University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

Computer Science and Engineering: Theses, Dissertations, and Student Research

Computer Science and Engineering, Department of

Fall 12-4-2015

Dynamic Data Management In A Data Grid Environment

Björn Barrefors
University of Nebraska-Lincoln, barrefors@gmail.com

Follow this and additional works at: http://digitalcommons.unl.edu/computerscidiss

 Part of the Computer Sciences Commons

Barrefors, Björn, "Dynamic Data Management In A Data Grid Environment" (2015). *Computer Science and Engineering: Theses, Dissertations, and Student Research*. 97.
http://digitalcommons.unl.edu/computerscidiss/97

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in Computer Science and Engineering: Theses, Dissertations, and Student Research by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

DYNAMIC DATA MANAGEMENT IN A DATA GRID ENVIRONMENT

by

Björn Barrefors

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor David Swanson

Lincoln, Nebraska

December, 2015

DYNAMIC DATA MANAGEMENT IN A DATA GRID ENVIRONMENT

Björn Barrefors, M.S.

University of Nebraska, 2015

Adviser: David Swanson

A data grid is a geographically distributed set of resources providing a facility for computationally intensive analysis of large datasets to a large number of geographically distributed users. In the scientific community, data grids have become increasingly popular as scientific research is driven by large datasets. Until recently, developments in data management for data grids have focused on management of data at lower layers in the data grid architecture. With dataset sizes expected to approach exabyte scale in coming years, data management in data grids are facing a new set of challenges. In particularly, the problem of automatically placing and deleting data replicas to optimally use grid resources.

This thesis describes a dynamic data management framework to handle automatic replica creation and deletion in a data grid environment. The dynamic data manager uses machine learning to predict data popularity and balance the system for improved end-user performance. We implement the dynamic data manager for CMS, one of the largest high-energy physics experiments in the world, and evaluate the performance of the deployed system.

ACKNOWLEDGMENTS

I would like to thank my advisor, David Swanson. Every time I felt like I was in over my head, David was there to calm me down and keep me moving in the right direction. Also, I would like to thank him for hiring me as a graduate research assistant, giving me the chance to enter the world of distributed computing.

I want to thank my unofficial advisor, Brian Bockelman. I appreciate his guidance and excellent technical support during my time in HCC. He is a bottomless well of knowledge and I could never have done this without him.

I want to thank Ying Lu. Ying took me in and showed me the world of research when I was still new to Computer Science. Her kindness and support will forever inspire me to become a better person and help others like she has helped me.

I also want to thank all the people in the CMS experiment that have helped me during this project. Without their dedication to curing my lack of knowledge in anything CMS related I would never have made it this far. I would especially like to thank Christoph Paus and Maxim Goncharov for giving the opportunity to work on this interesting project and letting me create my own solution. Also, I would like to thank Nicolo Magini for his prompt replies to my never ending questions.

GRANT INFORMATION

This work was partially funded by NSF grant PHY-1104664.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A data grid is a geographically distributed set of resources providing a facility for computationally intensive analysis of large datasets to a large number of geographically distributed users [13]. Data grids are widely used within many scientific fields such as high-energy physics, earth sciences and climate modeling which all perform computationally intensive analysis on large datasets.

Unlike High Performance Computing (HPC) where the execution time of a single job is priority, High Throughput Computing (HTC) focuses on finishing as many jobs as possible during a longer period of time. Toward this goal, we strive for optimal utilization of storage, network and computing resources. As the amount of data is expected to approach exabyte scale in coming years [9] proper data distribution strategies will become increasingly important to achieve optimal data placement. A common higher level data management strategy is to request data on an as needed basis using either replication or remote streaming. When a site is full, a local system administrator will manually delete datasets. There are several issues with this approach (which will be exacerbated as we approach exabyte scale):

- **Decreased throughput**: Local system administrators have no system-wide

vantage point to properly balance the system when manually distributing and deleting data without a system wide resource picture available. This can cause decreased performance on sites containing popular datasets. Such decrease in performance include increased queue times for jobs and decreased throughput.

- **Slow turnaround times**: Users in today's scientific community are demanding instant access to any data at any time [9]. Current data management solutions cannot ensure users instant access to any data if there are insufficient replicas.

- **Inefficient use of storage resources**: With the increased amount of data, decisions to replicate data needs to be made with improved system wide resource utilization as the ultimate goal.

- **Inefficient use of human resources**: When grids reach exabyte scale it won't be feasible to have system administrators manually decide on which data to delete for improved system-wide resource utilization.

In this thesis, we design a generic dynamic data management framework to handle these issues based on common data grid architectures. We furthermore implement the dynamic data manager for one of the worlds largest scientific experiments, the Compact Muon Solenoid (CMS) experiment [12]. The CMS detector is one of four major detectors located on CERN's Large Hadron Collider (LHC) and is part of the World-Wide LHC Grid (WLCG) [28]. The experiment currently hosts approximately 25 petabytes of analysis data on disk at over 60 sites around the world. Datasets range from gigabytes up to several hundred terabytes. In the CMS experiment, datasets are collections of events where each event can be data from a recorded particle collision in the detector or a simulation. Events in the same dataset share a set of logical physics properties which are analyzed together. The amount of data and geographical

distribution of this data makes the CMS experiment part of the WLCG an excellent example of a data-intensive scientific grid.

Most data grids today are following the "hourglass" model, a layered approach where each layer is a grouping of components with similar tasks. The "neck" of the hourglass consists of the layers which are used by and uses most of the other layers [24]. In data grids, these layers are usually the connectivity and fabric layer (see Figure 2.1). Due to the crucial nature of the neck layers, significant work has been done on these layers, such as development of advanced data management systems to keep track of data and handle replications [3] [26] [36] [43]. Not as much attention has been given to data management in the collective layer, such as automatic data distribution and deletion strategies [30] [11]. The goal of this thesis is to lay a foundation in dynamic data management in any data grid environment. Using data collected from the fabric and connectivity layer to better understand data usage and user behavior, we can implement system-wide data placement strategies to improve resource utilization. The dynamic data management framework described in this thesis uses machine learning to predict dataset popularity to preemptively balance the system. The goal is to predict future dataset popularity to be able to replicate data ahead of time. A greedy balancing algorithm is used to distribute datasets in the grid to improve resource availability.

To evaluate this work, we run simulations on usage data collected from the CMS experiment. From this, the system balance and accuracy of popularity predictions was measured by looking at the number of replicas compared to dataset popularity for a set period time. Evaluation results are presented in Chapter 5.

Chapter 2

Background

This chapter introduces the layered architecture on which most grids today are based [24]. We extend this concept and describe how these layers are implemented in the WLCG and where dynamic data management fits in this model.

2.1 The Grid Architecture

2.1.1 The Hourglass Model

We model our grid architecture based on a layered approach [24]. Components in the same layer share some common characteristics and each layer in the grid can be built on top of the lower layers as shown in Figure 2.1. One assumption about the layered grid architecture is it follows the "hourglass" model [17]. In a general hourglass model, one or more layers depend on and is used by most of the other layers. These layers are called the "neck" of the hourglass. They are considered important as they form the basis for other layers. In our work the neck consists of the fabric and connectivity layer (see Sections 2.1.2 and 2.1.3).

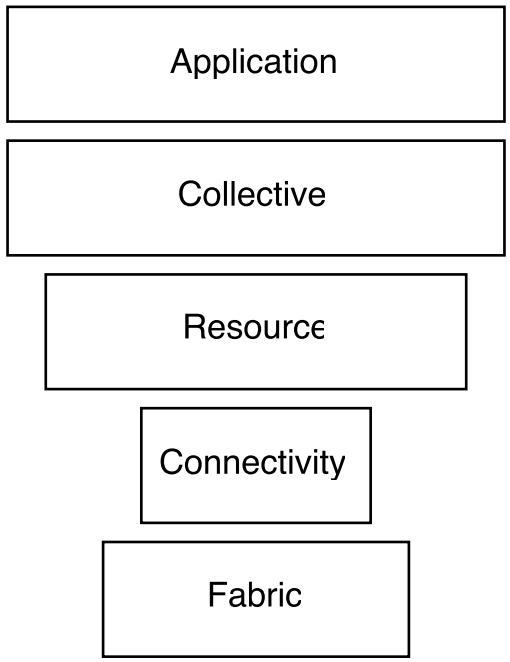

Figure 2.1: Layered grid architecture

2.1.2 Fabric Layer

The fabric layer consists of any shared resource such as computational, storage and network resources [24]. The fabric layer handles resources on an individual level and not with the connectivity or optimization of the system as a whole in mind.

2.1.3 Connectivity Layer

The connectivity layer enables exchange of data between fabric layer resources by implementing core communication and authentication protocols required for grid specific network transactions [24]. Without the connectivity layer resources are simply individual pieces which cannot communicate with each other. The connectivity layer is essential for resources to act as a grid in an efficient and secure manner.

2.1.4 Resource Layer

The resource layer takes the secure exchange of data in the connectivity layer and extends it to a higher level initiation, control and monitoring of shared operations on single resources [24]. There are two main tasks for the resource layer: gather state information about individual resources and negotiate access to single shared resources. Protocols in the resource layer work with individual fabric layer resources and also do not care about the global state of the system.

2.1.5 Collective Layer

The global state of the system is the responsibility of the collective layer. It focuses on the interactions between resources. Unlike the fabric and resource layer, the collective layer is not in the "neck" of the hourglass. Responsibilities of the collective layer include Virtual Organization (VO) management, allocation, scheduling and brokering of resources among users, monitoring and diagnostics and data management. Data management services should support the management of storage, network and computing resources to maximize data access performance with respect to metrics such as response time, reliability, and cost [26] [3]. However this can be approached in different ways: in [26], the focus is on the management of a universal namespace, efficient data transfer between sites, synchronization of remote copies, and wide-area data access and caching. Data management is considered the problem of actually performing the replication and keeping track of replicas across the grid. In [3] however, focus is on the mechanisms to accomplish these tasks.

2.1.6 Application Layer

The application layer is the final layer in the grid architecture considered here and it comprises of user applications in the grid environment. In our use cases, such user applications are mainly analysis jobs. We seek to improve the performance of these applications by improving underlying layers.

2.1.7 The Data Grid

The Data Grid is a specialization of a grid that is very commonly used within scientific computing, focused on processing large datasets. Many scientific disciplines have started generating datasets from the scale of gigabytes the terabytes. The number of users and the geographical distribution of users have significantly increased causing a demand for new grid data management solutions. Four principles are suggested for successful data management in a data grid [13]:

- **Mechanism neutrality**: Keeping data management architecture as neutral as possible of low-level mechanics such as data transfer, metadata storage and any other interfaces which deal with handling data on a lower level. Here, we create an abstract layer database to manage data to generate an overall picture of the status of the data grid regardless of collection method, making it possible to deploy for a wide variety of data grids.

- **Policy neutrality**: Policy neutrality means that design decisions such as replication policies are kept open to users for substitution via application specific code. However, defaults should still be provided. This work improves the default policies while users can still replicate datasets as needed.

- **Compatibility with existing grid infrastructure**: Many tools are available

for grid computing. The most well-known is the Globus Toolkit [23], used for authentication. A new data management system should use these tools to ease integration of new solutions into the existing system.

- **Uniformity of information infrastructure**: A new data management solution should use existing information systems to detect system and resource status. Here, we use existing services to collect metadata, dataset usage, and current system status. These are covered in greater detail in Section 2.2.

Based on these points, [13] suggest a more specific layered approach than the previous grid layer architecture. We divide the components into high and low level, and generic and data grid components. We add two components, *Dynamic Data Management* and *Usage Data Aggregation*. The job of a dynamic data manager is, based on current system status, create and delete replicas to improve future resource utilization. The usage data aggressor collect data about user application behavior. The full data grid architecture and corresponding CMS services is shown in Figure 2.2.

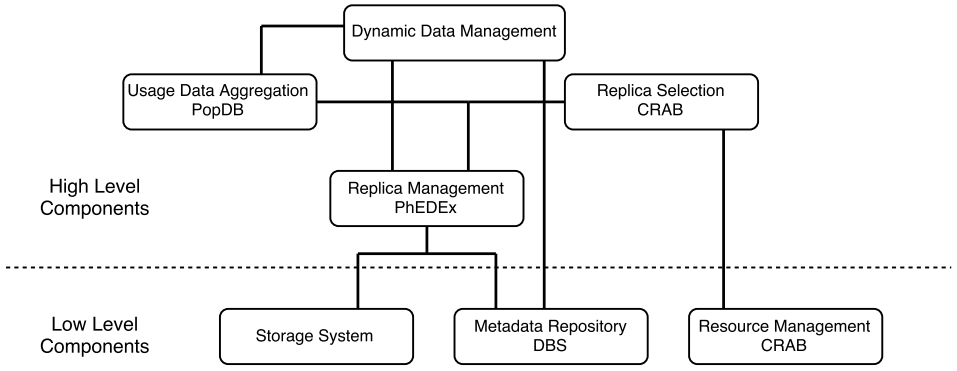

Figure 2.2: Major components and structure of a data Grid with corresponding CMS services

2.2 The CMS Experiment

2.2.1 Background

The Compact Muon Solenoid (CMS) detector is one of four major experiments located on CERN's Large Hadron Collider (LHC) [12]. The LHC boosts particles to close to the speed of light and make them collide with each other. Traces from these collisions are picked up by detectors, such as the CMS detector, with a goal of determine properties of particles. These collisions generate massive amounts of data; most is discarded in hardware immediately after the collision. Data not discarded is sent to a data center for digital reconstruction and stored as a collision event. During a run, millions of these events may be recorded. Each event can be up to 2MB [12]. Events with similar signatures are bundled together into datasets. Each dataset may be as large as a couple hundred terabytes. During one year, the CMS experiment generates tens of petabytes of data [28]. This data is made available for analysis by more than 3500 scientists, engineers, and students from 186 institutes in 42 countries [16].

Based on the definitions in Section 2.1, it is clear that the CMS experiment is a perfect fit for using a data grid. Indeed, the CMS experiment is part of the Worldwide LHC Computing Grid (WLCG). The WLCG was built as a distributed computing infrastructure for the storage and analysis of data from the four major LHC experiments [28]. Each experiment has independent resources within the grid and manages data and resources using mainly its own software and services. In the CMS experiment, data is distributed across 4 "tiers", Tier 0 through 3. Tier-0 and Tier-1 sites are mainly used for backup of data on disk and magnetic tapes, little analysis is done on these sites. Tier-3 are small sites used mainly locally by the institutions which are hosting them. The majority of analysis jobs are submitted to Tier-2 sites. There are currently more than 60 Tier-2 sites serving the CMS experiment. In the following

sections we will go over the core services and components used in the CMS experiment which are in some way utilized by the dynamic data manager. These are further explained in Section 3.1.

2.2.2 Metadata Repository

The CMS experiment keeps track of dataset metadata using the CMS Dataset Bookkeeping Service (DBS) [2]. The DBS catalog keeps dataset metadata such as run number, luminosity section number, algorithms used to process data, group name and mapping of datasets to files.

2.2.3 Replica Management

PhEDEx, short for Physics Experiment Data Export is the replica manager used by the CMS experiment [44]. PhEDEx initiates data transfers all the way down on a file level and tracks file replicas locations.

2.2.4 Replica Selection

When a user in the CMS experiment would like to analyze data, they use the CMS Remote Analysis Builder, or CRAB for short [14]. CRAB abstracts the underlying complexity of accessing CMS resources in the WLCG for end-users. Analysis workflow is submitted in the global job queue. CRAB determines the input data location and the global queue schedule the tasks at the best available site. In the case that there are no available CPU‘s at the possible sites CRAB can choose to stream data to another site with available CPU‘s or simply queue the jobs [9].

2.2.5 Popularity Aggregation

For accurate system status, user behavior needs monitoring. In CMS all activities are monitored and stored by Dashboard (DAS) [5]. Furthermore, this information is aggregated in an abstract database called the CMS Popularity Service (PopDB) [33].

2.3 Popularity Prediction

Machine Learning algorithms can be divided into classification, regression, and clustering algorithms [8]. Classification is the most common where given a set of features the algorithm tries to predict which class an object belongs to. Regression is very similar but instead of approximating a class a regression algorithm attempts to approximate a value. Finally, clustering attempt to bundle objects together based on a set of features. Clustering algorithms are a so called unsupervised learning algorithms, meaning no training data exists. Classification and regression algorithms however are considered supervised learning, training data with known results are used to train the algorithms.

2.4 Thesis Overview

The rest of this thesis first discusses the technology to create a dynamic data manager as well as previous work done in predicting popularity and dynamic data management in Chapter 3, and then describes the design and implementation in Chapter 4. Chapter 5 describe the evaluation of the popularity prediction models and the dynamic data management performance and presents the results. Chapter 6 presents conclusions and describes future work.

Chapter 3

Related Work

3.1 CMS Data Management Services

This section present previous work done in the CMS experiment to manage data.

3.1.1 CRAB3

In a data grid environment, end-users access computing and data resources transparently with no specific knowledge about the underlying complexities of the system. In CMS this abstraction layer is the CMS Remote Analysis Builder (CRAB) [14]. CRAB has been in production since Spring 2004 and is currently at its third iteration. CRAB takes end-user workflow requests and handles the management and scheduling of these using an analysis system composed of several services [14]. These services handle numerous tasks including receive and inject user requests, track and manage the requests in a global queue using HTCondor [47], prioritize requests and translate user requests into jobs. The workflow data is stored in an Oracle database. status of resources at sites are collected from so called machine ads. These are matched with class ads, these are jobs advertising the resources required to execute the job.

Class and machine ads are central to HTCondor matchmaking [15] and can be directly accessed using a HTCondor python module. Machine ads may not accurately represent the number of CPUs available at a site, most sites dynamically allocate CPUs to the CMS experiment on an as needed basis. The current need may be less than the total resources available to CMS. For accurate system status, a solution to this problem is presented in Section 4.3. CRAB reports usage data to the CERN Dashboard service [5].

3.1.2 PhEDEx

Traditionally data transfer management was an expensive activity; tasks such as ensuring data safety, large-scale data replication and tape migration/stage of data have relied on manpower [44]. As dataset sizes reached petabyte scale the manpower required scaled linear but available manpower remained flat. To solve this problem, the CMS experiment developed Physics Experiment Data Export (PhEDEx) [44] to automate these tasks. PhEDEx makes abstracts replication and deletion of data from the underlying infrastructure. Tasks such as file replication, routing decisions, tape migration, and file pre-staging are all dealt with using a suite of agents running on each site. Interaction with PhEDEx is done through a RESTful web API.

3.1.3 DBS

The CMS Dataset Bookkeeping Service (DBS) [2] provides access to a data catalog of all event data for the CMS experiment. Dataset information includes dataset size in bytes, number of files, physics group name, data tier, creation date, dataset type and mapping of datasets to files. The data in DBS is recorded from all Monte Carlo generations and analysis steps and can be used to track data back to the RAW

data or Monte Carlo generation it was originally generated from. However, these attributes can also be used by for improved data popularity predictions. Dataset information for CMS users is provided by the CMS Global DBS using an Oracle database. Authentication is handled using standard X.509 grid certificates and the database is accessed through a web API.

3.1.4 CMS Popularity Service

The CMS Popularity Service (PopDB) [33] is a popularity aggregation service made available for other CMS services to improve system performance. PopDB is developed to interface with different data sources and is currently gathering data from CRAB [14] and Xrootd [20]. Data gathered is stored in an Oracle database and made available for external systems through a web API and visual frontend. PopDB abstracts all the low level data collection from other services following the same grid architecture properties as this work.

3.2 MongoDB

In recent years, NoSQL databases have received significant attention due to their relative simplicity in design and easy scaling with clusters [29]. This is of great use for managing unstructured data. Instead of SQL, NoSQL often require no predetermined schema. Data is often treated as documents of data that are added to the database and can be accessed using key/value pairs. The majority of processing is moved to the data retrieval stage. Using MapReduce [19] style techniques, users can extract data over multiple machines in a cluster. One popular NoSQL databases, MongoDB [34], has a powerful aggregation framework that is a MapReduce style query language. MongoDB has tight integration with Python [40] using PyMongo [35] and

the key/value pair and array style data of MongoDB maps well with Python dictionaries and lists, this is important based on our implementation which is described in Section 4. The unstructured style of NoSQL databases makes MongoDB dynamic and easy to use with any grid. Finally, as we fetch data from other services over network, it is useful as a local cache of these queries using indexes. The MongoDB cache is described in greater detail in Section 4.1.2.

3.3 Other Dynamic Data Managers

Dynamic data management in a data grid environment is a fairly new concept with similar work done mainly for experiments on the LHC at CERN. Some early work was done without popularity prediction [30] [11]. Both of which approach dynamic data management as a caching problem where used data is kept in a cache for quicker concurrent accesses. However a different approach using machine learning to predict popularity has gained interest as the area of Machine Learning has received a lot of attention lately due to the discovery of new algorithms and the increase in data and cheap computational resources [27]. The advantages of using popularity prediction instead of just caching is improved decision of what data need increased availability, for improved storage resource utilization, and the possibility to increase data availability preemptively. Some work has been done on the merits of using data popularity predictions in multimedia content caching [22] to see if accurate predictions would improve caching performance. The authors found that perfect popularity predictions can give caches a relative performance gain of up to 18% making it a valid tool to use in online content caching. During the last two years two of the other experiments on the LHC at CERN have done some work on dynamic data management in a data grid using popularity prediction.

3.3.1 ATLAS Distributed Data Manager

The ATLAS [1] experiment, one of the other major experiments on the LHC at CERN, has developed a couple dynamic data managers [31] [6] on top of the ATLAS Production and Distributed Analysis System (PanDA) [32]. PanDA is a workload management system similar to CRAB, it handles the distribution and scheduling of user jobs at the ATLAS experiment. PD2P [31] was the first dynamic data manager developed for PanDA to better utilize ATLAS grid resources. PD2P had no popularity prediction component. Two years later the ATLAS Distributed Data Manager (DDM) [6] was developed.

The ATLAS DDM system attempts to forecast data popularity using a machine learning model called Artificial Neural Networks (ANN) [25] to give the data distribution manager time to delete and replicate data [7]. Their approach uses a regression algorithm to predict future number of dataset accesses. One of the downsides to ANN is the large amount of training data required for accurate models, something that will not always be available. The data redistribution consists of two parts, the cleanup process which deletes unpopular replicas and the replication process which creates new replicas of popular data. The replication process uses a total number of accesses per job slot as a metric to redistribute data as evenly as possible among sites. Number of accesses per sites are normalized with respect to the number of job slots for each site. Only datasets which have been unpopular for a set amount of weeks are considered for deletions, though at least one replica is always kept to avoid data loss. For current datasets at least two replicas are required on disk. The popularity prediction pre-filter datasets with very little usage to decrease prediction time. Furthermore, to increase accuracy of the prediction different models are trained for datasets of different core types. Initial evaluation looked at evolution of waiting time.

Results indicated that in general the system performance would improve as more data was moved up until 1000TB for basic replication strategies. Above 1000TB of moved data more advanced replication strategies proved to be advantageous. The authors also found that most jobs were submitted during the middle of the week while very few jobs were submitted during the weekends. The ATLAS Distributed Data Manager uses a similar approach to dynamic data management as in this work, but uses different metrics for dataset and site popularity and different algorithms for popularity prediction and system balancing.

3.3.2 The LHCb Data Storage System

The LHCb [4] experiment is another one of the major experiments on the LHC at CERN. The experiment recently developed a dynamic data manager using popularity prediction. The LHCb experiment have less disk space than the ATLAS and CMS experiment. Therefore, their dynamic data manager focus more on deletion of data, including complete removal of data from disk with a backup copy kept on tape. The LHCb data storage system uses a gradient boost classifier [25] to predict when a dataset will never be used again. Based on these predictions datasets are completely removed from disk. The data storage manager uses Nadaraya-Watson smoothing [25] and rolling mean values [25] to predict popularity of datasets which are predicted to be used again. Creation and deletion of new replicas are decided using a cost function. The cost function considers predicted popularity, cost of removing dataset from disk and cost of keeping a low number of replicas. Though tape storage is a lot cheaper than disk, staging data back to disk from tape is a slow process. Having a “false positive” for data expected to not be used again can be very costly, often taking hours or days. The evaluation of their algorithm showed that the number

of false positives could be significantly reduced compared to a least recently used algorithm. As the CMS experiment has more disk space than LHCb this is not a situation expected to occur regularly.

Chapter 4

Design and Implementation of the Dynamic Data Manager

This chapter describes the design of a dynamic data management framework and the implementation of a dynamic data manager for the CMS experiment. The design follows all four data grid properties described in Section 2.1.7 for successful grid data management. The dynamic data management framework is split into three parts, described in Sections 4.1, 4.2 and 4.3. The full framework is implemented using Python [40] and is installed as a package on the local system using the standard package builder in Python, distutils [38]. The installer also includes a unittest suite [42]. The package is highly configurable to fit the need of the local installation. Options such as maximum number of threads, storage thresholds, URLs to services and many other features are all easily managed using config files. These are automatically imported and parsed using ConfigParser [37]. Finally, all activities are logged in rotating log files on the local file system using the Python logging module [39].

4.1 Abstraction Layer

The first part of the dynamic data management framework is the abstraction layer. The abstraction layer is designed to handle collection, storage and access of data. It is called the abstraction layer as it abstracts all the grid specific tasks from the rest of the dynamic data manager to comply with the principles described in Section 2.1.7. The separation provided by the abstraction layer is clearly shown in Figure 4.1.

Information collected in the abstraction layer includes dataset properties, dataset popularity, replica distribution, site status and site performance. The abstraction layer keeps local data storage updated with current system information using regular update calls to all grid specific services. Data is reorganized and stored in a local MongoDB database for quick access. All service calls are done over the network which is very slow. To avoid waiting for network calls all data calls are done in parallel using the Python threading module [41]. All services are accessed using standard VO management policies which are described in Section 4.1.3.

4.1.1 Grid Services

The following sections describe what data is collected from which CMS service. The services themselves are covered in more detail in Section 3.1.

4.1.1.1 CRAB3

CRAB3 is the workflow manager used in the CMS collaboration and keeps track of all jobs. However, we are interested in the information it keeps about the performance of each site. All sites need to advertise the amount of CPUs they have available. This information is highly dynamic as many sites will allocate CPUs on an as needed basis, since most sites are at institutions delivering computational resources to many

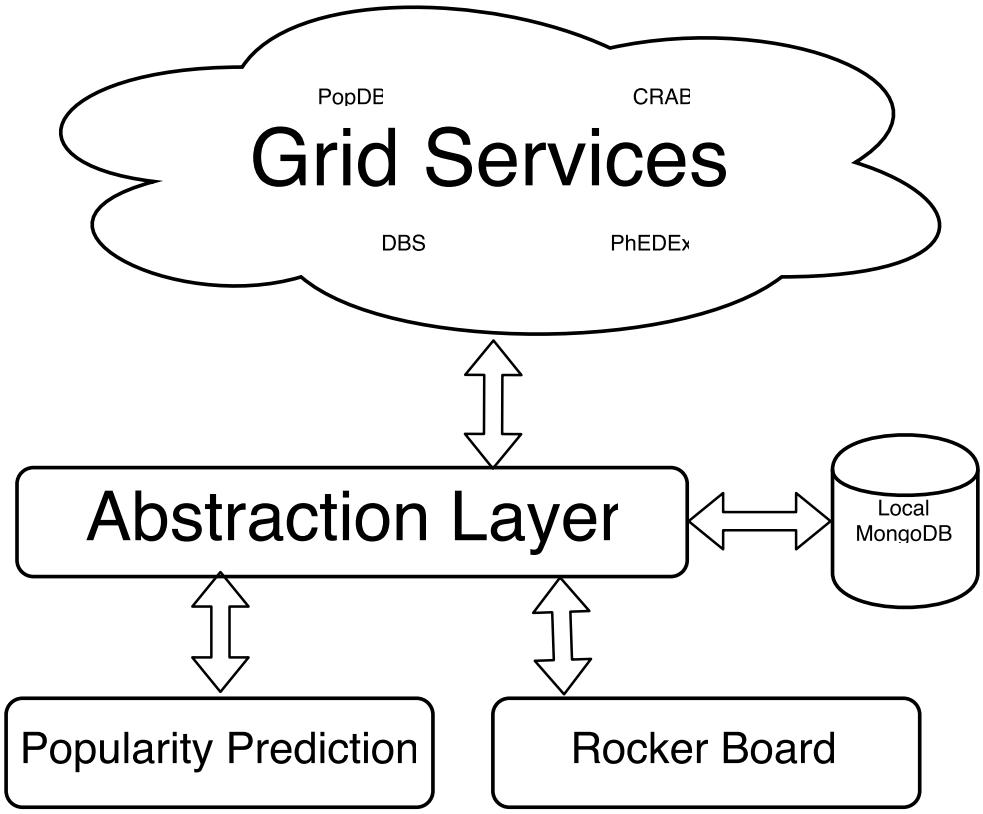

Figure 4.1: Abstraction layer workflow

scientists outside of the CMS collaboration. Since computational resources are too expensive to not be used, CPUs not used by the CMS collaboration will be allocated to external users. If a CMS user is requesting resources, however, external jobs will be stopped and the CPU re-allocated to CMS. To handle this we developed a script which collects the current number of CPU‘s advertised for each site once every hour. Data is kept for 30 days and the maximum value is used as an estimation of the current site performance. The number of days for which data is stored can easily be increased or decreased by modifying a single variable.

4.1.1.2 DBS

DBS is the data bookkeeping service for CMS. Data collected from DBS is general dataset information that can be of used as features in machine learning algorithms for more accurate results. This data includes data tier, number of files, dataset size, physics group and dataset type. Not all of these features are currently used but keeping them makes for a very flexible system where machine learning algorithms can easily be extended to use additional features.

4.1.1.3 PhEDEx

PhEDEx is the CMS replica manager. It keeps track of replicas and handles replicating and deleting data. Based on this data a system wide image of which datasets have replicas at which site is updated on a daily basis. PhEDEx can even further reduce the amount of lower level communication needed in the dynamic data manager. Using its built in data transfer and deletion tools the dynamic data manager can safely send replication and deletion requests to the PhEDEx API without communicating with underlying protocols.

4.1.1.4 PopDB

PopDB is a data usage aggregation service that handles the lower level collection of number of file accesses and CPU hours spent analyzing a certain dataset. PopDB also collects how many users are running jobs on a specific dataset. This data is collected but not used right now as it is not uncommon in physics groups to have one user submit all group jobs. However, other data grids may have other policies.

4.1.2 Local Data Storage

Having local data storage is important for two reasons:

- Quick data access
- Uniformity of information infrastructure

All services are accessed over the network making data accesses very slow. To minimize the impact of this, data can be cached in a local database. However, just caching data only reduce access time but does not handle uniformity of information infrastructure. Services available will be different for every data grid. Therefore, data collected is also stored separately from the caches in a restructured format for standardized data access within the dynamic data manager. Our implementation uses MongoDB for the local database. Caching in MongoDB is implemented such that each service has its own cache where the keys are an aggregation of the API call and the parameters passed. Such a system enables easy checks for cached data when doing service calls. The cache is easily maintained using indexes where data is automatically deleted if creation time is older than a certain limit. The MongoDB aggregation framework is heavily used for quick popularity calculations over thousands of datasets using MapReduce style queries.

4.1.3 Authentication

Most grid services require authentication to make API calls. This is handled using proxies. Proxies can be automatically generated by the Globus Toolkit, a common tool for Virtual Organization (VO) management in data grids. Such design decisions keeps our framework compatible with general grid infrastructure as described in Section 2.1.7.

4.2 Popularity Prediction

The popularity prediction module is in charge of predicting dataset popularity using previous usage patterns. To follow the policy neutrality described in Section 2.1.7 we designed a generic ranking class. This class can be extended by any kind of ranking algorithm, making it easy to implement many different algorithms and compare the results. The implementation we used classifies datasets as becoming more popular, less popular or staying unchanged. Datasets classified as increasing or decreasing in popularity are passed to a regression algorithm to predict the future magnitude of popularity. The whole popularity prediction process is shown in Figure 4.2.

4.2.1 Data Preprocessing

To properly train a machine learning algorithm it is important to have both training and test data for training and evaluation of the algorithm. Accurate training data can significantly affect the performance of both classifiers and regressor. To generate as accurate training and testing data as possible we developed a tool to visually display popularity metrics using D3.js [10]. Allowing for easy classification of when a dataset is becoming more or less popular as a trend and when increase in popularity is simply a short spike of usage that should not be treated as a long term increase in popularity. Using these classifications, data from a marked date and a week forward is collected and normalized as features. These are then paired up with either 0, -1 or 1 based on the classification. For the regressor the features are the same but the prediction is an average popularity for one week starting 7 days after the given date. A screen shot of the data classification tool is shown in Figure 4.3. A clear decrease in popularity can be seen during the first three weeks of the collected data for this dataset.

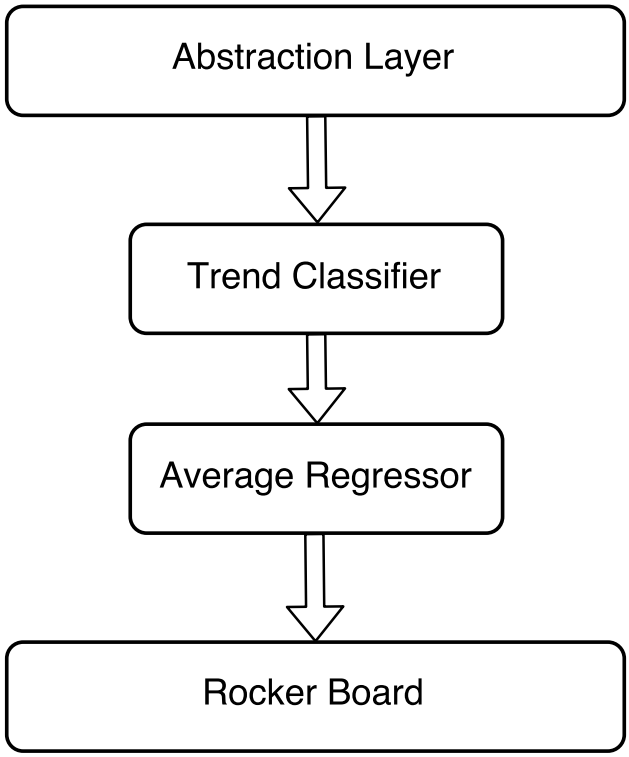

Figure 4.2: Popularity prediction workflow

4.2.2 Popularity Trend Classification

To find the best classification algorithm we implemented two different classifiers. These are both presented in Section 5.1 together with the results.

From these we chose a common machine learning algorithm called Support Vector Machines (SVM) [18]. Support Vector Machines are very popular for both linear and nonlinear classification problems for their relative power compared to ease of use.

The specific implementation use a Support Vector Classifier (SVC) with a polynomial kernel and $C = 0.5$. The SVC is trained using data generated from the data

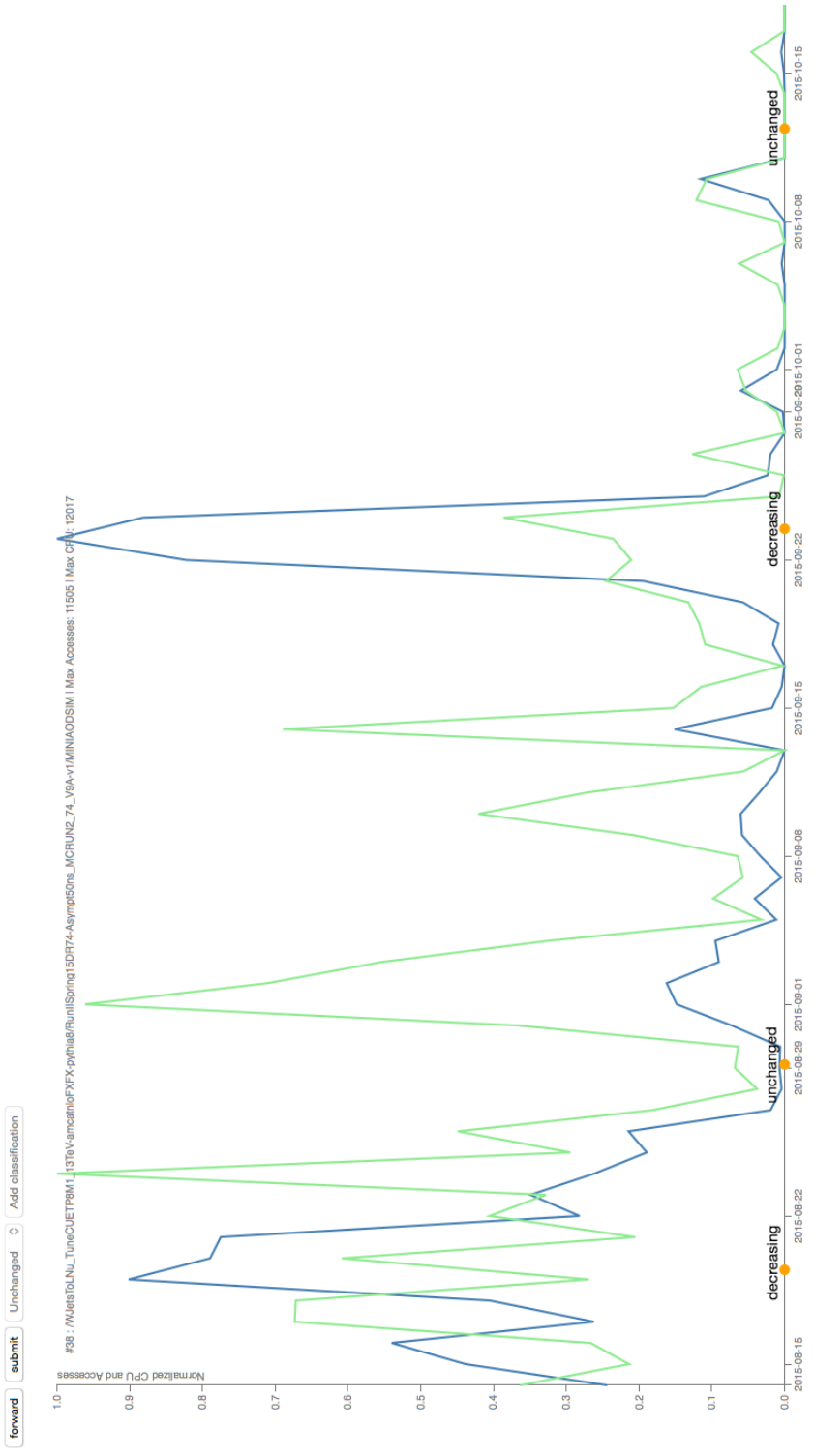

Figure 4.3: Data preprocessing tool

classification and preprocessing tools. Features is the same as when trained the classifier. The first seven days of popularity data for a section of increased, decreased or unchanged popularity and the classifications are any one of these three options of popularity. Popularity is defined in Equation 4.1 Example training datasets are shown in Table 4.1. During data preprocessing it became obvious different data tiers have different access patterns. This is leveraged using separate classifiers for each data tier.

$$popularity = \log(n\_cpu\_hours * n\_accesses) \tag{4.1}$$

Class	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
increasing	2.3	5.2	8.9	6.6	7.4	8.0	9.1
decreasing	10.5	4.2	8.4	2.4	3.1	2.8	1.4
unchanged	0.0	1.1	2.1	0.0	0.3	1.3	1.2

Table 4.1: Example machine learning training data for trend classifier

4.2.3 Average Popularity Regression

Predicting a dataset will become popular is a good start but is not sufficient to accurately balance the system. To balance the system, each replica needs to have about the same amount of accesses and CPU hours per gigabyte. Therefore, it is important to not only classify general popularity trends, but also estimate how popular a dataset will be during the upcoming weeks.

To do this we implemented a regressor. The reason for this choice was the accuracy of the regressor. Training a dataset popularity model for all datasets in the whole system and for their whole time span is inefficient and hard to accurately train. Many datasets are rarely used and may have next to no accesses for months at a time. Training a classifier on all this data is not useful for estimating increasing

and decreasing popularity. Furthermore, data popularity in a scientific data grid environment can be sporadic. We wanted to train the classifier to not predict increased popularity for short spikes of popularity. This is because in a data grid environment network capacity is limited. Creating replicas for short spikes of popularity is a waste of network resources. By using our first classifier we try to eliminate a large amount of the datasets which have no change in popularity or simply have a short spike and should not be considered for replication. Such behavior is called so called overfitting and is a common issue in machine learning [8]. This makes it much more possible to build an accurate model for those cases where a dataset is in fact increasing in popularity and will stay popular for some time.

The average popularity regressor is trained by taking the average dataset popularity for the following week after an increase or decrease. A log function is applied to the values to keep a consistency of scale with the features which are the same as in the trend classifier. The algorithm used is a Bayesian Ridge Regression algorithm [8]. Based on our tests the SVM based algorithms can be very powerful in classification problems while Bayesian probability based algorithms are better for solving time-series regression problems. The accuracy of the average popularity regressors is presented in Section 5.1.

Both machine learning algorithms are implemented using the scikit-learn toolkit for Python [46].

4.3 Rocker Board

Rocker Board module is the system balancing algorithm. The algorithm uses information produced in the other two main modules and makes the final decisions on which datasets should be replicated and which replicas can be deleted. Furthermore, the

rocker board algorithm does site selection for suggested replications and deletions. In short, the rocker board algorithm makes sure the data grid is balanced for improved resource utilization

The idea of the rocker board algorithm is simple and, as the name implies, is based on the a rocker board. A rocker board is a simple balancing tool consisting of a circular board balancing on top of a hemisphere (See Figure 4.4). The goal is to balance the rocker board such that no part of the actual board touches the ground. The problem being that if too much force is put on one side the board starts tipping and quickly reaches the ground unless appropriate adjustments are made to redistribute the force across the board. This concept can be applied to a data grid too. The goal is to distribute jobs across the system such that one site does not get overloaded. This could happen if several very popular datasets only have a few replicas on the same sites. These sites become overloaded with jobs and the system starts tipping until finally jobs are put in a queue, one part of the board is now touching the ground. For a visual description see Figure 4.5.

Figure 4.4: Rocker board

The strategy of the rocker board algorithm is to rank all datasets based on popularity predictions. The predictions are then mapped to a suggested number of replicas.

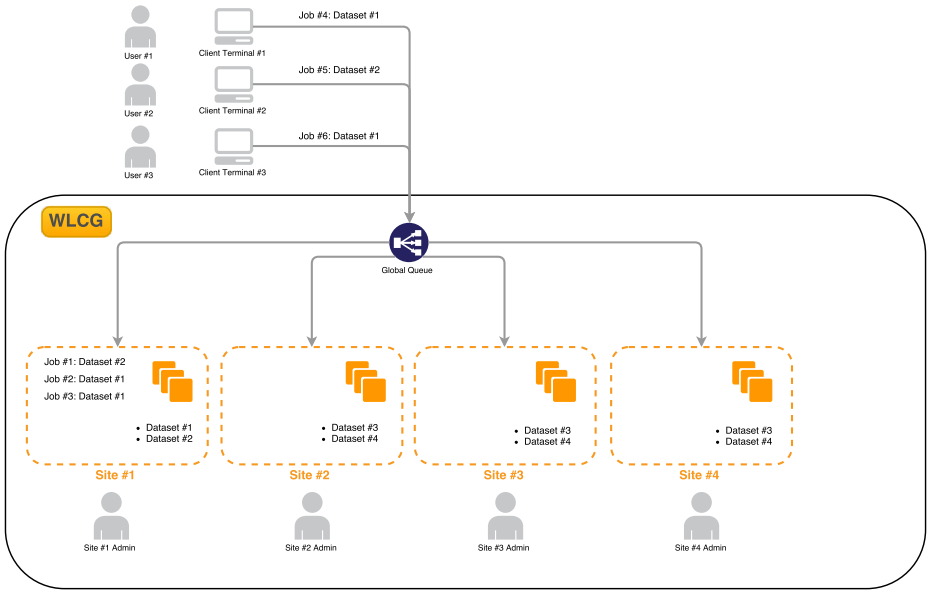

Figure 4.5: Two popular datasets have only one replica at the same site. All users submit jobs to these datasets and that site becomes overloaded while computational resources on the other two sites are not used at all.

Based on the suggested number of replicas the rocker board algorithm will create and delete replicas to achieve these values.

The beauty of this algorithm is that the popularity rankings can be based on any kind of data and works for both past, current and future data. If there is no historical data to train a machine learning classifier the rocker board algorithm can still be used on the current status of the system. If, however, prediction data exists the algorithm will balance the system before it has even started tipping. Anyone who has ever stood on a rocker board knows that once it starts tipping it takes a lot more effort to balance it again rather then when you are just standing on it when it is already balanced. By predicting popularity the system can be kept balanced making data available to users as it is needed instead of as a reaction to lack of data.

4.3.1 Dataset Rankings

Dataset rankings are calculated based on the popularity data from the popularity prediction module. Popularity values are normalized based on dataset sizes as we are trying to balance the system in terms of number of accesses and CPU hours per gigabyte. These new popularity values are then mapped into the discrete range $(min\_replicas, max\_replicas)$. $min\_replicas$ and $max\_replicas$ can be set in the config file following the policy neutrality principle.

From this, dataset rankings are the suggested changes in number of replicas calculated as $suggested\_replicas - current\_replicas$. These rankings are used in the selection process where the highest ranked datasets are selected and suggested for a new replica. The selection process keeps going until $max(rank) \leq 0$ or the maximum allowed gigabytes of transfers per day is reached. The full selection algorithm is shown in Algorithm 1, the site selection part is described in Section 4.3.2.

Algorithm 1 Replica Selection

```
1: procedure REPLICASELECTION(dataset_rankings)
2:     subscriptions ← [ ]
3:     subscribed_gb ← 0
4:     while subscribed_gb < max_gb do
5:         dataset_name ← max(dataset_rankings)
6:         if dataset_rank < 1 then
7:             break
8:         end if
9:         site_name ← GETSITE( )
10:        subscriptions ← (dataset_name, site_name)
11:        subscribed_gb ← subscribed_gb + dataset_size
12:        UPDATESITERANKINGS( )
13:        dataset_rankings[dataset_name] ← dataset_rankings[dataset_name] − 1
14:    end while
15:    return subscriptions
16: end procedure
```

The deletion selection is very similar to the replication process. The dataset

with the lowest ranking is selected and one of the replicas is deleted based on the site selection algorithm. The procedure is continued until all sites are below a certain threshold. If no sites are above the recommended threshold no replicas will be deleted. The deletion algorithm is shown in Algorithm 2. Once again the site selection is described in Section 4.3.2.

Algorithm 2 Replica Deletion

```
1:  procedure DELETIONSELECTION(dataset_rankings)
2:      deletions ← [ ]
3:      site_name ← GETSITE( )
4:      while site_name do
5:          dataset_name ← min(dataset_rankings)
6:          deletions ← (dataset_name, site_name)
7:          UPDATESITERANKINGS( )
8:          dataset_rankings[dataset_name] ← dataset_rankings[dataset_name] + 1
9:          if dataset_name have minimum replicas then
10:             delete dataset_rankings[dataset_name]
11:         end if
12:         site_name ← GETSITE( )
13:     end while
14:     return deletions
15: end procedure
```

4.3.2 Site Rankings

To control sites the system manager keeps two limits in terms of storage utilization. The soft limit is the site utilization goal. However, when the rocker board algorithm is creating new replicas it can fill sites up to the hard limit. This forces out unpopular replicas by pushing new popular replicas to the sites. If a site is above the soft threshold the rocker board algorithm will delete unpopular replicas until the storage utilization is below the soft threshold.

For replica selection, sites are ranked based on three criteria:

- **Popularity**: The popularity is a simple sum of popularity per gigabyte for all datasets at the site times the total storage in gigabytes used by those datasets.
- **Performance**: Performance is the maximum number of available CPUs divided by the total storage at the site.
- **Available storage**: Is a safety guard to make sure sites are not filled above the hard limit. If the site is filled above the hard limit this term becomes 0, otherwise it is 1.

The popularity metric is the metric used for system site balance. The performance component makes sure that not all sites are assumed equal, some sites have a lot more CPUs per gigabyte than other sites, which needs to be taken into consideration. The final rank is calculated according to Equation 4.2. The equation is inverted because sites are selected using weighted random selection where higher values are more likely to be chosen.

$$rank = \frac{performance * available\_storage}{popularity} \tag{4.2}$$

As we can see in Algorithm 1, after each replica selection site rankings are recalculated to make sure a site with unpopular data is not selected continuously.

For deletion selection the sites are ranked in a different way. The idea is that the replication algorithm will take care of accurately distributing data to balance the system. The cleaning procedure is simply used to make sure sites are not above their soft limit. If the selection algorithm did a good job, popular datasets should have been distributed to unpopular sites, meaning there are replicas at the site that can be deleted. As the site is cleaned up to go below the soft limit the lowest ranked datasets at sites above the threshold will be selected. Because of this the site rankings for the

deletion selection is simply the amount of data above the soft limit for each site. If it is not above the soft limit the site will not be considered for deletions; as soon as a site ends up below the limit it will be removed from the rankings.

The rocker board algorithm is evaluated in Section 5.2

Chapter 5

Evaluation

This chapter is divided into two main sections. In Section 5.1 we evaluate the performance and accuracy of the classification and regression algorithms used to predict data popularity. In Section 5.2 we evaluate the performance of our rocker board algorithm and the dynamic data manager implemented for the CMS experiment.

5.1 Popularity Prediction

This section evaluates the accuracy of several popularity prediction algorithms. We implemented two classifiers and two regressors and compared the performance of these on test data generated from the CMS experiment using our data preprocessing tool.

One of the problems we had was a small amount of training data. This is a common problem when training machine learning algorithms, as was the case in this work. At the beginning of May 2015 Run2 started at CERN. In the beginning analysis activity was spiking for certain data tiers to slow down significantly after two months. See Figure 5.1. We decided not to use data from before August 2015 to get more accurate training data, by this time usage patterns had settled down somewhat. Furthermore,

a lot of datasets are used so little and sporadically that when usage increase is detected it will already be unpopular again. We here talk about a couple of days of usage. These datasets are of no use when generating training data of increasing popularity, only to teach classifiers that such short spikes in popularity should be considered unchanged popularity.

To see the impact of increasing training data we ran the classifier on 60% up to 100% of all our training data sets. The results are shown in Figure 5.2. We still think our results were good but believe the accuracy could be increase even more with an increase of training data.

5.1.1 Data Preprocessing Tool

The visual popularity data classification tool ended up having a huge impact on our work. At first, we had problems generating good training data using automated mathematical models. Having this visual tool made it possible for us to intuitively generate more accurate training data for the machine learning algorithms. However, this wasn't the only improvement we got from this tool. When classifying datasets we noticed that different data tiers had specific usage patterns. See Figure 5.3 for example. This insight made us realize that we could make more powerful classifiers and regressors by training a distinct classifier and regressor for each data tier. It also led to some ideas for future work which is further discussed in Chapter 6.

5.1.2 Popularity Trends

Accurately predicting popularity trends in a data grid is one of the main goals of this thesis. To evaluate our popularity trend classifiers we generated a dataset of known classifications using the visual preprocessing tool from Section 4.2.1. A general rule

widely used in machine learning is to use 80% of the data for training and 20% for testing. The first classifier is a Support Vector Classifier (SVC). The full implementation is described in Section 4.2.2. The second is a Naive Bayes classifier using a Gaussian distribution (GaussianNB). We ran two sets of tests on both classifiers. The first kept the number of accesses and number of CPU hours per day as separate features, while the second combined them using multiplication. Due to the massive amount of accesses and CPU hours of some datasets all of these values were reduced by applying a log base ten operation. The results are shown in Figure 5.4 and 5.5.

From this data we can conclude the following:

- For our data the SVC is by far the superior classifier over the GaussianNB classifier which performs significantly worse and was therefore not used in the final design.
- Some data tiers such as MINIAODSIM and MINIAOD are much easier to predict usage patterns for than other data tiers. From this we can see the effectiveness of splitting up the classifiers between data tiers.
- Keeping the number of accesses and the number of CPU hours separate was 0.7% more accurate than combining the two as one feature. However, as we show in the next section, the performance of combining the number of accesses and the number of CPU hours as one feature proved to be significantly better in the popularity regressor. To keep consistency we used the combined single feature in the classifier as well.

5.1.3 Average Popularity

To perform system balancing some sort of estimate of the expected popularity is required. Without actual values the trend prediction itself becomes almost pointless.

We once again used one regressor based on Support Vector Machines and one based on Bayesian probability. The first one is a Support Vector Regressor (SVR) while the second is a Bayesian Ridge Regressor (BRR). Evaluation of regressors is based on the coefficient of determination (R^2) of the prediction [45]. The maximum and optimal score is 1 making a score as close as possible to 1 desired. We tested both regressors the same way as we tested the classifiers. We first ran the tests keeping the number of accesses and the number of CPU hours separate. After that we compared the results to combining the number of accesses and the number of CPU hours. Results are shown in Figures 5.6 and 5.7. Unlike the classifier, the regressors had significant difference in accuracy between the two different features, combining CPU hours and accesses as one feature proved to be much more accurate. Because of this we combined the number of accesses and the number of CPU hours and used that measurement as the feature in both the classifier and regressor in the final design. For the regressor the Bayesian probability based method significantly outperformed the Support Vector Machine approach. Using accesses and CPU hours as a combined feature the Bayesian Ridge Regressor had an average score of 0.943, while the Support Vector Regressor only had an average score of 0.445.

5.2 System Balance

In this section we evaluate our rocker board algorithm. The goal was to better balance the system to decrease user job queue times. However, queue times are heavily dependent on the scheduling algorithm. In our case this is outside of our control as this is done by CRAB. Because job scheduling was outside the scope of this thesis, this metric was not an accurate measurement of the performance of our work. Instead, we focused on measuring the system balance based on old popularity data from the CMS

experiment. We ran our algorithm once a week on old popularity data and recorded the changes in data replicas. For each week we calculated the average popularity per gigabyte on each site in the grid. The popularity was normalized based on site performance.

The simulation starts on an unbalanced system and is then balanced for 90 days by the rocker board algorithm. Figure 5.8 shows the ratio of standard deviation and average site popularity. The ratio is to account for changes in total data usage. For a perfect balancing algorithm the result would approach 0. The expected result was to see a decrease in standard deviation compared to average site popularity. This would imply data usage is getting more balanced over the system as a whole. As we can see in Figure 5.8, this is in fact what did happened in the general case. However, the results show significant fluctuation. This is most likely due to popularity predictions being too reactive to an increase or decrease in popularity. We can in Figure 5.9 see that in fact the LHE data tier had a large spike in number of accesses which quickly reduced down to almost no accesses just as the system balance decreased. In Chapter 6 we further discuss future work to improve popularity predictions.

Figure 5.1: CPU hours consumed in the CMS Experiment before and after Run2 started in May 2015. Source [21]

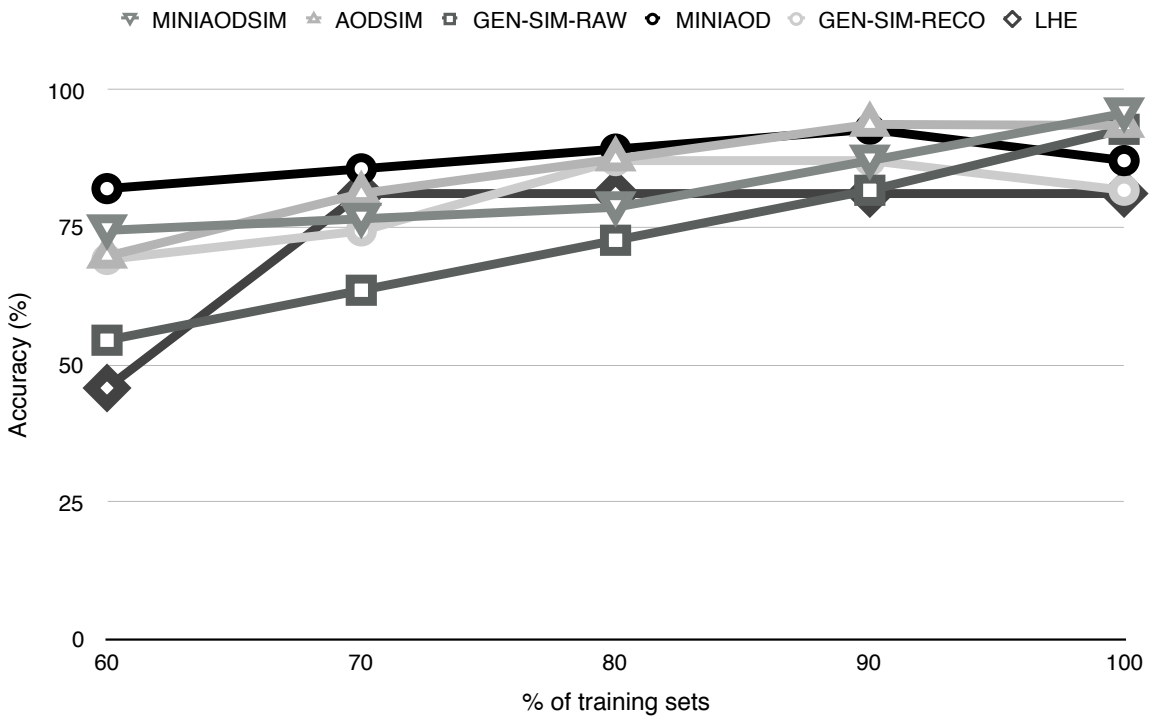

Figure 5.2: Accuracy of Support Vector Machines as number of training sets increase

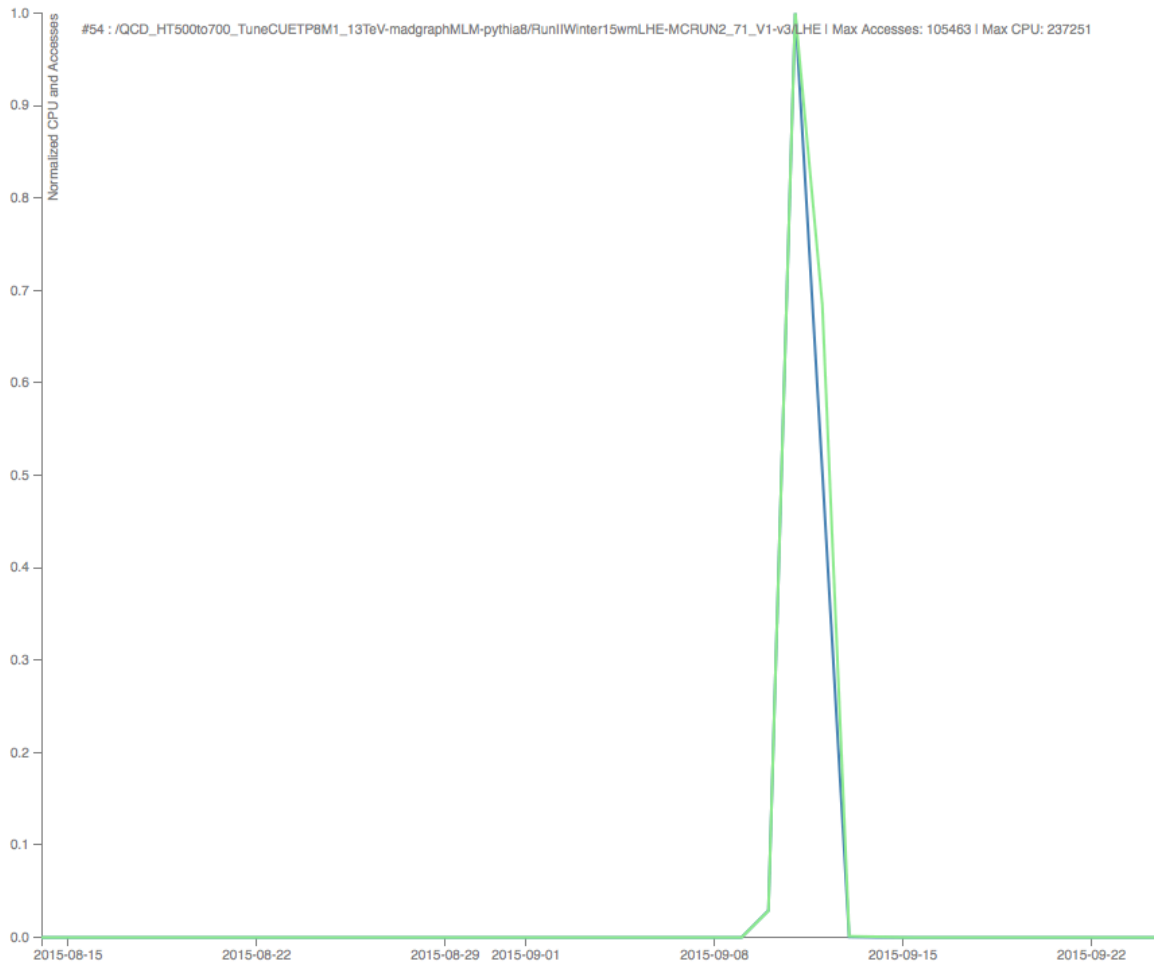

Figure 5.3: While generating training data using the visual preprocessing tool we noticed some data tiers had very specific usage pattern. The LHE data tier shown above for example tended to have very short bursts of popularity.

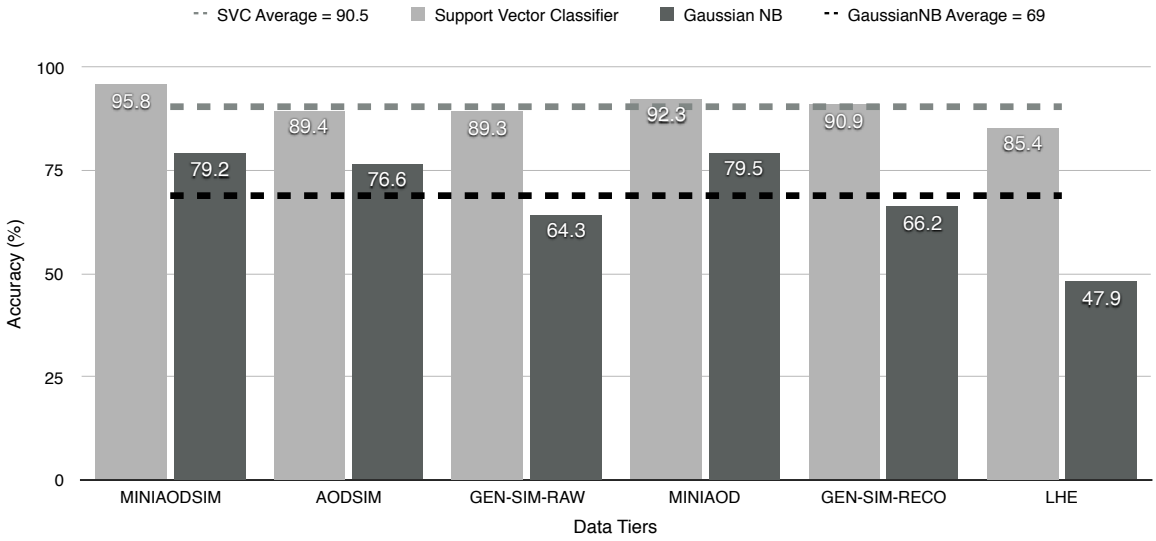

Figure 5.4: Accuracy of SVC and GaussianNB with accesses and CPU hours as separate features

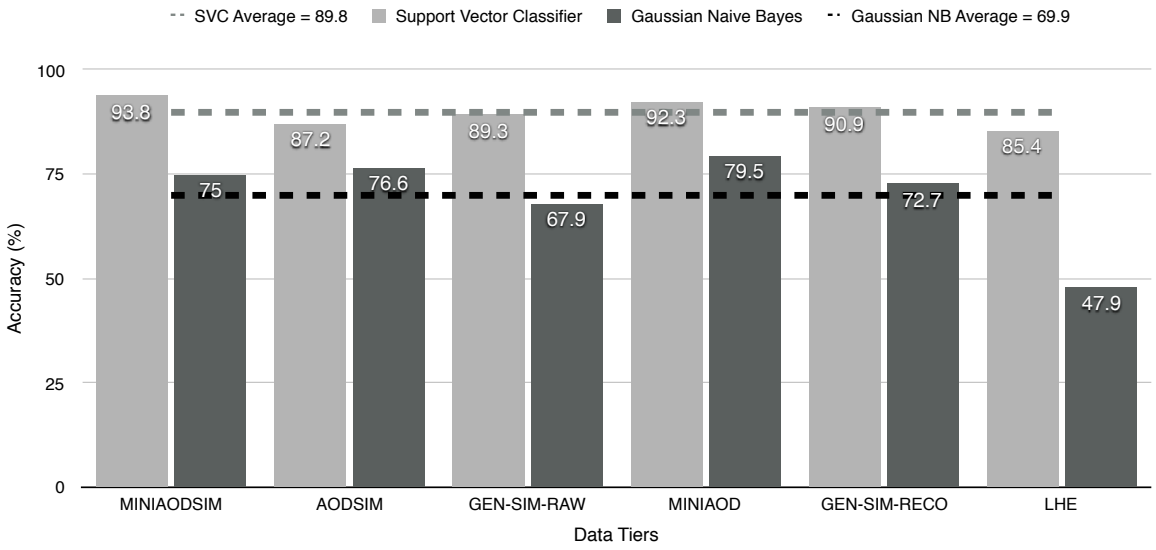

Figure 5.5: Accuracy of SVC and GaussianNB with accesses and CPU hours as a combined feature

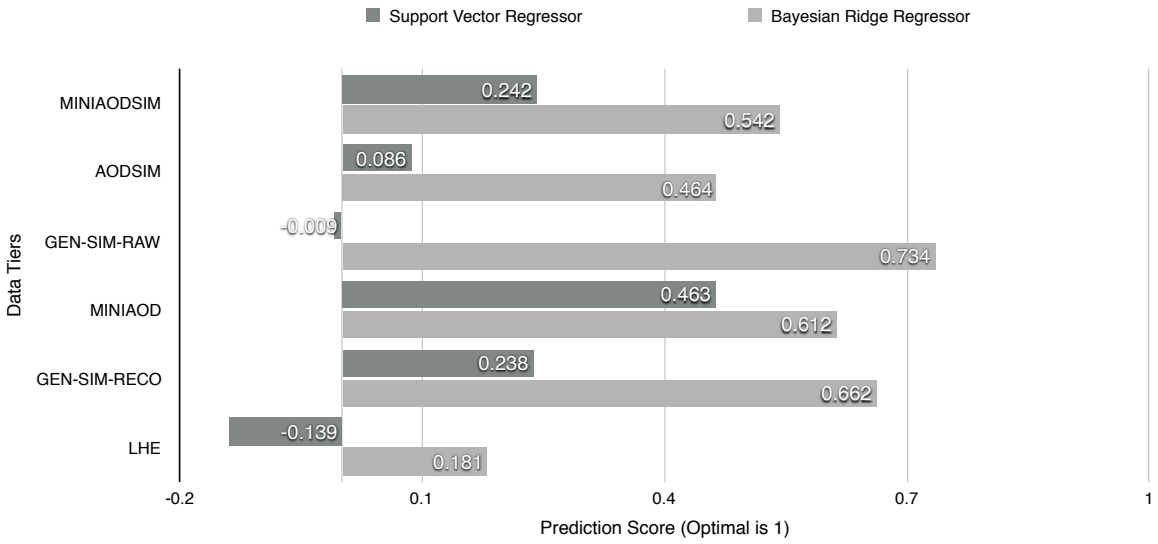

Figure 5.6: Score of Support Vector Regressor and Bayesian Ridge Regressor with accesses and CPU hours as separate features where optimal score is 1. SVR average: 0.147, BRR average: 0.533

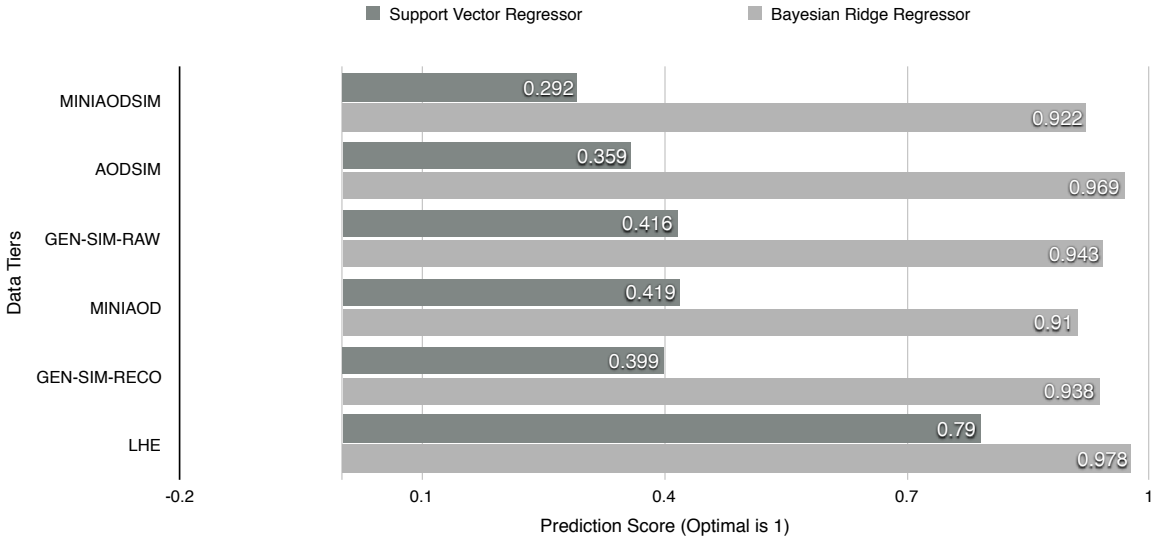

Figure 5.7: Score of SVR and Bayesian Ridge Regressor with accesses and CPU hours as combined a feature where optimal score is 1. SVR average: 0.445, BRR average: 0.943

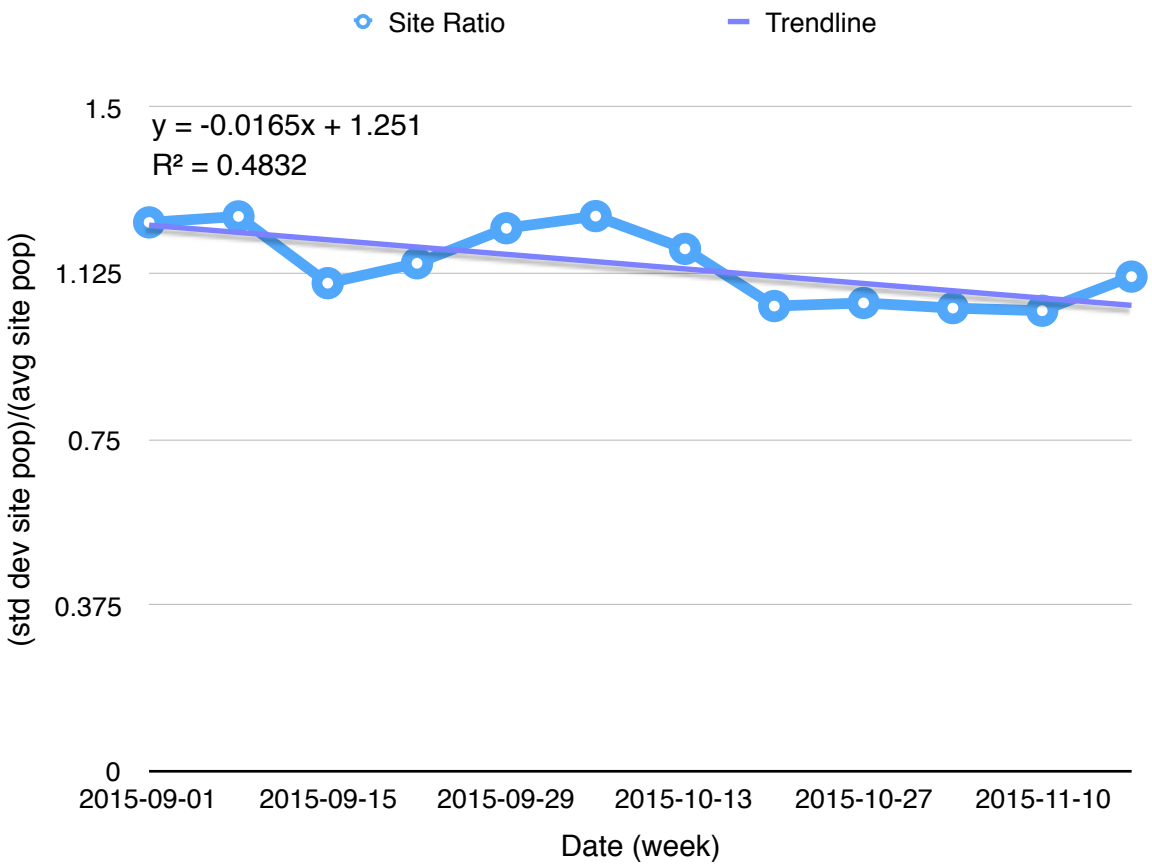

Figure 5.8: Ratio of standard deviation and average site popularity normalized based on site performance and storage capacity.

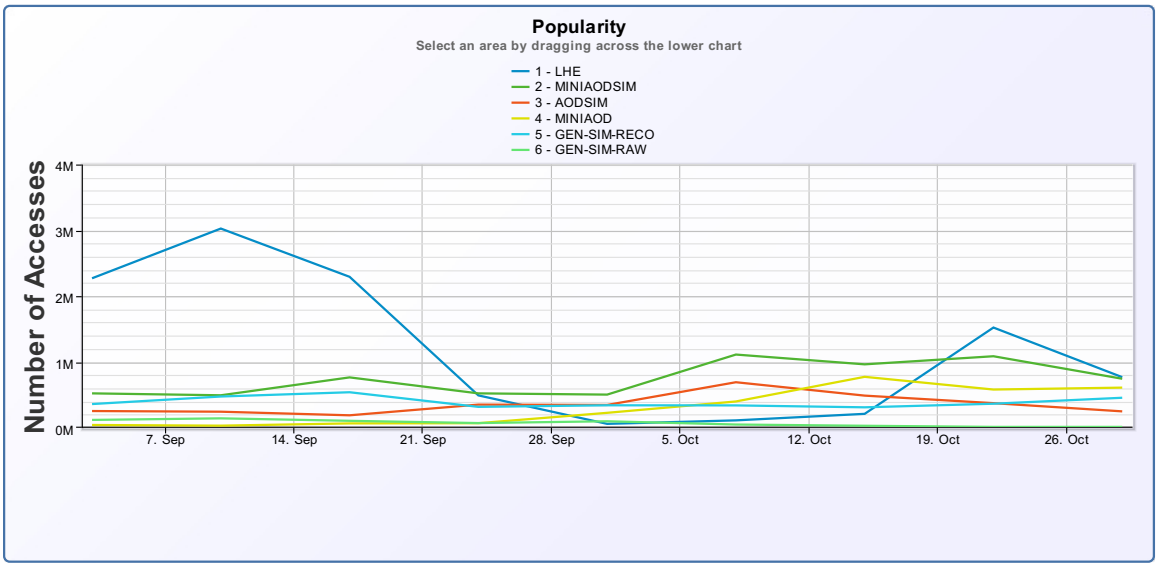

Figure 5.9: Total number of accesses for all data tiers. Source [21]

Chapter 6

Conclusions and Future Work

Dynamic data management in data grids is a new concept that is becoming more popular. In this thesis we describe and develop a general framework for dynamic data management in scientific data grids using a layered architecture. Following common standards suggesting highly generic implementations of data grid components, we created important ground work for a flexible and generic dynamic data manager in a data grid environment.

The middleware data manager allows for grid specific data collection without having to remodel the dynamic data manager and local data storage interface. The use of common grid authentication methods allows for easy addition of grid services making it highly flexible. The data analysis module provides data popularity prediction using machine learning techniques. A generic implementation makes it easy to implement more powerful machine learning algorithms for optimal grid specific performance. System balancing is done using a novel balancing algorithm called rocker board. This algorithm emulates a person balancing on a rocker board for better load balancing and utilization of Grid resources.

In the future, we want to further investigate the performance of different machine

learning algorithms and increase the amount of training data for better popularity prediction. In particularly we want to better train the trend classifier to not react on short bursts of increase or decrease in popularity. Furthermore we believe surveying physicists could prove useful for developing even more accurate usage models. Finally, we are planning on expanding the work to include a web interface for easier monitoring.

Initial data placement and disk cleaning was not in the scope of this work. Some work is currently being done, though not yet published, on estimating general dataset popularity for the CMS experiment. This work would be of great interest to use in an initial data placement algorithm. We are also looking at the work done by the LHCb experiment on completely removing datasets from disk automatically.

In the past the experiments at the LHC at CERN have developed many grid services separately. However, recent effort has been done to combine efforts in grid service development. We believe a collaboration between the ATLAS, CMS, and LHCb experiments different dynamic data managers could lead to significant improvements. It is the hope of the author that the general framework developed in this thesis can help making such collaborations a reality.

Bibliography

[1] Georges Aad, E Abat, J Abdallah, AA Abdelalim, A Abdesselam, O Abdinov, BA Abi, M Abolins, H Abramowicz, E Acerbi, et al. The ATLAS experiment at the CERN large hadron collider. *Journal of Instrumentation*, 3(08):S08003, 2008.

[2] Anzar Afaq, Andrew Dolgert, Yuyi Guo, Chris Jones, Sergey Kosyakov, Valentin Kuznetsov, Lee Lueking, Dan Riley, and Vijay Sekhri. The CMS dataset bookkeeping service. In *Journal of Physics: Conference Series*, volume 119, page 072001. IOP Publishing, 2008.

[3] Bill Allcock, Joe Bester, John Bresnahan, Ann L Chervenak, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, Steven Tuecke, and Ian Foster. Secure, efficient data transport and replica management for high-performance data-intensive computing. In *Mass Storage Systems and Technologies, 2001. MSS'01. Eighteenth IEEE Symposium on*, pages 13–13. IEEE, 2001.

[4] A Augusto Alves Jr, LM Andrade Filho, AF Barbosa, I Bediaga, G Cernicchiaro, G Guerrer, HP Lima Jr, AA Machado, J Magnin, F Marujo, et al. The LHCb detector at the LHC. *Journal of instrumentation*, 3(08):S08005, 2008.

[5] Julia Andreeva, S Belov, A Berejnoj, C Cirstoiu, Y Chen, T Chen, S Chiu, MDFD Miguel, A Ivanchenko, B Gaidioz, et al. Dashboard for the LHC experi-

ments. In *Journal of Physics: Conference Series*, volume 119, page 062008. IOP Publishing, 2008.

[6] Thomas Beermann, Graeme Andrew Stewart, and Peter Maettig. A Popularity Based Prediction and Data Redistribution Tool for ATLAS Distributed Data Management. *PoS*, page 004, 2014.

[7] Thomas Alfons Beermann. A study on dynamic data placement for the ATLAS Distributed Data Management system. Technical report, ATL-COM-SOFT-2015-012, 2015.

[8] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[9] Kenneth Bloom, Tommaso Boccali, Brian Bockelman, Daniel Bradley, Sridhara Dasu, Jeff Dost, Federica Fanzago, Igor Sfiligoi, Alja Mrak Tadel, Matevz Tadel, et al. Any Data, Any Time, Anywhere: Global Data Access for Science. *arXiv preprint arXiv:1508.01443*, 2015.

[10] Mike Bostock. D3.js: Data-Driven Documents. `http://d3js.org/`.

[11] Yonny Cardenas, Jean-Marc Pierson, and Lionel Brunie. Uniform distributed cache service for grid computing. In *Database and Expert Systems Applications, 2005. Proceedings. Sixteenth International Workshop on*, pages 351–355. IEEE, 2005.

[12] S Chatrchyan, G Hmayakyan, V Khachatryan, AM Sirunyan, W Adam, T Bauer, T Bergauer, H Bergauer, M Dragicevic, J Erö, et al. The CMS experiment at the CERN LHC. *Journal of Instrumentation*, 3(08):S08004, 2008.

[13] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The data grid: Towards an architecture for the distributed manage-

ment and analysis of large scientific datasets. *Journal of network and computer applications*, 23(3):187–200, 2000.

[14] M Cinquilli, D Spiga, C Grandi, JM Hernàndez, P Konstantinov, M Mascheroni, H Riahi, and E Vaandering. CRAB3: Establishing a new generation of services for distributed analysis at CMS. In *Journal of Physics: Conference Series*, volume 396, page 032026. IOP Publishing, 2012.

[15] Nicholas Coleman. Distributed policy specification and interpretation with classified advertisements. In *Practical Aspects of Declarative Languages*, pages 198–211. Springer, 2012.

[16] CMs Collaboration. CMS Collaboration. `http://cms.web.cern.ch/content/cms-collaboration`.

[17] NRENaissance Committee et al. *Realizing the Information Future:: The Internet and Beyond*. National Academies Press, 1994.

[18] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[19] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[20] Alvise Dorigo, Peter Elmer, Fabrizio Furano, and Andrew Hanushevsky. Xrootd-a highly scalable architecture for data access. *WSEAS Transactions on Computers*, 1(4.3), 2005.

[21] CMS Experiment. PopDB: CMS Popularity Service. `https://cmsweb.cern.ch/popdb/popularity/dataTier`.

[22] Jeroen Famaey, Tim Wauters, and Filip De Turck. On the merits of popularity prediction in multimedia content caching. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, pages 17–24. IEEE, 2011.

[23] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of High Performance Computing Applications*, 11(2):115–128, 1997.

[24] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International journal of high performance computing applications*, 15(3):200–222, 2001.

[25] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin, 2001.

[26] Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data management in an international data grid project. In *Grid ComputingGRID 2000*, pages 77–90. Springer, 2000.

[27] MI Jordan and TM Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[28] Massimo Lamanna. The LHC computing grid project at CERN. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 534(1):1–6, 2004.

[29] Neal Leavitt. Will NoSQL databases live up to their promise? *Computer*, 43(2):12–14, 2010.

[30] Ming Lei, Susan V Vrbsky, and Xiaoyan Hong. An on-line replication strategy to increase availability in data grids. *Future Generation Computer Systems*, 24(2):85–98, 2008.

[31] T Maeno, K De, and S Panitkin. PD2P: PanDA dynamic data placement for ATLAS. In *Journal of Physics: Conference Series*, volume 396, page 032070. IOP Publishing, 2012.

[32] Tadashi Maeno. PanDA: distributed production and distributed analysis system for ATLAS. In *Journal of Physics: Conference Series*, volume 119, page 062036. IOP Publishing, 2008.

[33] FH Barreiro Megino, M Cinquilli, D Giordano, E Karavakis, M Girone, N Magini, V Mancinelli, and D Spiga. Implementing data placement strategies for the CMS experiment based on a popularity model. In *Journal of Physics: Conference Series*, volume 396, page 032047. IOP Publishing, 2012.

[34] MongoDB. MongoDB: NoSQL Database. `https://www.mongodb.org/`.

[35] MongoDB. PyMongo: MongoDB API. `https://api.mongodb.org/python/current/`.

[36] Ekow Otoo and Arie Shoshani. Accurate modeling of cache replacement policies in a data grid. In *Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pages 10–19. IEEE, 2003.

[37] Python. ConfigParser: Python module. `https://docs.python.org/2/library/configparser.html`.

[38] Python. distutils: Python module. https://docs.python.org/3/library/distutils.html.

[39] Python. logging: Python module. https://docs.python.org/2/library/logging.html.

[40] Python. Python: Programming Language. https://www.python.org/.

[41] Python. threading: Python module. https://docs.python.org/2/library/threading.html.

[42] Python. unittest: Python module. https://docs.python.org/2/library/unittest.html.

[43] Kavitha Ranganathan and Ian Foster. Identifying dynamic replication strategies for a high-performance data grid. In *Grid ComputingGRID 2001*, pages 75–86. Springer, 2001.

[44] J Rehn, T Barrass, D Bonacorsi, J Hernandez, I Semeniouk, L Tuura, and Y Wu. PhEDEx high-throughput data transfer management system. *Computing in High Energy and Nuclear Physics (CHEP) 2006*, 2006.

[45] scikit learn. Regressor score function in scikit-learn. http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.BayesianRidge.html#sklearn.linear_model.BayesianRidge.score.

[46] scikit learn. scikit-learn: Machine Learning in Python. http://scikit-learn.org.

[47] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The condor experience. *Concurrency-Practice and Experience*, 17(2-4):323–356, 2005.