Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZAI-MSC-2013-03

Masterarbeit

im Studiengang "Angewandte Informatik"

A MapReduce Input Format for Analyzing Big High-Energy Physics Data Stored in ROOT Framework Files

Fabian Glaser

am Institut für Informatik
Gruppe Softwaretechnik für Verteilte Systeme

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

31. Mai 2013

Georg-August-Universität Göttingen
Zentrum für Informatik

Goldschmidtstraße 7
37077 Göttingen
Germany

Tel.	+49 (5 51) 39-17 20 00
Fax	+49 (5 51) 39-1 44 03
Email	office@informatik.uni-goettingen.de
WWW	www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 31. Mai 2013

Masterthesis

A MapReduce Input Format for Analyzing Big High-Energy Physics Data Stored in ROOT Framework Files

Fabian Glaser

May 31, 2013

Supervised by
Prof. Dr. Jens Grabowski
Dr. Thomas Rings
Institute of Computer Science
Georg-August-University of Göttingen, Germany

Prof. Dr. Helmut Neukirchen
Faculty of Industrial Engineering, Mechanical Engineering
and Computer Science
University of Iceland, Iceland

Acknowledgments

I would like to thank Prof. Dr. Jens Grabowski for offering me to write this thesis and accompanying me during my master studies. With his help I was able to spend some fruitful time at the university of Iceland, where the idea for this thesis was born. Many thanks to Prof. Dr. Helmut Neukirchen, who welcomed me in Iceland and contributed a lot to my graduate experience. We had a lot of fruitful discussions regarding the project addressed in this thesis. Back in Göttingen, Dr. Thomas Rings took care that I was able to continue my work and contributed a lot with suggestions and good advice.
I also thank Prof. Dr. Michael Schmelling and Dr. Markward Britsch from the LHCb group at the Max Planck Institute for Nuclear Physics (MPIK) in Heidelberg for providing the case study, consisting of a toy Monte-Carlo program for event data generation and the ROOT-based sample C++ analysis code underlying the evaluations.

The cloud resources have been provided by Amazon.

Abstract

Huge scientific data, such as the petabytes of data generated by the Large Hadron Collider (LHC) experiments at CERN are nowadays analyzed by grid computing infrastructures using a hierarchic filtering approach to reduce the amount of data. In practice, this means that an individual scientist has no access to the underlying raw data and furthermore, the accessible data is often outdated as filtering and distribution of data only takes places every few months. A viable alternative to perform analysis of huge scientific data may be cloud computing, which promises to make a "private computing grid" available to everyone via the Internet. Together with Google's MapReduce paradigm for efficient processing of huge data sets, it provides a promising candidate for scientific computation on a large scale.
This thesis investigates the applicability of the MapReduce paradigm, in particular as implemented by Apache Hadoop, for analyzing LHC data. We modify a typical LHC data analysis task so that it can be executed within the Hadoop framework. The main challenge is to split the binary input data, which is based on ROOT data files, so that calculations are performed efficiently at those locations where the data is physically stored. For Hadoop, this is achieved by developing a ROOT-specific input format. Services of the Amazon Elastic Compute Cloud (EC2) are utilized to deploy large compute clusters to evaluate the solution and explore the applicability of the cloud computing paradigm to LHC data analysis. We compare the performance of our solution with a parallelization of the analysis using the PROOF framework, a standard tool specialized in parallelized LHC data analysis. Our results show that the Hadoop-based solution is able to compete with the performance using PROOF. Additionally, we demonstrate that it scales well on clusters build from several hundred compute nodes inside the EC2 cloud.

Contents

1. Introduction 1
1.1. Goals and contributions . . . 3
1.2. Outline . . . 3

2. Foundations 5
2.1. High Energy Physics data analysis . . . 5
2.1.1. The Large Hadron Collider . . . 5
2.1.2. Data recorded at the Large Hadron Collider (LHC) . . . 6
2.1.3. The Worldwide LHC Computing Grid . . . 6
2.2. ROOT/PROOF and SCALLA . . . 7
2.2.1. Architecture . . . 7
2.2.2. The TSelector framework . . . 10
2.2.3. ROOT File Format and TTrees . . . 10
2.2.4. Structured Cluster Architecture for Low Latency Access . . . 11
2.3. MapReduce and the Google File System . . . 12
2.3.1. Architecture . . . 13
2.3.2. The Google File System . . . 13
2.4. Hadoop and the Hadoop Distributed File System . . . 13
2.4.1. Architecture . . . 14
2.4.2. The Hadoop File System . . . 15
2.4.3. Hadoop Streaming . . . 16
2.4.4. Hadoop Pipes . . . 17
2.5. Cloud Computing . . . 17
2.5.1. Characteristics . . . 17
2.5.2. Virtualization . . . 18
2.5.3. Deployment and service models . . . 19
2.6. The Amazon Web Services . . . 20
2.6.1. Availability Zones . . . 20
2.6.2. Simple Storage Services and Elastic Block Store . . . 20
2.6.3. Elastic Compute Cloud . . . 21
2.6.4. Virtual Private Clouds . . . 21
2.6.5. Elastic MapReduce . . . 22

2.6.6. Usage fees 22

3. Analysis **25**
3.1. Comparison between Hadoop and PROOF 25
3.2. The example HEP analysis 26
3.2.1. Input data 27
3.2.2. Implementation 28
3.3. Application of MapReduce to the example analysis 29
3.4. The physical structure of the input data 32
3.5. Definition of HDFS block selection workflow 33
3.6. Remarks on general applicability 34

4. Implementation **37**
4.1. The HDFS plugin for ROOT 37
4.2. Analysis implementation in Hadoop 38
4.2.1. Design decisions 38
4.2.2. The StreamingMapper 38
4.2.3. The StreamingReducer 39
4.3. The input format implementation 40
4.3.1. Design decisions 40
4.3.2. The RootFileMapper 41
4.3.3. The RootFileInputFormat 42
4.3.4. The StreamingInputFormat 45
4.4. Analysis implementation in PROOF 45
4.4.1. MySelector::SlaveBegin() 46
4.4.2. MySelector::Process() 46
4.4.3. MySelector::Terminate() 47
4.5. Parallelization of event data generation 47

5. Deployment **49**
5.1. Remarks on Elastic MapReduce 49
5.2. Cluster node configuration 49
5.2.1. Conducted benchmark tests 50
5.2.2. Benchmark results 52
5.2.3. Final node configuration 52
5.2.4. Software 54
5.3. Cluster monitoring 54
5.4. Cluster layout 55
5.4.1. Hadoop cluster layout 55
5.4.2. PROOF cluster layout 56

5.5. Cluster management 57
5.5.1. Authentication 57
5.6. Limitations and costs of utilizing the AWS services 58

6. Evaluation **61**
6.1. Metrics 61
6.1.1. Total execution time 61
6.1.2. Events per second 61
6.1.3. Average network throughput 62
6.1.4. Average CPU utilization 62
6.1.5. Remarks on scalability 62
6.2. Evaluation of the RootFileInputFormat 63
6.3. Comparison of the Hadoop and PROOF cluster performance 65
6.4. Scalability of Hadoop-based solution 68

7. Related Work **71**
7.1. Application of MapReduce to scientific computing 71
7.2. Different MapReduce implementations 72
7.3. Hadoop/HDFS in the HEP community 73

8. Conclusions and Outlook **75**
8.1. Future research directions 76

A. Abbreviations and Acronyms **79**

B. Changes to the ROOT HDFS plugin **81**

C. Steps to deploy clusters in EC2 **83**

List of Figures **85**

Listings **86**

List of Tables **86**

Bibliography **87**

1. Introduction

The scientific experiments and simulations at the *Large Hadron Collider* (LHC) at CERN, Geneva, produce data on a petabyte scale. In the early state of planning the computational analysis infrastructure, it became clear that it is not be feasible to implement a local storage and analysis cluster which is able to fully handle the sheer quantity of the produced data. Instead, it is distributed to a globally deployed hierarchical storage and analysis network called the *Worldwide LHC Computing Grid* (WLCG) [64]. This network is formed by computational capacities of physical institutions all over the world. The data itself consists of millions of tracked particle beam crossings, called *events* that need to be analyzed to extract statistical qualities. Due to political and technical constraints, several restrictions exist on access to the data and it is filtered before being distributed. Depending on these constraints, individual scientists might not have access to the raw data or may suffer from several months delay. To address these problems, other approaches need to be investigated regarding their applicability for LHC data analysis.

Currently the facilities involved in the WLCG apply different methods and frameworks to parallelize the computational tasks. Since each event can be investigated separately the analyses jobs can be highly parallelized while the output can be merged at the end. Often simple batch systems are used to split and distribute computational tasks to nodes in clusters build from commodity hardware for parallelization. This approach can be cumbersome since it often includes the manual development of scripts that semi-automatically distribute the workload.
Another approach was developed with the *Parallel ROOT Facility* (PROOF) framework [40] at CERN itself. PROOF is based on ROOT [23], a framework especially designed for high energy physics data analysis. PROOF parallelizes jobs at event level by automatically splitting up the input data, balancing the load on the different compute nodes and merging the results at the end. It relies on a distributed storage system called *Structured Cluster Architecture for Low Latency Access* (SCALLA) [21] that provides a hierarchical solution for storing and accessing huge amounts of data.

Apart from the scientific community, the growing size of collected user and application data fostered the search for efficient analysis frameworks in the industry. In 2004, Google published a white-paper [33] on a new parallel computing paradigm, which they developed for the analysis in their data centers. This new paradigm is called *MapReduce* and was

quickly adapted by other big companies including Facebook, Yahoo! and IBM. MapReduce heavily relies on the *Google File System* (GFS) [43], which aims to provide reliable and distributed storage and access to data.
Since Google's implementation of MapReduce and the underlying file system is proprietary, the Apache Software Foundation developed an open source implementation of MapReduce, called *Hadoop* [11]. Hadoop comes with its own distributed file system called *Hadoop File System* (HDFS) which tries to mimic the properties of GFS.

Similar to the idea of using a computing grid [38] for the analysis of LHC data, the *cloud computing* [54] paradigm is utilized by industrial companies. Driven by the need to serve peak loads on their computing infrastructure (e.g. caused by Christmas sales on online marked places) companies can hire extra computing resources over the Internet to extend their own capabilities.
Since it is arguable that grid computing and cloud computing are based on similar ideas, the adaptation of using cloud computing in scientific analysis is conservative. Its applicability for the scientific community therefore needs to be investigated further.

The purpose of this thesis is twofold: On the one hand, we investigate the applicability of the MapReduce paradigm to the LHC data analysis. We discuss how Hadoop can be used for efficient and reliable parallelization and identify HDFS as a solution for storing the huge amounts of data during the analysis.
On the other hand, we utilize cloud computing resources of the *Amazon Web Services* (AWS) [4] deploying computing clusters to test the developed solutions. This allows us to investigate how commercial cloud computing services can be exploited by the scientific community.
The access to the original data produced at the LHC is restricted to members of the collaborations conducting experiments at the LHC. Therefore, the data to be analyzed is based on Monte Carlo events generated by an event simulator called PYTHIA-8 [60] and made available by colleagues from the Max Planck Institute for Nuclear Physics (MPIK) in Heidelberg, who are members of the LHCb [26] collaboration. They also provided an example analysis implementation that serves as the case study for this thesis. The data itself is stored in ROOT files in a compact binary format developed at MPIK which is also used for exploratory access to real data or centrally produced Monte Carlo data. Since Hadoop is initially designed to process large text files, adapting it to the ROOT binary format is one of the main challenges addressed in this thesis.

The initial idea of applying the MapReduce paradigm to LHC data analysis was tested and described in a student research project [44], which served as a trial study for the problem addressed in this thesis.

1.1. Goals and contributions

The goals and contributions of this thesis can be summarized as follows:

- application of the MapReduce paradigm to an example HEP analysis representing a typical LHC data analysis task,
- development and evaluation of an efficient Hadoop input format for data stored in the binary ROOT format,
- evaluation of the scalability of the Hadoop-based solution,
- comparison with a PROOF-based solution,
- remarks on the applicability of cloud computing to scientific data analysis.

1.2. Outline

After this introduction, we introduce the basic principles and techniques utilized in this thesis in Chapter 2. We cover LHC data analysis in general terms, introduce PROOF and the underlying file access system SCALLA and describe MapReduce and the Hadoop implementation. Additionally, we introduce the basic principles of cloud computing and provide an overview of the Amazon Web Services. Chapter 3 describes how the concept of MapReduce can be used to parallelize event-based LHC analysis and how data stored in the binary ROOT format can be processed efficiently by Hadoop. In Chapter 4, we discuss the implementation of the prototype analysis in Hadoop and PROOF. In Chapter 5, we describe how we utilized AWS to deploy compute clusters to evaluate the provided solutions. Subsequently, Chapter 6 discusses the outcome of the evaluation in terms of performance and scalability. An overview of different MapReduce implementation and the impact of the MapReduce paradigm on the scientific community in general and especially on LHC data analysis is provided in Chapter 7. Chapter 8 summarizes the outcome of this thesis and gives ideas for possible future research directions.

2. Foundations

This chapter introduces the concepts and tools necessary for this thesis. In Section 2.1, we discuss data analysis in *High Energy Physics* (HEP) and its challenges in general terms. Section 2.2 introduces ROOT and the Parallel ROOT Facility (PROOF), commonly used tools for HEP data analysis. Subsequently, we discuss the programming paradigm *MapReduce* in Section 2.3 and its widely used open-source implementation *Hadoop* in Section 2.4, which provides an alternative to PROOF systems.
Cloud computing models can extend the general model in HEP data analysis and the evaluation done in this thesis heavily relies on cloud resources. Therefore, Section 2.5 introduces the basics of cloud computing. Subsequently, Section 2.6 describes the services of one particular cloud service provider, namely the *Amazon Web Services* (AWS) offered by Amazon. Parts of this chapter are adapted from our project report [44].

2.1. High Energy Physics data analysis

The four big experiments *ALICE* [2], *ATLAS* [1], *CMS* [28] and *LHCb* [3] at the *Large Hadron Collider* (LHC) located at CERN, Geneva produce particle collision data at a rate of around 15 PB per year. In addition to its computing and storage complexity, access to the data must be provided to around 5000 scientists in around 500 research institutes and universities all around the globe [64]. In the early state of planning it became clear that deploying a local cluster at CERN would not be a feasible solution. Therefore the *Worldwide LHC Computing Grid* (WLCG) project was initiated. The LHC, the structure of the recorded data and how the data is handled by the WLCG is described in the next subsections.

2.1.1. The Large Hadron Collider

The LHC located at *Conseil Européen pour la Recherche Nucléaire* (CERN) is a circular particle collider that collides proton beams with a center of mass energy of around 7TeV [24, p.3-6][1]. The proton beams are accelerated to travel nearly at the speed of light and two beams are traveling around the collider in opposite directions, crossing each

[1]Note that the LHC is also used to collide lead ions at 2.76TeV, but since we are interested in the general data flow, we only analyze the proton-proton collisions for simplicity.

other at four defined crossing points. Since the total circumference of the accelerator is around 27 km, each beam travels around the whole accelerator 11245 times per second. Each beam consists of 2808 bunches, where each bunch contains around 1.15×10^{11} protons [24, p.3-6]. The bunches are spaced with around 25 ns [18, p.86]. Due to technical constraints the average crossing rate reduces to "number of bunches" × "revolution frequency" $= 2808 \times 11245 = 31.6$ MHz. When two bunches cross there are around 19.02 [24, p.3-6] proton-proton collisions leading to maximum number of 600×10^6 collisions per second. A bunch crossing is called ***event***.
The products of these events are detected using a whole chain of different detectors specific for each experiment. Only a very small fraction of the observed events have interesting properties, so the collected data is passed through filters immediately.

2.1.2. Data recorded at the LHC

The raw data size collected each second varies between the experiments: ATLAS produces around 320 MB/s, CMS around 225 MB/s, LHCb around 50 MB/s and ALICE around 50 MB/s. The data size accumulates to around 15 PB of raw data over the year [64]. Before the data is available for analysis at physical institutions it goes through two reduction steps [40]: First the event data is reconstructed from the raw detector data into *Event Summary Data* (ESD) and then all information that is not related to the analysis is removed. The resulting data is called *Analysis Object Data* (AOD). The total size of AOD is about 30-200 TB per year, while the total size of ESD data accumulates to about 10 TB per year. Additionally Monte-Carlo simulations are conducted. Depending on the experiment the production rate of simulated data, which is treated the same way as described above, is 20% to 100% of the original raw data size. For some analysis studies and distribution the data is further reduced to a format, called *Derived Physics Data* (DPD).

2.1.3. The Worldwide LHC Computing Grid

As described in its initial technical design report [64] the WLCG implements a globally distributed computing and storage infrastructure divided into four tiers. The responsibilities for the different Tier-x centers vary slightly between experiments, but in general they can be summarized as follows. The original raw data produced at the LHC is stored on tape and initially processed at the Tier-0 center at CERN itself. Later on it is distributed to Tier-1 centers around the globe that are responsible for its permanent storage and for providing the computational capacity for processes that require access to huge amounts of original data. Currently there are eleven Tier-1 centers worldwide. The Tier-1 centers redistribute the data to Tier-2 centers that consists of one or more collaborating computation facilities. Tier-2 centers are meant to provide capabilities for end-user analysis and Monte-Carlo simulations. All other institutes and computing facilities that need access to the LHC data and take part

in its analysis are called Tier-3 centers.
While the exact software stack deployed on the different computing sites may vary, the *Baseline Services Working Group* defined a set of agreed services and applications that must be provided [19]. It includes but is not limited to storage and computing services, catalog services, workload management and authentication and authorization services.
In this thesis we focus on problems that arise at Tier-2/Tier-3, where physicists are interested in analyzing a subset of the experimental data according to their field of study.

2.2. ROOT/PROOF and SCALLA

ROOT [23] is a software framework developed at CERN that provides tools to handle and analyze large amounts of data. The data is defined in a specialized set of objects. ROOT provides functionality for histograms, curve fitting, minimization, graphics and visualization classes. The root architecture consists of around 1200 classes organized in 60 libraries and 19 categories (modules). The parallel version of ROOT is called *Parallel ROOT Facility* (PROOF). It aims to provide an alternative to the commonly used batch systems for data analysis used at Tier-2/Tier-3 centers. To access and process huge amounts of data, PROOF uses the *Structured Cluster Architecture for Low Latency Access* (SCALLA) as an underlying file access system. In the following we introduce PROOF's scheduling architecture and its file format. Subsequently, we discuss the basic concepts of SCALLA and the framework to parallelize event-based analysis with PROOF, called *TSelector*.

2.2.1. Architecture

PROOF is designed to address end-user analysis scenarios at Tier-2 or Tier-3. In these scenarios single tasks process up to 100 TB of event data on computing facilities with around 100 nodes [40]. In the motivation for PROOF [40], Gerardo et al. distinguish between three different task categories: *fully-interactive* tasks with short responsive times and high level of interactivity, *batch* tasks that have "long execution times and require very little interactivity" [40] and *interactive-batch* that are a mixture of both. According to the authors, PROOF was implemented to address the third category. While in the traditional batch approach the job partitioning is done before the execution and as a result the total execution time is determined by the slowest sub-job, PROOF tries to minimize these "long-tails" and provide real-time feedback [40].

As shown in Figure 2.1, PROOF implements a *master/slave* architecture, whereby the master-level can have multiple tiers, allowing the approach to scale on large distributed systems. A *client* initiates a session by contacting the *master*. The master is responsible for starting the *workers*, distribute the job to the workers and collect their output at the end. A multi-tier master architecture can help to distribute the workload when the results from

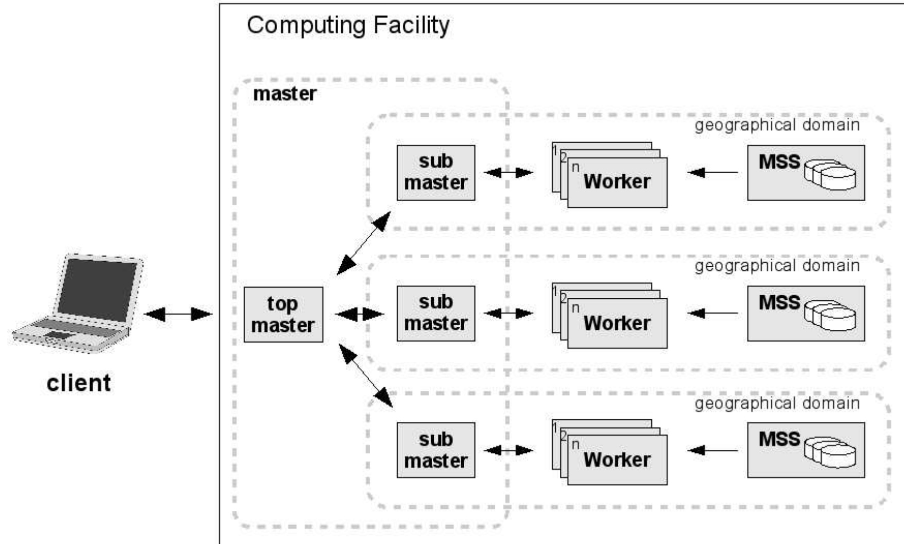

Figure 2.1.: PROOF's multi-tier architecture [40].

many workers are merged in a final stage of the analysis and allows to split a cluster into multiple *geographical domains*, thus optimizing the access to the underlying *Mass Storage System* (MSS) [40].

Figure 2.2 shows the central part for job distribution and load balancing in PROOF, which is the packet generator or `packetizer`: "It decides where each piece of work -called *packet*- should be processed" [41]. Thereby it tries to schedule subtasks to nodes, where the data is stored to reduce network load. Instead of *pushing* the subtasks to the workers, PROOF uses a *pull* architecture to distribute the work: When a worker is finished with processing a part of the job, it asks the `packetizer` for the next subtask. Figure 2.2 depicts the general workflow of the job parallelization: After an initialization phase, where the job description is send to the worker nodes (`Process(''ana.C'')`), the workers are requesting packets by contacting the master (`GetNextPacket()`). The master answers with a packet, indicating the location of the data (the first number indicates the index of the event data in the input file), and the size of the packet. Thereby the packet size can be varied. After the processing is finished, the workers are sending their results to the master (`SendObject(histo)`), which merges the results and returns them to the client.

PROOF is an official part of the ALICE computing model [32] and is deployed in the *CERN Analysis Facility* (CAF) at CERN [39]. To provide event-level parallelism for analysis jobs it offers the *TSelector* framework, which is described in the next subsection.

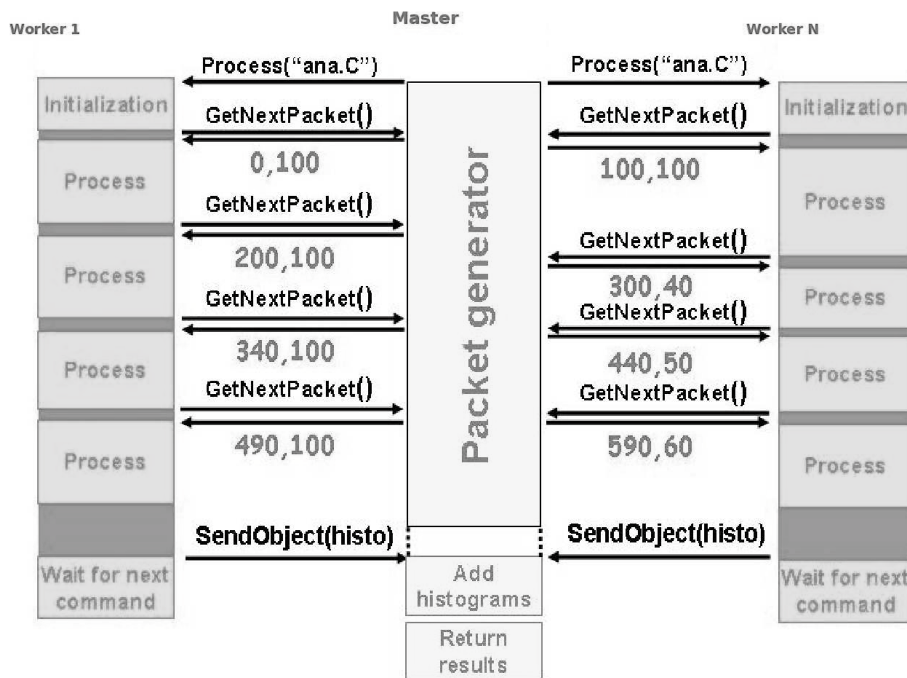

Figure 2.2.: The pull architecture of the PROOF packetizer [41].

2.2.2. The TSelector framework

The *selector framework* offers a convenient way to analyze event data stored in `TTrees` (see next subsection) using PROOF. In the framework the analysis is implemented as a `TSelector` and split up into the following methods:

- `Begin(), SlaveBegin()`: `Begin()` is called on the PROOF session client and `Slave-Begin()` on each worker node. All the initialization needed for the `Process()` method is done here.
- `Init()`: The `Init()` method is called each time a new `TTree` is attached to the input data.
- `Notify()`: This method is called when the first entry of a new file in the input data is read.
- `Process()`: `Process()` is called for each event in the input data. The actual analysis of the event must be implemented here. This is called on the worker nodes.
- `SlaveTerminate(), Terminate()`: These functions define the counterpart to `Slave-Begin()` and `Begin()` and are called at the end of the analysis. `SlaveTerminate()` is called of each slave and `Terminate()` is called on the session client. It is typically used to write the collected output data to a file.

An event-based analysis that is parallelized with PROOF must implement a `TSelector`. We refer to this framework in the description of implementation in Chapter 4.

2.2.3. ROOT File Format and TTrees

To store large amounts of data that need to be accessible in an efficient way, ROOT implements its own data format. Figure 2.3 shows the basic structure of a ROOT file. The data in ROOT files is always stored in a machine independent format (ASCII, IEEE floating point, big endian byte ordering) [65].
Up to the first 64 bytes are reserved for the file header. It is followed by *data records* of variable length. Among others the header contains information about the position of the first data record (`fBEGIN`), the position of the first data record marked as deleted (`fSeekFree`) and its length (`fNBytesFree`), and the postition of the first free word at the end of file (`fEND`) for efficient data access.
Each data record contains its own record header, called `TKey` followed by the data itself. Since ROOT is an object-orientated framework its data consists of objects that are serialized when written to files. The object data in a data record can be compressed using a compression algorithm based on *gzip*.
ROOT provides the classes `TTree` and `TNtuple` for storing large numbers of the same object

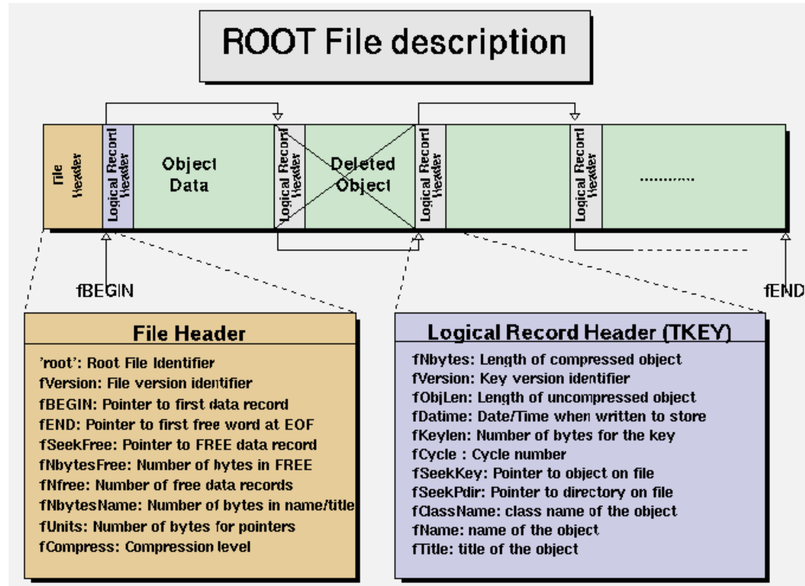

Figure 2.3.: Physical layout of the ROOT file format [25].

type, which are "optimized to reduce disk space and enhance access speed" [65]. `TTree` and `TNtuple` implement trees of objects, whereby the former can store any class of objects while the later itself is a `TTree` that is limited to only store floating point numbers. When filling a `TTree` with objects of the same kind, these objects are first written to a buffer instead of writing them to the file directly. When the buffer is full, it is compressed before being written. Since all objects are of the same type, the object header can also be compressed leading to a much higher compression factor when compared to compressing each object individually. In addition to optimized storage space `TTrees` also enhance the access speed. Data members of the object class that can be treated separately during the analysis can be stored in separate tree *branches*. Each of these branches is assigned to its own output buffer, which is filled with the same member variable of different objects, compressed and written to the file. A buffer holding same type member variables is called *basket*. This design allows to efficiently access the same member variable for all objects on disk, because only the output buffers that hold these members variables need to be read and decompressed.

2.2.4. Structured Cluster Architecture for Low Latency Access

The *Structured Cluster Architecture for Low Latency Access* (SCALLA) [48] is a distributed file access system that is broadly used in the high-energy physics community to build large distributed data clusters with commodity hardware. In SCALLA each node runs a low latency access server called `xrootd` ("eXtended rootd", since it originated from ROOT's file

access mechanism) and a corresponding cluster management service, called `cmsd`. SCALLA implements a redirector service that liberates the user from knowing the exact location of a file. A SCALLA cluster is organized in a 64-ary tree, which means that each node has at most 64 children. The root node acts as a *manager* and the inner nodes act as *supervisors* for its child nodes. This architecture enables efficient data lookup. When a client needs to access a file in the cluster, it contacts the manager node of the tree. This node forwards the query to its children asking if they have the file, which either hold a copy of the file or forward the request to their child nodes. This procedure is followed until the leaf nodes in the cluster tree are reached. If a node holds a copy of the file it responds to the query. If there a multiple copies of a file reported by child nodes the supervisor merges the responses into one single response stating that it holds the file. When the cluster manager receives the response with the location of the file, it redirects the client request to the corresponding node.
The SCALLA file access system is used by the LHC experiments ALICE, ATLAS and CMS [48]. The `xrootd` not only allows the redirection to files in a common namespace, but instead implements a plug-in based multi-layer architecture for file access to generic systems [34]. PROOF itself is realized as a protocol implementation inside this architecture.

2.3. MapReduce and the Google File System

MapReduce is a framework for implementing data intensive applications that was first introduced by Jeffrey Dean and Sanjay Ghemawat at Google in 2004 [33] and attracted a lot of attention since then. In a divide-and-conquer manner the programmer writes code in terms of a *map* and *reduce* function. Both functions take a *key/value* pair as input. The map function takes an input key/value pair generated from the input data and emits one or more key/value pairs, which are then sorted and passed to the reduce function.

```
map(String key, String value): // key: document name
                               // value: document contents
   for each word w in value:
      EmitIntermediate(w, "1");

reduce(String key, Iterator values): // key: a word
                                     // values: a list of counts
   int result = 0;
   for each v in values:
      result += ParseInt(v);
      Emit(AsString(result));
```

Listing 2.1: Word Count example [33].

The original example provided by Dean and Ghemawat for the application of the MapReduce paradigm is counting the appearance of words in a large text file. The pseudo-code is given in Listing 2.1.

The map function takes the document name as the key and file contents as a value. For each word in the context it emits a key/value pair, whereby the key is the word itself and the value is the current count of the word, which is one. These pairs are now sorted and passed to the reduce function. For each key (word) the reduce function is called once and the count of the words is summed up. In the end reduce emits a final key/value pair, whereby the key is still the word and the value holds the count of the word's appearance in the document.

2.3.1. Architecture

Typically the input data is split up and several map and reduce tasks are executed in parallel. Figure 2.4 shows the typical execution procedure in the MapReduce framework. A master node is assigning map and reduce tasks to worker nodes. The input data is divided into *splits* and each split is passed to a worker node executing a map task. These tasks write intermediate key/value pairs to the local disk of the worker node. A reduce worker is informed of the data location and reads the data remotely. Thereby it merges sorts the intermediate key/value pairs according to their keys. For each individual key and the set of corresponding values, the reduce function is called and the final output is produced.

2.3.2. The Google File System

Network bandwidth is a limiting resource in the execution described above. Dean and Ghemawat describe how the *Google File System* (GFS) [43] is used to preserve bandwidth usage. The GFS implements a distributed way of storing large data files on a cluster. The file is hereby split up into blocks (typically 64 MB in size) and each block is stored several times (typically three times) on different nodes in the cluster. Google's MapReduce implementation uses the locality information of these file blocks by assigning map task operating on one of the blocks to a node, which holds the block already or is nearby to a node that holds the block (e.g. in the same rack in the cluster). This way, huge amounts of bandwidth usage can be preserved.
Google's implementation of the MapReduce framework is proprietary and not available for the public. The next section introduces *Hadoop*, which is an open-source framework implementing the MapReduce programming model.

2.4. Hadoop and the Hadoop Distributed File System

Hadoop [11] is an open-source Java framework developed by the Apache Software Foundation, that implements the MapReduce programming model. It has been deployed on huge clusters and is used by several well-known companies, including *Facebook*, *Amazon* and *AOL* [15]. It comes with the *Hadoop File System* (HDFS), which mimics the properties of

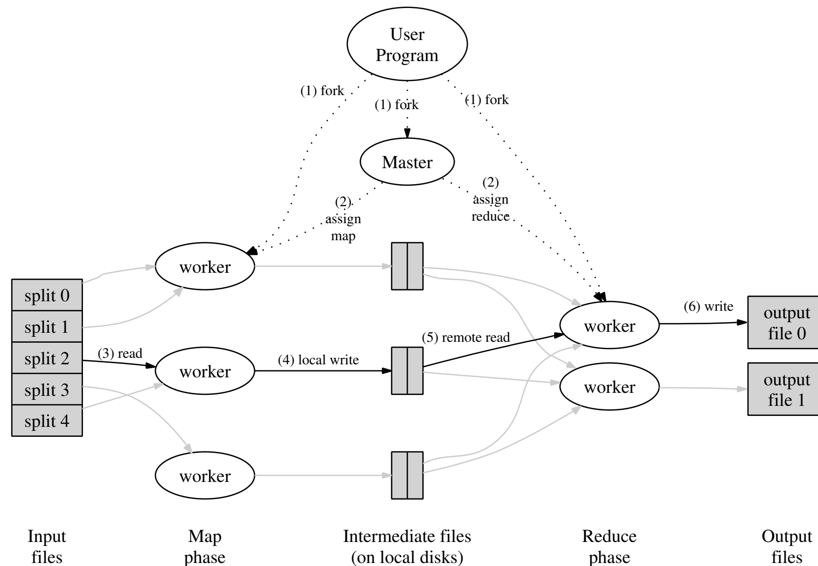

Figure 2.4.: MapReduce job execution overview [33].

GFS described above. After describing Hadoop's Architecture and HDFS in this section we introduce *Hadoop Pipes* and *Hadoop Streaming*, two possibilities to parallelize applications with Hadoop which are not written in Java.

2.4.1. Architecture

The architecture of a cluster running Hadoop orientates on Google's approach shown in Figure 2.4. It is a master/slave architecture, where one or two master nodes are assigning tasks to several worker/slave nodes.
The master nodes are serving as *NameNode*s[2] and *JobTracker*s, while the worker nodes are serving as *DataNode*s and *TaskTracker*s. These daemons are described below.

- **NameNode**: The `NameNode` is keeping track of the stored data. It does not store data itself, but manages the add/copy/delete and move operations that are executed on the data distributed over the `DataNodes`. It is a single point of failure in the current Hadoop implementation, because a failure of the `NameNode` causes the whole file system to become unavailable. Hadoop provides an optional daemon, which is

[2] In the scope of this thesis, we interchangeably use the name of the daemon either for the daemon process itself and for the node running the daemon process.

called SecondaryNameNode. Nevertheless it does not provide full redundancy, because it only creates snapshots of the namespace. There are some plans to implement a Backup Node and an additional Checkpoint Node in future versions of Hadoop, but they have not been realized in the current stable version of Hadoop (Hadoop-1.0.4).

- **DataNode**: The data stored in the Hadoop cluster is stored on the DataNodes. Data stored in HDFS is internally split up into blocks and then distributed and replicated around the DataNodes.

- **JobTracker**: The JobTracker is responsible for assigning map and reduce tasks to nodes running TaskTrackers. It therefore asks the NameNode where the data resides and tries to assign the corresponding tasks to TaskTrackers, which are nearby. The JobTracker monitors the work of all TaskTrackers and updates its status when a job is completed or failed.

- **TaskTracker**: To each node that is configured as a TaskTracker the JobTracker can assign certain tasks. The number of tasks a TaskTracker can accept is limited by the number of *slots* it offers. Each task is assigned to a certain slot. The TaskTracker spawns a separate Java Virtual Machine (JVM) for each task to make sure that it does not get affected if a task crashes. While assigning the tasks to certain nodes the JobTracker takes care that tasks that work on certain data sets get preferably executed on the DataNode where the data persists. If the corresponding TaskTracker does not offer a free slot, it chooses another TaskTracker running on a machine in the same rack.

2.4.2. The Hadoop File System

The Hadoop File System (HDFS) orientates on the description of GFS given in [43]. HDFS is a distributed file system, designed to store large amounts of data on clusters build of commodity hardware. As in GFS the data is split up into blocks and replicated to several nodes on the cluster. While HDFS is mostly POSIX compatible it relaxes a few of its requirements to enable streaming access to data. Its design goals and implementation are described in the HDFS Architecture Guide [22] and can be summarized as follows:

- **Hardware Failure**: In large clusters with thousands of nodes, the chances are pretty high that several cluster nodes suffer from hardware failure. HDFS aims to detect and recover from these failures as fast as possible. The DataNodes described above are sending regular heartbeat-messages to the NameNode, informing it that they are still available and which data chunks they hold. If the NameNode does not receive any message from a certain DataNode for a certain period of time, it initializes a replication of the data that was stored on the node to other DataNodes.

- **Streaming Data Access**: HDFS is designed to enable high throughput access to data stored in the cluster, rather than enabling low latency access. Its usage scenario is batch processing of jobs working on huge amounts of data without the need for interactivity.
- **Large Data Sets**: The typical size of files stored in HDFS is a few gigabytes up to some terabytes. While the files are split up into blocks and replicated over several `DataNode`s, they still appear to the user as single files.
- **Simple Coherency Model**: HDFS is designed for data, which is written once and read many times. This leads to a simplified data coherency model.
- **"Moving computation is cheaper than moving data"**: As we have seen in the description of GFS (Section 2.3.2), we can speed up computation by moving it to a worker node where or close to where the data is stored. Therefore HDFS provides an interface, which can be used by applications to obtain these locality information.
- **Portability across Heterogeneous Hardware and Software Platforms**: HDFS is written in Java and aims to be deployable on all major platforms.

The size of the file blocks in HDFS is set to 64 MB by default. The number of replications of each file block can be controlled by the *replication factor*. A replication factor of two means that the HDFS cluster always keeps two copies of each file block.

2.4.3. Hadoop Streaming

Hadoop Streaming is an API that enables the user to work with map and reduce functions that are not written in Java. Both functions can be any application that read input from the standard input stream (stdin) and write their results to standard output stream (stdout). The standard input format hereby is text-based and read line-by-line, whereby each line represents a key/value pair separated by a tab. Hadoop hereby makes sure that the lines passed to the reduce function are sorted by keys. Programs that are written in this manner can be sequentially tested with the help of UNIX Pipes, which of course do not parallelize the computation:

```
cat input | map | reduce
```

Splitting up the input and parallelizing map and reduce calls in the cluster, whereby using the locality information of the input data is done with the help of a special input format for Hadoop.

2.4.4. Hadoop Pipes

A second way to implement a MapReduce program on Hadoop, not written in Java, is *Hadoop Pipes*. Programs written with Hadoop Pipes link against a thin Hadoop C++ library, which enables the written functions to be spawned as processes and communicate over sockets. The C++ interface is SWIG [62] compatible and can be generated for other source languages as well. Unlike Streaming, Hadoop Pipes does not use the standard input and output channels. The key/value pairs are passed to the mapper and reducers as C++ Standard Template Library (STL) strings so any data that should be used as keys or values need to be converted beforehand. To use Hadoop Pipes the code needs to inherit from certain base classes and reimplement some of their functionality.

2.5. Cloud Computing

In the last years, a paradigm shift took place in the IT industry. It affects the way in which IT infrastructure is utilized on a commercial level as well as in private households. This new paradigm is called *cloud computing*.
Cloud computing services range from simple data backup services in the Internet to the possibility of deploying whole compute clusters or data centers in a remote environment. Individuals use cloud computing to store music, pictures and diaries in the Internet, making it accessible from everywhere and freeing them from keeping the data in sync on different devices. Companies can utilize cloud computing to serve peak-loads on their IT infrastructure by renting the needed resources *on-demand* from a *cloud computing provider*.
Some cloud computing services might also offer a viable model for solving computational challenges in the scientific community.
For our evaluations we are utilizing cloud computing services. This section summarizes its characteristics, virtualization as its core technique and different deployment and service models.

2.5.1. Characteristics

Cloud computing as defined by Armbrust et al. [16] “refers to both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services”. Hereby those services fulfill the following characteristics [54]:

- **On-demand self-service**: Consumers are able to utilize cloud services in an automated manner without human interaction.
- **Broad Network access**: Service capabilities are available over the network for heterogeneous client platforms.

- **Resource pooling**: Different resources (e.g. storage, processing, network bandwidth) are pooled with the help of virtualization. The exact location and physical nature of those resources might be unknown to the consumer.
- **Rapid elasticity**: Consumers can easily utilize and release resources according to their demand.
- **Measured service**: The cloud service providers use metrics to automatically control the usage of resources. The use of resources and the accumulating costs are transparent to the consumers.

Cloud computing services are often implemented by using *virtualization*, which is described in the next subsection.

2.5.2. Virtualization

One of the core enabling technologies in cloud computing is *virtualization*. It describes the "abstraction of IT resources that masks the physical nature and boundaries of those resources from resource users" [42]. All kinds of technical resources can be virtualized, e.g. memory, storage, network or even full machines. It can have several benefits to use virtualized resources, including [30]:

- **Server Consolidation:** Combine multiple servers on one physical machine for saving hardware costs and management and administration workload.
- **Sandboxing:** Virtual machines can provide fully isolated environments for running untrusted applications.
- **Virtual hardware:** Hardware, which is not present, can be simulated.
- **Debugging and Testing:** Virtualization can help with debugging complex software like operating systems by providing virtual machines with full software control. Arbitrary systems can be deployed for testing purposes.

Cloud computing heavily relies on virtualization such that as a user acquiring a service over the Internet may be completely unaware of the underlying physical infrastructure. Often cloud computing systems are distinguished by the level of abstraction of the physical infrastructure to the user. This leads us to the different deployment and service models in cloud systems.

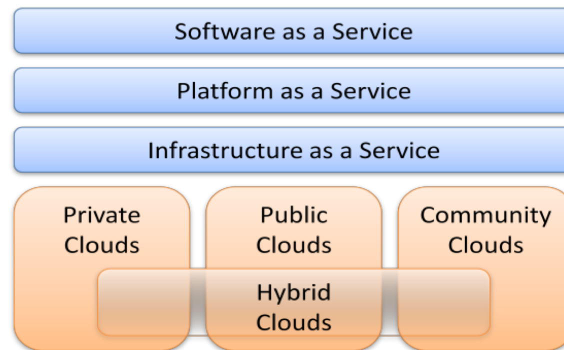

Figure 2.5.: Cloud computing deployment and service models [61].

2.5.3. Deployment and service models

Different deployment and service models exist for cloud systems. They are commonly classified as shown in Figure 2.5. The level of abstraction in the different service models increases from bottom to top. *Infrastructure as a Service* (IaaS) provides the lowest level of abstraction. Here the user can acquire (virtualized) resources such as processing power, memory or storage directly. Users are typically able to deploy their own software stack on top of the resources and are responsible for keeping the software up-to-date. One example for an IaaS infrastructure are the *Amazon Web Services* (AWS) [4] described in the next section. *Platform as a Service* (PaaS) describes a service level, which offers the users a platform to develop and deploy their own applications. They are liberated from managing the infrastructure themselves. The *Google App Engine* [45] and Mirosoft's *Windows Azure* [55] are well-known examples of PaaS.
At the highest level of abstraction *Software as a Service* (SaaS), offers full grown applications as services over the Internet, typically accessible through a standard Web browser. Similar as in PaaS, the users have no control over the infrastructure itself. One example for SaaS is *Google Drive* [46], where a group of users can collaborate to edit and share text documents online.
All these different service models can be arbitrarily deployed on top of the depicted cloud deployment models. The cloud deployment models are differentiated depending on who owns and uses them [54, 61]. *Private Clouds* are used exclusively by a single company in contrast to *Public Clouds*, which can be utilized by the public (usually for a fee). *Community Clouds* are clouds that are shared among several organizations often build around special requirements. *Hybrid Clouds* form a mixture of the introduced cloud deployment models.

2.6. The Amazon Web Services

With *Amazon Web Services* (AWS) [4], Amazon provides a scalable, on demand, IaaS environment, which can be used to provide large amounts of computing power, storage and data management infrastructure on a short timescale. It thereby consists of several different services, which can be used separately or in combination to provide the desired functionality. We briefly introduce the services used in the context of this thesis: *Availability zones*, *Elastic Block Store* (EBS), the *Elastic Compute Cloud* (EC2), the *Virtual Private Cloud* (VPC) and *Elastic MapReduce* (EMR). In addition, we comment on usage fees.

2.6.1. Availability Zones

Each service of AWS is bound to an *Availability Zone*. Availability Zones are designed to be independent of the failure of one another. Multiple Availability Zones form a *region*. Currently there are nine available regions: US East (Northern Virginia), US West (Oregon), US West (Northern California), EU (Ireland), Asia Pacific (Singapore), Asia Pacific (Tokyo), AWS GovCloud (US) and South America (Sao Paulo). Zones and regions also limit the accessibility of certain service instances to one another.

2.6.2. Simple Storage Services and Elastic Block Store

The Amazon Web Services (AWS) contain two services that are used for persistent data storage in the Amazon cloud: the *Simple Storage Service* (S3) [8] and the *Elastic Block Store* (EBS) [5].
Simple Storage Service (S3) is the general storage mechanism for AWS. It provides accessibility over the web and is theoretically unlimited in size from the users' viewpoint. All objects stored in S3 are contained in *buckets*. Each bucket has a unique name and can be made accessible over HTTP. Objects in buckets can be grouped into subfolders. Amazon provides access control to buckets in two ways: Access Control Lists (ACLs) and bucket policies. ACLs are permission lists for each bucket while policies provide a more generalized way of managing permissions that can be applied to users, user groups or buckets.
Elastic Block Store (EBS) provides block storage and can in particular be used to supply persistent storage for the Amazon Elastic Cloud (see next section). The EBS volumes can be formatted with a desired file system and are suitable for applications that need to use persistent storage e.g. databases. Each volume is created in a particular availability zone and can only be attached to virtual machines within the same zone. They can be anything from one GB to one TB in size. EBS provides the ability to back-up snapshots of the current volume state to S3. Snapshots get stored incrementally that means that for consecutive snapshots only the changed blocks are backed-up and not the whole volume again. The total Input/Output operations per second (IOPS) a standard EBS volume can deliver is around

100. For higher Input/Output demands, e.g. for database services, special EBS volumes with up to 2000 *provisioned* IOPS can be used.

2.6.3. Elastic Compute Cloud

Amazon's *Elastic Compute Cloud* (EC2) [6] is a Web service that provides scalable IaaS computing resources on demand. The basic building blocks of EC2 are the Amazon Machine Images (AMIs) that provide templates for the boot file system of virtual machines that can be hosted in EC2. Users can launch several instances of a single AMI as different instance types, which represent different hardware configurations. The CPU power of the instances is measured in Amazon EC2 Compute Units (ECUs). According to Amazon, one ECU provides the capacity of a 1.0-1.2 GHz 2007 Intel Opteron or Xeon processor. Amazon and several third party developers already provide different AMIs. During the time this document was created, there were around 10000 different AMIs to choose from. In case the available AMIs do not fulfill the users requirements, it is possible to create own AMIs, which can be made available to public or just used for private purposes. It is also possible to import local virtual machine images created by a Citrix Xen, Microsoft Hyper-V or VMware vSphere virtualization platform into EC2 and launch them as instances.
To provide access control to the launched instances, Amazon introduces the concept of *security groups*. A security group is a collection of rules that define what kind of traffic should be accepted or discarded by an instance. Instances can be assigned to numerous groups. Changes to the rules can be made at any time and are propagated to all instances automatically.
EC2 uses both the Simple Storage Service (S3) and the Elastic Block Store (EBS) as background storage. When designing a custom AMI or choosing from the available AMIs one need to decide on the kind of storage that should be used by the launched instances. If an AMI only uses *ephemeral storage*, also sometimes referred as *instance storage* the data stored on the instance is only available until the instance is terminated. For persistent storage (i.e. even when an instance is terminated), EBS-backed AMIs can be used. The instances of these AMIs are backed up by an EBS volume, which is connected to the instance on start up. Amazon also provides services for database storage, address mappings and monitoring of instances, which are not further described in this document.

2.6.4. Virtual Private Clouds

While Amazon offers instances on demand with EC2, Amazon's *Virtual Private Cloud* (VPC) [9] simplifies the deployment of a whole network infrastructures. EC2 instances are launched inside an isolated section of the cloud, sub-networks can be created and routing tables can be automatically deployed in a Virtual Private Cloud (VPC). Additionally, VPCs offer *gateways* that implement the ability to communicate with the Internet. Instances

launched in a VPC can be launched as *dedicated*, causing them to be executed physically isolated from instances that are launched by other AWS users. With *Elastic IPs* Amazon offers users to reserve public IP addresses for their instances and automatically assign them to an instance in their VPC that should be reachable from outside.

2.6.5. Elastic MapReduce

Amazon *Elastic MapReduce* (EMR) [7] offers users that have written their code for the Hadoop Framework to run their programs in a MapReduce environment on top of EC2. EMR therefore automatically launches AMIs that have a preinstalled Hadoop framework and takes care of the user's job submission. Amazon uses the MapR distribution [17] of the Hadoop framework. Jobs can be specified by the user over a web interface, a command line tool or the *Elastic Map Reduce API.* The user specifies either a jar file for running a Java job on Hadoop or a mapper and reducer written in another programming language to run as Hadoop Streaming Job. EMR does not offer the utilization of Hadoop Pipes over its interface so far. The jar files, scripts or binaries that should be executed, need to be uploaded to S3 beforehand. For configuration purposes EMR offers the possibility to the user to define *Bootstrap Actions*, which are scripts that get executed on each Hadoop node before the job is finally started. These scripts can be written in any language that is already installed on the AMI instances, such as Ruby, Perl or bash.
Elastic MapReduce (EMR) typically utilizes a combination of the following file systems: HDFS, Amazon *S3 Native File System* (S3N), local file systems and the *legacy Amazon S3 Block File System.* S3 Native File System (S3N) is a file system, which is implemented to read and write files on S3 and the local file system refers to the file system of the AMI instance itself. The user can choose on which kind of AMI instance types his job should be executed depending on the type of file system that is used to store the data.

2.6.6. Usage fees

The usage fees incurring when using AWS are differing by regions. For the work of this thesis, the region EU (Ireland) was used and the pricing was calculated as follows:

- **S3 and EBS**: For data stored in S3 or EBS Amazon charges a fee of 0.095$ per GB and month. Transferring data to the cloud was free. Transferring data out of the cloud was free for the first GB per month, the next 10 TBs of data cost 0.120$ per GB.

- **EC2** and **EMR**: The costs for the instances started in EC2 depend on the instance type that is started and are calculated per hour. Instances belong to one of the three service categories: *on demand instances* are acquired when they are needed and paid on an hourly basis. *Reserved instances* are reserved beforehand (typically for a time

period of one year) for a fixed reservation fee and can then be utilized for a reduced hourly fee. The fee depends on the usage type and prices are distinguished between *light utilization*, *medium utilization* and *heavy utilization*. *Spot instances* form the last category and are automatically deployed for a fixed low fee when there are a lot of free capacities in the EC2 cloud and automatically stopped when the capacity decreases beyond a certain level. This service model can be used to acquire cheap extra resources during times when the EC2 cloud is less utilized e.g. to speed up computation, which does not rely on permanent provisioned resources. For example a small on demand Linux instance (1.7 GB memory, 1 EC2 Compute Unit, 160 GB instance storage, 32 Bit or 64 Bit) was priced $0.065 per hour in the EU West (Ireland) region. The same instance type could also be reserved for one year for $60-$170 depending on the usage scenario and then utilized for an hourly rate between $0.022 and $0.042. To utilize small spot instances the hourly rate was between $0.016 and $0.040. Note that especially the prices for the spot instances are calculated on the total demand in the cloud and are subject to large fluctuations.
EMR is built on EC2 internally and the costs for EC2 instances automatically started by EMR are calculated per instance.

The actual pricing is subject to fluctuations and can be found on the Web sites for S3 [8], EC2 [6] and EMR [7].

3. Analysis

As we described in the previous chapter, data analysis of LHC data is a very challenging task. Distributed systems and large compute clusters need to be exploited to satisfy the computational requirements. While the physics community uses a grid system and specialized frameworks for the analysis, there are fewer experiences with applying the de-facto industrial standards MapReduce and cloud computing to LHC data. Since the example HEP analysis, which is subject to parallelization, is based on ROOT, we also consider the PROOF framework for parallelization. In Section 3.1, we analyze the main similarities and differences between Hadoop and PROOF. Subsequently, we describe the example HEP analysis in more detail in Section 3.2. The application of the MapReduce paradigm to the example HEP analysis is given in Section 3.3. Section 3.2.1 and Section 3.4 analyze the logical and physical structure of the input data respectively. In Section 3.5 we discuss, how the input data can be efficiently distributed inside the Hadoop framework and Section 3.6 discusses the validity of the developed method for other kinds of input data.

3.1. Comparison between Hadoop and PROOF

While PROOF is a specialized framework for LHC data analysis, Hadoop implements a general solution for distributed computing. Nevertheless, there are similarities: Hadoop and PROOF both implement a master/worker architecture, where the master is scheduling subtasks on worker nodes. They are both based on the assumption that a job can be divided into highly independent subtasks, which do not need to communicate with each other during execution.
The user's code is divided into specific methods that are called in a well defined order on the worker nodes: `map` and `reduce` in Hadoop, `Begin`, `Process` and `Terminate` in PROOF. In Hadoop, the `InputFormat` defines how the input data is split into `InputSplits` (see Section 3.3 for more detail) for the subtasks on the worker nodes while in PROOF the `TPacketizer` is assigning data *packets* to the workers.
Both frameworks follow the "Moving computation is cheaper than moving data" paradigm (see Section 2.4.2), where subtasks are tried to be scheduled on worker nodes that actually store the data locally.
Nevertheless the frameworks differ by important characteristics. The PROOF packetizer scheduling mechanism is based on the assumption, that files are stored as a whole at the same location. SCALLA hereby does not provide an automatic file replication mechanism

and hence no automatic recovery in case of data corruption or loss. Storing the data only on one node in the storage system can also cause bottlenecks, when multiple cluster nodes need to access the same part of the data at the same time. Hadoop heavily relies on HDFS and hence the data is replicated automatically in the cluster. This provides not only automatic recovery in case of data corruption or loss, but also gives more flexibility when scheduling the subtasks close to where the corresponding data resides.
While Hadoop does the splitting into subtasks before the computation is started, the PROOF packetizer can adapt the packet size during the computation using the feedback of individual worker nodes. Thereby, it can react to slow workers by reducing the size of the packets that are assigned to these nodes. Hadoop uses another mechanism to overcome the problem of waiting for slow worker nodes at the end of an analysis: When there are further `InputSplits` are available for processing, already assigned `InputSplits` are also assigned to workers that otherwise become idle. The results of the process that finishes first are collected and the other processes that are processing the same `InputSplits` are terminated. Hadoop's map and reduce tasks are executed on different worker nodes allowing to start the reduce phase while there are still map tasks computed. The job execution in PROOF is less flexible: After the job setup is steered by the master, the `Process()` method is called for each packet on one worker node. The results are collected by the client, who submitted the job, after the processing on the worker nodes is done.
PROOF is especially designed to process data organized in `TTrees`, which makes adaption of data analysis that is already processing a `TTree` straightforward. However for adapting the MapReduce paradigm for ROOT-based data analysis is more challenging. Possible key/value pairs need to be identified and the structure of the input data must be further investigated.

3.2. The example HEP analysis

To investigate the applicability of MapReduce and the cloud computing paradigm to LHC data analysis, we inspect an example HEP analysis that is representative for many of the analyses tasks conducted at Tier-2/Tier-3 facilities. The example analysis is provided by researchers from the Max Planck Institut für Kernphysik (MPIK) in Heidelberg, who are members of the LHCb collaboration [26]. They also provided a toy Monte Carlo event generation program that uses a standalone version of the PYTHIA-8 [60] event generator. This program generates simulated particle collision data similar to the data collected at CERN. Additionally, it simulates how the products of the events would be detected by the LHCb detector. The output data of this simulation contains the traces of the involved particles inside the detector for each event. These traces are called *tracks* and contain additional information, e.g. the charge and momentum of the traced particles.
The provided example HEP analysis processes the event data and counts the appearance

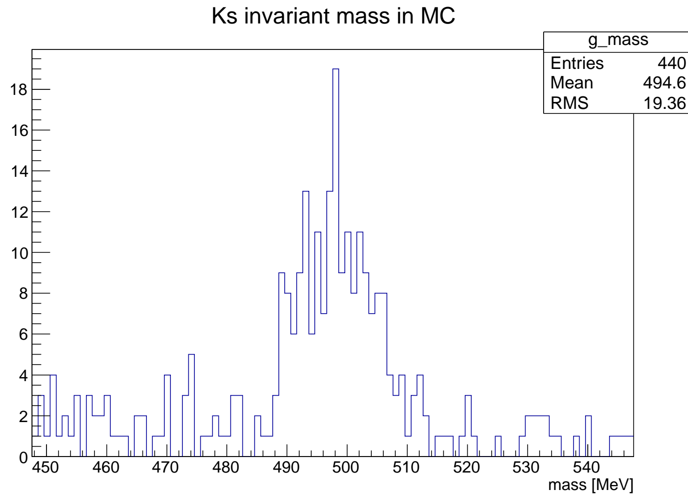

Figure 3.1.: Output of the example HEP analysis running on 20000 events.

of a specific particle, namely the *K-Short* particle. These particles can be identified by the decay into two particles called *pions*. One positively charged pion and one negatively charged pion, which do not decay any further inside the (simulated) detector. The K-Short particles are identified by their mass, which can be extracted from the pion tracks. Therefore each pair of tracks, where one track belongs to a positively charged particle and the other one belongs to a negatively charged particle, are considered. For each of these pairs the point, where the particles that are associated to the tracks have the closest distance (*point of closest approach*), is calculated. If the particles come close enough, the point of closest approach is considered to be the point, where a K-Short particle decayed into the two pions. The outcome of the analysis is a histogram that depicts the reconstructed particle masses in a certain range. Figure 3.1 shows the typical output histogram of the analysis. It depicts the result of analyzing 20000 events.

3.2.1. Input data

The input data for the example HEP analysis, which is generated by the generation program, is stored in a ROOT-based file format called Simple Data Format (SDF), which was developed by our colleagues from Heidelberg. In the SDF file format, data is stored in a specialized set of data classes, implemented in C++. It is designed to store the most important information of reconstructed tracks as well as the associated Monte Carlo data. As

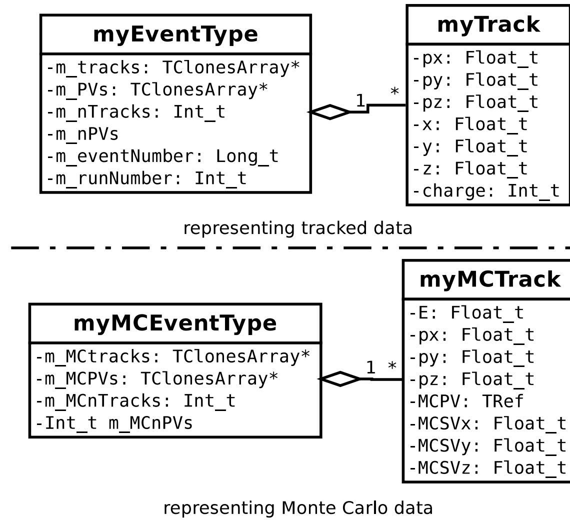

Figure 3.2.: Data class hierarchy of example study.

shown in Figure 3.2, the data class hierarchy consists of four classes:

- `myEventType`: This class stores the event data that was recorded with the help of the (simulated) detector. It also holds a collection of the reconstructed particle tracks.
- `myTrack`: The class `myTrack` implements the reconstructed particle tracks.
- `myMCEventType`: The class `myMCEventType` stores the original event data that was generated with the Monte Carlo event generator. It also stores a collection of generated particle tracks.
- `myMCTrack`: The original particle tracks, as generated by the Monte Carlo generator, are stored in the class `myMCTrack`.

Instances of these classes are stored in a `TTree` (c.f. Section 2.2.3), named `m_outTree`. The `m_outTree` has two branches, the `event` branch, which stores class instances of the type `myEventType` and the `MCevent` branch, which stores instances of `myMCEventType`.

3.2.2. Implementation

The analysis is implemented by a C++ program called *SDFreader* that is linked to the ROOT framework to exploit its functionality. The analysis implemented in the SDFreader reads data stored in events. We give an overview of its main building blocks in the following. Listing 3.1 shows the main loop for the event analysis. Since we are interested in a particle that decays into a positively and a negatively charged particle, we need to investigate all

```
for (ev = 0; ev < maxEntries; ev++){
  ...
  tree->GetEntry(ev);
  for( int i = 0; i < event->getNTracks(); i++){
    tmp = event->getTrack(i);
    if(tmp->getCharge() > 0){
      pTracks.push_back(*tmp);
    }
    else{
      nTracks.push_back(*tmp);
    }
  }
  loopOverTracks(event, pTracks, nTracks, hasMC);
}
```

Listing 3.1: Event loop in the original example HEP analysis code.

pairs of positively and negatively charged tracks. The outer loop (lines 1–14) runs over all events in the input data. In line 3 the complete event data is read from the input file. The inner for-loop (lines 4–12) runs over all tracks in the event and divides them into two lists: `pTracks`, the tracks belonging to positively charged particles and `nTracks`, the tracks belonging to negatively charged particles. In line 13, the method `loopOverTracks(...)` is called, which is shown in Listing 3.2. This method implements the looping over all pairs of tracks, where the first track belongs to a positively charged particle and the second track belongs to a negatively charged particle. The outer loop (lines 4–23) runs over the tracks of particles with positive charge and the inner loop (lines 6–22) runs over all tracks from negatively charged particles. The squared particle mass is calculated in the lines 9–13 with help of the information given in the tracks. Since a K-Short particle has a mass of approximately 498 MeV/c^2, [1], particles with a calculated mass below 400 MeV/c^2 or higher than 600 MeV/c^2 are not considered any further (lines 16–18). The mass is filled into the histogram in line 20. Note that additional selection mechanisms are implemented which are not shown in the listings. For example, only track pairs are considered that are close enough to be originated from the same point, since they must belong to pions that are the products of the same decaying K-Short particle.

3.3. Application of MapReduce to the example analysis

Since the events are analyzed independently, we can accelerate the analysis by processing events in parallel. Each event is investigated whether it contains particles with certain characteristics (in the example study: a certain mass) and at the end the output is merged. Since this is a two-step procedure, it can be formulated in a MapReduce manner: Events are

[1] $1eV/c^2 \approx 1.783 \times 10^{-36}kg$

```
1  void loopOverTracks(myEventType *event, list<myTrack> &pTracks, list<myTrack> &
      nTracks, Bool_t hasMC){
2    ...
3    // loop on positive tracks:
4    for(itPTrack = pTracks.begin(); itPTrack != pTracks.end(); itPTrack++){
5      // loop on negative tracks:
6      for(itNTrack = nTracks.begin(); itNTrack != nTracks.end(); itNTrack++){
7          ...
8          // calculate mass square:
9          mass2 = (m_pi*m_pi + pp2) + (m_pi*m_pi + pn2)
10           + 2.0*sqrt(m_pi*m_pi + pp2)*sqrt(m_pi*m_pi + pn2)
11           - (itPTrack->px + itNTrack->px)*(itPTrack->px + itNTrack->px)
12         - (itPTrack->py + itNTrack->py)*(itPTrack->py + itNTrack->py)
13         - (itPTrack->pz + itNTrack->pz)*(itPTrack->pz + itNTrack->pz);
14
15        // mass cut
16        if(mass2 < 400*400 || mass2 > 600*600){
17          continue;
18        }
19        ...
20        g_mass.Fill(sqrt(mass2)); // fill mass histogram
21        ...
22      }
23    }
24  }
```

Listing 3.2: Per-track pair analysis in the original example HEP analysis code.

passed to map functions, which do the event-level analyses and produce new key/value pairs (in the example case study representing the mass), which are passed to reduce functions that do the statistical analyses (in the example case study producing histograms).
Since we are going to process multiple input files and the events are numbered inside the file scope, we can provide this information in the key/value pairs passed to the mapper. Therefore we define the input key/value pairs as

```
key   := <file path of the event to be processed>
value := <event number in file>
```

The intermediate key/value pairs need to be defined such that they represent the characteristics of interest. In the example case study we are looking for particles with a certain mass. Therefore the intermediate key value/pairs are chosen as

```
key   := <mass of the particle>
value := <number of the observed particles with mass>
```

In the reduce phase of the job, the intermediate key/value pairs are collected and a histogram is created with the help of ROOT.
The main challenge is distributing the input data and producing the initial key/value pairs efficiently. As described in Section 2.3, MapReduce was initially developed for data processing in Google's compute clusters. Therefore, its initial application was text-based input such

as logs or index-files. As a result, input data stored in a ROOT-based file format cannot be processed by Hadoop out of the box due to its complex binary structure.
To be able to process ROOT files with Hadoop, we need to define how the input data should be read by the framework. The Hadoop API, in general, offers three base classes that define how input data is handled:

- `InputSplit:` One `InputSplit` defines the input that is passed to a single mapper. It typically consists of several key/value pairs that serve as input for a `map` function call.
- `RecordReader:` This class defines how an `InputSplit` that is passed to a mapper should be read. It especially defines getters, which return single key/value pairs that are read in `map` function calls.
- `InputFormat:` This class provides the base for all input formats defined for Hadoop. The most important methods are `getSplits(...)` which returns an array of `InputSplits`, and the method `getRecordReader(...)` which returns a `RecordReader` which is able to read a corresponding `InputSplit`.

For example, for text based input, the `getSplits(...)` method could split a text file into several blocks. One `InputSplit` would represent a number of lines from the input data and the `RecordReader` would define how these lines are converted into key/value pairs.
Splitting up ROOT-based files however is more complex, since it is not clear where the data belonging to one event is stored in the file. For input data consisting of several files, one could be tempted to represent each file by a single `InputSplit`, but this leads to two problems.
First, we have no control over the granularity of the subtasks assigned to each mapper. If we work with big input files, the subtasks assigned to the mappers could take a long time to complete. This could result in a situation, where the analysis completion is delayed by a single slow worker node in the cluster. Additionally, if the number of input files is smaller than the number of worker nodes in the cluster, some of the nodes do not get any subtasks assigned, which keeps them idle during the analysis.
The second problem is also connected to the file size. The `JobTracker` tries to schedule subtasks to worker nodes, where the corresponding input data is stored locally. This means that `InputSplits` should be stored completely on single nodes. If the data summarized in one `InputSplit` is spread across multiple blocks in HDFS, it becomes very likely it is also spread across multiple `DataNodes`. Assuming now that the input file size is larger than the block size in HDFS and each file is regarded as one `InputSplit` they cannot be handled efficiently.
As a result, we need to analyze the physical structure of the input data to be able to define an input format that splits the data efficiently.

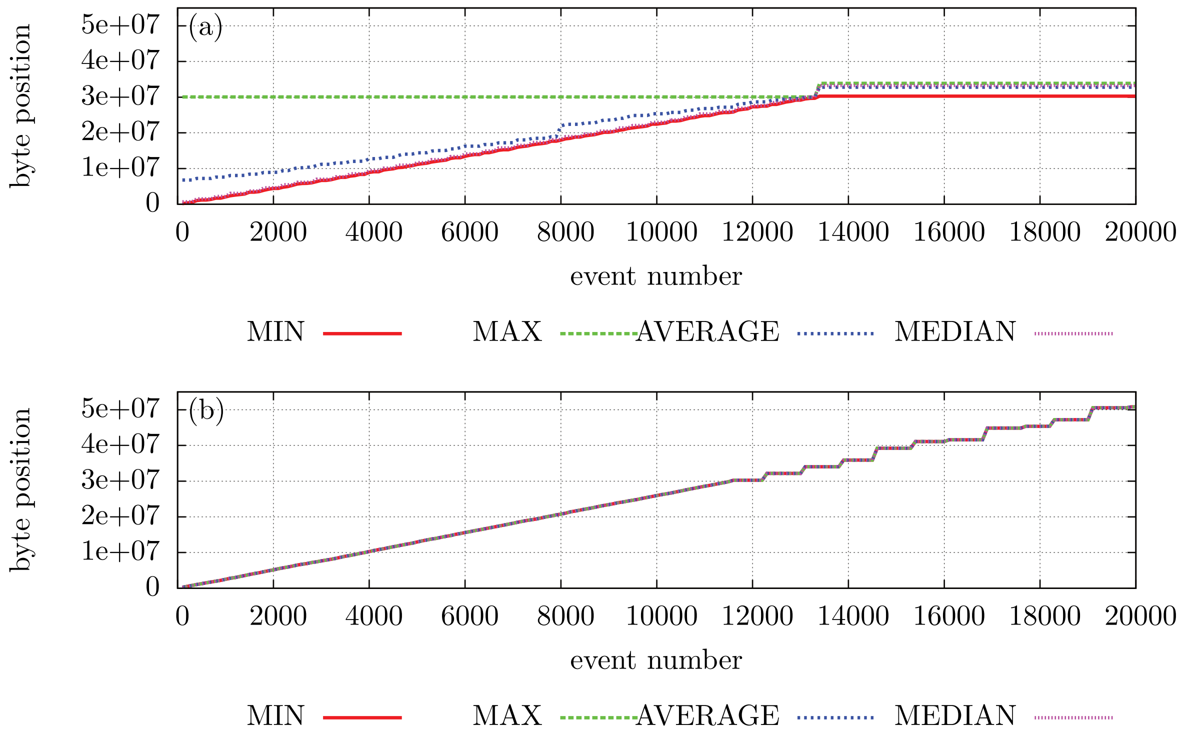

Figure 3.3.: Data distribution in an example SDF file storing 20000 events.

3.4. The physical structure of the input data

As described in Section 2.2.3, ROOT uses a complex binary format to store data on disk. `TTree`s are utilized and the member variables of classes are split into different branches. Variables inside the same branch are written to fixed-size buffers first, which are compressed and written to disk when they are full. Since member variables can be of complex compound types themselves, subbranches with associated output buffers can be created. The depth, to which subbranches are created, is controlled by the `splitlevel`, which is set when branches are created. By default, ROOT uses the maximum `splitlevel` of 99, when a branch is created. In the SDF data format instance of the classes `myEventType` and `myMCEventType` are stored in two separate branches. By setting the `splitlevel` in the data generation process we can control the depth to which subbranches are created for their member variables. We define the *byte position* as the relative offset of a byte from the file beginning. The analyzed file consists of 20000 events. This implies that it stores 20000

instances of `myEventType` as well as 20000 instances of `myMCEventType`. The example HEP analysis processes the data of the reconstructed events stored in instances of `myEventType`. Therefore, we refer to the data of a single `myEventType` instance as an *event* in the further proceeding.
For the analysis of the physical data structure, all bytes in the input file belonging to a single event (all data from its member variables) are considered. Figure 3.3 shows a comparison of the event data structure inside a SDF file, once created with the default `splitlevel=99` (plot (a)) and once with a `splitlevel=0` (plot (b)). The total file size is around 50 MB. The minimum observed byte position (MIN), the maximum observed byte position (MAX), the average byte position (AVERAGE), and the median byte position (MEDIAN) of the per-event data is calculated. When the `splitlevel` is set to 99, a subbranch is created for each member variable. Plot (a) of Figure 3.3 shows that this results in the distribution of the event data throughout the whole file. However, when we consider the average and median values, we observe that most of the per-event data is stored within some proximity. When we reduce the `splitlevel` to 0, the per-event data is stored sequentially at a certain byte position inside the file. Since not only the reconstructed event data is stored, but also the Monte Carlo truth (stored as instances of `myMCEventType`), the data for the events is not distributed throughout the whole file in plot (a).
We can observe that even when the data belonging to one event is not stored continuously inside a file, the event associated data is clustered around certain byte positions. This allows us to assign certain byte positions to each event.

3.5. Definition of HDFS block selection workflow

When we know the byte positions of the event associated data, we can use this information to find the blocks in HDFS. Knowing the blocks in HDFS, we are able to split the data efficiently with a customized `InputFormat` in Hadoop. Figure 3.4 shows the general per input file work flow for mapping the event data. Events are numbered inside the file scope starting with 0. In the flow chart `currEvent` is the index of the event that is investigated and `totEvents` is the total number of events stored in the input file. For each event in the input data, we use the information in the ROOT file record headers to find the byte positions of the baskets where the event member variables are stored (step 1). Subsequently, we use this information to extract the minimal (MIN), maximal (MAX), average (AVERAGE) and median (MEDIAN) per event byte position (step 2). The outcome is similar to the one we have shown in the Figure 3.3 of the last section. In step 3 we utilize this information to map the event under consideration to a certain file block in HDFS. Here, different strategies are possible. For example we can map the event to the block, that holds the byte at the median byte position or we try to combine the calculated per event metrics to optimize the mapping

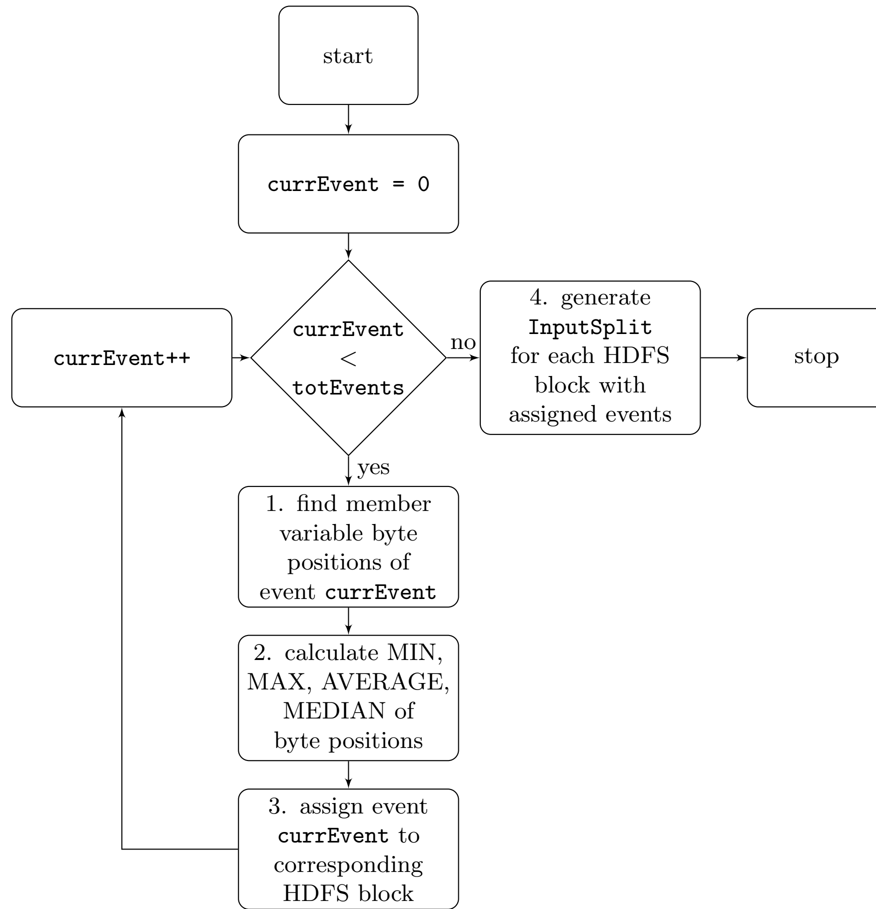

Figure 3.4.: Workflow of the HDFS block selection.

further. Having mapped all events to certain blocks in HDFS, we generate one `InputSplit` for each block containing the indexes of the associated events (step 4).

3.6. Remarks on general applicability

The analysis of the physical file structure in Section 3.4 revealed that, even if the per-event data is split into several branches, it is stored around certain byte positions inside the file. This led to the definition of the general HDFS block selection workflow in Section 3.5. Here we use the calculated byte positions to assign events to certain blocks in HDFS. This approach is not limited to the SDF data format, but can also be used for all kinds of binary

input data, provided that the bytes belonging to the logical data records are clustered inside the file. However, if the per-record data is evenly distributed over several blocks in HDFS, it becomes impossible to split the data such that it can be efficiently processed by Hadoop. The probability for such a scenario becomes higher, if the size of the individual data records increases. This problem can also appear with ROOT-based data formats, since the stored class structures can get arbitrarily complex. For the given SDF data format however, the individual data records are small (about a few KB) and there is a good chance that most of its data is stored on a single HDFS block. Thus the input data can be split efficiently with a customized input format in Hadoop. The next chapter describes how our approach is implemented.

4. Implementation

In this chapter we describe the implementations of the analysis in the Hadoop framework as well as in the `TSelector` framework in PROOF. For reading ROOT files from HDFS, an already existing ROOT plugin is used. We start by introducing this plugin in Section 4.1. The description of the implementation of the map and reduce function in the Hadoop framework is given in Section 4.2. Subsequently, we describe our customized input format in Section 4.3. In Section 4.4, the analysis implementation within the PROOF framework is discussed. Since we need to generate big amounts of input data for our evaluations, we parallelized the Monte Carlo data generation within the Hadoop framework. Section 4.5 introduces the implementation of this parallelization.

4.1. The HDFS plugin for ROOT

The base class for all file operations in ROOT is `TFile`. Derived classes that handle the file operations on different underlying file systems exist. These classes are registered with the help of a plugin manager and loaded when they are needed. Rudimentary HDFS support for ROOT is implemented within this framework. The `TFile` class provides is the base class for all file operations in ROOT. It provides the method `Open(const char* name, ...)` that implements the opening or creation of files. The scheme of the determined URL passed by `name` is used internally to determine the subclass to which the open operation is redirected. If, for example, the `name` starts with "root:" a `TNetFile` object is created which handles data operations on the SCALLA file systems. If the name starts with "http:", the `TWebFile` class is responsible for handling the request. The subclasses managing the file operations are managed by corresponding subclasses of `TSystem`, e.g. `TNetSystem`, which represent the underlying file system itself. The rudimentary HDFS support is implemented by the classes `THDFSFile` and `THDFSSystem`. Data can be read from files stored in HDFS, but it currently does not provide any write support. Unfortunately `THDFSFile` and `THDFSSystem` are developed against a deprecated version of the HDFS C-API. Small adjustments had to be made as part of this thesis to tweak the classes to work with the new API and be able to read data directly from HDFS with ROOT and PROOF. The adjustments are documented in Appendix B.

4.2. Analysis implementation in Hadoop

We discuss our design decisions in the next section. Subsequently, we describe the implementations of the mapper and reducer for Hadoop in more detail. Thereby, we focus on the parts that are different from the implementation of the example HEP data analysis code.

4.2.1. Design decisions

Since we want to minimize the changes that need to be made to the original analysis code and need to link to the ROOT libraries, the analysis in the Hadoop framework is implemented in C++. As described in Section 2.4, there are two possibilities to implement map and reduce functions in other languages than Java: Hadoop Pipes and Hadoop Streaming. Some time was spend on developing a solution based on Hadoop Pipes in the early stage of our investigations, but it turned out to be slow and the approach was discarded. Additionally, Hadoop Pipes lacks of proper documentation and does not have a broad user community. Therefore we decided to use Hadoop Streaming.
The mapper and reducer in Hadoop Streaming are implemented as independent executables. The code modularity and the programming language can, therefore, be chosen freely. It must only be assured that both programs read their input from the standard input stream line by line, whereby each line corresponds to a key/value pair. The key/value pairs are defined as described in Section 3.3. In our implementation, both mapper and reducer are implemented as single C++ classes: the `StreamingMapper` and the `StreamingReducer`. The compiled programs are linked against the ROOT libraries.

4.2.2. The StreamingMapper

The looping through the events in the original HEP analysis code is transformed to reading the standard input line by line, where each corresponds to one key/value pair. The map function uses ROOT to read the event at the given index. The mapper loops through all particle tracks in one event and calculates the particle mass. If the mass is in the desired range, the mapper writes the value to standard output, from which it is read by the reducer. Listing 4.1 shows the shortened implementation of `StreamingMapper` main function. Data is read from standard input line by line in the while-loop in line 5. The filename and the event index are extracted from the input line in the lines 7 to 11. To not repeat the creation of a file object for each line, the last filename is stored in `oldname` and compared to the newly read filename (line 13). If the filename has changed, a new `TFile` object is created in line 15. In the lines 17 to 19, the address of the `TTree` is obtained from the file and the address of the event to be read is set to the correct branch address. The event is read from file in line 23. The for-loop starting in line 25 matches the for-loop of the original HEP analysis code provided in Listing 3.1.

```
 1  int main(int argc, char **argv) {
 2    ...
 3    myEventType *event = NULL;
 4    ...
 5    while (std::cin.getline(inputline, 256)) {
 6      ...
 7      int tab = line.find('\t', 0);
 8      if (tab != -1){
 9        filename = line.substr(0,tab);
10        ev = HadoopUtils::toInt(line.substr(tab,line.size()));
11      }
12      ...
13      if (oldname != filename){
14        ...
15        hfile = TFile::Open(TString(filename));
16        ...
17        tree = (TTree*) hfile->Get("m_outTree");
18        branch = tree->GetBranch("event");
19        branch->SetAddress(&event);
20        ...
21        oldname = filename;
22      }
23      tree->GetEntry(ev);
24
25      for (int i = 0; i < event->getNTracks(); i++){
26        ...
27      }
28      loopOverTracks(event, pTracks, nTracks, hasMC);
29    }
30  }
```

Listing 4.1: Implementation of the `StreamingMapper`.

The method `loopOverTracks(...)` is altered such that, instead of writing the mass to a histogram, it is written to standard output from where it can be read by the reducer.

4.2.3. The StreamingReducer

The reducer is used to collect the data from the different map jobs and produce the histogram of the mass distribution. It reads the intermediate key/value pairs, which represent a certain mass and its count from standard input. For the creation of the histogram, the ROOT libraries are used. Since the HDFS plugin for ROOT is only able to read data, but not to write files directly to HDFS, the file containing the histogram is written to the local disk first and subsequently transferred to HDFS with the Hadoop Command Line Interface (CLI).

```
1  int main(int argc, char* argv[]){
2    ...
3    TH1D outputHist = TH1D("g_mass", "Ks invariant mass", 100,
4                           497.61 - 50, 497.61 + 50);
5    ...
6    outputFile = TFile::Open("/tmp/mapreduce_out.root", "RECREATE");
7
8    while(std::cin.getline(filesplit, 256)){
```

```
 9      std::string line = std::string(filesplit);
10      int tab = line.find('\t', 0);
11      if (tab != -1){
12        mass = HadoopUtils::toFloat(line.substr(0,tab));
13        ...
14        outputHist.Fill(mass);
15      }
16    }
17    ...
18    system("hadoop dfs -put /tmp/mapreduce_out.root data-out/mapreduce_out.root");
19    ...
20  }
```

Listing 4.2: Implementation of the `StreamingReducer`.

Listing 4.2 shows the important parts of the `StreamingReducer`. The histogram is created in line 3. In line 6, the output file in the local file system is opened. The while-loop in the lines 8 to 16 reads the mass values from standard input, one per line and adds them to the output histogram. In line 18, the Hadoop CLI is used to transfer the created file to HDFS.

4.3. The input format implementation

In this section, the implementation of the customized input format is described. The main challenge that needs to be addressed is connected to the fact that input formats for Hadoop are implemented in Java but the libraries we need to utilize are programmed in C++. The next subsection describes the resulting design decisions. The customized input format is called the `RootFileInputFormat` and is introduced subsequently. We also describe the implementation of an input format, called `StreamingInputFormat`, which was done in the scope of our preceding feasibility study [44], since we use it in a comparison in the later evaluation.

4.3.1. Design decisions

While Hadoop Streaming can be used to implement map and reduce functions in C++, it does not provide a way to define a customized `InputFormat` in a language other than Java. This leads to the following problem: Utilizing a library written in C++ to obtain information about the input files during the splitting of the data in the `InputFormat` is not possible without further ado. For the steps 1. and 2., depicted in Figure 3.4, we need very specific information about the input file structure which can be only obtained with the help of specialized code libraries.

For reading ROOT files with Java, the SCaVis framework [29] exists. It has been developed to provide a fully featured Java framework for scientific computations. Unfortunately it is not able to read ROOT based files directly from HDFS. This problem could be solved by mounting HDFS with the help of FUSE [14]. Since this leads to a performance degradation

of 20–30% and we are trying to be as independent as possible from other code libraries this approach was discarded.
Instead we are following another approach for the implementation: step 1. and 2. of Figure 3.4 are removed from the responsibility of the `InputFormat` and implemented in as a tool we call *RootFileMapper*. This tool is responsible for the mapping process and creates files that we call *event maps*. One event map is a simple text file, which contains the minimal, maximal, average and median byte position of the events. Thereby each line corresponds to the data of one event and the line number corresponds to the index of the event in the ROOT file. These files are than read by the `InputFormat` implementation, which does the final mapping of events to blocks in HDFS and creates the splits for the data distribution to the mappers.

4.3.2. The RootFileMapper

The C++ RootFileMapper reads input file paths from the standard input line by line and creates the event maps described above. It reads the input file information from HDFS using the `THDFSFile` and stores the event maps in the same folder in HDFS utilizing the HDFS C-API.
Listing 4.3 shows how the functionality of the ROOT libraries is exploited to extract the byte positions in the `mapHDFSFile(...)` function of the RootFileMapper.

```
1  void FileMapper::mapHDFSFile(const char* inputFileName, char* host, int port){
2    ...
3    // loop over events
4    for (int i = 0; i < inputTree->GetEntries(); i++){
5      std::vector<Long64_t> posVec;
6      TIter next(&branchList);
7      TBranch* currBranch;
8      ...
9      // loop over branches
10     while ((currBranch = (TBranch*) next())){
11       Long64_t basket = TMath::BinarySearch(
12         (Long64_t) (currBranch->GetWriteBasket() + 1),
13          currBranch->GetBasketEntry(), (Long64_t)i);
14       TBasket* currBasket = currBranch->GetBasket(basket);
15       if (currBasket == 0)
16         continue;
17       Long64_t seekkey = currBranch->GetBasket(basket)->GetSeekKey();
18       Long64_t size = currBranch->GetBasket(basket)->GetNbytes();
19       ...
20       Long64_t end = seekkey + size;
21
22       pos = (seekkey + end)/2;
23       posVec.push_back(pos);
24     }
25     ...
26   }
27 }
```

Listing 4.3: Utilizing ROOT for the event mapping.

The method takes the file name, the IP address of the NameNode instance and the port on which the HDFS file system can be reached as input arguments to connect to the file system and open the file. Instead of reading the whole file to get the byte positions, the header information stored in the record headers in the ROOT files is used to directly obtain the desired information. For each event, we loop over the branches that store the event data. The byte positions are stored in the vector posVec defined in line 5. For each branch, we obtain the basket in which the entry is stored in line 11 to 13. In line 17 and 18 the byte position of the basket in the file (seekkey) and the size of the basket are obtained. The last byte of the basket is calculated in line 20. The position of the basket is defined as its middlemost byte position in line 23 and the position is stored in the vector in line 24. The positions stored in the vector are later on used to calculate the minimal, maximal, average and median byte positions.

4.3.3. The RootFileInputFormat

The class architecture of the Java input format implementation is depicted in 4.1. The class InputFormat defines how the input is divided into InputSplits and returns them by a call to its getSplits() method. The StreamingInputSplit class implements an InputSplit that stores a range of event indexes and the corresponding file. The StreamRecordReader implements the corresponding RecordReader that is used by the mappers to extract the information from the defined splits. The class FileInputFormat is Hadoop's base class for file based input formats. Each InputSplit defines the input that is processed by a single map job. It provides the information on which nodes of the cluster the data can be found (method getLocations()). Each mapper uses a RecordReader class which knows how the InputSplit needs to be read. The RecordReader offers the method next(), which is called by the Hadoop framework to get the next key/value pair. A RecordReader can also implement the getProgress() method, which reports the progress through the InputSplit, which is processed by the RecordReader. The method isSplittable() offered by the FileInputFormat returns whether an input file can be split or needs to be processed as a whole.

The RootFileInputFormat is derived from the FileInputFormat. Internally it uses an EventMapSplitter and an EventMapImporter. The EventMapImporter is used to import the event maps generated by the RootFileMapper. One event map is represented by a data class called EventMap. The method getSplits(...) of the RootFileInputFormat delegates the splitting to an implementation of the EventMapSplitter, which has a generate-InputSplits(...) method that takes the imported event map, the file for which the InputSplits should be generated and the locations of the blocks in HDFS as input arguments. It returns a list of generated InputSplits that define how the input data should be divided among the different map jobs.

The BasicEventMapSplitter implements a simple heuristic to finally map the

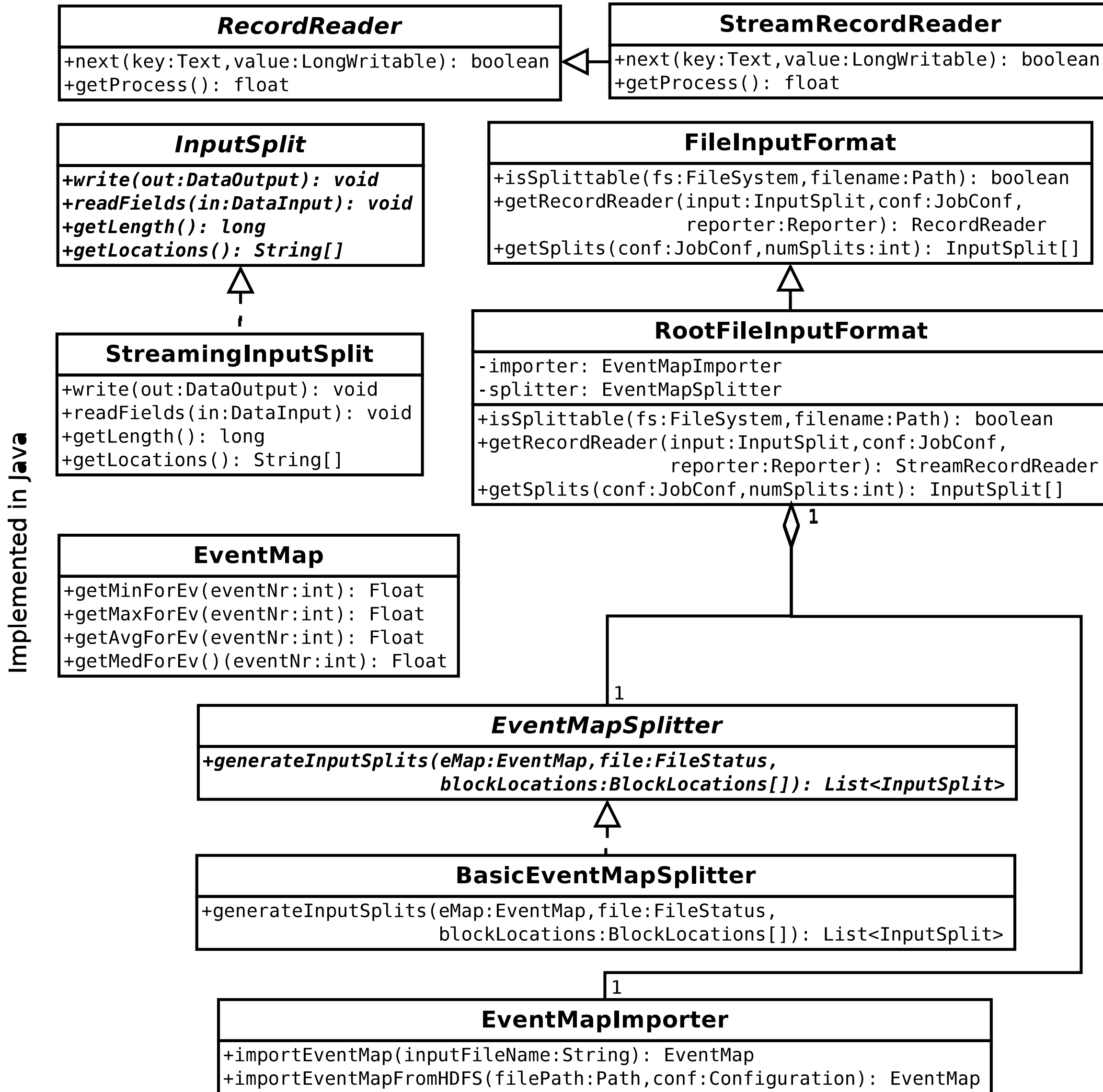

Figure 4.1.: Class diagram for `RootFileInputFormat`.

```
1  public List<InputSplit> generateInputSplits(FileStatus file, EventMap eMap,
2          BlockLocation[] blockLocations) throws IOException{
3    ...
4    for (int eventNr = 0; eventNr < eMap.getSize(); eventNr++){
5    // just in case there is a block which contains no data.
6      while (eMap.getAvgforEv(eventNr) > limit){
7        if (startEvent < endEvent - 1){
8          endEvent = eventNr - 1;
9          splits.add(new StreamingInputSplit(rootCompatiblePath,
10             startEvent, endEvent,
11             currentBlockLocation.getHosts()));
12         startEvent = eventNr;
13       }
14       currentBlockIndex++;
15       currentBlockLocation = blockLocations[currentBlockIndex];
16       limit = currentBlockLocation.getOffset() +
17           currentBlockLocation.getLength();
18     }
19   }
20   splits.add(new StreamingInputSplit(rootCompatiblePath,
21     startEvent, eMap.getSize() - 1, currentBlockLocation.getHosts()));
22
23   return splits;
24 }
```

Listing 4.4: Basic heuristic for mapping events to blocks in HDFS.

events to blocks inside HDFS. Listing 4.4 shows the important parts of the `generateInputSplits(...)` method. The loop starting in line 4 is looping over all events that are listed in the event map. We are assigning the events to the blocks where their average byte position is stored. Since the array with the block locations stores the blocks in ascending order of their position in the file, we can start from the beginning of the array. The `limit` is set to the last byte of the first block that contains data at the beginning and `startEvent` is set to 0. We increase the event index `eventNr` until we find the first event, which average byte position is not on the current HDFS block (line 6). Since we know that the events have increasing average byte positions, we can now create the `StreamingInputSplit` for the current HDFS block (line 9–11). After that the `startEvent` is set to the current event number (line 12) and we take the next block in HDFS into account (line 14–15). In line 16 the next limit is set to the last byte of the new HDFS block.
The increasing of the current block index is done in a while-loop (line 6–18), since it might be that there are blocks that contain no event data at all. For this case, we also need the if-condition in line 7 to avoid adding each `StreamingInputSplit` more than once.

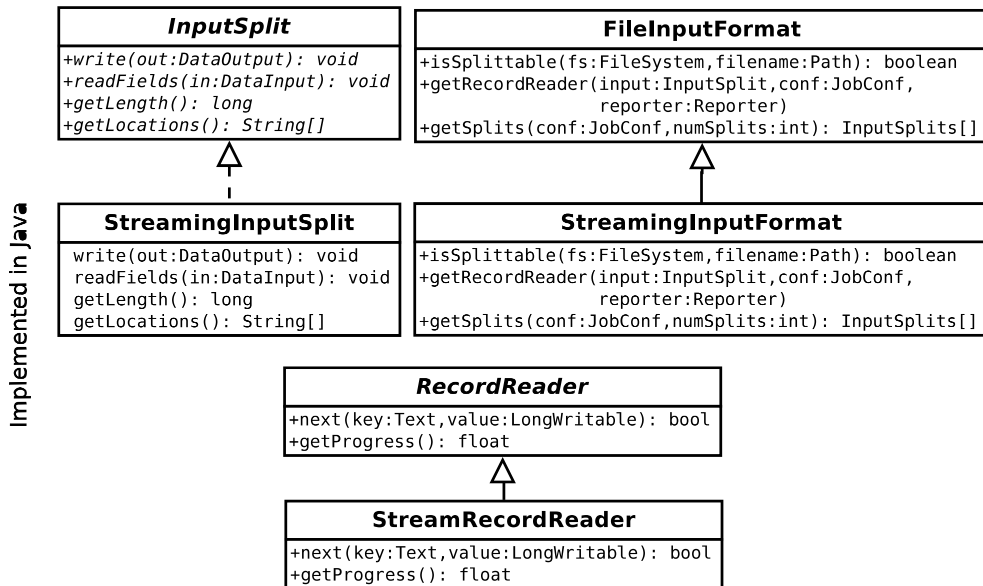

Figure 4.2.: Class diagram for `StreamingInputFormat`.

4.3.4. The StreamingInputFormat

The `StreamingInputFormat` implements our first approach to split the ROOT data and was developed during our preceding feasibility study [44]. Figure 4.2 shows its class hierarchy. It uses the same `StreamingInputSplit` and `StreamRecordReader` as the `RootFileInputFormat`. The problems that arise when the input files are bigger than the blocks in HDFS are not addressed. The `getSplits(...)` method of the `StreamingInputFormat` returns `StreamingInputSplits` with a fixed size of 25000 events.

4.4. Analysis implementation in PROOF

Given a file, which stores data in a `TTree`, we can utilize ROOT to automatically generate a skeleton for a `TSelector` that is able to process the data. Listing 4.5 shows the steps that need to be executed on the ROOT command line to generate a `TSelector` for a given input file.

```
1 root [0] TFile *f = TFile::Open("treefile.root")
2 root [1] TTree *t = (TTree *) f->Get("T")
3 root [2] t->;MakeSelector("MySelector")
4 root [3] .!ls MySelector*
5 MySelector.C  MySelector.h
```

Listing 4.5: Generating a `TSelector` *[25].*

Hereby “treefile.root” is the name of the file containing the input data. “T” is the name of the TTree and the created TSelector is called “MySelector”.
The analysis code is divided into the methods described in Section 2.2.2 and the implementation is described in more detail in the next subsections. The analysis is submitted to the PROOF cluster with a wrapper C++ program that is defining the input data and the TSelector which should be utilized for the processing.

4.4.1. MySelector::SlaveBegin()

The SlaveBegin() method is used to create a histogram which is later filled with the calculated particle mass in a specific range. The difference to the original analysis code is that each worker creates its own histogram object. These objects are transferred to the client that submitted the job, which then creates the final output. The skeleton of the SlaveBegin() method is given in Listing 4.6. The histogram is created in line 4 and added to the output in line 5.

```
1  void MySelector::SlaveBegin(TTree * tree)
2  {
3     ...
4     gmass = new TH1D("g_mass","Ks invariant mass", 100, 497.61 - 50, 497.61 + 50);
5     fOutput->Add(gmass);
6  }
```

Listing 4.6: Implementation of MySelector::SlaveBegin().

4.4.2. MySelector::Process()

The Process(Long64_t entry) method is used to implement the per-event analysis. It is called for each event in the TTree in the input data, whereby entry is the event index in the file. The per-event analysis is implemented as in the original analysis. Listing 4.7 shows the skeleton of the Process() implementation. The TTree, which stores the input data, is automatically read by the TSelector framework and is accessible through the fChain variable. In line 4, the complete event is read from the input data. The for-loop, which starts in line 6, implements the event based analysis as described in Listing 3.1.

```
1  Bool_t MySelektor::Process(Long64_t entry)
2  {
3    ...
4    fChain->GetTree()->GetEntry(entry);
5    ...
6    for( int i = 0; i < event->getNTracks(); i++){
7      ...
8    }
9    loopOverTracks(event, pTracks, nTracks, hasMC);
10   ...
11 }
```

Listing 4.7: Implementation of MySelector::Process().

4.4.3. MySelector::Terminate()

The `Terminate()` method implements the generation of the final histogram file. It is called on the client and the final histogram is written to the directory from which the job was submitted to the PROOF master. The implementation is shown in Listing 4.8. The final histogram is obtained from `fOutput` in line 2. Note that the histograms, created by the workers, are automatically merged by the framework. Final formatting of the histogram is done in the lines 8 and 9 and the output file is created in line 12. In line 14 the histogram is written to the output file.

```
1  void MySelector::Terminate(){
2    gmass = dynamic_cast<TH1D*>(fOutput->FindObject("g_mass"));
3    ...
4    if (gmass == 0){
5      Error("Terminate", "gmass = %p", gmass);
6      return;
7    }
8    gmass->SetMinimum(0.0);
9    gmass->GetXaxis()->SetTitle("mass [MeV]");
10
11   TString fileName = "test.root";
12   TFile* outputFile = TFile::Open(fileName, "RECREATE");
13
14   gmass->Write();
15 }
```

Listing 4.8: Implementation of `MySelector::Terminate()`

4.5. Parallelization of event data generation

The Monte Carlo data generation is a time-consuming task. Generating 3×10^5events on m1.medium-instance in the Amazon EC2 cloud takes around halve an hour to complete. Since we aim to scale the input data up to 10^8 events, using just one instance to generate the data could easily take a few days. Additionally, storing the whole generated data in the local storage of one instance could easily result in data loss. Therefore the generation and storage of the generated data was mapped to Hadoop and HDFS. As described in section 2.4.3 programs that read data from standard input line-by-line could be used in a MapReduce manner with the help of Hadoop Streaming. The fact that the `bash` script, used to generate the data, reads the number of events from standard input makes it a candidate to be used as a `map` function. The total number of events to be generated is split up into chunks of equal size. For each chunk a text file is created that just contains the number of events in this chunk. When defining these files as input in Hadoop, they are treated as individual `InputSplits` and thus assigned to individual `map` functions by default. Hadoop can be used without specifying a `reduce` function. The `bash` script for generating events was modified, such that it copies the ROOT files that are generated in the second step to

a specified folder in HDFS. In addition it forks a second `bash` script as a subprocess which reports the status of the generation to standard error, which can be read by the Hadoop framework to provide progress feedback during the generation process.
Since we are going to produce a lot of input files in parallel, which should be stored in a common namespace in HDFS, we need to make sure that they receive globally unique names. Therefore we name the files by combining the cluster-unique IP address of the host generating the data with the current timestamp. Since we are making sure that only one map job is running on a node at a time during data generation, this combination is enough to provide global unique names in the common namespace.

5. Deployment

To evaluate the solution developed in the previous chapters and to exploit the applicability of cloud computing to LHC data analysis, we deployed Hadoop and PROOF clusters in Amazon's EC2 cloud. To decide on a suitable (virtual) hardware configuration, we evaluated the different instances types offered by the EC2 cloud. Before deploying a cluster in the cloud, a customized AMI was build and scripts to ease cluster management were developed. A short remark, why we did not use Amazon's EMR is given in Section 5.1. Section 5.2 describes the benchmarks that were conducted on the different instance types and discusses the AMI configuration. Subsequently, Section 5.3 introduces the Ganglia monitoring system, which is used to monitor the Hadoop and PROOF clusters and Section 5.4 discusses the cluster layout in more detail. In Section 5.5, the deployment and management of the clusters is described and in Section 5.6, we summarize the limitations and costs that occur when using Amazon's EC2.

5.1. Remarks on Elastic MapReduce

Initially, the Amazon's EMR was considered to deploy a Hadoop cluster in the cloud. After several trials it turned out that its API does not provide all the functionality that is needed for the project: the Hadoop versions used in EMR are not state of the art and installing all the required software by Bootstrap Actions (compare section 2.6.5) is not very handy. Additional to installations of the Hadoop framework, non-trivial installation of ROOT/PROOF and SCALLA are needed. Therefore the clusters were deployed on top of EC2 manually as described in the following.

5.2. Cluster node configuration

A customized AMI was build with the software required for the evaluation. Additionally the performance of different Amazon instance types was evaluated to decide on the suitable (virtual) hardware configuration for the cluster nodes. We were especially interested in determining the performance difference between instance storage and EBS storage, since we are going to analyze huge amounts of data.
Instance storage is technically realized by the hard disks that reside inside the host computers where the virtual instance is launched. Thus several instances launched on the same

host computer might use isolated parts of the same hard disk. Instance storage volumes cannot be detached from one instance and reattached to another since they are bound to the host that is launching the instance. Data stored on instance storage is lost, when the corresponding instance is terminated.
EBS storage volumes can be used to store data persistently. Snapshots and incremental backups are provided. Technically, they are network drives, which make it possible to detach EBS volumes from one instance and reattach it to another. EBS-storage volumes can also be launched with a guaranteed number of IOPS. The downside of utilizing EBS are extra fees.
We evaluated the usability of instance storage and EBS-storage by performing some read-/write tests on the corresponding volumes on different instance types. Additionally the original HEP analysis described in Section 3.2 was launched on different instance types to get a feeling for the overall instance performance specific to the targeted problem. The configurations of the benchmarked instance types are given in Table 5.1. Each instance comes with a certain amount of instance storage that can be utilized. To evaluate the performance of EBS storage an extra EBS volume was connected to the instance. Note that the highly specialized compute instances in the EC2 cloud were not considered, since we are aiming to build a distributed cluster build from a few hundred nodes instead of launching only a few nodes with high computational capacity. The prices given in the column are the actual prices for running the corresponding instance type in the EU West (Ireland) region for one hour on demand.

5.2.1. Conducted benchmark tests

The conducted benchmarks tests are described in more detail in the following.

Write test with `dd`

For the write test we used the standard Linux tool `dd` to write approximately 1GB of data to the targeted volume. The tool is called from the command line as given below.

```
dd if=/dev/zero of=bigfile bs=512k count=2000 conv=fdatasync
```

Thereby `if` specifies the device where the data is read from, `of` is the name of the output file, `bs` is the block size for the individual blocks written to the volume and `count` is the number of blocks that are written to the device. The option `conv=fdatasync` is used to force `dd` to physically write all data to disk before finishing.

Read test with `dd`

We used `dd` again to perform a read test on the targeted volume. Thereby the written file is read again to measure the read speed. Before the test is conducted, it is made sure that the file did not reside in the memory anymore, by writing another file that is big enough to

Type	Memory	CPU	Inst. Storage	Platform	Price/hour
Standard Instances					
m1.small	1.7 GiB	1 ECU	160 GB	32-bit/64-bit	$0.065
m1.medium	3.75 GiB	2 ECUs	410 GB	32-bit/64-bit	$0.130
m1.large	7.5 GiB	4 ECUs	850 GB	64-bit	$0.260
m1.xlarge	15 GiB	8 ECUs	1690 GB	64-bit	$0.520
High-CPU Instances					
c1.medium	1.7 GiB	5 ECUs	350 GB	32-bin/64-bit	$0.165
c1.xlarge	7 GiB	20 ECUs	1690 GB	64-bit	$0.660
High-Memory Instances					
m2.xlarge	17.1 GiB	6.5 ECUs	420 GB	64-bit	$0.460
m2.2xlarge	34.2 GiB	13 ECUs	850 GB	64-bit	$0.920
m2.4xlarge	68.5 GiB	26 ECUs	1690 GB	64-bit	$1.840
High I/O instances					
hi1.4xlarge	60.5 GiB	35 ECUs	2x1024 GB (SSD)	64-bit	$3.410

Table 5.1.: Configuration of the benchmarked instance types.

occupy all memory of the instance. For the read test the tool is called from the command line as given below.

```
dd if=bigfile of=/dev/null bs=512k
```

The parameters are the same as described above.

Read test with `hdparm`
In the third test we used the standard tool `hdparm` to measure the read speed. The command line call is given below.

```
hdparm -t /dev/xcdf
```

The option `-t` is set to perform timed readings. The second option `/dev/xcdf` is the device from which the data is read. The result is an indication how fast uncached data can be read from the device.

Performance of the example HEP analyses on 20000 events
For the last test we executed the original example HEP analysis code given as a program called `SDFreader` that was reading data from the targeted volume. The program reads data from a file in the same directory containing 20000 events by default and conducts the analyses described in section 3.2. We used the Linux tool `time` to measure the execution time. The program call is given below.

```
time −o result −f ''%e'' ./SDFreader
```

The option `-o` sets the output file. The output format is specified with `-f`. Here `''%e''` determines the time it takes for the program to finish. The program `SDFreader` is executed in the local directory, which was either on an EBS volume or on instance storage.

5.2.2. Benchmark results

Figure 5.1 shows the outcome of the performance benchmarking. The benchmarking was done both on EBS volumes and instances storage and repeated five times for each test. The average value as well as the minimum and maximum measured values are plotted for each instance type. We observe huge performance fluctuations in the read/write speed benchmarks and it is hard to deduce general trends. The instance storage performance varies because of other instances launched on the same host and the EBS storage performance varies because of other traffic in the network. Instance storage delivers better performance in most of the cases but in general its performance is even less predictable when compared to EBS. The measurements with `hdparm` (plot (c)) show much less fluctuations. The peak for the hi1.4xlarge instance happens because of its internal SSD (see Table 5.1), which offers much higher read performance. We have no explanation, why this is not reflected by the `dd` test (plot (a)).
Plot (d) shows the time measurements for analyzing 20000 events with the original example HEP analysis program. When comparing the average measured values, there is only little difference between the underlying storage systems. Performing the analysis on instance storage tend to be a little faster than on EBS storage. The highest performance difference is between the m1.small instance and the m1.medium instance, while the differences to the other instance types are not that distinctive.
Summarizing our results, we observe that the performance of the underlying storage system is subject to some fluctuations and is affected by other processes running in the EC2 cloud. The impact on the analyses execution however is limited.

5.2.3. Final node configuration

The analysis conducted above suggests to use the m1.medium instance for the basic cluster nodes, since it offers fairly good performance for a moderate price. We have seen that instance storage tend to offer better performance in comparison to EBS storage but that this has limited impact on the overall performance of the analysis. When choosing the underlying storage, we also need to consider that we are going to generate data that is stored on the cluster during the whole analysis. If we are forced to restart the cluster, all data stored in instance storage is lost. To provide more flexibility and reliability EBS seem to be the better storage solution for our problem. The final virtual hardware configuration for the basic cluster nodes is given below.

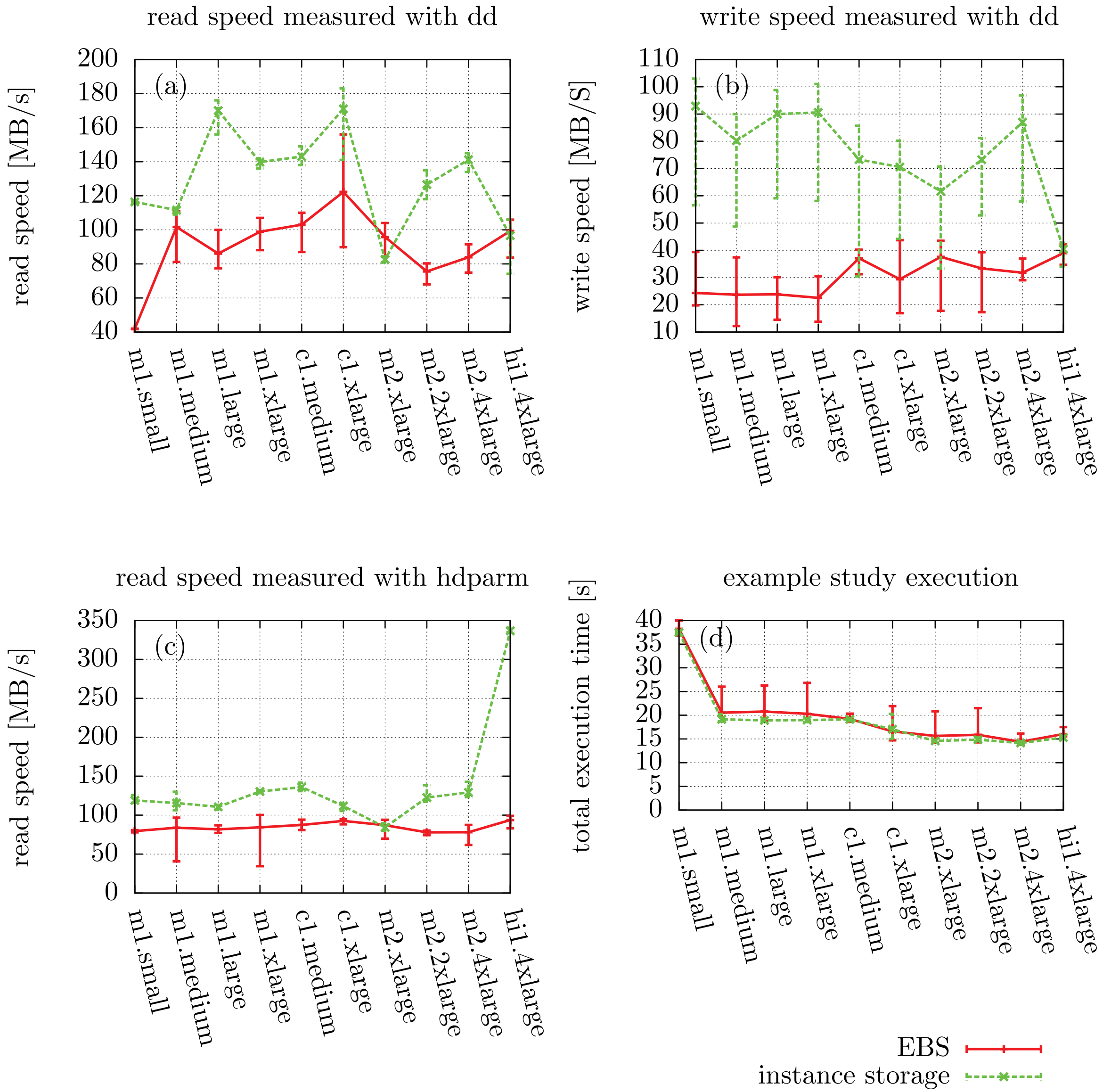

Figure 5.1.: Benchmarking results on different instance types in EC2.

Instance type:	m1.medium
CPU:	2 ECUs
Architecture:	64-Bit
Boot volume:	EBS
Memory:	3,75 GB RAM
Intance-storage:	410 GB
EBS-storage:	100 GB

This hardware configuration is used for all cluster nodes except for the monitoring master (c.f. Section 5.3 for more details). With increasing cluster size, the number of scheduled IOPS that are directed to the database of the monitoring server becomes too large to be handled by a standard EBS volume. Standard EBS volumes can handle around 100 IOPS. For larger cluster sizes EBS volumes with 1000 IOPS and 2000 IOPS were used respectively.

5.2.4. Software

Hadoop, HDFS, ROOT, SCALLA, PYTHIA and all their dependencies are installed on the customized AMI. The versions of the deployed software are listed below.

Ubuntu:	12.04 with Linux kernel 3.2.0-27-virtual
SCALLA:	3.2.0
Hadoop:	1.0.4
ROOT/PROOF:	5.34/05
PYTHIA:	8.1
Ganglia:	3.5.2

In our first trials, we used PROOF 5.34/01 but it turned out that it suffered from a bug that caused the worker processes to deadlock under certain conditions. The bug was reported to the PROOF developers and seems to be fixed with version 5.34/05 that was then used instead.

5.3. Cluster monitoring

To monitor the status of the deployed cluster and to be able to record metrics like network throughput and CPU utilization, the Ganglia monitoring system [52] is used. It provides a highly scalable solution to monitor distributed systems in the order of a few thousands nodes. Ganglia consists of three daemons: the monitoring daemon `gmond`, the meta daemon `gmetad` for collecting the data and `gweb`, a daemon for summarizing cluster information on a web page.
Each node in the cluster collects metrics e.g. CPU load or free memory of its own by

running the gmond daemon. The monitored nodes are organized into groups. In the normal Ganglia mode each of these groups share a common UDP multicast address. The metrics are stored locally and reported to the other nodes in the group using the multicast address every specified second. With this approach, each node in the group knows the status of all other nodes in the group at any time. This provides redundancy on the one hand and on the other hand, it allows Ganglia to be highly scalable, since the monitored hosts can be split up in multiple groups, which share one multicast address per group.
To collect and store the metrics at a single point, a node is running the gmetad. This daemon polls the metric data from an arbitrary node of each group and stores it in *round-robin*-databases. These databases store only a fixed number of values representing a certain time period. When the number of stored values exceeds the specified amount, the old database values are overwritten by new values following the round-robin scheduling principle. This design is well suited for monitoring since often only a given time period is of interest.
The gweb daemon reads the data from the round-robin databases and represents it on a PHP-based Web site.
In our cluster deployments Ganglia is configured in a slightly different manner. Since we want to minimize the use of resources on the worker nodes for other tasks except the analysis, storing the monitoring information on all nodes did not seem suitable. Except we use a Ganglia configuration, where each node is sending the metrics directly to the node running the gmetad using a Transmission Control Protocol (TCP) socket. This approach is less scalable since it puts a high I/O load on the gmetad node, but it reduces the load on the rest of the worker nodes.

5.4. Cluster layout

All nodes are running in Amazon's EC2. Since we are running evaluations with different cluster sizes and no fixed usage pattern, *on demand* instances seemed to be the most suitable solutions. To simplify the management of the configurations a VPC is created and Elastic IP addresses are assigned to master nodes additionally. Each master is launched within an associated security group (compare Section 2.6.3) to establish basic security mechanisms and firewall configurations. The next two subsections describe the framework specific configurations for the clusters.

5.4.1. Hadoop cluster layout

The Hadoop cluster layout is shown in Figure 5.2. The master daemons JobTracker and NameNode are running on separate instances to distribute the load and ease fault recovery. Since the namespace is small in our evaluations, the effort of restoring the file system state by hand does is comparably small to maintaining an instance running a SecondaryNameNode, so

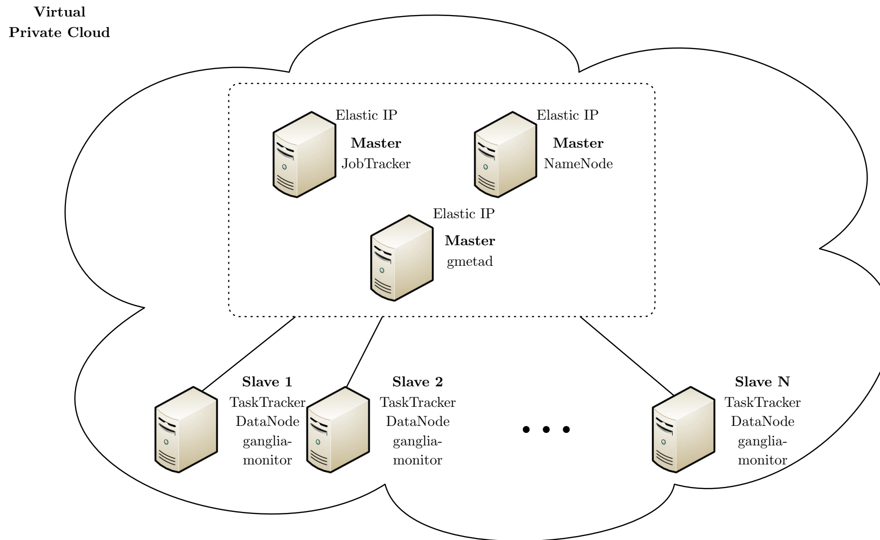

Figure 5.2.: Hadoop cluster layout.

no `SecondaryNameNode` is deployed. Elastic IP addresses are assigned to the instances running the `JobTracker` and the `NameNode` respectively. Both the `JobTracker` and `NameNode` daemons are hosting Web sites that represent their current status. The `JobTracker` website collects scheduling information about the submitted jobs, while the `NameNode` website represents the current status of the HDFS file system.
The slave instances are running both a `JobTracker` and a `DataNode`. The 100 GB of EBS-storage available per instance is used as a boot volume as well it is integrated in HDFS. So if the number of slave nodes is m the total capacity of HDFS sums up to 100 GB $\times m$. The actual capacity available for storing data of the deployed file system however is smaller, since each instance is using around 7 GB for the operating system and installed applications.
The replication factor in HDFS was set to 3 throughout all experiments with a block size of 64 MB.

5.4.2. PROOF cluster layout

Figure 5.3 shows the layout for the PROOF cluster. All nodes are running the `cmsd` and `xrootd` daemons. One node is taking the role of a PROOF master and is responsible for scheduling the workload on the worker nodes. It also acts as a redirector for the SCALLA distributed file system. As described in section 2.2.4 the cluster is organized in a 64-ary tree structure. When launching more than 64 worker nodes an additional supervisor is needed for each 64 extra nodes. In Figure 5.3 **Worker 2** is acting as a supervisor. Similar to the HDFS configuration described above the 100 GB of EBS-storage is integrated into the file system.

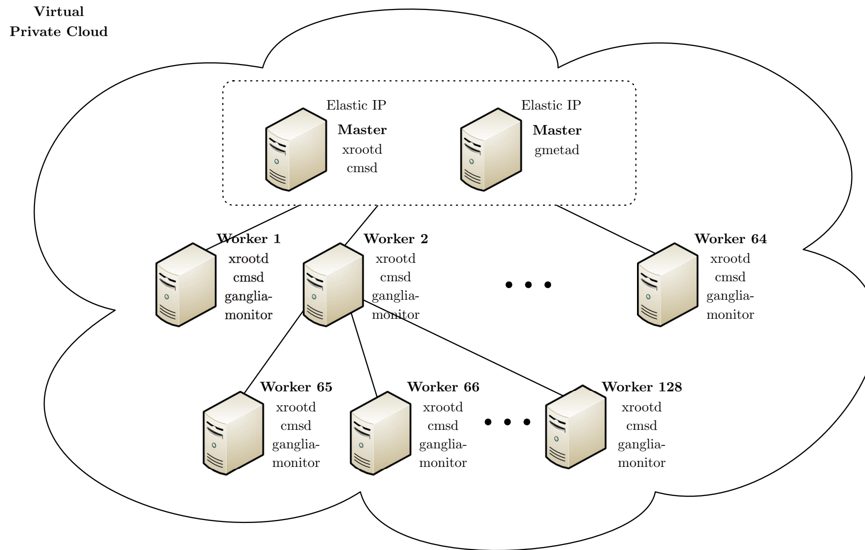

Figure 5.3.: PROOF cluster layout.

5.5. Cluster management

To ease the cluster management a suite of `bash` scripts was developed. They are based on scripts that come with the Hadoop framework that utilize the EC2 CLI to deploy Hadoop clusters in the EC2 cloud. Their functionality was extended to provide automatic configuration of the PROOF/SCALLA cluster deployment. More details on the steps taken to deploy the clusters are given in the Appendix C. Additionally the master daemons for the Hadoop clusters are deployed on different instances and the Ganglia monitoring system is configured automatically. The Hadoop daemons can be started and stopped with the help of wrapper scripts that come with the framework. For starting managing the `xrootd` and `cmsd` daemons of the SCALLA file system, System V scripts were developed and deployed on the cluster nodes.

5.5.1. Authentication

There are three different authentication mechanism used by the different AWS services: *access keys*, *X.509 certificates* and *Amazon EC2 key pairs*. For the services we utilize, no X.509 certificates are required, so we are only going to briefly mention the other two.
Access keys are based on symmetric key cryptography and are mainly used to make API calls to EC2. An access key consists of an *access key id*, that is publicly known and a *secret access key* that is only known by Amazon and the user. When making a service request the secret access key is used to create a *digital signature*. This signature and the access key ID

are send alongside the request to Amazon. Amazon now used the access key id to lookup the secret access key, which is then used to verify the signature.
Amazon EC2 key pairs are used to protect SSH access to EC2 instances. They are based on asymmetric key cryptography and consist of a *key pair name*, a *private key* and a *public key*. The key pair name is used to assign the key pair to instances on launch time. The public key part of the specified key pair is thereby automatically copied to the instance. The private key is then used to securely log into the instance.

5.6. Limitations and costs of utilizing the AWS services

Certain service limitations are exposed to AWS customers by default. While this thesis is written, customers are limited to run only 20 EC2 instances at the same time and use 20 TB of EBS storage. For the total cluster sizes we are aiming to deploy in EC2, we needed to apply for increased limits. Finally we got permissions to start 1005 EC2 instances using a total of 60 TB of EBS storage in the region EU West (Ireland) of EC2.
Sometimes we were not able to successfully request a larger number of on demand instances at the same time, because of capacity limitations in the EC2 cloud. In this case the API request was answered by the message given in Figure 5.4. Since we are deploying our clusters in a VPC, we are bound to the Availability Zone that was specified when the VPC was created. Hence choosing another Availability Zone as suggested in the message was not an option. By repeating the request with a smaller number of instances from time to time, we were able to increase our cluster size up to the desired number of nodes incrementally. The largest number of instances we use in parallel is 483. Note that even when we got permission to use 1005 instances, even requesting halve the number of instances on demand already led to problems.

```
Server.InsufficientInstanceCapacity: We currently do not have sufficient
m1.medium capacity in the Availability Zone you requested (eu-west-1a).
Our system will be working on provisioning additional capacity. You can
currently get m1.medium capacity by not specifying an Availability Zone
in your request or choosing eu-west-1b, eu-west-1c.
```

Figure 5.4.: Capacity limitations in the EC2 cloud.

For this thesis we are using a Amazon research grant to cover the EC2 fees. Since we are financially bound by this grant, also our compute time in the cloud is limited. To not exceed the research grant, rough costs estimates are conducted, before the clusters for the evaluations are deployed. Table 5.4 shows a cost estimation for conducting the scalability evaluations of the Hadoop-based solution described in Section 6.4 of the next chapter. The number of nodes (#Nodes) is the number of worker nodes plus the number of master nodes.

#Nodes	GB/Node	#Hours	Instance costs	Storage costs	Total costs
124	100	35	$564.20	$64.17	$628.37
244	100	30	$951.60	$108.23	$1059.83
484	100	20	$1258.40	$143.12	$1401.52
				Sum	$3089.72

Table 5.4.: Cost estimation for scalability evaluation of the Hadoop-based solution.

Each node corresponds to a m1.medium instance ($0.13/hour). Only the Ganglia master is deployed on an m1.large instance which is double the price ($0.26/hour) and hence counts as two nodes. The number of hours (#Hours) is a rough estimate of how long the given cluster size is needed. The main costs that occur are the costs of running the instances per hour (Instance costs) and the costs for the EBS storage (Storage costs). Amazon advertises the EBS storage price as $0.11 per GB and month for the EU Ireland region. Experiences show that the charges are calculated on an hourly usage base, such that we can estimate the storage price also per usage hour. Note that additional charges apply for using Elastic IPs, provisioned IOPS and EBS-optimized volumes in the Ganglia master and per million I/O requests. Nevertheless, the table provides a rough estimate of the costs that apply, when launching clusters of the given size for the given time period in the EC2 cloud. The estimated costs correspond to the costs for generating and analyzing 1 TB of event data with Hadoop on three different clusters with 120 worker nodes, 240 worker nodes and 480 worker nodes respectively. The table shows that running a 480 worker node cluster for 20 hours already costs around $1400. Since we are bound by the research grant, generating and analyzing more input data is not possible in the scope of this thesis. Doubling the input data effectively doubles the analysis time and therefore the costs for the experiments. Additionally the problems, when launching a large number of on-demand instances which are described above, keeps us from scaling the cluster sizes further.

6. Evaluation

To evaluate the concepts developed in this thesis, we deployed Hadoop and PROOF clusters in Amazon's EC2 cloud. In Section 6.1, we introduce the metrics that are used to measure the performance of the implemented systems. Section 6.2 compares the performance of the `RootFileInputFormat` and the `StreamingInputFormat`. The performance of the Hadoop-based solution is compared with the performance of the PROOF-based solution in Section 6.3. Subsequently, in Section 6.4, we investigate the scalability of the parallelized example HEP analysis on large Hadoop clusters.

6.1. Metrics

For analyzing the performance of the implementations in Hadoop and PROOF, we use a couple of metrics. These are described in the following.

6.1.1. Total execution time

The total execution time T_{tot} is the total time in seconds $[s]$ it takes to analyze the events given in the input data. In Hadoop, we measure T_{tot}^{Hadoop} that is the elapsed time from submitting a job until all output data is written. Generation time of the event map files is not included. The event map files are created only once after generating the input data and are kept in the same directories in HDFS as the corresponding data files. Therefore we do not add the time it takes to generate the event map files to T_{tot}^{Hadoop}. Similarly, we define T_{tot}^{PROOF} which is the time in seconds it takes from submitting a job to the PROOF cluster until all output data is collected and written.

6.1.2. Events per second

P is the average number of events that were analyzed per second, i.e.

$$P = \frac{N_{tot}}{T_{tot}}$$

, where N_{tot} is the total number of events in the input data. P is measured in $[1/s]$.

6.1.3. Average network throughput

We measure the average network throughput throughout the whole cluster during the analysis. This is measured in bytes per second $[Bytes/s]$ and includes all traffic that is transferred over the network. The average network throughput is a good indicator on how much data was read from remote locations.

6.1.4. Average CPU utilization

We use the CPU utilization of the cluster nodes as an indicator for the scheduling behavior of Hadoop and PROOF. If the subtasks are scheduled properly, the CPU utilization should be high on all worker nodes throughout the whole analysis.

6.1.5. Remarks on scalability

We are interested in the behavior of the distributed systems using Hadoop and PROOF when we *scale up* their computational resources by adding more worker nodes. *Scalability* is an often cited property of distributed systems, but it lacks of a formal definition [49], which is necessary to measure it quantitatively. Therefore we stick to an intuitive notion of scalability:
We say a solution is *scalable*, if the average number of analyzed events per second P increases linearly with the number of worker nodes.
However, this definition is a bit loose, because the speedup is clearly limited in general: In theory, when using Hadoop, the number of worker nodes working in parallel is limited by the number of `InputSplits` generated for the given input data. Increasing the number of worker nodes any further does not result in any speedup. When working with PROOF, the number of worker nodes processing data in parallel is limited by the number of `packets` generated by the `packetizer`. Increasing the number of worker nodes beyond the number of generated `packets` does not lead to any further speedup. Since an event is the smallest possible unit to be processed independently, the number of `InputSplits` and `packets` is limited by total number of events in the input data.
In addition to the constraints exposed by the splitting of the input data, other factors limit the speedup. Amdahl already stated in 1967 [10] that the speedup that can be expected by parallelism is limited by parts of a program that cannot be parallelized. Furthermore parallelism adds management overhead to the execution. For example, setting up the job in the cluster and merging results does not benefit from adding more worker nodes, since it requires additional set up of network connections between the nodes and data transfer over the network.

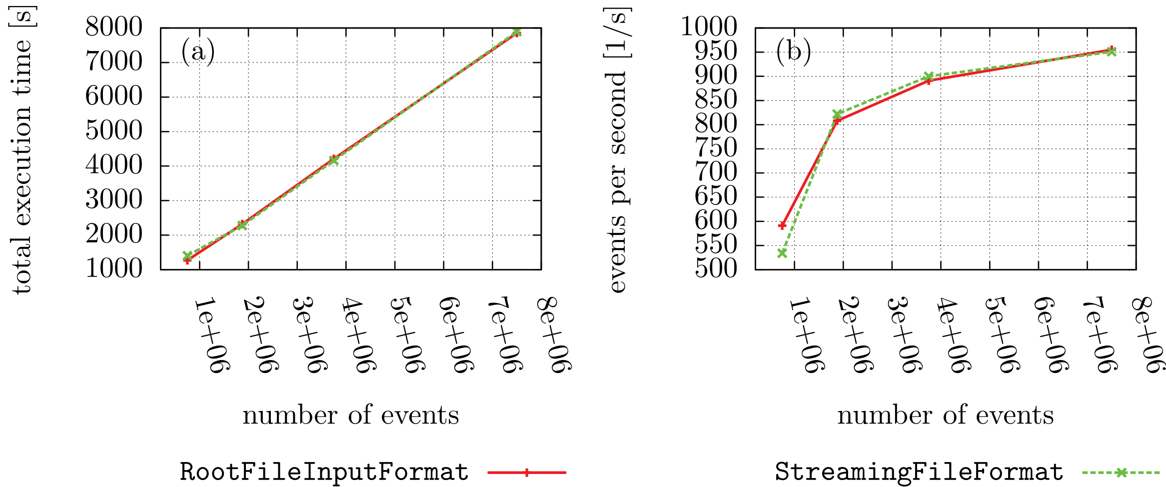

Figure 6.1.: Comparison of the total execution time and analyzed events per second of the `StreamingInputFormat` *and* `RootFileInputFormat` *on a 15 node cluster with varying input size.*

6.2. Evaluation of the RootFileInputFormat

We measured the performance of `RootFileInputFormat` by performing several test runs with varying input size and compare the results to evaluations done with the `StreamingInputFormat`. For each of the test runs, the input data was stored in a single file. This input file contained 7.5×10^5 events (1.8 GB), 1.875×10^6 events (4.4 GB), 3.75×10^6 events (8.8 GB) and 7.5×10^6 events (18 GB) respectively. For the evaluation, we fix the cluster size to 15 worker nodes. The data size was chosen, such that the `StreamingInputFormat` generates 30, 75, 150 and 300 `InputSplits` respectively leading to an average number of maps assigned to each worker node of 2, 5, 10 and 20. We repeat the evaluations two times for each input size to be able to detect negative impact of other processes running in the cloud. Note that, since we are financially bound by our research grant, we have to limit the number of iterations for our evaluations.

Figure 6.1 shows the total execution time (plot 6.1.a) and the analyzed events per second (plot 6.1.b) for both input formats. Both formats show a similar performance. Investigating the total execution time in plot 6.1.a, it appears that it scales linear with growing input size. Nevertheless, by investigating the number of analyzed events per second (plot 6.1.b), we deduce that the performance increases with growing input size, indicating that Hadoop needs a certain number of maps per worker node to schedule a job efficiently.

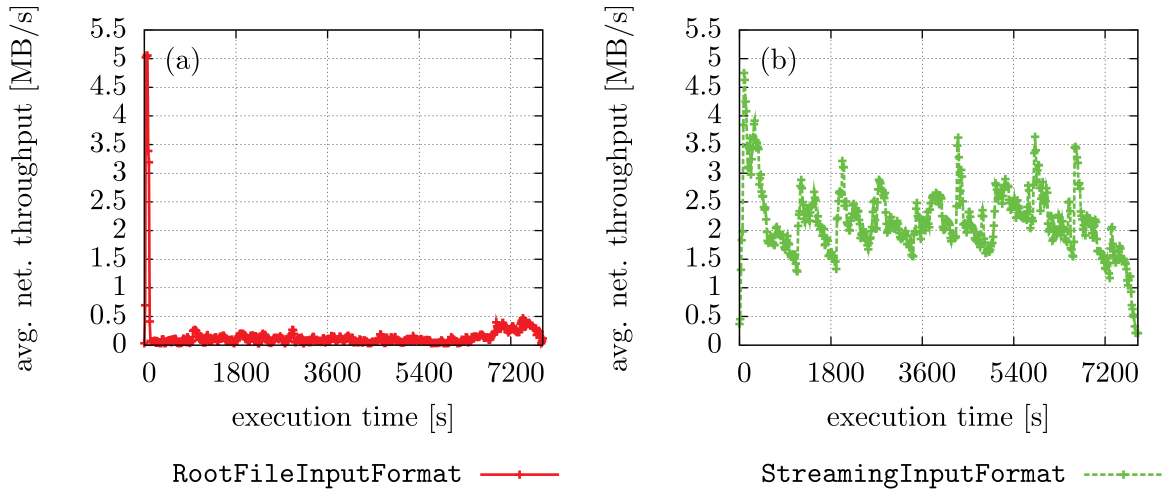

Figure 6.2.: Average network throughput while analyzing 7.5×10^6 *events with the* `StreamingInputFormat` *and* `RootFileInputFormat` *on a 15 node cluster.*

Figure 6.2 shows the network throughput while analyzing the input file with 7.5×10^6 events. Plot 6.2.a shows the throughput that is caused by Hadoop using the `RootFileInputFormat`, while plot 6.2.b depicts the throughput of Hadoop utilizing the `StreamingInputFormat` The results are satisfying for the `RootFileInputFormat`: Hadoop causes much less network traffic when using the `RootFileInputFormat` indicating that our approach successfully helps to schedule the map jobs efficiently. When comparing the network utilization of different Hadoop jobs using the `RootFileInputFormat`, we are able to identify a pattern: During the initial job setup, the evaluation of the event map files causes some network traffic. This is due to the fact that these files are stored in HDFS and the `JobTracker` is reading them remotely to calculate and assign the `InputSplits`. The network utilization could be reduced in this setup phase by storing the event map files locally on the `JobTracker`. However, for the general work-flow, it is more convenient to keep them in the same directory and file system as the files from which they are created.
We observe a second increase in the network utilization at the end of the analysis when the results are combined by the reducer (c.f. plot 6.2.a) after 6800 seconds), because the intermediate key/value pairs are read remotely. During the analysis most of the data is read locally when using the `RootFileInputFormat`. Nevertheless, the more efficient scheduling is not reflected in the total execution time of the evaluation. As describe in Section 5.2.2, the total execution time of the analysis seem to be relatively independent of the performance of the underlying storage system. The evaluations of this section reveal that it does not have a measurable performance difference when reading the data remotely. Regarding our example

HEP analysis, we conclude that its total performance is rather affected by the computing capabilities than by the I/O performance.
As a result, we determined that with the `RootFileInputFormat`, we can dramatically reduce the network utilization and it performs well in the given scenario.

6.3. Comparison of the Hadoop and PROOF cluster performance

In this section, we compare the performance of the Hadoop-based data analysis with the analysis in the PROOF framework. The PROOF analysis was conducted using two different storage solutions: the first time reading input data from SCALLA and the second time reading data from HDFS. We generated a total of 75×10^6 events on 15 nodes. The data generation took about 8 hours to complete. We created the event map files on a single instance which took about one hour to finish. The resulting data is stored in 250 ROOT files, each containing 3×10^5 events. Each file was around 750 MB in size, accumulating to a total input size of around 190 GB. Because of the block size of 64 MB, each file was split up into 12 blocks internally in HDFS. Since each block is treated as an `InputSplit`, the total number of `map` jobs summed up to 3000.
We started with a cluster size of 15 worker nodes and scaled up to 30, 60 and 120 nodes. The analysis of the generated events was done on both Hadoop and PROOF and repeated two times for each cluster size to gain trust in the solution and make sure that the effect of oscillations due to other processes running in the cloud is limited. After each scale up of the cluster size, the data stored in HDFS and SCALLA was rebalanced such that each data-node roughly stored the same amount of data. The maximum number of map and reduce operations running on a node in parallel was set to two and each node was running a maximum number of two PROOF workers in parallel.

Figure 6.3 shows the total execution time (plot 6.3.a) and the analyzed events per seconds (plot 6.3.b) of the three different cluster configurations. The total execution time and the corresponding number of analyzed events per second indicate that Hadoop introduces some overhead to the calculations. Comparing its performance to the PROOF-based solutions, we identify a deceleration by 14%-27%. When using PROOF, the performance differences between the two underlying file systems are negligible.
We observe that the number of analyzed events per second approximately doubles, when we double the cluster size. Therefore we argue that all three solutions *scale well* on the investigated cluster sizes.

Figure 6.4 shows the average network throughput (left plots) and the average CPU utilization (right plots) on the 60 worker node cluster in its different configurations during the analysis. Since the results are similar for all cluster sizes, we limit the presented results to

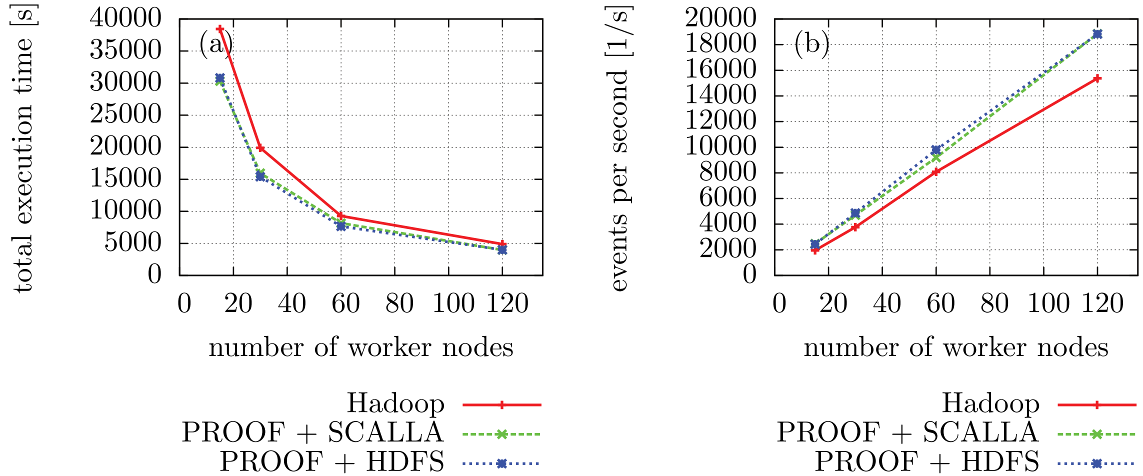

Figure 6.3.: Comparison of the total execution time and analyzed events per second in Hadoop and PROOF with varying cluster size.

the evaluations on the 60 worker node configurations. The plots 6.4.a and 6.4.b show the results for the Hadoop in combination with HDFS. The results for PROOF and SCALLA are depicted in the plots 6.4.c and 6.4.d, and the analysis outcome of PROOF and HDFS is shown in the plots 6.4.e and 6.4.f. The average CPU utilization in the plots 6.4.b, 6.4.d and 6.4.f indicate that all three solutions evenly distribute the workload in the cluster so that all worker nodes are busy most of the time. Note the average CPU utilization does not reach 100%, since also the master nodes that are less busy during the analysis are part of the calculations.
The Hadoop-based solution and PROOF reading data from SCALLA show comparably small network utilization (plots 6.4.a and 6.4.c), since the computation is preferably scheduled on nodes where the data is actually stored. When PROOF reads data from HDFS, it is not aware of the data locality and all data is treated as remote data. This results in scheduling the subtasks on arbitrary nodes in the cluster which is reflected by a higher network throughput (plot 6.4.e). Nevertheless, this is not reflected by the total execution time, since the full network capacity is not exploited and the delays caused by remote reads are negligible in comparison to computations time.
Hadoop and PROOF show different network utilization patterns. As stated in the basic input format evaluation in Section 6.2, the `RootFileInputFormat` causes comparably high network utilization in the beginning of the analysis when the event map files are read and in the end of the analysis when analysis results are transferred. PROOF, when reading data from SCALLA, is able to keep the network utilization low, but causes minor network

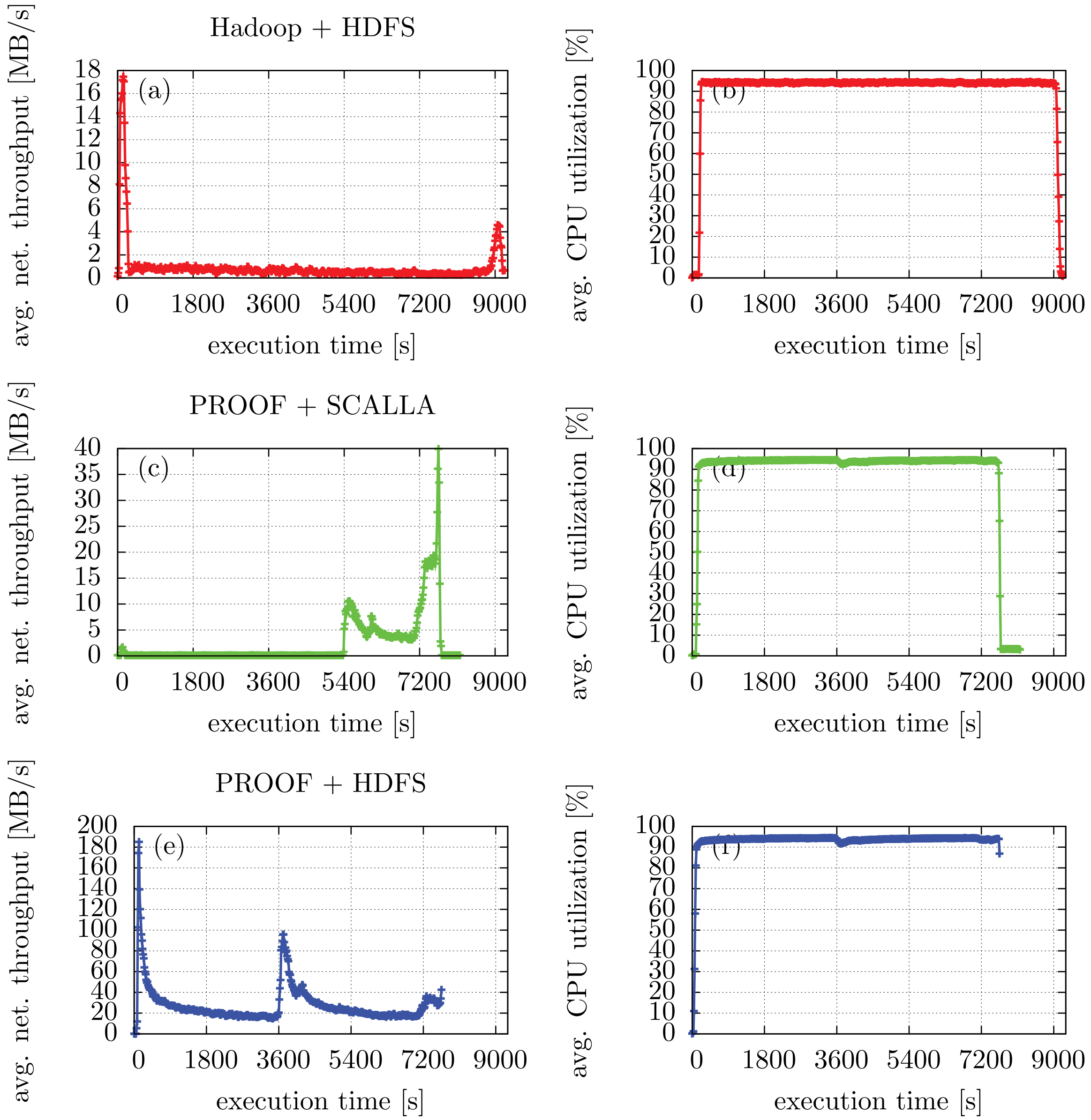

Figure 6.4.: Average network throughput and average CPU utilization on a 60 node cluster using Hadoop and HDFS, PROOF and SCALLA, and PROOF and HDFS respectively.

traffic in a later state of the analysis (c.f. plot 6.4.c after 5400 seconds) since the default PROOF packetizer schedules the subtasks not only based on locality information, but also on the performance of single worker nodes. That could result in scheduling a subtask to a node that needs to read the data remotely. A similar explanation holds for the peaks in the network utilization of PROOF reading data from HDFS (c.f. plot 6.4.e at 0 seconds and 3600 seconds). The packetizer schedules subtasks to nodes where the data is not stored. However, due to the performance feedback of the nodes themselves, this effect decreases over time. Since subtasks are performing better on nodes, where the data resides, these nodes are preferred, for scheduling new subtasks that read data from the same file.
The evaluations in this section revealed that Hadoop adds some overhead to the computation. When using Streaming, it spawns a JVM for each mapper and reducer, which then executes the binary that was specified as a map or reduce function. This approach increases robustness, since a failing subtasks can not affect the `TaskTracker` daemons, which is running in another JVM, but it also introduces some overhead. In addition to the analysis code implemented in C++, also the Java code implementing the `RecordReader` and the corresponding `InputSplit` is executed, which causes additional computational load.
Surprisingly, PROOF performed well when reading data from HDFS. This indicates that this combination is a valuable subject for further development.

6.4. Scalability of Hadoop-based solution

We showed that the Hadoop-based solution behaves well on clusters with up to 120 worker nodes. Subsequently, we evaluate its behavior for bigger input data on larger clusters. In HEP analysis, the Hadoop solution should scale well for input data in the order of a few TB that is analyzed on a cluster of a few hundred nodes. In the next experiment we generated 1500 files with 3×10^5 events per file. The generation was conducted on a 120 worker node cluster in parallel and took about 7 hours to complete. Since we learned that the event map file generation takes a considerable amount of time for larger input data (c.f. Section 6.3), we parallelized the generation of the event map files with Hadoop Streaming, whereby the input for the map jobs are the files for which the event maps are created. With this approach, creating the event map files for all 1500 input files took 750 seconds to complete on a cluster with 120 worker nodes. The total event number accumulates to 450×10^6 events with a total size of around 1 TB of input data. Due to a replication factor of 3, the total HDFS storage needed to store the input data was around 3 TB. Since each input file is around 750 MB in size, it is divided into 12 blocks in HDFS internally and the total number of `InputSplit`s accumulates to 18000. The cluster size was doubled twice starting with 120 worker nodes to 240 and 480 worker nodes. After each scale up, the data was rebalanced in the cluster so that each `DataNode` roughly contained the same amount of data.

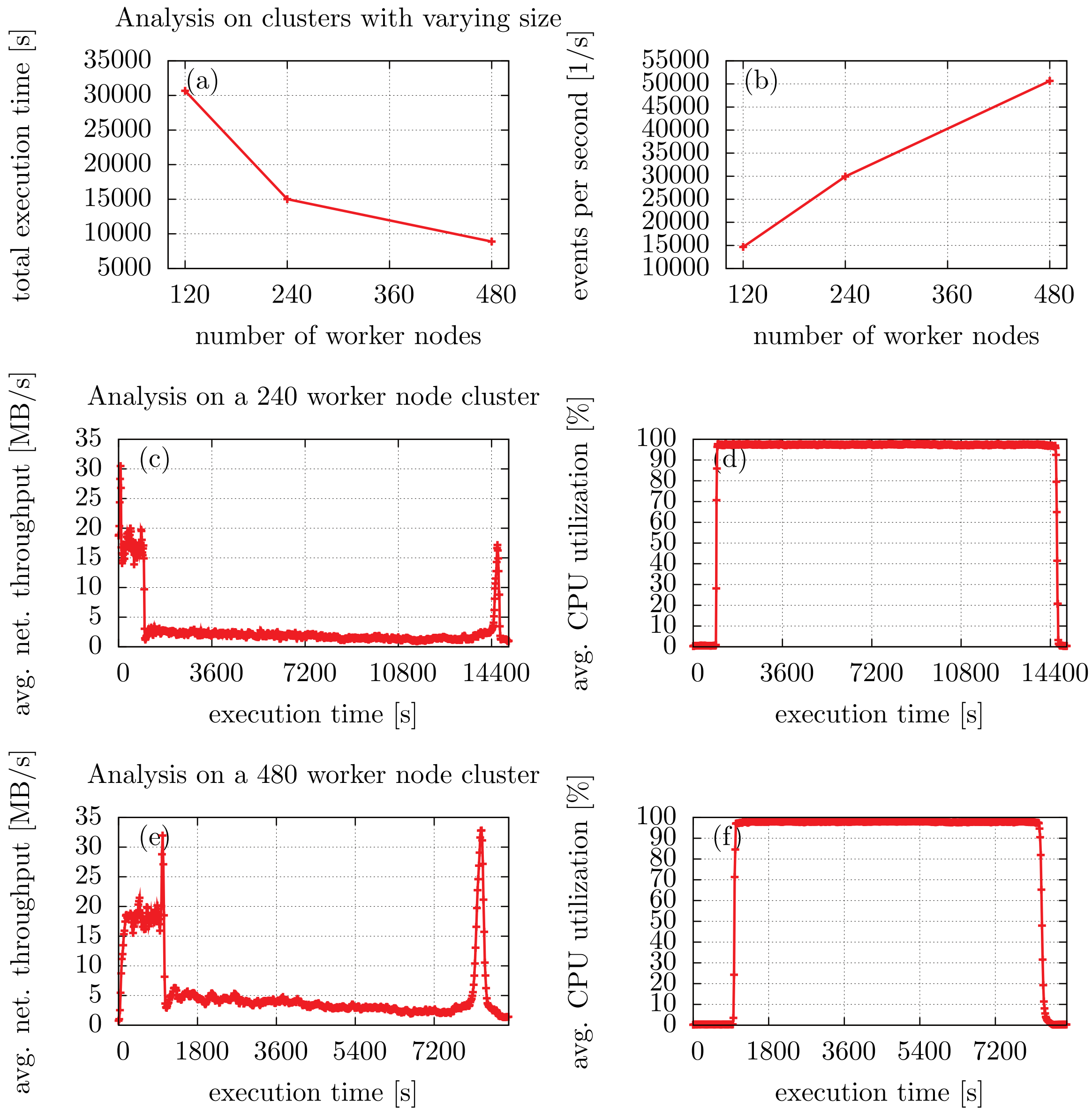

Figure 6.5.: Analysis of 1 TB of input data with Hadoop.

The results are shown in Figure 6.5. We observe that the total execution time (plot 6.5.a) is halved when scaling the cluster size from 120 to 240 nodes. This implies that the number of analyzed events per second (plot 6.5.b) effectively doubles when having a fixed size of input data.
When scaling the cluster size further from 240 nodes to 480 nodes we are only able to achieve a performance gain of 70%. The plots 6.5.c and 6.5.d show the average cluster throughput and the CPU utilization during the analysis in the 240 node cluster. The corresponding metrics for the evaluation on the 480 node cluster are depicted in the plots 6.5.e and 6.5.f respectively. Since the average network throughput is low in both cluster configurations, the limited performance gain cannot be explained by network latency or saturation. The average CPU utilization (plots 6.5.d and 6.5.f) indicate that the workload was scheduled properly.
The low CPU utilization (plots 6.5.d and 6.5.f) and the moderate network throughput (plots 6.5.c and 6.5.d) in the first 900 seconds are caused by the `JobTracker`, which is using the `RootFileInputFormat` to analyze the input data and calculate the `InputSplits`. The last 300 seconds in the 240 node configuration and 600 seconds in the 480 node configuration are used to collect the data and produce the final output histogram in the reduce phase.
These observations confirm the assumptions from Section 6.1.5: First the setup phase does not benefit from parallelization. Secondly, some parts of the job, e.g. the data collection phase in the end might even suffer from increasing the number of nodes due to the communication overhead that is introduced. The generation of the `InputSplits` in Hadoop is done on the `JobTracker` and cannot be parallelized within the framework. Because sequential parts of the job, such as the `InputSplit` generation, do not experience any speedup, they relatively gain weight in the total execution time when scaling the cluster size. This could be an explanation of the reduced performance gain, when we scale the cluster size from 240 to 480 nodes. Nevertheless, the performance gain of 70% is still satisfying. Due to several constraints, we were not able to scale the cluster size any further. One reason was the service constraints of the Amazon EC2 cloud described in Section 5.6.

7. Related Work

The approach to use MapReduce for scientific computations is not new and several studies exist on its applicability to different problems. Apart from Hadoop several other public-available implementations of the MapReduce paradigm exist, which often aim to optimize or extend it for certain application areas.
Additionally Hadoop and also HDFS, as a stable distributed file system, have attracted some attention of the HEP community. Section 7.1 gives an overview of applying MapReduce to scientific computing. In Section 7.2, different implementations of the MapReduce paradigm are described and Section 7.3 discusses Hadoop and HDFS related studies in the HEP community.

7.1. Application of MapReduce to scientific computing

Several studies explore how MapReduce can be used to solve problems in scientific computing. We just name a few. Chu et al. apply the paradigm to machine learning algorithms on multicore systems [31]. Among others, the algorithms they investigate include *Locally Weighted Linear Regression*, *k-means clustering* and a small implementation of a *neural network*. In their evaluations they almost achieve linear speedups on multicore systems in comparison to sequential implementations and conclude that MapReduce is a promising approach for exploiting multicore systems in machine learning.
Another more detailed investigation of applying MapReduce to k-means clustering is given by Zhao et al. [67]. They adapt the clustering algorithm to MapReduce with the help of the Hadoop framework and show that it scales well on large data sets.
McKenna et al. introduce a MapReduce based framework for genome analysis in bioinformatics [53] and claim that its adaption eases the development of robust and scalable DNA sequencing tools.
Hama [12] is another Apache project that is build on top of Hadoop to provide efficient calculations of matrix, graph and network algorithms. While it uses other paradigms for graph and network algorithms, MapReduce is utilized for matrix calculations [59].

7.2. Different MapReduce implementations

In this thesis we focused on Hadoop, one particular public-available MapReduce implementation. Several other implementations exist that often aim to optimize or extend the MapReduce paradigm for certain application areas. As the MapReduce implementation by Google and Hadoop they are often coupled to special distributed file systems.

Ekanayake et al. identify certain extensions for the classic MapReduce paradigm to adapt it to more classes of applications [35]. Their main critique is that MapReduce does not offer efficient support for iterative algorithms. Therefore they introduce *Twister*, a Java framework, optimized to run MapReduce operations iteratively.
Microsoft also follows the idea of iterative MapReduce: *Project Daytona* [56] is a runtime environment for iterative MapReduce jobs in the Microsoft Azure Cloud [55]. Programs for Daytona are written in C#.

Disco [57] is a MapReduce implementation developed by the Nokia Research Center. In contrast to Hadoop it is not implemented in Java, but uses a combination of Python and Erlang. Because of this language combination the code is relatively small and fairly easy to understand. In the Disco framework Map and Reduce functions are normally implemented in Python, but can also be written in arbitrary languages using the *Disco Worker Protocol*. Disco comes with its own distributed file-system, called *Disco Distributed Filesystem* (DDFS).

Gu and Grossman present the design and implementation of *Sector* and *Sphere* [47]: Sector is a distributed file system while Sphere implements a framework for executing User Defined Functions (UDFs) within one data center or across distributed data centers. While Sphere is not exactly a MapReduce implementation it can mimic its programming style. The user can simply define two UDFs: one map and one reduce function. These UDFs can be used on top of Sector to be executed in a MapReduce manner. While Sphere and Sector supports the MapReduce programming paradigm it is more general approach. Additionally it does not strictly assume that bandwidth is a scarce resource, but that different computing sides are connected by high speed networks (10 Gbps or higher). Sector and Sphere can be utilized by using a C++ API.

The MapReduce implementations described above focus on applying the paradigm to single clusters. In contrast Wang et al. introduce *G-Hadoop* [66], an implementation that provides executing MapReduce tasks across distributed clusters. G-Hadoop operates on *G-Farm* [63], a file system that interconnects distributed data centers. While Hadoop is designed to schedule map tasks to nodes where the corresponding `InputSplit` is stored, G-Hadoop schedules tasks across data-centers. The G-Hadoop API thereby remains com-

patible with the Hadoop API such that existing Hadoop applications do not need to be modified. A framework of this kind could be used to scale the implementation done in this work across multiple clusters.

7.3. Hadoop/HDFS in the HEP community

In their evaluation of the applicability of MapReduce to scientific computing [36], Ekanayake et al. also describe an adoption of Hadoop for HEP data analysis. However in their approach they do not address the problem of splitting the data residing in HDFS efficiently. Instead data files are processed as a whole. This leads to the problems described in Section 3.3 when the individual file size becomes larger than the block size in HDFS. They evaluate their solutions on small clusters (up to 12 compute nodes) and claim that for larger cluster sizes the speedup diminishes. This contradicts our results, since we are able to achieve satisfying speedups even when we scale the cluster up to 480 compute nodes (c.f. Section 6.1.5).

Similar to the approach described above, Riahi et al. describe how Hadoop and HDFS can be utilized for computing within the WLCG in a small/medium Grid site (Tier-2/Tier-3) [58]. Again the problem of splitting ROOT files into several `InputSplits` is not solved, but whole files are assigned to single mappers. Similar to Ekanayake at al. they limit their evaluation to very small cluster sizes (three compute nodes with a total of 72 cores).

HDFS is not bounded to Hadoop and can be used as a reliable distributed file system without exploiting the MapReduce paradigm. Bockelman identifies HDFS as a possible storage solution in the context of the CMS experiment [20]. He concludes that HDFS is a viable solution for grid storage elements inside the WLCG, especially with regards to scalability, reliability and manageability. He is also the author of the HDFS plugin for ROOT.

8. Conclusions and Outlook

In this thesis we investigated the applicability of MapReduce and the cloud computing paradigm to LHC data analysis. We have developed an input format that splits ROOT based input data in a way that it can be efficiently processed within the Hadoop framework. By successfully deploying large compute clusters in the EC2 cloud, we demonstrated how cloud computing can be utilized for LHC data analysis. We scaled the utilized clusters up to sizes that are realistic in high performance computing centers. Thereby, we achieved a good performance gain for the analysis. In general, our evaluations show that Hadoop and HDFS provide a viable solution for parallelizing ROOT based data analysis.

Nevertheless, there are some downsides of utilizing Hadoop for LHC data analysis. The comparisons with PROOF show that Hadoop introduces a computational overhead of 14% to 27% in our evaluations. This percentage does not include the time it takes to calculate the mapping information for the ROOT files. Hadoop is especially designed for data analysis that is Input/Output intensive and where the network might become the limiting bottleneck. During the evaluation it became clear that the example study from LHC is CPU intensive. Therefore, it had only minor benefits from the data locality information given by HDFS. The mapping of the events to HDFS blocks worked well in our example. In the example HEP analysis, the underlying data classes have a simple structure. With increased complexity, e.g. when other complex data classes are referenced by member variables, assigning events to HDFS blocks becomes more complex.

Utilizing IaaS as with EC2 comes with the flexibility of renting resources on demand. It eases cluster management, since tasks like address and firewall configuration can be done centrally. The concept of a single machine image that is deployed several times simplifies software installation and maintenance. Backups can be achieved easily by using snapshots. Nevertheless, deploying a cluster requires proper knowledge of the underlying technical principles. Even when the physical infrastructure is hidden to the customer, the configuration of the IaaS services is similar to setting up real physical networks or clusters.

We showed how ROOT-based data analysis can be mapped to MapReduce. The approach to map data records to byte positions can also be a suitable solution for all kinds of binary input data that should be analyzed within MapReduce frameworks. Additionally we gave insights on how the EC2 cloud can be utilized to deploy clusters for computations.

8.1. Future research directions

While we have investigated how LHC associated data analysis can be executed in the cloud on a large scale, it is not clear how these EC2- based solutions perform in comparison to local cluster deployments. Several studies exist that address the general performance of the EC2 cloud for high performance computing [37, 50, 51]. All of them indicate that we must expect a performance drop when utilizing the cloud. The main drawback in the cloud computing environments seem to be an increased communication latency between the nodes [51], which renders it unsuitable for highly coupled scientific computing applications. Since the LHC event analysis is highly decoupled, it is less affected by increased communication latency. To understand the cloud-related performance issues better, benchmarks should be conducted that offer comparability with local cluster deployments.
In general performance comparisons are not trivial, since they need a generalized framework that defines the comparable objects clearly. For benchmarking the EC2-based solutions developed in this thesis, the existing benchmarking frameworks for Hadoop and PROOF can be utilized. Using these frameworks has the advantage of achieving comparability with already existing benchmarking results. Hadoop offers the `TeraSort` package [13] that is commonly used to benchmark storage and MapReduce performance. It measures the time it takes to sort 1 TB of generated data. For PROOF cluster benchmarking the `TProofBench` framework [27] can be utilized. It offers both CPU and IO-intensive benchmarks. The CPU intensive benchmark generates random numbers that are filled into several histograms, while the IO-intensive benchmark is based on reading data from `TTree`s. Utilizing these frameworks gives more insights into the general performance that could be expected when utilizing the EC2-cloud for LHC data analysis.

As we have seen in Section 6.3, PROOF in conjunction with HDFS delivers a good general performance. Nevertheless, since the locality information in HDFS is not utilized in the existing PROOF packetizers all data files are treated as remote data and the subtasks are not assigned to the nodes where the data resides. This violates the "Moving computation to data" paradigm described earlier in this thesis. Therefore, developing a packetizer for PROOF that fully utilizes the HDFS data locality information could be an interesting and promising approach to increase the performance further.

With the growing size of distributed systems, it becomes more likely that one or more of its components fail, e.g. because of hardware failure. Thus *fault-tolerance* becomes an important property. Hadoop is designed to provide fault-tolerance out of the box. If a worker node is failing, the assigned task is automatically rescheduled on another worker node. HDFS replicates the stored data automatically to other `DataNode`s in the cluster. If a `DataNode` becomes unavailable, e.g. because of hardware failure, the data that was stored on this particular `DataNode` is automatically replicated from the remaining copies in the

cluster to other `DataNodes`.
While PROOF can reschedule failed subtasks to other worker nodes, SCALLA does not provide automatic replication of stored data. This results in data loss if one of the worker nodes that are storing data becomes unavailable, which can cause the submitted job to fail. During our evaluations we were not confronted with failing worker nodes. Some initial tests were performed with Hadoop by disconnecting one of the worker nodes by hand. The tasks that were running on this particular worker node were rescheduled automatically as expected and the data was replicated. It would be interesting to evaluate the fault-tolerance further, especially with the combination of PROOF and HDFS.

During the evaluation in this thesis, it became clear that the performance of the provided example HEP analysis is rather dependent on CPU power than on I/O performance. Hadoop was able to perform well, but the PROOF-based solution achieved better performance. Hadoop is designed to reduce the network load and is optimized for data-intensive applications. Therefore it would be interesting to see if Hadoop shows comparably better performance if the considered application relies on high data throughput. Reevaluating the `ProofFileInputFormat` with such an application would give more insights on the possible advantages that come by utilizing Hadoop for HEP data analysis.
As seen in Section 6.3, Hadoop is 14% to 27% slower than PROOF in our evaluations, which is caused by the overhead introduced by spawning JVMs and the execution of additional Java code for reading the input data. Additionally, the fact that Hadoop only offers a Java API for defining input formats, complicates the processing of binary input data that can only be read by libraries written in other programming languages. For our example HEP data analysis, being able to link to the ROOT framework, is necessary. As future work it should be implemented in a MapReduce framework that fully provides C++ support, e.g. in Sphere (see Section 7.2), to evaluate the possible performance drawbacks of the Java-based Hadoop solution further.

PROOF is designed for compute clusters consisting of a few hundred nodes and might lose its advantages over Hadoop, when scaling further. In the proceeding of this thesis, we investigated PROOF clusters up to a size of 120 worker nodes. The scaling evaluations that are provided in Section 6.4 for the Hadoop-based solution are omitted for PROOF in this thesis due to initial software problems and time constraints and can be conducted as part of future research.

Even with the abstraction offered by cloud services, deploying compute clusters in the cloud is still a challenging task. Utilizing IaaS for deploying and administrating a cluster is quite like deploying and administrating a physical cluster. The complexity is reduced in some kind but new challenges appear which are connected to mastering the specific tools of the chosen IaaS provider. Non-computer-scientists, which want to deploy infrastructures

to run simulations in the cloud, are confronted with overcoming a high technical barrier, before IaaS might provide a helpful tool for their research. Higher levels of abstraction are needed to reduce technical difficulties and address especially the needs of scientists that want to deploy the infrastructure for their research. Services like Amazons EMR might be too inflexible to satisfy the needs of scientists. Therefore the requirements of simulation applications need to be further investigated. Developing concepts for categorizing these applications and automatically mapping their requirements to cloud infrastructures might open a whole new field of research.

A. Abbreviations and Acronyms

ACL Access Control List

AMI Amazon Machine Image

AOD Analysis Object Data

API Application Programming Interface

AWS Amazon Web Services

CAF CERN Analysis Facility

CERN Conseil Européen pour la Recherche Nucléaire

CLI Command Line Interface

DDFS Disco Distributed Filesystem

DPD Derived Physics Data

EBS Elastic Block Store

EC2 Elastic Compute Cloud

ECU EC2 Compute Unit

EMR Elastic MapReduce

ESD Event Summary Data

GFS Google File System

HDFS Hadoop File System

HEP High Energy Physics

IaaS Infrastructure as a Service

IOPS Input/Output operations per second

JVM Java Virtual Machine

LHC Large Hadron Collider

MPIK Max Planck Institut für Kernphysik

MSS Mass Storage System

PaaS Platform as a Service

POSIX Portable Operating System Interface

PROOF Parallel ROOT Facility

S3N S3 Native File System

S3 Simple Storage Service

SaaS Software as a Service

SCALLA Structured Cluster Architecture for Low Latency Access

SCaVis Scientific Computation and Visualization Environment

SDF Simple Data Format

SSD Solid State Drive

SSH Secure Shell

STL Standard Template Library

TCP Transmission Control Protocol

UDF User Defined Function

UDP User Datagram Protocol

URL Uniform Resource Locator

VPC Virtual Private Cloud

WLCG Worldwide LHC Computing Grid

B. Changes to the ROOT HDFS plugin

This appendix lists the changes that were made to the ROOT HDFS-plugin (5.34/05) to work with Hadoop 1.0.4. `ROOTSYS` is the root directory of the ROOT installation. The changes made to `$ROOTSYS/io/hdfs/inc/THDFSFile.h` are documented in Listing B.1.

```
1  $ diff THDFSFile.h-original THDFSFile.h
2  81a82
3  >     void *  GetDirPtr() const{return fDirp;};
```

Listing B.1: Changes in THDFSFile.h

Only the `GetDirPtr()`, which is inherited from the `TSystem` class, needs to be implemented such that calls directed to the HDFS file system are handled properly by an instance of `THDFSystem`.

The changes that were made to `$ROOTSYS/io/hdfs/src/THDFSFile.cxx` are described in listing B.2. Since the method call, listed in line 5 and 10 is deprecated, it is replaced by the lines 7 to 12 and 16 to 23 respectively. Instead of the HDFS file system, set in the configuration files, the local file system is used, when the function `hdfsConnectAsUser(...)` is called with "default". To provide the correct information for the plugin, the two environmental variables `HDFS_SERVER` and `HDFS_PORT` are used.

```
1  $ diff THDFSFile.cxx-original THDFSFile.cxx
2  73d72
3  <
4  97c96,101
5  <       fFS = hdfsConnectAsUser("default", 0, user, groups, 1);
6  ---
7  >       const char * server = getenv("HDFS_SERVER");
8  >       int port = atoi(getenv("HDFS_PORT"));
9  >       if (server && port)
10 >     fFS = hdfsConnectAsUser(server, port, user);
11 >       else
12 >     fFS = hdfsConnectAsUser("default", 0, user);
13 301d304
14 <
15 310c313,318
16 <       fFH = hdfsConnectAsUser("default", 0, user, groups, 1);
17 ---
18 >       const char * server = getenv("HDFS_SERVER");
19 >       int port = atoi(getenv("HDFS_PORT"));
20 >       if (server && port)
21 >     fFH = hdfsConnectAsUser(server, port, user);
22 >       else
23 >     fFH = hdfsConnectAsUser("default", 0, user);
```

```
24 376a385
25 >
26 398c407
27 <         delete fUrlp;
28 ---
29 >         // delete fUrlp;
30 423c432
31 <      if (fDirCtr == fDirEntries-1) {
32 ---
33 >      if (fDirCtr == fDirEntries) {
34 430c439,440
35 <      TUrl tempUrl;
36 ---
37 >
38 >      /*TUrl tempUrl;
39 434c444
40 <      result = fUrlp[fDirCtr].GetUrl();
41 ---
42 >      result = fUrlp[fDirCtr].GetUrl();*/
43 437c447
44 <      return result;
45 ---
46 >      return (strrchr(result, '/') + 1);
47 459a470,474
48 >      if (fileInfo->mKind == kObjectKindFile)
49 >    buf.fMode |= kS_IFREG;
50 >      else if (fileInfo->mKind == kObjectKindDirectory)
51 >    buf.fMode |= kS_IFDIR;
52 >
```

Listing B.2: Changes in THDFSFile.cxx

The `delete fUrlp` in line 27 causes a segmentation fault and was hence commented out. The subtraction of 1 in line 31 causes the plugin to not list all entries in a HDFS directory. The changes described in lines 38 to 46 are made because the wrong path is returned by the `GetDirEntry(...)` method implementation of the `THDFSSystem` class. The changes described in line 48 to 51 are made to set the meta information for files in HDFS correctly.

C. Steps to deploy clusters in EC2

Several bash scripts that are build around the commands above have been implemented to ease the cluster deployment. They are used to start the required instances, deploy the configuration files and start the daemon processes on the corresponding instances. Figure C.1 shows the steps taken to deploy a Hadoop cluster in EC2. The Steps 1. to 4. correspond

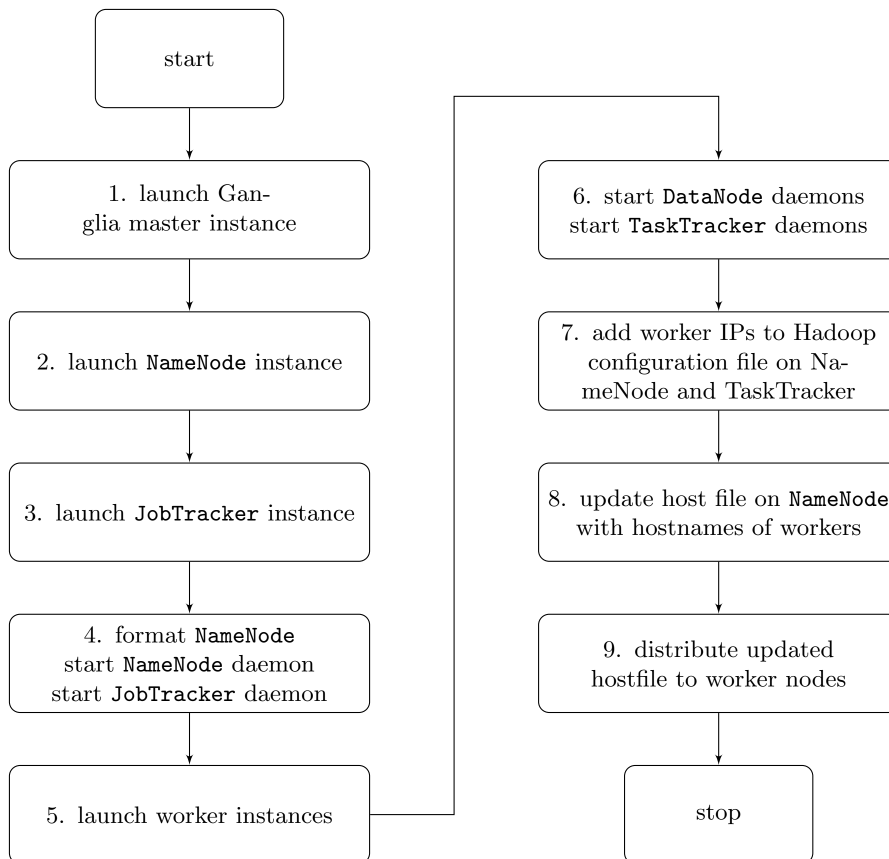

Figure C.1.: Steps to deploy a Hadoop cluster in EC2.

to deploying the master nodes. The nodes running the Ganglia `gmetad`, Hadoop `NameNode`

and Hadoop `JobTracker` are started and configured. The Steps 5. to 9. are used to add worker nodes to the cluster. Their daemons are started and their host names and IP addresses are propagated to the other nodes in the cluster. When increasing the cluster size, these steps can be repeated iteratively until the desired cluster size is reached.

List of Figures

2.1. PROOF's multi-tier architecture [40]. 8
2.2. The pull architecture of the PROOF packetizer [41]. 9
2.3. Physical layout of the ROOT file format [25]. 11
2.4. MapReduce job execution overview [33]. 14
2.5. Cloud computing deployment and service models [61]. 19

3.1. Output of the example HEP analysis running on 20000 events. 27
3.2. Data class hierarchy of example study. 28
3.3. Data distribution in an example SDF file storing 20000 events. 32
3.4. Workflow of the HDFS block selection. 34

4.1. Class diagram for `RootFileInputFormat`. 43
4.2. Class diagram for `StreamingInputFormat`. 45

5.1. Benchmarking results on different instance types in EC2. 53
5.2. Hadoop cluster layout. 56
5.3. PROOF cluster layout. 57
5.4. Capacity limitations in the EC2 cloud. 58

6.1. Comparison of the total execution time and analyzed events per second of the `StreamingInputFormat` and `RootFileInputFormat` on a 15 node cluster with varying input size. 63
6.2. Average network throughput while analyzing 7.5×10^6 events with the `StreamingInputFormat` and `RootFileInputFormat` on a 15 node cluster. . . 64
6.3. Comparison of the total execution time and analyzed events per second in Hadoop and PROOF with varying cluster size. 66
6.4. Average network throughput and average CPU utilization on a 60 node cluster using Hadoop and HDFS, PROOF and SCALLA, and PROOF and HDFS respectively. 67
6.5. Analysis of 1 TB of input data with Hadoop. 69

Listings

2.1. Word Count example [33]. 12

3.1. Event loop in the original example HEP analysis code. 29
3.2. Per-track pair analysis in the original example HEP analysis code. 30

4.1. Implementation of the `StreamingMapper`. 39
4.2. Implementation of the `StreamingReducer`. 39
4.3. Utilizing ROOT for the event mapping. 41
4.4. Basic heuristic for mapping events to blocks in HDFS. 44
4.5. Generating a `TSelector` [25]. 45
4.6. Implementation of `MySelector::SlaveBegin()`. 46
4.7. Implementation of `MySelector::Process()`. 46
4.8. Implementation of `MySelector::Terminate()` 47

List of Tables

5.1. Configuration of the benchmarked instance types. 51
5.4. Cost estimation for scalability evaluation of the Hadoop-based solution. . . . 59

Bibliography

[1] G. Aad, E. Abat, J. Abdallah, et al. The ATLAS Experiment at the CERN Large Hadron Collider. *Journal of Instrumentation*, 3:S08003, 2008.

[2] K. Aamodt, A. A. Quintana, R. Achenbach, et al. The ALICE experiment at the CERN LHC. *Journal of Instrumentation*, 0803:S08002, 2008.

[3] A. A. Alves Jr, L. Andrade Filho, A. Barbosa, et al. The LHCb Detector at the LHC. *Journal of Instrumentation*, 3:S08005, 2008.

[4] Amazon. Amazon Web Services. [Online; `http://aws.amazon.com/` fetched on 05/21/2013].

[5] Amazon. Elastic Block Store. [Online; `http://aws.amazon.com/ebs/` fetched on 05/21/2013].

[6] Amazon. Elastic Compute Cloud. [Online; `http://aws.amazon.com/ec2/` fetched on 05/21/2013].

[7] Amazon. Elastic MapReduce. [Online; `http://aws.amazon.com/elasticmapreduce/` fetched on 05/21/2013].

[8] Amazon. Simple Storage Service. [Online; `http://aws.amazon.com/s3/` fetched on 05/21/2013].

[9] Amazon. Virtual Private Cloud. [Online; `http://aws.amazon.com/vpc/` fetched on 05/21/2013].

[10] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[11] Apache Software Foundation. Apache Hadoop. [Online; `http://hadoop.apache.org/` fetched on 05/21/2013].

[12] Apache Software Foundation. Apache Hama Project. [Online; `http://hama.apache.org/` fetched on 05/21/2013].

[13] Apache Software Foundation. Hadoop TeraSort. [Online; https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/examples/terasort/package-summary.html fetched on 05/21/2013].

[14] Apache Software Foundation. Hadoop Wiki: MountableHDFS. [Online; http://wiki.apache.org/hadoop/MountableHDFS fetched on 05/21/2013].

[15] Apache Software Foundation. Hadoop Wiki: PoweredBy. [Online; http://wiki.apache.org/hadoop/PoweredBy fetched on 05/21/2013].

[16] M. Armbrust, A. Fox, R. Griffith, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[17] MapR Technologies. [Online; http://www.mapr.com/ fetched on 05/21/2013].

[18] M. Benedikt, P. Collier, V. Mertens, et al. *LHC Design Report*, volume v.3 : the LHC Injector Chain. CERN, Geneva, 2004.

[19] I. Bird, L. Bechtev, M. Branco, et al. LCG- Baseline Services Group Report . [Available online; http://lcg-archive.web.cern.ch/lcg-archive/peb/bs/BSReport-v1.0.pdf fetched on 05/21/2013].

[20] B. Bockelman. Using Hadoop as a grid storage element. In *Journal of physics: Conference series*, volume 180, page 012047. IOP Publishing, 2009.

[21] C. Boeheim, A. Hanushevsky, D. Leith, et al. Scalla: Scalable Cluster Architecture for Low Latency Access Using xrootd and olbd Servers. Technical report.

[22] D. Borthakur. Hadoop Architecture Guide. [Available online; http://hadoop.apache.org/docs/r1.0.4/hdfs_design.pdf fetched on 05/21/2013].

[23] R. Brun and F. Rademakers. ROOT - An Object Oriented Data Analysis Framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 389(1):81–86, 1997. See also http://root.cern.ch.

[24] O. S. Brüning, P. Collier, P. Lebrun, et al. *LHC Design Report*, volume v.1: the LHC Main Ring. CERN, Geneva, 2004.

[25] CERN. ROOT: A Data Analysis Framework. [Online; http://root.cern.ch/drupal/ fetched on 05/21/2013].

[26] CERN. The LHCb collaboration. [Online; http://lhcb.web.cern.ch/lhcb/ fetched on 05/21/2013].

[27] CERN. TProofBench. [Online; `http://root.cern.ch/drupal/content/proof-benchmark-framework-tproofbench` fetched on 05/21/2013].

[28] S. Chatrchyan, G. Hmayakyan, V. Khachatryan, et al. The CMS experiment at the CERN LHC. *Journal of Instrumentation*, 0803:S08004, 2008.

[29] S. Chekanov. SCaVis: Scientific Computiation and Visualization Environment. [Online; `jwork.org/scavis` fetched on 05/21/2013].

[30] S. N. T.-c. Chiueh and S. Brook. A survey on virtualization technologies. *RPE Report*, pages 1–42, 2005.

[31] C. Chu, S. K. Kim, Y.-A. Lin, et al. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, 19:281, 2007.

[32] P. Cortese, F. Carminati, C. W. Fabjan, et al. *ALICE computing: Technical Design Report*. Technical Design Report ALICE. CERN, Geneva, 2005. Submitted on 15 Jun 2005.

[33] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, page 10, Berkeley, CA, USA, 2004. USENIX Association.

[34] A. Dorigo, P. Elmer, F. Furano, and A. Hanushevsky. XROOTD- A highly scalable architecture for data access. *WSEAS Transactions on Computers*, 1(4.3), 2005.

[35] J. Ekanayake, H. Li, B. Zhang, et al. Twister: a runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[36] J. Ekanayake, S. Pallickara, and G. Fox. MapReduce for Data Intensive Scientific Analyses. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 277–284, Washington, DC, USA, 2008. IEEE Computer Society.

[37] C. Evangelinos and C. Hill. Cloud Computing for parallel Scientific HPC Applications: Feasibility of running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2. *ratio*, 2(2.40):2–34, 2008.

[38] I. Foster. What is the grid?-a three point checklist. *GRIDtoday*, 1(6), 2002.

[39] J. F. G-Oetringhaus. The CERN analysis facility—a PROOF cluster for day-one physics analysis. volume 119, page 072017. IOP Publishing, 2008.

[40] G. Ganis, J. Iwaszkiewicz, and F. Rademakers. Data Analysis with PROOF. Number PoS(ACAT08)007 in Proceedings of XII International Workshop on Advanced Computing and Analysis Techniques in Physics Research.

[41] G. Ganis, J. Iwaszkiewicz, and F. Rademakers. Scheduling and Load Balancing in the Parallel ROOT Facility. Number PoS(ACAT)022 in Proceedings of XI International Workshop on Advanced Computing and Analysis Techniques in Physics Research.

[42] Gartner. IT Glossary. [Online; `http://www.gartner.com/it-glossary/virtualization/` fetched on 05/21/2013].

[43] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. volume 37, pages 29–43, New York, NY, USA, October 2003. ACM.

[44] F. Glaser and H. Neukirchen. Analysing High-Energy Physics Data Using the MapReduce Paradigm in a Cloud Computing Environment. Technical Report VHI-01-2012, Engineering Research Institute, University of Iceland, Reykjavik, Iceland, 2012.

[45] Google. Google App Engine. [Online; `https://appengine.google.com/` fetched on 05/21/2013].

[46] Google. Google Drive. [Online; `https://drive.google.com/` fetched on 05/21/2013].

[47] Y. Gu and R. L. Grossman. Sector and Sphere: The Design and Implementation of a High Performance Data Cloud. volume 367, pages 2429–2445. The Royal Society, 2009.

[48] A. Hanushevsky and D. L. Wang. Scalla: Structured Cluster Architecture for Low Latency Access. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1168–1175. IEEE, 2012.

[49] M. D. Hill. What is scalability? *ACM SIGARCH Computer Architecture News*, 18(4):18–21, 1990.

[50] A. Iosup, S. Ostermann, M. N. Yigitbasi, et al. Performance analysis of cloud computing services for many-tasks scientific computing. *Parallel and Distributed Systems, IEEE Transactions on*, 22(6):931–945, 2011.

[51] K. R. Jackson, L. Ramakrishnan, K. Muriki, et al. Performance analysis of high performance computing applications on the amazon web services cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 159–168. IEEE, 2010.

[52] M. Massie et al. The Ganglia Monitoring System. [Online;http://ganglia.sourceforge.net/ fetched on 05/21/2013].

[53] A. McKenna, M. Hanna, E. Banks, et al. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome research*, 20(9):1297–1303, 2010.

[54] P. Mell and T. Grance. The NIST definition of Cloud Computing. *NIST special publication*, 800:145, Sept. 2011.

[55] Microsoft. Windows Azure. [Online; http://www.windowsazure.com/en-us/ fetched on 05/21/2013].

[56] Microsoft Research. Project Daytona. [Online; http://research.microsoft.com/en-us/projects/daytona fetched on 05/21/2013].

[57] Nokia Research Center. Disco Project. [Online;http://discoproject.org fetched on 05/21/2013].

[58] H. Riahi, G. Donvito, L. Fanò, et al. Using Hadoop File System and MapReduce in a small/medium Grid site. volume 396, page 042050, 2012.

[59] S. Seo, E. J. Yoon, J. Kim, et al. Hama: An efficient matrix computation with the mapreduce framework. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 721–726. IEEE, 2010.

[60] T. Sjöstrand, S. Mrenna, and P. Skands. A brief introduction to PYTHIA 8.1. *Computer Physics Communications*, 178(11):852–867, 2008.

[61] I. Sriram and A. Khajeh-Hosseini. Research Agenda in Cloud Technologies. *CoRR*, abs/1001.3259, 2010.

[62] Simplified Wrapper and Interface Generator. [Online; http://www.swig.org/ fetched on 05/21/2013].

[63] O. Tatebe, K. Hiraga, and N. Soda. Gfarm grid file system. *New Generation Computing*, 28(3):257–275, 2010.

[64] The LCG TDR Editorial Board. LHC Computing Grid Technical Design Report. Technical Report 1.04, June 2005.

[65] The ROOT team. ROOT User Guide 5.26. [Available online http://root.cern.ch/download/doc/ROOTUsersGuide.pdf fetched on 05/21/2013].

[66] L. Wang, J. Tao, H. Marten, et al. MapReduce across Distributed Clusters for Data-intensive Applications. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2004–2011, May.

[67] W. Zhao, H. Ma, and Q. He. Parallel k-means clustering based on mapreduce. In *Cloud Computing*, pages 674–679. Springer, 2009.