which is ideal when only one server is used. Also it comes closer to the MySQL database, because it can be run on only one server. The performance on re-indexed data was not significantly better and an even worse performance drop is observable when fetching a large number of documents. The queries contained from 2 to 6 conditions (consulted 2 to 6 different metadata).

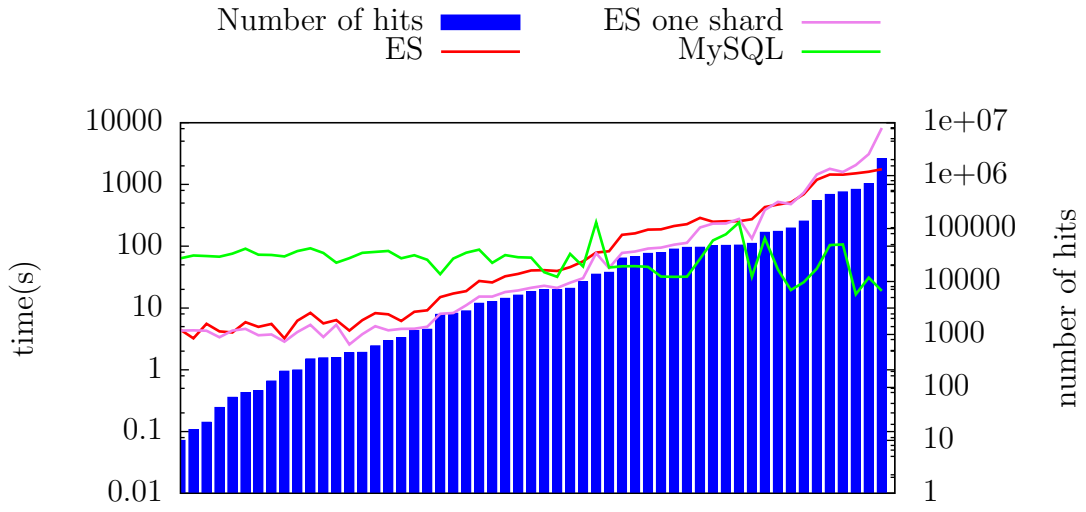


Figure 5.6: Comparison between the currently deployed MySQL and ES on a larger set of queries. ES performance with data re-indexed to one shard is also graphed (ES one shard).

[14] A shard is a single Lucene instance. It is a low-level "worker" unit which is managed automatically by Elasticsearch. An index is a logical namespace which points to primary and replica shards.

6. NoSQL Integration into DFC

To prove the concept of connecting the metadata part of the DIRAC File Catalog to a NoSQL database modules `fmeta` and `dmeta` that manage the metadata in the FileCatalog service were modified. As discussed in the previous chapter, the best available NoSQL database for this particular purpose is Elasticsearch.

6.1 Query Strategy

The current query search algorithm cuts the query per element of the disjunction. With each element the File Catalog finds directories satisfying the directory part of the query. Then it finds the files satisfying the file part. With the list of files and list of directories it filters the files that are not in the sub-tree of one of the satisfying directories (the algorithm is described by Algorithm 2).

Algorithm 2 Find files satisfying query

```
1:  function FIND(queryList)
2:      outFiles = [ ]
3:      for all element in queryList do
4:          dirMeta ← getDirectoryMeta(element)
5:          dirs ← findDirectories(dirMeta)              ▷ including sub-trees
6:          fileMeta ← getFileMeta(element)
7:          files ← findFiles(fileMeta)
8:          files ← dropFilesNotInDirs(files, dirs)
9:          outFiles.append(files)
10:     end for
11:     return removeDuplicates(outFiles)
12: end function
```

The procedure is so complicated, because the metadata for directories and for files are managed by different modules (in the algorithm, function `findDirectories` is in module `dmeta` and `findFiles` is handled by `fmeta`). This can result in unnecessary fetching of files, which are then dropped because they happen to be located in a directory that does not satisfy the query. This is especially relevant because as the tests done by this project suggest, the query speed of Elasticsearch unlike MySQL depends mainly on the number of hits it has to retrieve and not on the complexity of the query (see Figure 5.6).

The solution could be letting the database engine deal with the complexity of the search algorithm. This would mean storing all the metadata associated with a file (its own as well as the ones inherited from the directory structure) in the files document. This would increase disk space (but as Table 5.5 suggests, Elasticsearch is much more space efficient when compared to MySQL) as well as the difficulty of metadata management. The inheritance of metadata through the directory structure has to be maintained when setting and unsetting them. The result would be that finding files would only consist of translating the metaquery from the internal representation to an Elasticsearch compatible form (which is

now done per element in functions `findDirectories` and `findFiles`) and submitting the query. What would be sacrificed is the fact that directory and file metadata can be handled in a completely different manner.

6.2 The New Implementation

A new module has been developed to create a specialized interface between the database and the metadata catalog. This also minimized the changes that had to be done in the current metadata managing modules (the practice where one module manages directory metadata and another manages file metadata was preserved). As discussed above, the module stores all the metadata in one document to improve the speed of queries. Setting and un-setting directory metadata is a rather complicate procedure, which requires fetching all the documents associated with files in the particular sub-tree and updating them with the new metadata (respectively removing the one removed from the directory).

Removing a property from a document in Elasticsearch is not one of the basically supported operations. There are two ways to implement it:

1. get the whole document from the database, remove the property and insert it again;

2. submit a script that deletes the property from the document inside the database in a update-like operation.

Since the scripting has to be enabled by the database administrator and it is not guaranteed for the DIRAC administrator to have administrator access to the database as well, the first approach was chosen. This means that removing a metadata from a directory that has lots of files in its sub-tree is a non-trivial operation and shall be executed as rarely as possible. Updating and setting new metadata can be done by a simple command.

For searching the metadata managing modules from the file catalog had to be changed. The new algorithm simply converts the query to the correct format, submits it and reads the result.

7. User Documentation

To start working with DIRAC (or the Grid in general), the user should join some grid Virtual Organization and obtain a Grid Certificate. Before a user can work with DIRAC, the user's certificate proxy should be initialized and uploaded to the DIRAC ProxyManager Service [11]. This is done by running command `dirac-proxy-init`. After that, all the DIRACs' functionality is ready for use and all the actions will be signed by the users certificate.

The primary interface to the DIRACs' file and metadata catalog is the command-line interface provided by script `dirac-dms-filecatalog-cli`. Commands that use the modules implemented by this project are `meta`, `find` and `dataset`.

7.1 Command Meta

Command used for all the metadata manipulating operations. The sub-commands of command `meta` are: `index`, `get`, `set`, `remove` and `show`.

7.1.1 index

Usage: `meta index [-d|-f|-r] <metaname> <metatype>`

This command adds new a metadata name to available metadata names. The user then can assign a value to the metaname for a concrete file or directory. The parameters `-f` and `-d` specify, whether the new metaname will be associated with files or directories, respectively. The `-r` parameter is used, when the user wants to remove the metadata name. `<metatype>` can be one of the following: int, float, string, date. The user has to be in the `dirac_admin` group to be allowed to use this command.

7.1.2 set

Usage: `meta set <path> <metaname> <metavalue>`

This command sets a value to a metaname for a directory or a file specified by the path parameter.

7.1.3 get

Usage: `meta get [<path>]`

This command gets all user defined metadata for a directory or file specified by the path parameter. When getting metadata for a file, it also gets all the inherited directory metadata. When getting metadata for a directory, the output specifies which are defined on that concrete directory and which are inherited:

```
>meta get dir1
          !testDirStr : foo
           *unitFloat : 1.5
          *testDirInt : 1
```

The metadata prefixed with the `!` mark are defined on the directory, others (prefixed with *) are inherited.

7.1.4 remove

Usage: `meta [remove|rm] <path> <metaname>`

This command removes the previously defined metaname for a directory or a file specified by the path parameter.

7.1.5 show

Usage: `meta show`

This command shows all the available metadata names, divides them between the file and the directory metadata and supplies the type.

7.2 Command Find

Usage: `find [-q] [-D] <path> <metaQuery>`

This command finds all files satisfying the given metadata query. When the command is invoked it parses the query and, unless the `-q` parameter is used, prints it out in its internal representation. After the query results return, the list of all the files satisfying it is printed out. When the `-D` parameter is used, the command prints only the directories that contain the files satisfying the meta-query.

7.3 Command Dataset

Command used for all the operations involving the dataset functionality. Its sub-commands are `add`, `annotate`, `check`, `download`, `files`, `freeze`, `locate`, `overlap`, `release`, `replicate`, `rm`, `show`, `status` and `update`.

7.3.1 add

Usage: `dataset add [-f] <dataset_name> <meta_query>`

This command adds a new dataset containing all files satisfying the query supplied by the `<meta_query>` parameter. In the `<dataset_name>` the user specifies the path where the dataset should be located. When parameter `-f` in used, the dataset is immediately frozen.

7.3.2 annotate

Usage: `dataset annotate [-r] <dataset_name> <annotation>`

This command adds annotation to a dataset. There is no indexing involving annotations, they are just for the user. The annotation is a string up to 512 characters long. When the `-r` parameters is used, the annotation is removed from the specified dataset. A dataset can have only one annotation.

7.3.3 check

Usage: `dataset check <dataset_name> [<dataset_name>]*`

This command checks the correctness of the cached information about the dataset. It can be supplied with one dataset, or a list of dataset names separated with spaces. The information can be outdated by recent file addition or removal in the file catalog and can be updated using the sub-command `update`.

7.3.4 download

Usage: `dataset download <dataset_name> [-d <target_dir>] [<percentage>]`

This command invokes download of the dataset files. If the `<percentage>` parameter is used, a subset of the datasets files are downloaded. The size of the downloaded part is approximately the inserted percentage of the size of the whole dataset. Unless the `-d` parameter supplies the target directory, the current working directory is used. The command creates a directory with the same name as the dataset and all the files are saved there.

7.3.5 files

Usage: `dataset files <dataset_name>`

This command lists all the files that the specified dataset groups. In case of a frozen dataset, the files saved in the database associated to the dataset are listed. When the dataset is dynamic, the command returns the same result, as using the `find` command with the datasets metaquery.

7.3.6 freeze

Usage: `dataset freeze <dataset_name>`

This command changes the status of a dataset to frozen. All the files that satisfy the datasets metaquery at the moment of the command invocation are associated with the dataset. For releasing the dataset, the sub-command `release` is used.

7.3.7 locate

Usage: `dataset locate <dataset_name>`

This command shows the distribution of the dataset files over the storage elements providing the absolute size and percentage of the dataset size used per storage element.

7.3.8 overlap

Usage: `dataset overlap <dataset_name1> <dataset_name2>`

This command checks if the two datasets have the same files. The metaqueries are checked first so that when there cannot be files satisfying both, the check does not compare lists of files.

7.3.9 release

Usage: `dataset release <dataset_name>`

This command changes the status of a dataset to dynamic. All the records associating concrete files to the dataset are deleted from the database. For freezing the dataset, the sub-command `freeze` is used. However, if there were files added to the file catalog that satisfy the datasets metaquery after the dataset was frozen, the subset of files satisfying the metaquery not including those files cannot be recreated after releasing the dataset.

7.3.10 replicate

Usage: `dataset replicate <dataset_name> <SE>`

This command initiates a bulk replication of the datasets files to a storage element specified by the parameter `<SE>`. The replication is handled by the Request Management System.

7.3.11 rm

Usage: `dataset rm <dataset_name>`

This command deletes the dataset from the file catalog. If the dataset is frozen, its files are cross-checked with all other frozen datasets and if there are some files that are grouped by the deleted dataset only, the user is offered the option to delete those files from the File Catalog. The deletion is handled by the Request Management System.

7.3.12 show

Usage: `dataset show [-l] [<dataset_name>]`

This command lists the names of all the existing datasets. When the `-l` option is used, other data about the datasets are printed as well. When a `<dataset_name>` is provided, the command restricts itself to this dataset.

7.3.13 update

Usage: `dataset update <dataset_name> [<dataset_name>]*`

This command updates the cached parameters for a dataset or for a space-separated list of datasets.

7.3.14 status

Usage: `dataset status <dataset_name> [<dataset_name>]*`

This command prints details about a specified dataset or a space-separated list of datasets.

```
> dataset status testDataset
testDataset:
========
    Key               Value
=========================================================
  1 NumberOfFiles     13
  2 MetaQuery         testDirInt = 1
  3 Status            Dynamic
  4 DatasetHash       A89F08D23140960BDC5021532EF8CC0A
  5 TotalSize         61562
  6 UID               2
  7 DirID             5
  8 OwnerGroup        dirac_admin
  9 Owner             madam
 10 GID               1
 11 Mode              509
 12 ModificationDate  2015-09-24 11:50:28
 13 CreationDate      2015-09-24 11:50:28
 14 DatasetID         71
```

Conclusion

This project successfully added dataset support to the DIRAC File Catalog making DIRAC even more versatile. The dataset support was actively requested by two experiments, that can now start using DIRAC. Closely coupled with the dataset functionality went the development of the new MetaQuery class that extended the metaquery language by adding more logical operators. The new MetaQuery also supports normalization and optimization of the query input, opening new possibilities for future usage in the DIRAC Data Management system as well as in the Workflow Management system.

This project also tackled the problem of storing metadata. Trying to enhance the current solution, a couple of NoSQL databases were tested on sample data similar to the DIRAC production metadata. The tests proved that connecting the metadata part of DFC to a NoSQL database could improve query performance, especially on more complex queries. A better performance is observable when applying 4 and more constraints and retrieving less than 10 000 hits.

To prove the concept of connecting a NoSQL database to DIRAC, a new module was developed to provide an interface between DFC and the database. As the back-end database Elasticsearch was used, because it performed best in the conducted tests. To improve the query performance, the complexity was moved from the python code of DIRAC to the database engine. This was traded by adding complexity to the management procedures. These do not have to be, unlike the query mechanism, optimized for time performance.

In future if the DIRAC collaboration decides to use Elasticsearch as the database back-end for the metadata catalog, more functionality can be added using Elasticsearch specific features. For example when a query is executed, the number of documents satisfying it is returned before all the documents are fetched. This could be used for example when checking the properties of a dynamic dataset or when trying to predict the time for fetching all the results of a particular query. Also new comparison operators translating to one of Elasticsearch query types[1] can be added to extend the metaquery language.

[1] `https://www.elastic.co/guide/en/elasticsearch/reference/current/term-level-queries.html`

Bibliography

[1] B. JONES. *Towards the European Open Science Cloud* [online] *Zenodo* 2015-03. [ref. 2015-12-01] `http://dx.doi.org/10.5281/zenodo.16001`

[2] J. HAYES. *Happy 10th Birthday, WLCG!* [online] International Science Grid This Week [ref. 2014-09-18] `www.isgtw.org/feature/happy-10th-birthday-wlcg`

[3] *The Grid: A system of tiers* [online] CERN [ref. 2014-09-18] `home.web.cern.ch/about/computing/grid-system-tiers`

[4] *Grid Computing - Technology and Applications, Widespread Coverage and New Horizons.* edited by Maad, Soha; InTech, 2012. ISBN 978-953-51-0604-3

[5] *User Interfaces* [online] EGIWiki [ref. 2014-09-29] `wiki.egi.eu/wiki/User_Interfaces`

[6] A. A. ALVES, L. M. FILHO, A. F. BARBOSA, I. BEDIAGA, G. CERNICCHIARO, G. GUERRER, H. P. LIMA, A. A. MACHADO, et al. The LHCb Detector at the LHC. *Journal of Instrumentation*. 2008-08-01, vol. 3, issue 08. DOI: 10.1088/1748-0221/3/08/S08005.

[7] A. TSAREGORODTSEV, M. BARGIOTTI, N. BROOK, et al. DIRAC: a community grid solution. *Journal of Physics: Conference Series*. 2008-07-01, vol. 119, issue 6. DOI: 10.1088/1742-6596/119/6/062048.

[8] T. ERL. *Service-oriented architecture: concepts, technology, and design*. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference, 2005, xxviii, 760 p. ISBN 01-318-5858-0.

[9] A. SHOSHANI, A. SIM, J. GU. Storage Resource Managers: Middleware Components for Grid Storage. 2002.

[10] P. DUBOIS. *MySQL: the definitive guide to using, programming, and administering MySQL 4.1 and 5.0*. 3rd ed. Indianapolis, Ind.: Sams Pub., 2005, xxii, 1294 p. ISBN 06-723-2673-6.

[11] A. CASAJUS, R. GRACIANI and The Lhcb Dirac TEAM. DIRAC distributed secure framework. *Journal of Physics: Conference Series*. 2010, vol. 219, issue 4. DOI: 10.1088/1742-6596/219/4/042033.

[12] A. C. RAMO, R. G. DÍAZ. DIRAC Security Infrastructure. *Proceedings of the CHEP 2006 Conference*. 2006.

[13] A. CASAJUS, R. GRACIANI, S. PATERSON, A. TSAREGORODTSEV and The Lhcb Dirac TEAM. DIRAC pilot framework and the DIRAC Workload Management System. *Journal of Physics: Conference Series*. 2010-04-01, vol. 219, issue 6. DOI: 10.1088/1742-6596/219/6/062049.

[14] *Request Management System* [online] DIRAC administrators documentation [ref. 2015-10-16] `diracgrid.org/files/docs/AdministratorGuide/Systems/RequestManagement/rms.html`

[15] C. HAEN. *Federating LHCb datasets using the Dirac FileCatalog* (presentation, 21st International Conference on Computing in High Energy and Nuclear Physics 2015)

[16] A. TSAREGORODTSEV, S. POSS. DIRAC File Replica and Metadata Catalog. *Journal of Physics: Conference Series*. 2012-12-13, vol. 396, issue 3. DOI: 10.1088/1742-6596/396/3/032108.

[17] V. GARONNE, R. VIGNE, G. STEWART, M. BARISITS, T. B. EERMANN, M. LASSNIG, C. SERFON, L. GOOSSENS, et al. The ATLAS Distributed Data Management project: Past and Future. *Journal of Physics: Conference Series*. 2012-12-13, vol. 396, issue 3. DOI: 10.1088/1742-6596/396/3/032045.

[18] V. GARONNE, R. VIGNE, G. STEWART, M. BARISITS, T. B. EERMANN, M. LASSNIG, C. SERFON, L. GOOSSENS, et al. Rucio – The next generation of large scale distributed system for ATLAS Data Management. *Journal of Physics: Conference Series*. 2014-06-11, vol. 513, issue 4. DOI: 10.1088/1742-6596/513/4/042021.

[19] S. BAGNASCO, L. BETEV, P. BUNCIC, F. CARMINATI, C. CIRSTOIU, C. GRIGORAS, A. HAYRAPETYAN, A. HARUTYUNYAN, A. J. PETERS, P. SAIZ. AliEn: ALICE environment on the GRID. *Journal of Physics: Conference Series*. 2008-07-01, vol. 119, issue 6, s. 062012-. DOI: 10.1088/1742-6596/119/6/062012.

[20] P. BUNCIC, A. J. PETERS, P. SAIZ, J. F. GROSSE-OETRINGHAUS. The architecture of the AliEn system. *Proceedings of the Computing in High Energy Physics (CHEP 2004), Interlaken, Switzerland, 106.*

[21] A. TSAREGORODTSEV. DIRAC Distributed Computing Services. *Journal of Physics: Conference Series*. 2014-06-11, vol. 513, issue 3. DOI: 10.1088/1742-6596/513/3/032096.

[22] *DIRAC overview* [online] DIRAC documentation [ref. 2014-09-18] `diracgrid.org/files/docs/Overview/index.html`

[23] DATASTAX CORPORATION. Introduction to Apache Cassandra White Paper. 2013. Also available online: `www.datastax.com/wp-content/uploads/2012/08/WP-IntrotoCassandra.pdf`

[24] *Popularity ranking of database management systems* [online] DB-Engines Ranking [ref. November 2015] `http://db-engines.com/en/ranking`

[25] K. CHODOROW. *MongoDB: the definitive guide*. Second edition. Beijing: O'Reilly, [2013], xix, 409 pages. ISBN 14-493-4468-2.

[26] O. BEN-KIKI, C. EVANS, B. INGERSON. *YAML Ain't Markup Language (YAML™)* Version 1.1. Working Draft 2008-05, 2009, 11.

[27] C. GORMELY, Z. TONG. *Elasticsearch: the definitive guide*. S.l.: O'Reilly Media, 2014. ISBN 978-144-9358-549.

[28] Combined Communications and Electronics Board. *COMMUNICATION INSTRUCTIONS GENERAL*. OCTOBER2010, (121). Allied Communications Publications. Available on: `http://jcs.dtic.mil/j6/cceb/acps/acp121/ACP121I.pdf`. Section: 318

List of Tables

List of Abbreviations

CERN European Organization for Nuclear Research (name derived from Conseil Européen pour la Recherche Nucléaire) – European research organization that operates the largest particle physics laboratory in the world.

LHC Large Hadron Collider – the world's largest and most powerful particle collider built by CERN

WLCG The Worldwide LHC Computing Grid – global collaboration of more than 170 computing centers in 42 countries, linking up national and international grid infrastructures

ALICE A Large Ion Collider Experiment – one of the four big experiments for the LHC, hosted at CERN

ATLAS A Toroidal LHC Apparatus – one of the four big experiments for the LHC, hosted at CERN

CMS Compact Muon Solenoid – one of the four big experiments for the LHC, hosted at CERN

LCHb LHC beauty – one of the four big experiments for the LHC, hosted at CERN

CLI Command Line Interface – means of interacting with a computer program

DNF Disjunctive Normal Form – standard form of a logical formula, which is a disjunction of conjunctive clauses

DIRAC The Distributed Infrastructure with Remote Agent Control – middleware developed by one of the CERNs' collaborations

DFC DIRAC File Catalog – file catalog service in the DIRAC middleware

LFC LCG File Catalog – file catalog provided by CERN IT department

LFN Logical File Name – name of the file displayed to a file catalog user

PFN Physical File Name – complete file URL

CQL Cassandra Query Language – the primary language for communicating with the Cassandra database

WT WiredTiger – MongoDB storage engine, newly available in version 3.0

SRM Storage Resource Managers – middleware software modules managing available storage resource

A. DVD contents

The attached DVD contains:

- `Thesis.pdf` – this thesis,
- `/DIRAC/` – the whole DIRAC source code,
- `/data/` – a sample of the testing data,
- `/scripts/` – all the scripts used for database loading and testing.

B. Queries

This attachment lists the set of queries, which was used to get a basic benchmark of the databases performance. The queries are written in JSON form for convinience: a one level JSON is essentially a serialized form of the python dictionary data structure. When creating a meaningful query from the JSON the following algorithm is used:

- if the current key contains the substring `Int`, then according to the projects' conventions the field represents an integer typed metafield. The query condition in this case is `[key] > [value]`,
- otherwise (key contains substring `Str`) the field represents a string metafield adn the condition is `[key] = [value]`.

```
{'testInt2': 844, 'testInt8': 406}

{'testInt2': 125, 'testInt7': 101}

{'testInt5': 270, 'testInt6': 267}

{'testInt1': 154, 'testInt2': 69, 'testInt7': 305}

{'testInt2': 260, 'testInt4': 731, 'testInt5': 873}

{'testInt1': 185, 'testInt5': 389, 'testInt7': 561}

{'testInt2': 19,
 'testInt3': 745,
 'testInt5': 600,
 'testInt7': 321}

{'testInt1': 330,
 'testInt2': 168,
 'testInt3': 477,
 'testInt7': 515}

{'testInt2': 809,
 'testInt4': 339,
 'testInt7': 848,
 'testInt8': 562}

{'testInt5': 593, 'testStr1': 'foxtrot'}

{'testInt2': 258, 'testStr2': 'yankee'}

{'testInt6': 805, 'testStr3': 'tango'}
```

```
{'testInt4': 467, 'testInt5': 364, 'testStr2': 'juliet'}

{'testInt5': 85, 'testInt8': 385, 'testStr1': 'juliet'}

{'testInt3': 506, 'testInt5': 840, 'testStr2': 'victor'}

{'testInt2': 645,
 'testInt4': 57,
 'testInt5': 309,
 'testStr2': 'kilo'}

{'testInt1': 794,
 'testInt4': 190,
 'testInt8': 663,
 'testStr3': 'juliet'}

{'testInt2': 621,
 'testInt4': 495,
 'testInt8': 558,
 'testStr3': 'whiskey'}

{'testInt1': 833,
 'testInt6': 807,
 'testInt7': 336,
 'testInt8': 58,
 'testStr3': 'sierra'}

{'testInt3': 943,
 'testInt5': 292,
 'testInt6': 762,
 'testInt7': 160,
 'testStr1': 'charlie'}

{'testInt5': 339,
 'testInt6': 918,
 'testInt7': 752,
 'testInt8': 789,
 'testStr3': 'mike'}

{'testInt8': 157, 'testStr1': 'kilo', 'testStr2': 'hotel'}

{'testInt6': 347, 'testStr1': 'papa', 'testStr2': 'india'}

{'testInt3': 109, 'testStr1': 'victor', 'testStr3': 'juliet'}

{'testInt5': 240,
 'testInt6': 578,
```

```
 'testStr1': 'delta',
 'testStr3': 'golf'}

{'testInt3': 131,
 'testInt4': 160,
 'testStr1': 'kilo',
 'testStr2': 'tango'}

{'testInt3': 487,
 'testInt8': 856,
 'testStr1': 'charlie',
 'testStr3': 'whiskey'}

{'testInt4': 443,
 'testInt5': 103,
 'testInt7': 540,
 'testStr2': 'india',
 'testStr3': 'golf'}

{'testInt3': 303,
 'testInt7': 200,
 'testInt8': 866,
 'testStr1': 'foxtrot',
 'testStr3': 'golf'}

{'testInt1': 532,
 'testInt4': 371,
 'testInt6': 155,
 'testStr2': 'india',
 'testStr3': 'yankee'}

{'testInt1': 479,
 'testInt2': 432,
 'testInt3': 915,
 'testInt8': 33,
 'testStr1': 'quebec',
 'testStr2': 'alpha'}

{'testInt1': 826,
 'testInt4': 802,
 'testInt5': 824,
 'testInt8': 874,
 'testStr1': 'juliet',
 'testStr2': 'golf'}

{'testInt2': 604,
 'testInt4': 791,
 'testInt5': 354,
```

```
'testInt6': 668,
'testStr1': 'juliet',
'testStr2': 'xray'}
```

C. MongoDB explain index

This appendix lists output of the `db.collection.explain.find()` command applied to one of the queries from the testing set. The rewriting of that query can be seen in the first listing, the output in the second one.

```
SELECT FileID FROM Metas
        WHERE testInt5.Value > 364
        AND testStr2.Value='juliet'
        AND testInt4.Value > 467;
```

In the beginning the query planner writes some basic information about the search (the names `namespace`, `parsedQuery`, etc.). Then the details of the query plan selected by the optimizer are listed in a name `winningPlan`. The stage `FETCH` describes retrieving documents, in this particular case the documents are filtered so that the constraints that are minor according to the query planer are fulfilled. Then, in the next stage `IXSCAN`, the index is scanned. In this case the index over the property `testStr2` was chosen. After the winning plan, two more rejected plans are described. These plans differ from the winning one by using a different index. In the end some information about the current server was printed by the command, but was not included here because it has no relevance for this problem.

```
{
  "queryPlanner": {
    "plannerVersion": 1,
    "namespace": "fcall.files",
    "indexFilterSet": false,
    "parsedQuery": {
      "$and": [
        {
          "testStr2": {
            "$eq": "juliet"
          }
        },
        {
          "testInt4": {
            "$gt": 467
          }
        },
        {
          "testInt5": {
            "$gt": 364
          }
        }
      ]
    },
    "winningPlan": {
      "stage": "FETCH",
```

```
    "filter": {
      "$and": [
        {
          "testInt4": {
            "$gt": 467
          }
        },
        {
          "testInt5": {
            "$gt": 364
          }
        }
      ]
    },
    "inputStage": {
      "stage": "IXSCAN",
      "keyPattern": {
        "testStr2": 1
      },
      "indexName": "testStr2_1",
      "isMultiKey": false,
      "direction": "forward",
      "indexBounds": {
        "testStr2": [
          "[\"juliet\", \"juliet\"]"
        ]
      }
    }
  },
  "rejectedPlans": [
    {
      "stage": "FETCH",
      "filter": {
        "$and": [
          {
            "testStr2": {
              "$eq": "juliet"
            }
          },
          {
            "testInt4": {
              "$gt": 467
            }
          }
        ]
      },
      "inputStage": {
        "stage": "IXSCAN",
```

```
            "keyPattern": {
              "testInt5": 1
            },
            "indexName": "testInt5_1",
            "isMultiKey": false,
            "direction": "forward",
            "indexBounds": {
              "testInt5": [
                "(364.0, inf.0]"
              ]
            }
          }
        },
        {
          "stage": "FETCH",
          "filter": {
            "$and": [
              {
                "testStr2": {
                  "$eq": "juliet"
                }
              },
              {
                "testInt5": {
                  "$gt": 364
                }
              }
            ]
          },
          "inputStage": {
            "stage": "IXSCAN",
            "keyPattern": {
              "testInt4": 1
            },
            "indexName": "testInt4_1",
            "isMultiKey": false,
            "direction": "forward",
            "indexBounds": {
              "testInt4": [
                "(467.0, inf.0]"
              ]
            }
          }
        }
      ]
    },
    "ok": 1
}
```