Optimizing Data Management for MapReduce Applications on Large-Scale Distributed Infrastructures

Diana Moise

THÈSE / ENS CACHAN - BRETAGNE

sous le sceau de l'Université européenne de Bretagne

pour obtenir le titre de
DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN
Mention : Informatique
École doctorale MATISSE

présentée par

Diana Maria Moise

Préparée à l'Unité Mixte de Recherche 6074
Institut de recherche en informatique
et systèmes aléatoires

Optimizing data management for MapReduce applications on large-scale distributed infrastructures

Thèse soutenue le 16 décembre 2011
devant le jury composé de :

Frédéric DESPREZ,
Directeur de recherche - ENS Lyon, INRIA Grenoble Rhône-Alpes/ *rapporteur*

Guillaume PIERRE,
Professeur associé - Vrije Universiteit Amsterdam / *rapporteur*

Valentin CRISTEA,
Professeur des universités - University Politehnica Of Bucharest / *examinateur*

Luc BOUGÉ
Professeur des universités - ENS Cachan-Bretagne / *directeur de thèse*

Gabriel ANTONIU
Chargé de recherche, INRIA Rennes Bretagne-Atlantique / *directeur de thèse*

Résumé

Dans ce manuscrit, nous avons abordé la gestion de manière efficace des données traitées et produites par des applications MapReduce sur infrastructures distribuées à grande échelle.

La première partie du manuscrit est consacrée au contexte de notre travail. Dans un premier temps, la partie se concentre sur le paradigme MapReduce et ses mises en œuvre. Dans un second temps, elle présente l'environnement ciblé, les infrastructures à grande échelle. Nous détaillons deux environnements largement utilisés pour exécuter des applications data-intensive. La première plate-forme sur laquelle nous nous concentrons concerne les grilles informatiques, une approche bien établie dans le domaine de recherche du calcul parallèle et distribué. Nous présentons ensuite le cloud computing, un modèle récemment émergé, qui a gagné en popularité très rapidement au cours des dernières années.

La deuxième partie présente les contributions de cette thèse. Nous proposons un système de fichiers distribué, optimisé pour des accès hautement concurrents, qui puisse servir comme couche de stockage pour des applications MapReduce. Nous avons conçu le BlobSeer File System (BSFS), basé sur BlobSeer, un service de stockage distribué, hautement efficace, facilitant le partage de données à grande échelle. Nous étudions également plusieurs aspects liés à la gestion des données intermédiaires dans des environnements MapReduce. Nous explorons les contraintes des données intermédiaires MapReduce à deux niveaux: dans le même job MapReduce et pendant l'exécution des pipelines d'applications MapReduce. Enfin, nous proposons des extensions de Hadoop, un environnement MapReduce populaire et open-source, comme par example le support de l'opération append.

La troisième partie est consacrée à l'implémentation des travaux présentés dans la partie 2. Dans cette partie, nous décrivons la mise en œuvre du système de fichiers BSFS et son interconnexion avec Hadoop et BlobSeer. Nous nous concentrons aussi sur les extensions et les modifications réalisées afin d'améliorer l'environnement Hadoop MapReduce avec les caractéristiques précédemment mentionnées.

La quatrième partie présente une évaluation expérimentale étendue de nos contributions. Nous évaluons l'impact du système de fichiers BSFS au travers d'une série d'expériences avec des benchmarks synthétiques et des applications MapReduce réelles. Cette partie valide aussi notre contribution dans le contexte des données intermédiaires. L'évaluation expérimentale de l'opération append est également incluse. Enfin, nous fournissons une évaluation des coûts d'exécution des applications MapReduce dans le Cloud, en se basant sur le modèle de coût d'Amazon EC2.

La cinquième partie conclut ce manuscrit en présentant un résumé de nos contributions et en énumérant plusieurs directions de recherche futures.

Mots clés :

Applications data-intensive, MapReduce, plate-formes distribuées à grande échelle, grilles informatiques, cloud computing, gestion des données intermédiaires, Hadoop, HDFS, BlobSeer, haut débit, accès hautement concurrents.

Abstract

In this manuscript, we addressed the problem of efficiently managing data processed and produced by MapReduce applications, on infrastructures distributed at large scales.

The first part of the manuscript is dedicated to the context of our work. It first focuses on the MapReduce programming paradigm and its implementations, and then presents the targeted environment, i.e., large-scale infrastructures. We detail two environments that are most-commonly used for executing data-intensive applications. The first platform that we focus on is the Grid, a well-established approach to distributed computing. We further present Cloud computing, a recently-emerged model with a continuously-growing popularity.

The second part presents the contributions of this thesis. We propose a concurrency-optimized file system for MapReduce Frameworks. As a starting point, we rely on BlobSeer, a framework that was designed as a solution to the challenge of efficiently storing data generated by data-intensive applications running at large scales. We have built the BlobSeer File System (BSFS), with the goal of providing high throughput under heavy concurrency to MapReduce applications. We also study several aspects related to intermediate data management in MapReduce frameworks. We investigate the requirements of MapReduce intermediate data at two levels: inside the same job, and during the execution of pipeline applications. Finally, we show how BSFS can enable extensions to the de facto MapReduce implementation - Hadoop, such as the support for the append operation.

The third part details the implementation of the work presented in Part 2. In this part we describe the implementation of the BSFS file system and its interconnection with Hadoop and BlobSeer. We also focus on the extensions and modifications we carried out within the Hadoop MapReduce framework to enhance it with the aforementioned features.

The fourth part shows the extensive experimental evaluation of our contributions. We evaluate the impact of the BlobSeer File System by performing experiments both with synthetic microbenchmarks and with real MapReduce applications. This part further validates our contribution in the context of intermediate data. The experimental evaluation of the append operation is also included in this part. Finally, we provide a cost evaluation of running MapReduce applications in the Cloud, by taking into account Amazon's EC2 cost model.

The fifth part concludes this manuscript, by presenting a summary of our contributions and listing several future directions brought forth by our work.

Keywords:

Data-intensive applications, MapReduce, large-scale distributed platforms, grid, cloud computing, intermediate data management, Hadoop, HDFS, BlobSeer, high throughput, heavy access concurrency.

N° d'ordre :

École normale supérieure de Cachan - Antenne de Bretagne

Campus de Ker Lann - Avenue Robert Schuman - 35170 BRUZ

Tél : +33(0)2 99 05 93 00 - Fax : +33(0)2 99 05 93 29

Contents

Chapter 1

Introduction

Contents

1.1 Context

A large part of today's most popular applications are data-intensive. Whether they are scientific applications or Internet services, the data volume they process is continuously growing. This trend followed by more and more applications entails increasing demands in terms of the computational and data requirements. Some applications generate data volumes reaching hundreds of terabytes and even petabytes.

Two main aspects arise when trying to accommodate the size of the data: processing the computation in a manner that is efficient both in terms of resources and time, and providing storage capable to deal with the requirements of data-intensive applications. Since the input data is large, the computation, which is, in most cases straightforward, is distributed across hundreds or thousands of machines; thus, the application is split into tasks that run in parallel on different machines, tasks that will need to access the data in a highly concurrent manner.

Handling massive data has a strong impact on the design of the storage layer, which must be able to cope with storing huge files, while still supporting fine-grained access to data. Files are distributed at a large scale, I/O throughput must be sustained at a high level, even in the context of heavy concurrency.

Specialized abstractions like Google's MapReduce and Pig-Latin were developed to efficiently manage the workloads of data-intensive applications. These models propose high-level data processing frameworks intended to hide the details of parallelization from the user. Such frameworks rely on storing huge objects and target high performance by optimizing the parallel execution of the computation.

Google's MapReduce is a parallel programming paradigm successfully used by large Internet service providers to perform computations on massive amounts of data. A computation takes a set of input key-value pairs, and produces a set of output key-value pairs. The user of the MapReduce library expresses the computation as two functions: *map*, that processes a key-value pair to generate a set of intermediate key-value pairs, and *reduce*, that merges all intermediate values associated with the same intermediate key. The framework takes care of splitting the input data, scheduling the jobs' component tasks, monitoring them and re-executing the failed ones.

Hadoop's implementation of MapReduce follows the Google model. The framework consists of a single master *jobtracker*, and multiple slave *tasktrackers*, one per node. A MapReduce job is split into a set of tasks, which are executed by the tasktrackers, as assigned by the jobtracker. The input data is also split into chunks of equal size, that are stored in a distributed file system across the cluster. First, the map tasks are run, each processing a chunk of the input file, by applying the map function defined by the user, and generating a list of key-value pairs. After all the maps have finished, the tasktrackers execute the reduce function on the map outputs.

Both the input data and the output produced by the reduce function are stored in a distributed file system; the storage layer is a key component of MapReduce frameworks, as its design and functionalities influence the overall performance. MapReduce applications process data consisting of up to billions of small records (of the order of KB); storing the data in a large number of approximately KB-sized files would be impossible to manage and certainly inefficient. Thus, the data sets are packed together into huge files (hundreds of GB).

There are numerous challenges raised by managing data processed and produced by MapReduce applications, on infrastructures distributed at large scales. Although a large community has tackled these challenges over the past decade, existing solutions still face many limitations that need to be overcome.

1.2 Contributions

This work was carried out in the context of MapReduce applications, focusing on providing efficient data management on large-scale infrastructures. The contributions of this thesis are detailed below.

1. The work addresses in a first phase, the problem of how to efficiently provide storage for MapReduce frameworks. We particularly focus on the Hadoop MapReduce framework and its underlying storage layer, HDFS. Our contribution consists in building a file system that can successfully be used as storage backend for MapReduce applications executed with Hadoop. Our proposed file system can be used as both a standalone DFS and as a replacement for HDFS in the Hadoop framework. The benefits of

our proposal for efficiently storing MapReduce applications were validated through large-scale experiments.

2. We addressed the problem of efficiently managing intermediate data generated between the "map" and "reduce" phases of MapReduce computations. In this context, we propose to store the intermediate data in the distributed file system used as backend. The work brings a twofold contribution. First, we investigate the features of intermediate data in MapReduce computations and we propose a new approach for storing this kind of data in a DFS. In this manner, we avoid the re-execution of tasks in case of failures that lead to data loss. Second, we consider as storage for intermediate data, the file system we previously proposed, suitable for the requirements of intermediate data: availability and high I/O access.

3. In order to speed-up the execution of pipeline MapReduce applications (applications that consist of multiple jobs executed in a pipeline) and also, to improve cluster utilization, we propose an optimized Hadoop MapReduce framework, in which the scheduling is done in a dynamic manner. We introduce several optimizations in the Hadoop MapReduce framework in order to improve its performance when executing pipelines. Our proposal consists mainly in a new mechanism for creating tasks along the pipeline, as soon as the tasks' input data becomes available. This dynamic task scheduling leads to an improved performance of the framework, in terms of job completion time. In addition, our approach ensures a more efficient cluster utilizations, with respect to the amount of resources that are involved in the computation.

4. The work also focuses on the append operation as a functionality that can bring benefits at two levels. First, introducing append support at the level of the file system may be a feature useful for some applications (not necessarily belonging to the MapReduce class). Our proposed file system provides efficient support for the append operation. We further modified the Hadoop MapReduce framework to take advantage of this functionality. In our modified Hadoop framework, the reducers append their data to a single file, instead of writing it to a separate file, as it was done in the original version of Hadoop.

All the aforementioned contributions were implemented from scratch and then extensively tested on large-scale platforms, in two environments: Grids and Clouds.

1.3 Publications

Journals

- *BlobSeer: Next Generation Data Management for Large Scale Infrastructures.* Nicolae B., Antoniu G., Bougé L., Moise D., Carpen-Amarie A. Journal of Parallel and Distributed Computing 71, 2 (2011), pp. 168-184

Conferences and Workshops

- *Optimizing Intermediate Data Management in MapReduce Computations* Moise D., Trieu T.-T.-L., Antoniu G., Bougé L. The ACM SIGOPS Eurosys 11 conference, CloudCP 2011

1st International Workshop on Cloud Computing Platforms (2011)

- *A Cost-Evaluation of MapReduce Applications in the Cloud.* Moise D., Carpen-Amarie A., Antoniu G., Bougé L. To appear in the Grid'5000 Spring School Proceedings (2011)
- *BlobSeer: Bringing High Throughput under Heavy Concurrency to Hadoop Map-Reduce Applications* Nicolae B., Moise D., Antoniu G., Bougé L., Dorier M. The 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010) (2010)
- *Improving the Hadoop Map/Reduce Framework to Support Concurrent Appends through the BlobSeer BLOB management system* Moise D., Antoniu G., Bougé L. The 19th ACM International Symposium on High Performance Distributed Computing (HPDC'10), Workshop on MapReduce and its Applications (2010)
- *Large-Scale Distributed Storage for Highly Concurrent MapReduce Applications* Moise D. The 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010): PhD Forum (2010)
- *Resource CoAllocation for Scheduling Tasks with Dependencies, in Grid* Moise D., Moise I., Pop F., Cristea V. The 2nd International Workshop on High Performance in Grid Middleware (HiPerGRID 2008) (2008)
- *Advance Reservation of Resources for Task Execution in Grid Environments* Moise I., Moise D., Pop F., Cristea V. The 2nd International Workshop on High Performance in Grid Middleware (HiPerGRID 2008) (2008)

1.4 Organization of the manuscript

This manuscript is organized in five main parts.

The first part is dedicated to the context of our work. The first chapter of this part presents the target application domain of our work, i.e., data-intensive applications, more specifically, MapReduce applications. The chapter further focuses on the MapReduce programming paradigm and its implementations. The second chapter of the context of this work concerns large-scale infrastructures. We detail two environments that are most-commonly used for executing data-intensive applications. The first platform that we focus on is the Grid, a well-established approach to distributed computing. In a second part of the chapter, we present Cloud computing, a recently-emerged model with a continuously-growing popularity. The closing chapter of this first part introduces BlobSeer, a framework that was designed as a solution to the challenge of efficiently storing data generated by data-intensive applications running at large scales.

The second part presents the contributions of this thesis. In Chapter 5 we propose a concurrency-optimized file system for MapReduce Frameworks. Using BlobSeer as a starting point, we have built the BlobSeer File System (BSFS), with the goal of providing high throughput under heavy concurrency to MapReduce applications. Chapter 6 develops our

contribution in the context of intermediate data. We study several aspects related to intermediate data management in MapReduce frameworks. The work presented Chapter 6 addresses intermediate data at two levels: inside the same job, and during the execution of pipeline applications. Finally, Chapter 7 shows how BSFS can enable extensions to the de facto MapReduce implementation - Hadoop. The focus of this chapter is on introducing support for the append operation in Hadoop.

The third part consists of a chapter that details the implementation of the work presented in Part 2. In this part we describe the implementation of the BSFS file system and its interconnection with Hadoop and BlobSeer. We also focus on the extensions and modifications we carried out within the Hadoop MapReduce framework to enhance it with the aforementioned features. The last section of this chapter is dedicated to the deployment tools we developed to allow us to evaluate our work.

The forth part shows the extensive experimental evaluation of our contributions. Chapter 9 evaluates the impact of the BlobSeer File System by performing experiments both with synthetic microbenchmarks and with real MapReduce applications. The next chapter of this part validates our contribution in the context of intermediate data. The experimental evaluation of the append operation is also included in this part. In the final chapter of this part, we provide a cost evaluation of running MapReduce applications in the Cloud, by looking into several aspects: the overhead incurred by executing the job on the Cloud, compared to executing it on a Grid, the actual costs of renting Cloud resources, and also, the impact of the storage system used as backend by MapReduce applications.

The fifth part concludes this manuscript, by presenting a summary of our contributions and listing several future directions brought forth by our work.

Part I

Context: Data-intensive Applications on Large-Scale Distributed Platforms

Chapter 2

Data-intensive applications

Contents

APPLICATIONS in most of today's areas of both science and industry are *data-intensive*. The "data-intensive" term refers to applications that spend most of their execution time on I/O operations. An indirect but somehow, intrinsic meaning of the term is related to the amount of data that data-intensive applications usually manage. Whether they are scientific applications or Internet services, the data volume these applications process is continuously growing.

As applications are becoming increasingly complex in terms of both data and computation, special interest has been devoted by the community to analyzing the requirements of data-intensive applications. This class of applications has led to the emergence of *data-intensive scalable computing (DISC)* that is now considered a new paradigm in scientific discovery after empirical, theoretical, and computational scientific approaches. DISC includes applications where the main challenge comes from the data size, its complexity or the rate at which data is acquired. Data-intensive applications can be characterized through several attributes.

Large datasets. The size of the data handled by DISC applications is usually the factor that impacts the most the design and execution of this kind of applications. The huge volume of data processed by DISC applications is of the order of terabytes, at least. Nowadays, the magnitude of the data size goes beyond petabytes, reaching even exabytes.

High I/O rate. In data-intensive applications, a significant amount of time is dedicated to reading and writing data, as opposed to the actual computation that is far less time-consuming.

Heavy read/write concurrency. Typically, the processing exhibited by the applications is done in parallel, which yields heavy read access to multiple parts of the datasets. Heavy write concurrency is also a common feature of data-intensive applications that generate large amounts of data.

In practice, the aforementioned features give rise to real challenges when it comes to dealing with data-intensive applications. Performing actions such as data storing, analyzing, visualizing, etc., becomes a problematic issue that needs dedicated approaches. In this direction, new programming models and specialized frameworks have been developed from scratch to specifically meet the requirements of data-intensive applications. MapReduce [29], Hadoop [16], Dryad [46] are well-known examples of abstractions and execution environments that enable parallel processing of large datasets on large-scale infrastructures. Issues such as fault tolerance, performance, data storage, scheduling, represent a central design point in those frameworks.

The rest of this chapter is dedicated to MapReduce, the programming model that is nowadays widely acclaimed as a key solution to designing data-intensive applications. Additionally, we present two of the reference MapReduce implementations, along with the frameworks that provide support for the MapReduce abstraction.

2.1 MapReduce applications

2.1.1 The MapReduce programming model

Google introduced MapReduce [29] as a solution to the need to process datasets up to multiple terabytes in size on a daily basis. The goal of the MapReduce programming model is to provide an abstraction that enables users to perform computations on large amounts of data. Through its design, the MapReduce paradigm provides support for the following aspects:

Automatic parallelization of computations. The parallelization of the user's computations, as well as their parallel execution are automatically handled by the model. The user supplies the computations to be performed and the MapReduce framework orchestrates the dispatching of computations and their execution.

Large-scale data distribution. The MapReduce paradigm is designed to efficiently handle the processing of very large datasets. This implies that the distribution of the user datasets is inherently managed by the framework. The paradigm also aims at enabling users to utilize the resources of a large distributed platform.

Simple, yet powerful interface. The MapReduce model exposes a simple interface, that can be easily manipulated by users without any experience with parallel and distributed systems. However, the interface is versatile enough so that it can be employed to suit a wide range of data-intensive applications.

User-transparent fault tolerance. The typical environment targeted by MapReduce computations is one that deals with faults on a regular basis. Thus, the framework is expected to ensure fault tolerance through mechanisms that do not require user intervention.

Commodity hardware. The paradigm is engineered to work on clusters of inexpensive machines, therefore is does not require specialized hardware to run on.

The MapReduce abstraction is inspired by the "map" and "reduce" primitives commonly used in functional programming. When designing an application using the MapReduce paradigm, the user has to specify two functions: *map* and *reduce* that are executed in parallel on multiple machines. Applications that can be modeled by the means of MapReduce, mostly consist of two computations: the "map" step, that applies a filter on the input data, selecting only the data that satisfies a given condition, and the "reduce" step, that collects and aggregates all the data produced by the first phase.

The paradigm specifies that the "map" and "reduce" functions manipulate data in the form of key/value pairs. Thus, "map" takes as input a list of key/value pairs and produces a set of *intermediate* key/value pairs. Usually, the same intermediate key appears in several pairs, associated with different values. The "reduce" function also handles data as key/value pairs, thus the intermediate values that share the same key are grouped together and passed to the "reduce" phase for processing. The "reduce" function applies the user computation on each intermediate key and its corresponding set of values. The result of this phase is the final output of the MapReduce processing.

2.1.2 The Google implementation

Along with this abstraction, Google also proposed a framework that enables distributed processing of MapReduce computations. The framework implements the features introduced in Section 2.1.1, exposing the simple interface specified by the abstraction to the user.

A typical scenario of running an application with Google's MapReduce framework comprises two phases: first, the user supplies the input data and the "map" and "reduce" functions; second, the framework performs the computations and generates the result. These phases are implemented through a user-side library and the MapReduce framework. On the user's side, the MapReduce library performs several actions.

1. The library first splits the input data into fixed-size blocks, creating thus, a set of splits. The size of each split can be configured by the user and typically varies from 16 MB to 64 MB.

2. The "map" and "reduce" functions that will be invoked on multiple machines, are instantiated into *tasks* that are created by the library as follows: the "map" tasks are created one per input split, while the number of "reduce" tasks is specified by the user. The mechanism of distributing the "reduce" invocations relies on a *partitioning*

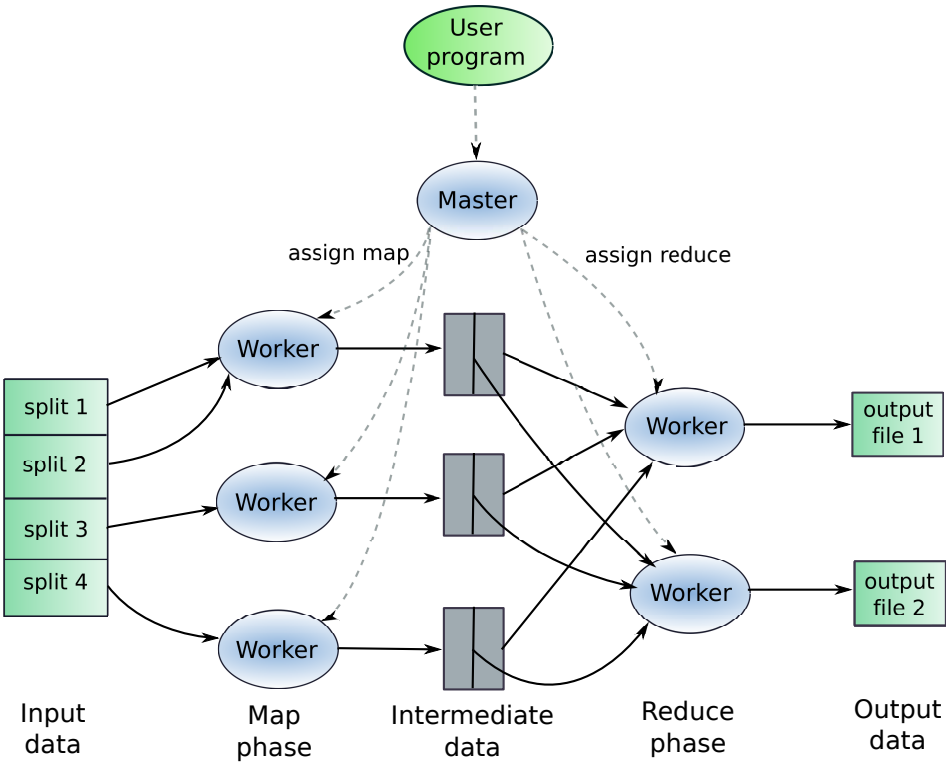


Figure 2.1: Google's MapReduce framework.

function supplied by the user. This function splits the intermediate key space into the same number of partitions as the number of "reduce" tasks provided by the user.

The Google MapReduce implementation is designed in the master-slave fashion: a centralized *master* assigns tasks to multiple *workers* that execute them in parallel. The steps carried out by the framework in order to execute a MapReduce application are illustrated on Figure 2.1, and can be summarized as follows:

1. The master dispatches the "map" and "reduce" tasks to idle workers. Each worker is assigned a task to run.

2. A worker that is assigned a "map" task is also allocated an input split. The mapper reads the input data in order to parse key/value pairs on which it then applies the "map" function, producing intermediate key/value pairs. The intermediate data is stored on the worker's local disk, partitioned into regions corresponding to each reducer. The locations of the intermediate pairs are reported back to the master.

3. A worker running a "reduce" task receives from the master the locations holding the partitions it was assigned to process. The "reduce" worker transfers all the intermediate pairs over the network, then sorts them by key, grouping together all the values associated with the same key. The sorting phase is an optimization employed by the framework for efficiency reasons, as it is very likely that the same intermediate key is produced by several mappers. The worker then takes one pair at a time and applies the "reduce" computation on the key and its associated values. The result of this

procedure is appended to a final output file. To avoid synchronization, each worker appends the data it produces to its own output file.

4. When all the "map" and "reduce" tasks have completed, the master notifies the user. The output of the MapReduce processing consists of as many files as the number of "reduce" tasks that were executed by the framework. The merging of the output files into a single file is an additional step that has to be performed by the user. However, most distributed applications (including the MapReduce class) are able to deal with multiple files as input data.

In the Google implementation, the master entity plays a key role in the system. In order to be able to dispatch tasks and coordinate workers, the master maintains several data structures. For each task, the master keeps its state and progress, along with the identity of the worker in charge of running the task. If the task is a "map" computation, the master also stores the locations and sizes of the intermediate data the task produced so far. This information is updated as the "map" task is being executed and it is simultaneously pushed to "reduce" workers. The first type of data structure helps the master to keep track of the execution progress, and also to handle failures. The second structure is needed so that the intermediate data produced by the mappers can be propagated to the reducers.

The MapReduce model is one of the promoters of a trend that has recently emerged in the distributed-computing community. This trend brings forward a computational model that consists in *shipping computation to data* instead of transferring large datasets across the network. This approach relies on the assumption that in commodity- hardware environments, network bandwidth is a rather scarce resource. It is thus preferable to avoid transferring massive amounts of input data over the network, from the nodes that store the data, to the nodes that perform the computation. MapReduce takes advantage of the deployment model that collocates storage and computation on the same nodes. When dispatching tasks to workers, the master takes into account the *locality* criterion, that is, it attempts to schedule a "map" task as close as possible to the node that stores the corresponding input data: the node itself, if available, or the node that is the closest to the data, in terms of the network topology. Google reports that the locality mechanism helps preserve network bandwidth during large MapReduce computations, as most of the input data is read locally.

2.1.3 Applications

Modeling an application with MapReduce requires the user to supply the definition of the "map" and "reduce" functions. These methods need to be specified in compliance with the MapReduce abstraction. We first give the algorithmic description of the canonical MapReduce example, i.e., the *word count* application. We also provide examples showing the specification of two of the most-commonly use MapReduce applications: *grep* and *sort*.

Word Count

The *word count* application is a simple example that illustrates how a basic and common problem can be solved using MapReduce. The application consists in counting the number of occurrences of each word in a large collection of documents. Example 1 provides a specification of the word-count problem in the MapReduce style. The "map" function parses each

Example 1 Word Count

```
map(K key, V value)
      //key: file name
      //value: file contents
      for word ∈ value
          emitIntermediate(word, 1)
reduce(K key, V[] values)
      //key: a word
      //value: a list of '1'-s
      sum ← 0
      for v ∈ values
          sum ← sum + v
      emit(key, sum)
```

Example 2 Distributed Grep

```
map(K key, V value)
      //key: file name
      //value: file contents
      for line ∈ value
          if line matches pattern
            emitIntermediate(key, line)
reduce(K key, V[] values)
      //key: file name
      //value: a list of lines matching the pattern
      for v ∈ values
          emit(key, v)
```

word of the input document and produces as intermediate data, a key/value pair consisting of a word and the value '1', suggesting that the word appeared once. As the MapReduce framework groups together all intermediate values related to the same key, the "reduce" function processes a key and its list of associated values, i.e, a list containing '1'-s. For each key, the "reduce" computation sums up all the values in the list. This means that the function computes the number of times the word appears, for each word.

Distributed grep

This application scans the input data in order to find the lines that match a specific pattern. The grep example can be easily expressed with MapReduce (Example 2). The "map" function processes the input file line by line and matches each single line against the given pattern. If the matching is successful, then the line is emitted as intermediate data. The "reduce function" simply passes the intermediate data as final result.

Example 3 Distributed Sort

```
map(K key, V value)
      emitIntermediate(key, value)
reduce(K key, V[] values)
      for v ∈ values
          emit(key, v)
```

Distributed Sort

Sorting datasets is a basic operation that is widely employed by data-processing applications. It consists in sorting key/value records based on key. The sorting is based on a *comparator*, a user-specified function that is applied on the keys of the records. With MapReduce, sorting is a straightforward operation. As shown by Example 3, both the "map" and "reduce" functions are trivial computations, as they simply take the input data and emit it as output data. The sort MapReduce implementation takes advantage of the default optimizations performed by the framework. First, the records are automatically parsed as key/value pairs and dispatched to mappers. Second, the framework implicitly sorts all the values associated with the same intermediate key, consequently, the list of values received by the "reduce" function is already sorted for each key. Third, sorting by key is achieved through the partitioning function that delegates a range of keys to each reducer. The partitioning function (hashing scheme) must satisfy the following condition:

$$\texttt{if } k_1 < k_2 \texttt{ => hash}(k_1) \texttt{ < hash}(k_2).$$

This ensures that the partitioning of the keys namespace to reducers leads to a sorted output: $reducer_1$ producing $output_1$ is assigned the first partition of intermediate keys, $reducer_2$ takes the second, and so on.

More examples

MapReduce is used to model a wide variety of applications, belonging to numerous domains such as analytics (data processing), image processing, machine learning, bioinformatics, astrophysics, etc.

A first class of application that can be designed with MapReduce is brought forward by Google and refers to *information extraction and text processing*. It contains operations that Google, as well as most Internet-services providers, execute on a daily basis [29]: grep, sort, inverted index construction, page rank, web access log stats, document clustering, web link-graph reversal, Bayesian classification, etc.

Another significant usage of the MapReduce paradigm relates to *search query analysis and information retrieval*. It targets applications that perform database-like operations, such as: joins, aggregations, queries, group-by operations, etc. A large part of the SQL-like applications can be easily expressed as MapReduce operations [51, 78]. However, some queries require a more complex design that, when written with MapReduce, may lead to a workflow that is complicated to manage and debug. In this direction, several frameworks and

abstractions were developed based on MapReduce, with the goal of providing a simple-to-use interface for expressing database-like queries [64, 6].

Bioinformatics is one of the numerous research domains that employ MapReduce to model their algorithms [69, 58, 56]. As an example, CloudBurst [69] is a MapReduce-based algorithm for mapping next-generation sequence data to the human genome and other reference genomes, for use in a variety of biological analyses.

Other research areas where MapReduce is widely employed, include: astronomy [76], social networks [26], artificial intelligence [39, 60], image and video processing, simulations, etc.

2.2 Hadoop-based applications

2.2.1 The Hadoop project

The Hadoop project [15] was founded by Yahoo! in 2006. It started out as an open-source implementation of the MapReduce model promoted by Google. By 2008, Hadoop was being used by other large companies apart from Yahoo!, such as Last.fm and Facebook. A notable Hadoop use-case belongs to the New York Times. In 2007, the company rented resources on Amazon's EC2 in order to convert the newspaper's scanned archives to PDF files. The data reached 4 TB in size and the computation took less than 24 hours on 100 machines. Hadoop was used to run this application in the cloud environment provided by Amazon. In 2008 and then in 2009, Hadoop broke the world record for sorting 1 TB. Using Hadoop, Yahoo! managed to sort a terabyte of data in 62 seconds. Currently, the world record is 60 seconds, held by a team from the University of California, San Diego.

The core of the Hadoop project consists of the MapReduce implementation and the Hadoop Distributed File System (HDFS). Along the years, several sub-projects have been developed as part of the Hadoop collection. The sub-projects include frameworks that cover a wide range of the distributed computing area. They were either built as complementary to the Hadoop core, or on top of the core, with the purpose of providing higher-level abstractions. Currently, the Hadoop project offers the following services.

MapReduce: a framework for large-scale data processing.

HDFS: a distributed file system for clusters of commodity hardware.

Avro: a system for efficient, platform-independent data serialization.

Pig: a distributed infrastructure for performing high-level analysis on large data sets.

HBase: a distributed, column-oriented storage for large amounts of structured data, on top of HDFS.

ZooKeeper: a service which enables highly reliable coordination for building distributed applications.

Hive: a data-warehouse system that provides data summarization, ad-hoc queries, and the analysis of large datasets stored in HDFS.

Chukwa: a system for collecting and analyzing data on large-scale platforms. It also includes tools for displaying, monitoring and analyzing results, in order to make the best use of the collected data.

Cassandra: a distributed database management system. It was designed to handle very large amounts of data, while providing a highly-available service with no single point of failure.

Ever since it was released, Hadoop's popularity rapidly increased, as a result of the features it yields, such as performance, simplicity and inexpensiveness. The list of Hadoop users [12] includes companies and institutes that employ one or several Hadoop projects for research or production purposes. Companies such as Adobe and EBay use Hadoop MapReduce, HBase and Pig for structured data storage and search optimization. Facebook makes use of HDFS and Hive for storing logs and data sources and performing queries on them. Twitter heavily uses Pig, MapReduce and Cassandra to process all types of data generated across Twitter. Reports from Yahoo! show that Hadoop is currently running on more than 100,000 CPUs, and on the largest Hadoop cluster (comprising 4500 nodes and several terabytes of RAM and petabytes of storage). Yahoo! also reports that more than 60 % of the Hadoop jobs it runs are Pig jobs.

In addition to its cluster usage, Hadoop is becoming a de-facto standard for cloud computing. The generic nature of cloud computing allows resources to be purchased on-demand, especially to augment local resources for specific large or time-critical tasks. Several organizations offer cloud compute cycles that can be accessed via Hadoop. Amazon's Elastic Compute Cloud (EC2) contains tens of thousands of virtual machines, and supports Hadoop with minimal effort.

2.2.2 The Hadoop MapReduce implementation

The Hadoop project provides an open-source implementation of Google's MapReduce paradigm through the Hadoop MapReduce framework [16, 75]. The framework was designed following Google's architectural model and has become the reference MapReduce implementation. The architecture is tailored in a master-slave manner, consisting in a single master *jobtracker* and multiple slave *tasktrackers*.

Architecture

Figure 2.2 shows the main entities of the Hadoop system and the flow of interactions triggered by a job-submission event. The Hadoop client submits a job to the jobtracker for execution. The jobtracker splits the job into a set of *tasks* that are either "map" or "reduce" computations. The input data is also split into fixed-size *chunks* that are stored in the distributed file system used as backend storage (by default, HDFS). Each "map" task is associated with a data chunk. This step involves inquiring the file system's namespace, so that the jobtracker becomes aware of the location of each input chunk. This information is required by the next step of the execution, which is the task scheduling.

The jobtracker's main role is to act as the task *scheduler* of the system, by assigning work to the tasktrackers. Each tasktracker disposes of a number of available *slots* for running

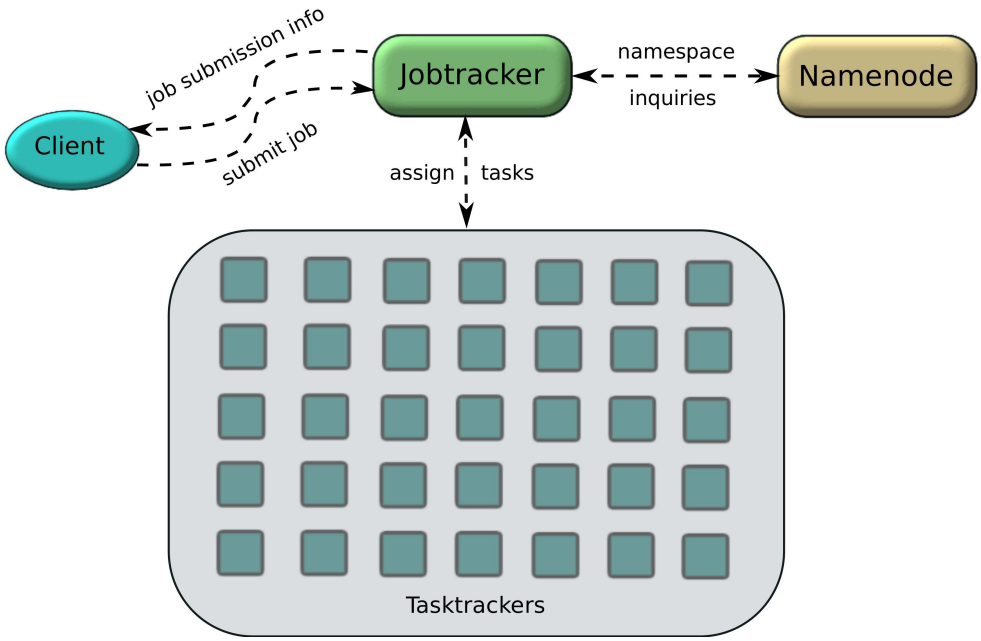


Figure 2.2: The Hadoop MapReduce framework.

tasks. Every active map or reduce task takes up one slot, thus a tasktracker usually executes several tasks simultaneously. When dispatching "map" tasks to tasktrackers, the jobtracker strives at keeping the computation as close to the data as possible. This technique is enabled by the data-layout information previously acquired by the jobtracker. If the work cannot be hosted on the actual node where the data resides, priority is given to nodes closer to the data (belonging to the same network rack). The jobtracker schedules first "map" tasks, as the reducers must wait for the "map" execution to generate the intermediate data.

Apart from data splitting and scheduling responsibilities, the jobtracker is also in charge of monitoring tasks and dealing with failures.

Job scheduling

The default scheduling policy implemented by Hadoop's jobtracker treats tasks on a first-in-first-out (FIFO) basis. However, this approach has obvious limitations when considering environment issues, such as heterogeneity, or special classes of applications. As a result, the Hadoop framework was modified to allow users to plug-in a scheduler that suits their execution environment and purposes. To take some examples, two of the major Hadoop production users, Yahoo! and Facebook, developed cluster-oriented schedulers to address their specific requirements.

The *FIFO Scheduler* keeps a queue of jobs that are processed each at a time. This very simplistic scheduling algorithm was based on the assumption that the execution of each job involves the whole cluster, so that jobs were processed sequentially. With this approach, it is however possible to specify a job priority that is taken into account by the jobtracker. When selecting the next job to execute, the jobtracker chooses the one with the highest priority in the queue.

The *Fair Scheduler* developed at Facebook allows users to assign jobs to pools, and allocates "map" and "reduce" slots to each pool. Yahoo!'s *Capacity Scheduler* addresses a scenario that involves a large number of users, and targets a fair resource allocation to users. The Capacity Scheduler keeps queues with configurable number of slots to which jobs are added on account of the submitting user. Overall, this type of scheduling enforces sharing of cluster capacity among users, as opposed to sharing resources among jobs, as was the case for the Fair Scheduler.

Failures

Hadoop was designed to tolerate failures in a user-transparent manner. One of the most-promoted benefits of using Hadoop is its ability to automatically deal with various types of failures caused by different sources. Hadoop identifies and manages several types of failures.

Task failure. If a tasktracker reports an error code when running a certain task, the jobtracker reschedules the execution of the failed task. The jobtracker attempts to dispatch the task to a different tasktracker than the one on which the task had previously failed. In addition, the jobtracker keeps track of the number of times a task had failed. If this number reaches a configurable threshold, the task will not be re-executed further.

Tasktracker failure. The communication between the jobtracker and the tasktrackers is implemented through a heartbeat mechanism: the tasktracker sends a message to the jobtracker every configurable amount of time. A crashed tasktracker can be identified when the jobtracker notices that it stopped sending heartbeat messages in a given time interval. This tasktracker is removed from the jobtracker's pool of workers. Furthermore, the jobtracker reschedules the "map" tasks that were run on the failed tasktracker, as the intermediate data they produced is no longer accessible.

Jobtracker failure. The jobtracker represents the single point of failure in the system, thus any failures at this level is critical and leads to a failed job.

2.3 Summary

This chapter presents the target application domain of our work, i.e., data-intensive applications, more specifically, MapReduce applications. In a first phase, we investigate the properties of data-intensive applications and provide a general presentation of the programming model proposed for designing DISC applications. We then focus on the MapReduce programming paradigm and its implementations. Introduced by Google, the MapReduce abstraction has revolutionized the data-intensive community and has rapidly spread to various research and production areas. An open-source implementation of Google's abstraction was provided by Yahoo! through the Hadoop project. This framework is considered the reference MapReduce implementation and is currently widely-used for various purposes and on several infrastructures.

Chapter 3

Infrastructures

Contents

In a previous chapter we presented a survey on data-intensive applications, mainly focusing on MapReduce applications. The second part of the context of this work concerns large-scale infrastructures. In this chapter we detail two environments that are most commonly used for executing data-intensive applications. The first platform that we focus on is the Grid, a well-established approach to distributed computing. In a second part of the chapter, we present Cloud computing, a recently-emerged model with a continuously growing popularity.

3.1 Grids

The main concept that lies behind the Grid was foreseen many years ago: in 1969 Len Kleinrock suggested that "We will probably see the spread of 'computer utilities', which, like present electric and telephone utilities, will service individual homes and offices across the country." A couple of decades later, in 2002, Ian Foster proposed a definition of the Grid concept, that became the "reference" definition in the domain. In [34], the author defines the Grid as "a system that coordinates resources which are not subject to centralized control, using standard, open, general-purpose protocols and interfaces to deliver nontrivial qualities of service". Several important features of the Grid emerge from this definition.

- A Grid integrates and coordinates resources and users that are located within different control domains.

- A Grid relies on multipurpose protocols and interfaces that address fundamental issues such as authentication, authorization, resource discovery, and resource access.

- A Grid allows its resources to be used in a coordinated manner with the purpose of providing various qualities of service (for instance, response time, throughput, availability, and security), and/or co-allocation of multiple resource types in order to fulfill complex user demands. Thus, the utility of the combined system is significantly greater than the sum of its components.

The end of the past century has witnessed a growth of the long-distance networks, allowing to interconnect computing units located in different institutions. The performance of these networks has become developed enough with regard to the latency and bandwidth aspects, to consider the execution of distributed applications on geographically remote computers. The idea of taking advantage of the computing power of several centers was first experimented in 1995 in United States, within the project *Information Wide Area Year* [31, 36].

Emergence of computational Grids

Aggregating computers is not a new idea: ever since the 80's, universities and enterprises have made use of computational clusters, strongly sustained by the increasing use of personal computers. These machines were interconnected by high-performance networks and put together in the same room, with the goal of building distributed super-computers. This approach has brought an alternative solution to centralized supercomputers, that are extremely expensive to build. The main particularity of computer clusters is their homogeneity, as they are composed of identical machines. Clusters aggregate a few hundreds of machines and take advantage of the interconnection link between universities and institutions, leading to the emergence of *cluster federations*.

The concept of *grid* appeared in 1998 in the book [37], edited by Foster and Kesselman. The term of "grid" was selected after an analogy between this type of computational infrastructure and the distributed electricity network, called the *Power Grid*. The correspondence between the two infrastructures comes from the purpose of rendering the usage of computational Grids as simple as the one of electricity networks: the machines are plugged into sockets, without knowing the sources of energy or the techniques of producing it. In the context of computational Grids, a user must be able to submit an application without having to manage its execution or even knowing about the resources involved in the computation.

Defining computational grids is an open subject, even nowadays, when it comes to deciding whether a platform can be viewed as a grid or not. Nonetheless, a prolific literature [38, 34] has been dedicated to profiling computational Grids. In order to provide a definition of the Grid, we first introduce two fundamental concepts.

Computational resource. A computational resource is a hardware or software element that allows to automatically generate, manage, store or exchange computational data.

Grid site. A grid site is a set of computational resources geographically placed in a same institute, university, research center, or a given user, that constitute a domain of an autonomic, uniform and coordinated administration.

Considering these two concepts as a starting point, we can define a computational Grid as a set of Grid sites. The users of a computational Grid are referred to as *virtual organizations (VOs)*. A virtual organization is defined by a set of users, represented either by an individual or by an enterprise, having the common interest of sharing computational resources. Virtual organizations are dynamic, as they are created and destroyed depending on the users needs.

A Grid classification

There exist several types of Grids. The classification of Grids is defined according to their goals and their structure. A Grid that has the *multi-user* property allows several users to access it and employ its resources. A *multi-application* Grid allows the submission of applications of different types.

Data Grids are specialized in the storage of large data. This type of Grids must guarantee the persistence and the availability of data, while providing efficient data access. Data Grids are multi-user, but mono-application. Typical applications for this Grid category include special purpose data mining that correlates information from multiple different data sources. Data Grids [45, 24] strive at efficiently supporting data management, access and organization while accommodating large scales.

Service Grids federate resources to provide for services that can not to delivered by any single machine. Sub-categories of service Grids include *on-demand*, *collaborative* and *multimedia* Grids.

Computational Grids are primarily used for providing support for computing-intensive applications. In this context, a crucial role is played by the performances of the Grid resources and of the interconnecting networks. Computational Grids are multi-user and multi-application. A further sub-classification of computational Grids leads to two categories: *distributed supercomputing* and *high- throughput* Grids. The first type of Grids is dedicated to a class of applications that raise substantial challenges. A distributed supercomputing Grid executes the application in parallel on multiple machines to reduce the total completion time. The high-throughput category targets applications consisting of a batch of jobs. High-throughput Grids aim at increasing the completion rate of the job stream.

A particular type of Grid systems refers to *desktop Grids*, systems composed of desktop computers connected to the Internet. This type of grids uses desktop computer instruction cycles that would otherwise be wasted at night, during lunch, or even in the scattered seconds throughout the day when the computer is waiting for user input or slow devices. Projects such as SETI@Home [18] show the benefits of using millions of machines that volunteer to perform a global computation. Since nodes are likely to go "offline" from time to time, as their owners use their resources for their primary purpose, this model must be designed to handle such challenges.

Middleware for Grids

The creation of a Grid requires a toolkit that supplies the core components and offers the basic interfaces for bringing together all the required infrastructures. A lot of effort was invested in developing Grid middleware across various groups from different communities. We further give examples of such Grid toolkits.

Globus. The Globus toolkit [35] is recognized as the standard Grid computing middleware, intensively used both in business and academia. Globus is an open-source software project, developed by the Globus Alliance. It integrates the basic building blocks required for the construction of a Grid. Globus also provides higher-level services including software for security, information infrastructure, resource and data management, communication and fault tolerance. This toolkit relies on an object-oriented approach and offers the users a large collection of services satisfying a wide range of requirements.

UNICORE. *UNiform Interface to COmputing Resources* [67] represents a Grid middleware system currently used in several supercomputer centers worldwide. Developed under the initiative of the German Ministry for Education and Research, UNICORE has become a solid support in many european research projects, such as EUROGRID, GRIP, VIOLA, etc. The UNICORE toolkit is an open-source technology with a three-layered structure: user, server and target system tier. The server level provides several services that are accessed by the user tier through the UNICORE Graphical User Interface (GUI). This interface allows a seamless and secure access to the server's services. UNICORE implements services based on a job model concept, called the *Abstract Job Objects* (AJO). Such an object contains descriptions of computational and data related tasks, resource information and workflow specifications. The UNICORE Client supplies the functionalities to create and monitor jobs that are launched on the UNICORE sites. The target system tier provides the interface with the underlying local resource management system.

gLite. The *Lightweight* middleware for grid computing [55] was developed as part of the EGEE Project with the purpose of providing a framework for designing grid applications. The gLite infrastructure is currently used in different scientific communities, eg., High Energy Physics, Earthscience, Fusion and Archeology. A notable gLite user is CERN that employs the middleware for many of its scientific projects. The core services of gLite include security, application monitoring, data and workload management. A user has access to the gLite services through a *User Interface* (UI). Apart from ensuring a secure and authenticated access to gLite, this interface provides a large set of Grid operations such as: list all the resources suitable to execute a job, submit jobs for execution, cancel jobs, retrieve the information logged about the job, retrieve the outcome of jobs etc.

Grid'5000 The Grid'5000 [47] project is a widely-distributed infrastructure devoted to providing an experimental platform for the research community. Several features distinguish Grid'5000 as a reliable, efficient, large-scale experimental tool.

Experiment diversity. Grid'5000 provides the users with a diversified set of software technologies at various levels: Grid middleware, application runtime, networking proto-

cols, operating systems mechanisms, etc. These services are supplied in such a way as to guarantee scalability, fault tolerance, performance and security.

Deep reconfiguration. In order to support experimental diversity while allowing users to tune the environment to suit their needs, Grid'5000 provide a deep reconfiguration mechanism that enables users to deploy, install, boot and run their customized software.

Security. Grid'5000 employs a two-level security scheme that specifies strong authentication, authorization checks and limited Internet connectivity.

A large homogeneous subset. Based on the remark that homogeneity enables a more facile and accurate performance evaluation, 2/3 of the total set of machines incorporated by Grid'5000 are homogeneous.

Control and measurement tools. Grid'5000 ensures experiment reproducibility and monitoring through a set of dedicated tools that enable users to run the experiments on a specific set of nodes, synchronize the execution on different machines, collect monitoring information regarding network traffic, machine load, disk usage, etc.

These features are built in the design of the Grid'5000 platform. Spread over 10 geographical sites located through on French territory and 1 in Luxembourg, the Grid'5000 testbed includes more than 20 clusters and 7000 CPU cores. We detail a set of tools provided by Grid'5000 that we also employed as heavy users of the platform.

OAR [10] is a batch scheduler that allows Grid'5000 users to make fine-grain reservations, ranging from one processor core to Grid-level reservations spanning over several sites and hundreds of nodes.

Kadeploy [8] enables users to deploy a customized operating system image on the Grid'5000 infrastructure, with administrator rights allowing users to install specific software for their experiments.

The Grid'5000 API is a set of well-defined interfaces that enable secure and scalable access to resources in Grid'5000 from any machine through standard HTTP operations.

Taktuk [25] is a tool designed for efficiently managing parallel remote executions on large-scale, heterogeneous infrastructures.

3.2 Clouds

Cloud computing is an emerging paradigm for enabling *on-demand access* to shared computing resources that can be rapidly provisioned and released. A *Cloud* is defined as a pool of easily usable and accessible resources such as storage, network, applications, services, etc. The Cloud model promotes resource availability and dynamic reconfiguration for achieving optimal resource utilization. Virtualized physical resources, virtualized infrastructure, as well as virtualized middleware platforms and business applications are being provided and consumed as services in the Cloud. The resources are exploited through a *pay-per-use* model

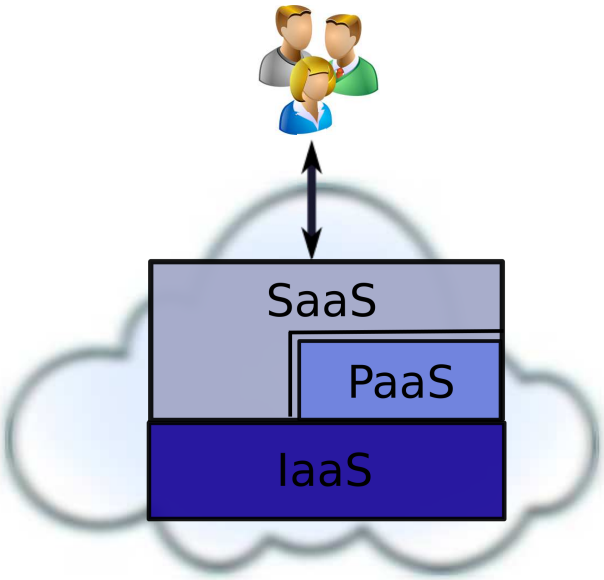


Figure 3.1: Cloud Service models.

and in compliance with customized *service-level agreements (SLAs)* specified by a contract between the cloud provider and the client. Several works in the literature focus on proposing a Cloud definition [54, 72, 20, 21, 41, 43].

Cloud-computing borrows core principles from the Grid, both computing paradigms sharing similar visions. Much the same as Grids, Clouds federate resources with the purpose of providing users with a tool they can instrument according to their needs. Whereas Grids facilitate the fair sharing of resources across organizations, Clouds provide the resources on demand, giving the impression of a single dedicated resource. Furthermore, in Clouds, the concept of *sharing* resources is replaced by the concept of *isolation* of the environment, achieved through *virtualization*. Grids and Clouds share common features such as resource heterogeneity, scalability, reconfigurability, etc.

The distinctions between the two infrastructures mainly arise from the features that the Grid failed to provide or poorly supported and that Cloud computing managed to supply and enhance. Virtualization is the key technology of Clouds. This mechanism enables several essential features such as on-demand resource provisioning and security through virtualization-based isolation. The Cloud is considered to be *user-friendly*, by providing a user-transparent deployment model, a flexible architecture and a software-independent tool for designing applications.

3.2.1 Cloud taxonomy

We provide a classification of Clouds that takes into account the type of services the Cloud provides and the scale at which the underlying infrastructure is employed.

Service models

A way to categorize Clouds is by characterizing them from the point of view of the services they provide. An architectural categorization of Cloud technologies has been proposed [54], as a stack of service types (Figure 3.1).

Infrastructure as a Service (IaaS). An IaaS Cloud enables on-demand provision of fundamental computational resources in the form of virtualized resources. Resources include basic computer networking, commodity data storage, and hosting virtualized operating systems. Users of Iaas Clouds typically deploy customized virtual machine images, run arbitrary software, and then explicitly save the data and the virtual machine image to remote storage options. The consumer does not manage nor control the underlying Cloud infrastructure, but he/she has full control over operating systems, storage, deployed applications, and possibly limited control in selecting networking components. Examples of IaaS Cloud providers include: Amazon Elastic Compute Cloud (EC2) [2], Eucalytus [63], Nimbus [9], OpenNebula [59].

Platform as a Service (PaaS). At the level of the platform as a service, the Cloud supplies programming and execution environments as a software platform. Customers are provided with a ready-to-use platform including tools and programming languages that enable users to design and execute their applications. The consumer does not manage nor control the underlying Cloud infrastructure such as storage or network, but he/she has full control over the deployed applications and hosting environment configurations. As examples of PaaS Clouds, we mention Google's App Engine [68] and Microsoft's Azure [17].

Software as a Service (Saas). By providing software as a service, SaaS Clouds enable consumers to directly use applications running on a Cloud infrastructure. The application developers can either use the PaaS layer to develop and run their applications or directly use the IaaS Cloud. The applications can be accessed through a simple client interface such as a web browser. The consumer has no control over the underlying infrastructure nor over the application capabilities. However, the client can supply configuration settings to adjust the application to fit his/her requirements. Some examples of applications in this layer are Google Docs, Microsoft Office Live, etc.

Deployment models

Private Clouds. In the case of private Clouds, the Cloud infrastructure is owned by a single organization and its usage is targeted solely to that organization. Private Clouds are basically an enhanced data-center model, that enables a flexible and efficient management of local infrastructure. The advantages of private Clouds include an internal, centralized resource management, architecture-aware virtualization optimizations, dynamic resizing and partitioning of the infrastructure, etc. Probably the greatest advantage of private Clouds over other types of Clouds, is the fact that organizations can handle security and control themselves, instead of commissioning service providers against cost.

Community Clouds. This type of Clouds is dedicated to a specific community that incorporates several organizations with shared interests. The Cloud infrastructure is shared by the organizations. Examples of domain-specific Clouds include *scientific Clouds* [13] and *HPC Clouds*. The first example refers to Clouds designed to provide experimental means to scientific and educational projects. HPC Clouds offer adequate support to the HPC community.

Public Clouds. The services offered against cost by commercial organizations to the general public are considered public Clouds. The infrastructure of a public Cloud is owned by a single, usually large company that provides customers with a simple interface for handling virtualized resources.

Hybrid Clouds. A hybrid cloud is defined as a combination of private Clouds requiring additional services that can be acquired from either of the 2 other Cloud deployment models. The combined infrastructures are bound together by standardized technologies that enable data and application portability. Usually, hybrid Clouds are created as extensions of private Clouds, for the sake of scalability.

3.2.2 Cloud examples

Amazon Elastic Compute Cloud (EC2)

Amazon's EC2 Infrastructure as a Service (IaaS) Cloud is the most widely-used and feature-rich commercial Cloud. Amazon defines EC2 as a web service that provides resizable compute capacity in the Cloud. Amazon EC2 allows users to rent compute or storage resources, in order to run their own applications. Typically, users first choose the type of virtual machine (VM) that suits their needs (application requirements, budget, etc.) and then boot the VM on multiple Amazon resources, thus creating what is referred to as *instances* of that VM. EC2 provides flexibility in terms of resource configuration as well as elasticity, by allowing users to dynamically adapt the number of VM instances.

Amazon also delivers a storage service where users can keep their data. The storage system introduced by Amazon, S3 [4], features a simple access interface that has become the IaaS standard for data transfers in and out of the Cloud.

EC2 image types are grouped together into a collection of pre-configured virtual machine images, referred to as *Amazon Machine Images (AMIs)*. Default AMIs can be deployed and customized according to users needs. To keep all modified configurations and settings, users can save AMIs in S3.

The Nimbus IaaS Cloud toolkit

The Nimbus project [9] is an open-source toolkit that provides an implementation of an Infrastructure-as-a-Service Cloud platform. Nimbus consists in an extensible IaaS implementation, that exposes to the users an EC2-like interface, as well as other features that enable the ease-of-use: customizable environments, cluster deployment and configuration, interfaces to other IaaS Clouds. The Nimbus Cloud was started as a project targeted mainly towards the scientific community, with the purpose of building an experimental testbed tailored for scientific needs.

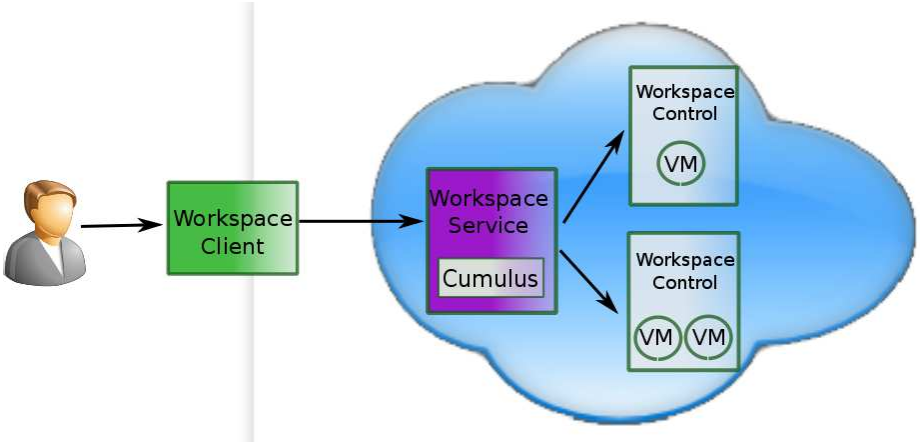


Figure 3.2: The architecture of a Nimbus Cloud.

The architecture of a Nimbus Cloud, as shown on Figure 3.2 is based on four modular components:

The Cloud Client provides the users with the commands for launching and managing virtual resources.

The Workspace service is a standalone site VM manager that represents also the Cloud entry point. It handles client requests for virtualized resources and manages VM deployment.

The Workspace Control is an agent running on each node, in charge of VM deployment, management and configuration.

Cumulus is an open source implementation of the S3 REST [33] API that plays the role of the front-end to the Nimbus storage repository for VM images. It allows users to deploy their own virtual machine images on the Cloud infrastructure, by uploading them to the storage repository through standardized interfaces.

Eucalyptus

The Eucalyptus system [63] is an open-source Cloud that provides on-demand computing instances and shares the same APIs as Amazon's EC2 cloud. Eucalyptus stands for *Elastic Utility Computing Architecture Linking Your Programs To Useful Systems*. The Eucalyptus Cloud was designed as a highly-modular framework in order to enable extensibility with minimal effort. Eucalyptus is advertised as being among the few providers of *a virtual network overlay* that brings benefits at two levels: it isolates network traffic of different users, and it unifies clusters such that their resources appear to the user as belonging to the same Local Area Network.

Four high-level components make up the architecture of Eucalyptus.

The Node Controller (NC) runs on each node with the purpose of hosting virtual machine instances, taking care of issues such as controlling and monitoring the VM execution.

The Cluster Controller (CC) is launched on the cluster front-end and manages several tasks: it schedules requests for launching VMs on specific NCs, it controls the virtual network overlay and it monitors the state of each NC, through status reports.

The Storage Controller (Walrus) provides a data-storage service for storing and accessing virtual machine images as well as application-level data. Walrus exposes an interface compatible with the S3 standard.

The Cloud Controller (CLC) acts as the Cloud entry-point by exposing and managing the virtualized resources. The CLC offers a series of web services oriented towards resources, data and interfaces.

OpenNebula

OpenNebula [59] is an open-source project that provides a toolkit for building a Cloud infrastructure. The OpenNebula tools enable the federation of heterogeneous, distributed infrastructures, in a flexible and user-friendly manner. The system provides an administration interface for the centralized monitoring and management of the infrastructure. An essential feature is that it enables users to plug-in workload and resource-aware allocation policies such as load balancing, affinity-aware, capacity reservation, etc. The toolkit also supports straightforward integration of management tools employed by the underlying, local infrastructure. OpenNebula's main advantage is that it supports a combination of deployment types: private, public, hybrid and community Clouds.

3.3 Summary

This chapter presented a survey of Grid and Cloud infrastructures, including features, classifications and popular examples. A good understanding of the targeted infrastructures is required for the design and evaluation of the contribution we will present in the following chapters. In this chapter we described in detail 2 platforms that we consider to be a pertinent environment for validating our work: the Grid'5000 testbed and the Nimbus Cloud.

Chapter 4

BlobSeer

Contents

In the previous chapters we described the application domain and the infrastructures targeted by our work. In this chapter, we focus on a framework that was designed as a solution to the challenge of efficiently storing data generated by data-intensive applications running at large scales.

BlobSeer [62, 61] is a data storage service specifically designed to deal with the needs of data-intensive applications: *scalable aggregation of storage space* from the participating nodes with minimal overhead; support to store *huge data objects*; *efficient fine-grain access* to data subsets; and ability to sustain a *high throughput under heavy access concurrency*. In order to achieve these goals, it relies on *data striping*, *distributed metadata management* and *versioning-based concurrency control*. These techniques avoid data-access synchronization and enable the distribution of the I/O workload at large scale both for data and metadata, which is crucial in achieving a high aggregated throughput under concurrency.

4.1 Design overview

In order to provide highly-scalable storage to applications that exhibit massively-parallel data access, BlobSeer's design relies on a set of principles and techniques, detailed below.

BLOBs. BlobSeer uses the concept of *BLOBs* (binary large objects) as an abstraction for data. A BLOB is a large, flat sequence of bytes - intrinsically, unstructured data. The size of a BLOB typically reaches the order of TBs. Each BLOB is assigned a globally unique identifier from the BLOB namespace. BlobSeer is targeted towards applications that process large datasets in a fine-grain manner, i.e., data is accessed in blocks of a few KBs. Managing data as BLOBs brings an essential benefit when it comes to *scalability*: manipulating huge BLOBs comprising small KBs-sized blocks is much more efficient than maintaining the small blocks themselves.

Data striping. In BlobSeer, each BLOB is split into even-sized blocks, called *chunks*, which are spread across the storage nodes. The chunk is the data-management unit, and its size can be configured for each BLOB.

Distributed metadata. The metadata incurred by the data striping mechanism is organized in BlobSeer, as a *distributed segment tree* [79]. Typically, applications access data by specifying an offset and a size, which define a *range* belonging to a certain BLOB. A segment tree is a binary tree in which each node is associated to a range of the BLOB, delimited by offset and size. Thus, a node *covers* the range (offset, size). The root covers the whole BLOB. For each node that is not a leaf, the left child covers the first half of the range, and the right child covers the second half. Each leaf covers a single chunk of the BLOB. Such a tree is associated to each BLOB. BlobSeer's metadata keeps the mapping of a given range to the physical nodes where the corresponding blocks are located. The metadata are distributed across a set of nodes, for scalability and data availability purposes.

Versioning-based concurrency control. BlobSeer employs the versioning mechanism as a main design principle. In BlobSeer, data and metadata are never modified. Instead, BLOB-update operations generate new *versions* of the same BLOB. The benefits of using versioning are twofold. First, by exposing the versions of each BLOB to the application, BlobSeer enables the user to efficiently design complex workflows, by parallelizing various stages of the application, so as to simultaneously process different versions of the same BLOB. Another advantage of supporting versioning at the application level is the straightforward implementation of *rollback* operations: the application can simply switch back to an older BLOB version.
Second, BlobSeer internally uses versioning to handle *concurrency*. When parallel accesses to the same BLOB are performed, BlobSeer manages to decouple the accesses and execute them with a minimal amount of synchronization. A read operation can thus access data and metadata in a fully parallel manner with respect to writers.

4.2 Architecture

BlobSeer's architecture (Figure 4.1) includes several entities whose implementation follows the design principles described in the previous section.

Data providers. Their role is to provide physical storage to applications. The providers store the chunks that contain application-level data. In a typical BlobSeer environment, data providers are started one per node.

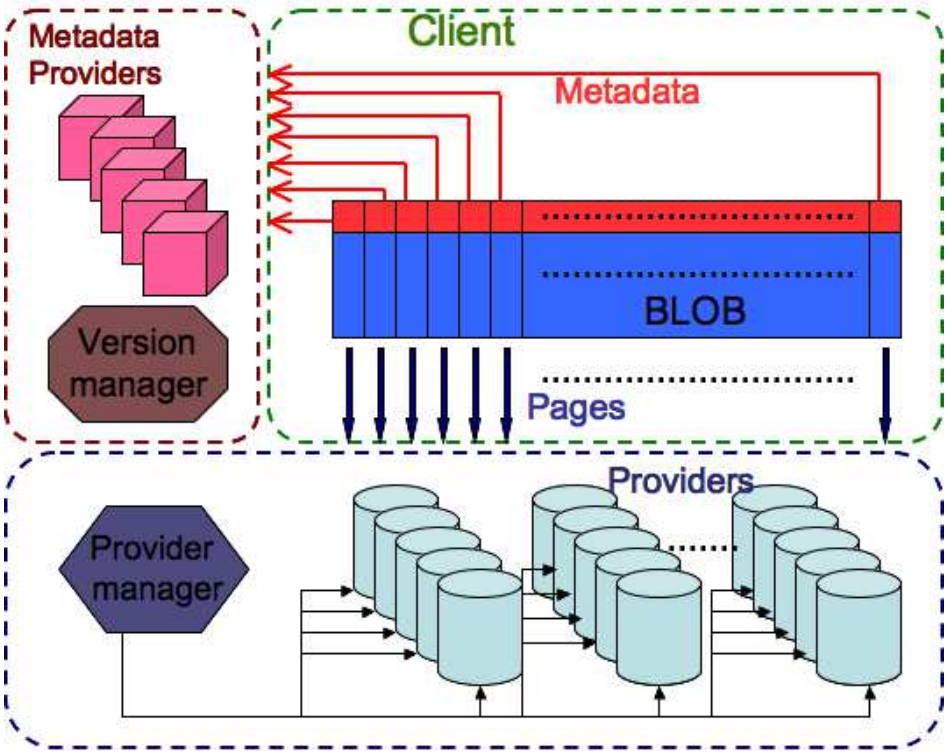


Figure 4.1: BlobSeer's architecture.

The provider manager. This centralized entity is in charge of allocating newly-generated chunks to data providers. It also maintains the information about the total storage space available on all data providers. The provider manager assigns chunks to providers in a round-robin manner, which leads to a *uniform data distribution* in terms of number of chunks stored by each provider. This strategy aims at achieving load balancing with respect to data providers.

Metadata providers. All the metadata regarding chunk location for each BLOB version are stored by entities called metadata providers. Managing metadata in a distributed fashion is a mechanism that enables the BlobSeer system to perform fast access to metadata.

The version manager. BLOB version numbers are assigned by a centralized version manager, which is also responsible for ensuring consistency when concurrent writes to the same BLOB are issued. It is typically hosted on a dedicated node.

Clients. A BlobSeer client is employed by the user to perform basic operations on BLOBs, such as create, read, write and append.

4.3 Access interface to BLOBs

A client of BlobSeer manipulates BLOBs by using a simple interface that allows to: create a new empty BLOB; append data to an existing BLOB; read/write a subsequence of bytes

specified by an offset and a size from/to an existing BLOB. Each BLOB is identified by a unique id in the system.

```
1 id ← CREATE()
```

By invoking the CREATE primitive, a new BLOB is created in the system: its size is set to 0 and its assigned identifier id is returned to the user for further access operations.

```
1 WRITE(id, buffer, offset, size)
2 APPEND(id, buffer, size)
```

Each write or append operation generates a new version of the BLOB, version that is assigned a number. The version numbers are incrementally generated by the version manager. Both WRITE and APPEND functions require the user to specify the BLOB id, and have as effect the copying of size bytes of data from the user-supplied buffer to the corresponding BLOB. In case of a write operation, the user must indicate the offset at which the data is to be written in the BLOB. For the APPEND function, the offset is assumed to be at the end of the BLOB.

```
1 READ(id, v, buffer, offset, size)
```

Reading data from a BLOB involves first identifying the BLOB and the required version, by supplying the id and v parameters. If the v parameter is missing, the system assumes the latest version of the BLOB is requested. The result of calling the READ function is the filling of buffer with size bytes of data representing the contents of the BLOB (id, v) starting from offset.

```
1 new_id ← CLONE(id, v)
```

The purpose of the CLONE primitive is to enable users to create a new BLOB starting from an existing one. The new BLOB is a duplicate of the BLOB identified by (id, v). This function creates a new BLOB with the same data as the specified BLOB and returns the identifier assigned by the version manager. Any further updates to the original and new BLOBs are independent from each other. By the means of CLONE, the two BLOBs can evolve in two separate directions.

BlobSeer provides additional primitives that enable users to switch between BLOB versions, find out the size of a BLOB, get the total number of BLOBs in the system, etc.

4.4 How reads and writes are performed

Zooming on reads

To read data, the client must specify the version number one wants to read from, as well as the offset and size of the range to be read. The client may also call a special primitive first, to find out the latest version available in the system at the time this primitive was invoked.

The corresponding distributed algorithm, describing the interactions between the client, the version manager, the distributed data and metadata providers are presented and discussed in detail in [62]. The main global steps can be summarized as follows.

- The client queries the version manager about the requested version of the BLOB.
- The version manager forwards the query to the metadata providers, which send to the client the metadata that corresponds to the chunks that make up the requested range.
- When the location of all these chunks is determined, the client fetches the chunks from the data providers.
- Those requests are sent asynchronously and processed in parallel by the data providers. If the requests involve parts of a chunk and not the entire chunk, only the required parts are sent to the client. If one of the requests fails, the whole read operation is considered to have failed and an error code is returned to the client.

Zooming on writes

The write operation involves the following steps.

- The client first splits the data to be written into a list of chunks that correspond to the requested range.
- Then, it contacts the provider manager, requesting a list of data providers capable of storing the chunks: one provider for each chunk.
- The chunks are then written in parallel to the providers allocated by the provider manager. If, for some reason, the writing of a chunk fails, then the whole write fails.
- Otherwise the client proceeds by contacting the version manager to announce its intent to update the BLOB.
- Subsequently, the version manager assigns a new snapshot version number to the write request. This number is used by the client to generate new metadata, weave them together with *existing metadata*, and store them on the distributed metadata providers, in order to create the illusion of a new standalone version.
- Once metadata were successfully written to the metadata providers, the client notifies the version manager of success, and returns to the user.

Concurrent write requests are dealt with at the level of the version manager. The version manager needs to keep track of all writers concurrently active, and delay completing a new version until all writers that were assigned a lower version number reported success. The detailed algorithm for writing is provided in [62].

The append operation is identical to the write operation, except for a single difference: the offset of the range to be appended is unknown at the time the append is issued. It is eventually fixed by the version manager at the time the version number is assigned. It is set to the size of the version corresponding to the latest version number.

4.5 Summary

This chapter introduces the *BlobSeer* system, an answer to the requirements yielded by the management of data-intensive applications on large-scale infrastructures. One such requirement is efficiently dealing with massive data in large-scale distributed systems while maintaining a high throughput for heavily concurrent, fine-grain data accesses. To address these requirements, BlobSeer relies on a set of principles such as data striping, versioning, distributed metadata management, and others. The core of BlobSeer's design is using versioning as a mechanism for concurrency control. This approach enables clients to perform read/write accesses to data in a concurrent manner, with minimal synchronization.

Thanks to its features, BlobSeer is a suitable candidate to provide storage for MapReduce applications. In the next part of this manuscript, we dedicate several chapters to exploring the benefits BlobSeer can bring in the MapReduce context.

Part II

Contribution

Chapter 5

Designing a Concurrency-Optimized File System for MapReduce Frameworks

Contents

THIS chapter presents our contribution at the level of the storage layer for MapReduce applications. We first investigate the specific features of MapReduce applications that have an impact on the storage backend. In this chapter we also briefly discuss file systems belonging to various communities, that are successfully used as storage for frameworks executing MapReduce jobs.

With this survey as a starting point, we designed and implemented a *concurrency-optimized file system for MapReduce frameworks*. The BlobSeer system introduced in Chapter 4 allowed us to efficiently deal with the challenges that arise along with the MapReduce paradigm. We built the *BlobSeer File System (BSFS)* with the goal of providing high throughput under heavy concurrency to MapReduce applications.

5.1 Dedicated file systems for MapReduce applications

5.1.1 Requirements for the storage layer

There are a number of challenges to be addressed by the storage layer of MapReduce applications. *Fine-grained access* to huge files is required, since MapReduce applications deal with a very large number of small records of data. Completing the application in a reasonable amount of time requires the storage layer to sustain *high throughput*, while a large number of clients access the same file concurrently.

One of the optimization techniques employed by the MapReduce framework is to ship the computation to nodes that store the input data; the goal is to minimize data transfers between nodes. For this reason, the storage layer must be able to provide the information about the location of the data. This *data distribution exposure* helps the framework to schedule tasks as close as possible to the data they need.

5.1.2 File systems for MapReduce applications

Data intensive scalable computing. Several distributed file systems emerged from the Internet services community, to provide the right abstraction for the MapReduce paradigm. These file systems were most often built from scratch, tailored to offer high performance in specific usage scenarios and for specific application workloads.

To meet its storage needs, Google introduced the **Google File System (GFS)** [40], a distributed file system that supports large-scale data processing on commodity hardware. In GFS, a file is split into 64 MB chunks that are placed on storage nodes, called *chunkservers*. A centralized *master server* is responsible for keeping the file metadata and the chunk locations. To access a file, a client first contacts the master to get the chunkservers that store the required chunks; all file I/O operations are then performed through a direct interaction between the client and the chunkservers. GFS is optimized for access patterns involving huge files that are mostly appended to, and then read from. Fault tolerance is ensured through chunk replication and data checksumming.

Google introduced MapReduce as a solution to the need to process datasets up to multiple terabytes in size on a daily basis. Although MapReduce is not restricted to work exclusively with GFS as storage, there are however some advantages when using a DFS with a design similar to GFS's. Firstly, the MapReduce framework takes advantage of data striping and chunk-level replication to efficiently schedule mappers. Secondly, GFS's fault tolerant techniques improve the execution of MapReduce applications when a failure occurs. For example, since the reducers write their output to GFS, data availability is ensured; in case of a reducer failure, the data generated up to the point of failure is not lost and the reduce computation can resume on another node. Thirdly, using the same nodes for both processing and storing data (thus placing computation close to data) brings significant performance benefits in terms of high throughput.

High performance computing. Distributed file systems belonging to the HPC community are also good candidates for supporting data-intensive workloads. Such file systems that were adapted to fit the needs of MapReduce applications are PVFS (Parallel Virtual File System) [11] and GPFS (General Parallel File System) [70].

GPFS [70] is part of the *shared-disk file systems* class that use a pool of block-level storage, shared and distributed across all the nodes in the cluster. The shared storage can be directly accessed by clients, with no interaction with any intermediate server. The integration of GPFS with the Hadoop framework has to address two main issues. Firstly, GPFS supports a maximal block size of 16 MB, whereas Hadoop often makes use of data in 64 MB chunks; IBM solved this issue by adding a new concept, *metablocks*, that meant keeping the small block size (512 KB-2 MB), but changing the block allocation policy, so that contiguous blocks are placed on the same node. The second problem concerns the data layout the Hadoop's jobtracker must be aware of. Since GPFS exposes a POSIX interface, this aspect was solved by introducing a new GPFS function.

PVFS [11] is an *object-based file system* that separates the nodes that store the data from the ones that store the metadata (file information, and file block location). When a client wants to access a file, it must first contact the metadata server and then directly access the data on the data servers indicated by the metadata server. In order to be used as a storage back-end for Hadoop, some functionalities were added to PVFS [77], through an additional layer built on top of it: readahead buffering, data layout exposure and replication emulation.

Cloud computing. The MapReduce paradigm has also been adopted by the cloud computing community as a support to those cloud-based applications that are data-intensive. Cloud providers support MapReduce computations so as to take advantage of the huge processing and storage capabilities the cloud holds, but at the same time, to provide the user with a clean and easy to use interface. There are several options for running MapReduce applications in clouds: renting cloud resources and deploying a cluster of virtualized Hadoop instances on top of them, using the MapReduce service some clouds provide, or using MapReduce frameworks built on cloud services.

The first option consists in using cloud resources to create a Hadoop cluster. Setting up a Hadoop cluster is the typical scenario of using Hadoop, and implies the same process in a virtualized environment as it does in a non-virtualized one: assigning Hadoop roles to the cluster nodes and then launching the processes as configured by the first step.

Amazon released **Elastic MapReduce** [3], a web service that enables users to easily and cost-effectively process large amounts of data. The service consists in a hosted Hadoop framework running on Amazon's Elastic Compute Cloud (EC2) [2]. Amazon's Simple Storage Service (S3) [4] serves as storage layer for Hadoop. The S3 file system stores files as objects, using the key-value abstraction: the filename is used as the key, whereas the file content is the value. Files can be created, listed, and retrieved using either a REST-style HTTP interface or a SOAP interface. S3's design aims to provide scalability, high availability and low latency at commodity costs. There are two ways of using S3 with Elastic MapReduce. The first option is to use S3 as a replacement for HDFS, and thus, having Hadoop's mappers read the input data directly from S3, while the reducers write the output data to S3. The second possibility is to have the input data in S3, but transfer it to HDFS before performing the computation with Hadoop, and then retrieve the output files from HDFS and store them in S3. Since S3 is not optimized for MapReduce access patterns, using the first method is recommended only when the cost of moving the data from S3 to HDFS is too great (the data is too large) and for long-running applications that query the datasets periodically.

AzureMapReduce [42] is an implementation of the MapReduce programming model,

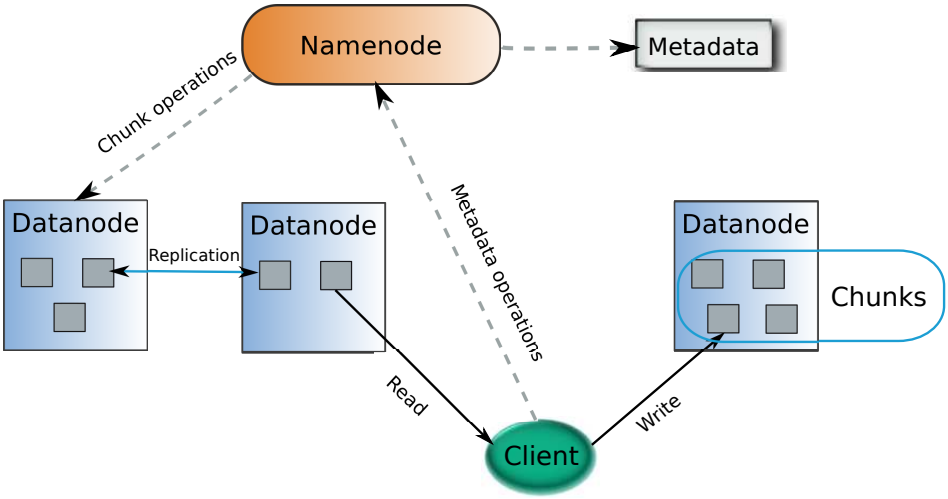


Figure 5.1: The Hadoop Distributed File System (HDFS).

based on the infrastructure services the Azure cloud [17] offers. The framework uses the Azure Blob as storage layer, which is a service that provides distributed storage where users can store and retrieve any type of data through a web services interface. The Azure platform offers two types of blobs as storage options: block blobs and page blobs. The user selects a type of blob, depending on the way the data will be accessed, that is, block blobs for streaming data access and page blobs for random read/write operations. Azure's infrastructure services are built to provide scalability, high throughput and data availability. These features are used by the AzureMapReduce runtime as mechanisms for achieving fault tolerance and efficient data processing at large scale.

5.1.3 The Hadoop Distributed File System - HDFS

The Hadoop Distributed File System (HDFS) [7] is part of the Hadoop project. Its design aims at providing storage for *huge files* with *streaming data access patterns*, while running on clusters of *commodity hardware*. Like most distributed file systems, HDFS stores files that can reach petabytes in size and that are processed in a write-once, read-many-times manner. In HDFS, files are typically generated or transferred from source and they are rarely modified over time; these datasets are usually read several times by applications that perform computations on them. HDFS is designed to work on inexpensive hardware, which implies that the system has to manage failures in a user-transparent way. HDFS uses the same design concepts as GFS: data is organized into files and directories, a file is split into fixed-size blocks that are distributed across the cluster nodes. The blocks are called *chunks* and are usually of 64 MB in size (this parameter specifying the chunk size is configurable).

Figure 5.1 shows the architecture of HDFS, designed after the master-slave model. The nodes in a HDFS cluster that store the chunks are called *datanodes*. A centralized *namenode* is responsible for keeping the file metadata and the chunk location. Figure 5.1 also shows the interactions between a client and the file system entities: when accessing a file, a client first contacts the master to obtain the list of datanodes that store the required chunks; all file I/O operations are then performed through a direct interaction between the client and

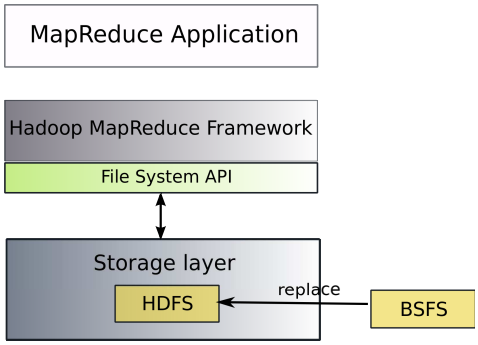


Figure 5.2: Hadoop's core: the MapReduce framework and the storage layer.

the datanodes. HDFS handles failures through chunk-level replication (3 replicas are kept by default). When distributing the replicas to the datanodes, HDFS employs a rack-aware policy: the first copy is always written locally, the second copy is stored on a datanode in the same rack as the first replica, and the third copy is shipped to a datanode belonging to a different rack (randomly chosen). The namenode decides and maintains the list of datanodes that store the replicas of each chunk.

Like most file systems developed by the Internet services community, HDFS is optimized for specific workloads and has different semantics than the POSIX-compliant file systems. HDFS does not support concurrent writes to the same file. Moreover, once a file is created, written and closed, the data cannot be overwritten nor appended to. HDFS is not optimized for small I/O operations, however it uses client- side buffering to improve the throughput. Clients buffer all write operations until the data reaches the size of a chunk (64 MB). HDFS also implements readahead buffering: when HDFS receives a read request for a small block, it prefetches the entire chunk that contains the required block. Another technique HDFS uses to achieve an overall high throughput, is to expose the data layout to the Hadoop scheduler (the jobtracker). When distributing the chunks among datanodes, HDFS picks random servers to store the data, which will often lead to a layout that is not load balanced. To make up for this, the scheduler will try to place the computation as close as possible to the needed data. HDFS provides the information about the location of each chunk, and the jobtracker will use it to execute tasks on datanodes in such way as to improve load balancing across all nodes.

5.2 The BlobSeer File System - BSFS

5.2.1 Integrating BlobSeer with Hadoop

The core of the Hadoop project consists of two main components: the Hadoop MapReduce framework and the Hadoop Distributed File System. The latter of the two, HDFS, can be used outside the project, as a stand-alone DFS; furthermore, Hadoop's implementation of the MapReduce programming model can operate with a storage layer other than HDFS. This is possible since the two components are connected through a set of file system interfaces specified within the Hadoop project. Figure 5.2 shows the general layout of the main Hadoop components. The file system API contains a set of Hadoop-specific interfaces that

basically define standard file system operations (create, read, write, etc). Through this interface, during the execution of an application, the Hadoop MapReduce framework accesses the underlying DFS for creating job-specific files, storing temporary files, reading the input data for processing and finally, writing the output data it produced.

The contribution presented in this chapter is at the level of the storage layer in the Hadoop stack. As Figure 5.2 shows, we replace the default backend of Hadoop, HDFS, with a file system built on top on BlobSeer, file system that we called BSFS.

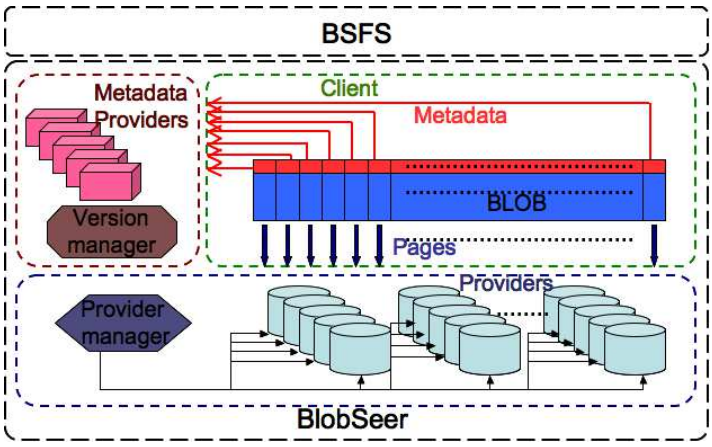


Figure 5.3: BlobSeer's architecture. The BSFS layer enables Hadoop to use BlobSeer as a storage backend through a file system interface.

In order to use BlobSeer as storage layer for the Hadoop MapReduce framework, we implemented this API on top of BlobSeer (Figure 5.3). Prior systems like the Kosmos File System (KFS) [5] and Amazon S3 [4] have used this file system API to build backend stores for Hadoop applications. The layer we added on top of BlobSeer, as well as some optimization techniques are detailed in the following.

5.2.2 The file system namespace manager

Our BlobSeer-based DFS consists in a layer added on top of the BlobSeer storage system, that also implements the file system operations specified by the Hadoop API. We designed a centralized entity that has several roles:

Managing the file system namespace : The BSFS layer keeps the file system metadata and directory structure, i.e., the information regarding file properties and the hierarchy of files and directories.

Mapping files to BLOBs : BlobSeer offers a flat storage space, in which data is kept in BLOBs uniquely identified in the system by a key. In order to use BlobSeer as a regular DFS, BSFS maps each file to a BLOB, by associating the file name to the BLOB id.

Implementing the Hadoop API : By implementing the file system interface through which Hadoop accesses its storage backend, BSFS can serve as storage for MapReduce applications run on top of Hadoop. Moreover, BSFS can act as a stand-alone DFS that can be accessed through an HDFS-like file system interface.

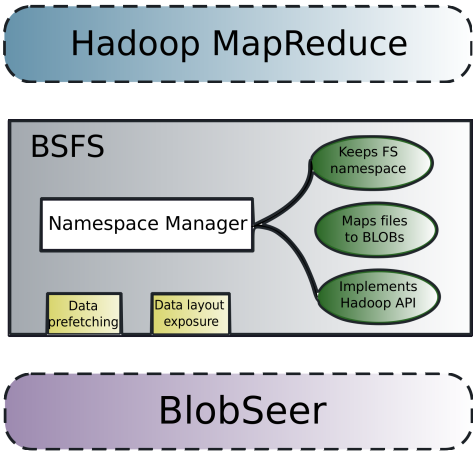


Figure 5.4: BSFS - the namespace manager.

The BSFS layer, shown in Figure 5.4, keeps the interaction between clients and the namespace manager to the minimum. Clients interact with the namespace manager only for operations concerning the file system metadata; data accesses are performed through a direct communication between the client and the BlobSeer storage nodes.

5.2.3 Data prefetching

When running MapReduce applications with Hadoop, the data stored in the backend DFS is processed in small records of 4 KB. More precisely, in the Hadoop MapReduce framework, each mapper has to execute the "map" function specified by the user, on its assigned chunk(s). However, the mapper reads each chunk sequentially in blocks of 4 KB and then applies the "map" function to this piece of data. Instead of performing small reads on the data stored in the file system, HDFS prefetches the entire chunk of 64 MB containing the requested data. The prefetching is done in an asynchronous manner, but the read requests are then served synchronously from the buffer. The same buffering mechanism is employed by HDFS when the reducers produce their outputs and write them to HDFS. Since it reduces the file system overhead, this technique, shown on Figure 5.5, becomes an important factor in achieving performance in terms of high I/O throughput.

Our BSFS layer implements this optimization technique as client-side buffering. This consists in prefetching a whole chunk when a read of 4 KB is issued, and in collecting the small records issued as final output until the data collected reaches at least the size of a chunk (64 MB by default). Hence, the actual reads and writes from/to the underlying storage layer are performed on data large enough to compensate for network traffic overhead.

5.2.4 Data layout exposure

In a regular Hadoop deployment, the physical machines in the cluster are used for both computation and data storage. The Hadoop MapReduce framework takes advantage of this deployment model to collocate computations with the data they work on. This is actually one of the goals of MapReduce frameworks, i.e., to minimize the transfer of large amounts

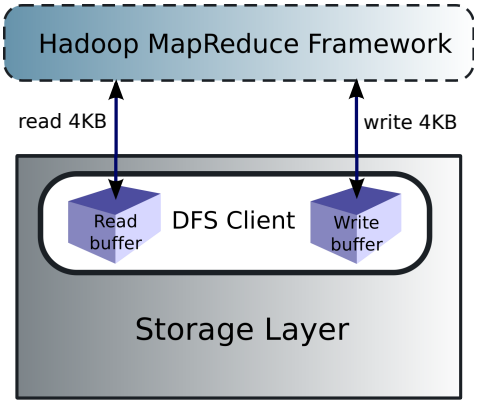


Figure 5.5: Client-side buffering. The DFS client buffers small data requests to reach a chunk size(64 MB).

of data by shipping computations instead. For Hadoop, this approach is beneficial since it comes as a mechanism for achieving overall load balancing. In Hadoop, there are two operations that determine the load of the nodes: the write operation and the scheduling mechanism. We further discuss how each of the two influences the load.

When a file is written to HDFS, the namenode employs a random chunk distribution policy which consists in randomly choosing the datanodes to store the chunks of a file. This policy does not aim at balancing the load of the nodes, furthermore, in most cases, it will lead to a non-uniform data layout. Since the namenode is aware of the mapping of chunks to datanodes, it can expose this information to the Hadoop MapReduce framework.

In particular, the jobtracker uses the chunk layout to determine how to dispatch map tasks to datanodes; each mapper is assigned a chunk to process and the jobtracker tries to run the mapper on the datanode storing the appropriate input chunk. This scheduling policy tries to compensate for the unbalanced data distribution across datanodes, and to achieve overall load balancing.

The Hadoop file-system API specifies a primitive that exposes the data layout in the underlying DFS. By calling this primitive, the jobtracker becomes aware of the way data was split into blocks and the exact location of each block. In order to provide the same functionality to Hadoop in our BSFS implementation, we extended the BlobSeer API with a new primitive. This primitive returns for a given data range in a BLOB, the list of pages comprised in that range, as well as the addresses of the providers that store the pages. This information is retrieved by the jobtracker when calling the primitive specified by the API, and is used to schedule map tasks in an efficient manner.

5.3 Summary

The work described in this chapter addresses the problem of how to efficiently provide storage for MapReduce frameworks. We particularly focus on the Hadoop MapReduce framework and its underlying storage layer, HDFS. Our contribution consists in integrating BlobSeer with Hadoop, by building a BlobSeer-based file system, BSFS, that can successfully be used as storage backend for MapReduce applications executed with Hadoop. BSFS can be

used as both a stand-alone DFS and as a replacement for HDFS in the Hadoop framework. BSFS takes advantage of BlobSeer's features to provide high I/O throughput and to sustain it under highly-concurrent access to data. The benefits of using BSFS as storage layer for MapReduce applications were validated through large-scale experiments; these tests detailed in Chapter 9 compare BSFS and HDFS in different scenarios. Apart from the significant speed-up of Hadoop's performance shown by the results we obtained, BSFS has additional functionalities that HDFS does not support: concurrent appends, concurrent writes at random offsets and versioning. In the next chapters of this manuscript, we show how some of these functionalities can be exploited as extensions and improvements at the level of the MapReduce framework. Contributors to this work include Bogdan Nicolae, Gabriel Antoniu and Luc Bougé.

Chapter 6

Optimizing Intermediate Data Management in MapReduce Computations

Contents

In this chapter, we study several aspects related to intermediate data management in MapReduce frameworks. We first define intermediate data in the MapReduce context and analyze its characteristics, with an emphasis on the issues this type of data raises. The work presented in this chapter addresses intermediate data at two levels: inside the same job, and during the execution of pipeline applications.

6.1 Intermediate data in MapReduce computations

MapReduce applications, as well as other cloud data flows, consist of multiple stages of computations that process the input data and output the result. At each stage, the computation produces *intermediate* data that is to be processed by the next computing stage. This

type of data is transferred between stages and has different characteristics from the ones of meaningful data (the input and output of an application). While the input and output data are expected to be persistent and are likely to be read multiple times (during and after the execution of the application), intermediate data is *transient* data that is usually *written once*, by one stage, and *read once*, by the next stage.

In this chapter, we consider two categories of intermediate data: the intermediate data generated inside the same MapReduce job and the intermediate data produced between successive MapReduce jobs that represent the stages of a pipeline application.

Intermediate data produced within a single MapReduce application take the form of key/value pairs generated by the map phase of the application. All intermediate values associated with the same intermediate key are grouped together and shipped to the reduce function. Section 6.2 of the chapter focuses on this type of intermediate data.

In the case of pipeline MapReduce applications, the data produced by one job represents the input to the next job in the pipeline. This data is considered to be intermediate data and in most cases, is not relevant for the user. We elaborate on the case of pipeline MapReduce applications in Section 6.3.

Motivating scenario - effect of failures

The importance of intermediate data management is best illustrated when considering failures. Actually, storing intermediate data on the local disk of the tasktrackers, impacts Hadoop's performance when failures occur. When running a MapReduce job with Hadoop, failures can have multiple causes: bugs in the user code, crashing processes and machines, etc. As far as intermediate data is concerned, failures can be fatal at two points during the job's execution: when the tasktracker is in the process of running the map function and is writing the intermediate data to disk, and when the reducers are copying the map output data to their local file system.

In both cases, a mapper-node failure makes the "map" output data unavailable. The intermediate data (partially or completely generated up to the moment the failure occurred) stored on the failed mapper's local disk is lost. Consequently, the reducers are not able to transfer the data and proceed further in the computation. The policy Hadoop uses in this situation, is to restart the execution of the failed mapper on another node. When the jobtracker becomes aware of a tasktracker failure, it simply reschedules all the "map" tasks that were run and successfully completed on that tasktracker, to be rerun if they belong to incomplete jobs. This restart mechanism is necessary as the intermediate output residing on the failed tasktracker's local file system is no longer accessible. Although this approach exempts the user from the burden of handling failures, it also generates unsatisfactory consequences. In our context, this approach implies re-generating the intermediate data, which is an overhead that results in additional runtime.

As it was targeted towards commodity hardware, MapReduce has to deal with frequent failures in a user-transparent manner. Several research studies show that failures are the norm, rather than the exception. In [27], it is reported that for a system with 10 thousands of super reliable servers (MTBF of 30 years), there is one failure per day. Also, every year, 1–5 % of disk drives crash, while servers crash at least twice with a failure rate of 2–4 %. According to Google's experiences with MapReduce, failures happen much more often on commodity

machines: an average of 5 worker deaths out of 268 machines per job in March 2006 [28], and at least one disk crash in every run of a 6-hour MapReduce job with 4,000 machines [14]. In [52], the authors study the effect of failures on the total execution time. Their results show that for a single Hadoop job, a single machine failure leads to a 50 % increase in job completion time. Also, the work in [74] reports that a single machine failure can increase the completion time of Tera-sort by 39 % and 44 % on a 72-node cluster and a 20-node cluster, respectively.

6.2 Intermediate data generated inside the same job

In order to address the issues described in section 6.1, we propose to store the intermediate data in a distributed file system (DFS) that is able to ensure *data availability* with a *minimal impact on efficiency* at the level of the framework. Distributed file systems designed for data-intensive applications provide data availability through replication, as well as high I/O throughput under heavy access concurrency. By storing intermediate data in a DFS, a mapper failure will have a far lesser impact on the job execution time, as the data produced up to that point will not be lost. The framework could schedule the failed map task to resume on another node, from where the failed tasktracker left off.

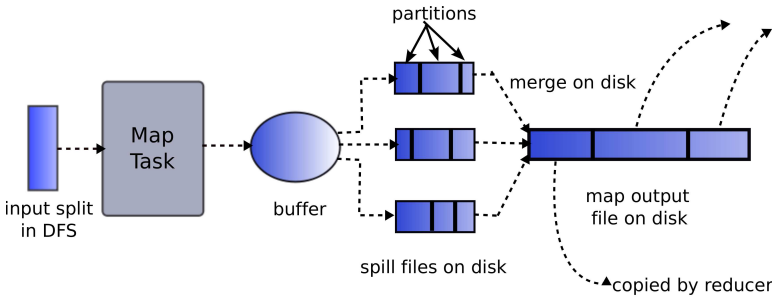
The real challenge is to select a DFS that fits the specific features of intermediate data and also optimally satisfies the availability and efficiency requirements.

We further focus on the Hadoop project - the reference implementation of the MapReduce paradigm - and we analyze how intermediate data is handled in the Hadoop framework.

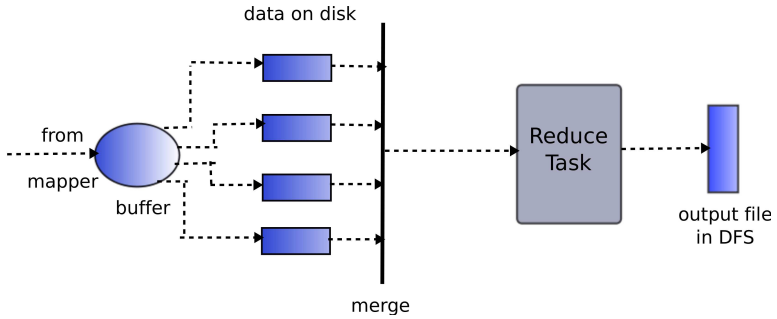
6.2.1 Intermediate data management in Hadoop

Each tasktracker executes the map user-defined function on its assigned data chunk; the output is sorted by key and then transferred to the reducers as input. The process through which the data is sorted and pipelined from the mappers to the reducers, is called the *shuffle* phase and is an essential part of the Hadoop core. On the map side, the outputs are written to the local filesystem of the tasktracker running the map function. The tasktracker notifies the jobtracker upon successful completion of a map task, so that the jobtracker becomes aware of the mapping between map outputs and the nodes that store them.

Each reducer is assigned a partition of keys to process in the reduce phase. The partition contains key/value pairs residing on the local disk of multiple tasktrackers across the cluster. Furthermore, the mappers will probably complete their execution at different times, so the reduce task starts copying the outputs it needs, as soon as they become available. The map outputs are copied to the local filesystem of the reducer via HTTP. Tasktrackers do not delete map outputs from disk as soon as the reducer has retrieved them, as the reducer may fail. Instead, they wait until the jobtracker notifies them to delete the copied files, which is done only after the job has completed. The output of the reduce phase is written to a distributed file system (by default, HDFS).



(a) Map side



(b) Reduce side

Figure 6.1: Intermediate data in the original Hadoop MapReduce framework.

6.2.2 Using BlobSeer as storage for intermediate data

Our proposal consists in using BlobSeer as storage layer for the intermediate data generated by MapReduce applications. We specifically focus on evaluating our approach within the Hadoop project. To meet this goal, we extended the Hadoop project at two levels: the MapReduce framework and the storage layer. Firstly, we modified the Hadoop MapReduce framework to store the intermediate data in a DFS. Secondly, we developed the BlobSeer File System, that allows BlobSeer to serve as storage layer for Hadoop. The latter of the two steps is described in Chapter 5, whereas the extensions at the level of the MapReduce framework are described in the following.

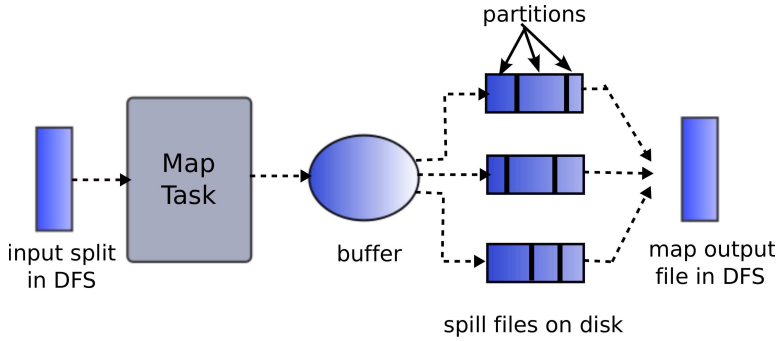
Modifying Hadoop to store intermediate data in a DFS

In the original Hadoop MapReduce framework, the output of a mapper goes through several phases from the moment it is produced and until it is written to the local disk of the tasktracker. Those steps are illustrated on Figure 6.1(a) and detailed below.

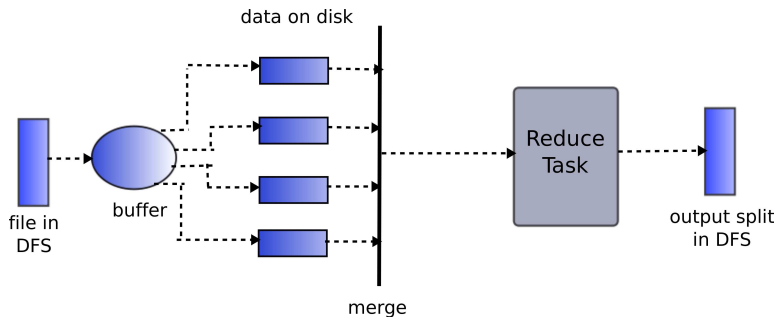
- As it is being generated, the output of each map task is written to a dedicated memory buffer of configurable size (default 100 MB).
- When the buffer reaches a certain threshold, the content is divided into *partitions* corresponding to the reducers that will process them. The data is partitioned according to a partitioning function specified by the user, or by default, a hash function on the keys.

- The data within each partition is then sorted by key.
- The partitions in the buffer are then flushed to a job-specific directory on disk; each time the buffer is flushed, a new *spill file* is created.
- Before the "map" task completes, all the spill files are merged into a single partitioned and sorted file.

When it successfully completes, each mapper sends a notification to the jobtracker. Thus, the jobtracker keeps track of the map output each tasktracker stores locally and is able to inform reducers where their assigned input data resides.



(a) Map side



(b) Reduce side

Figure 6.2: Intermediate data in the modified Hadoop MapReduce framework.

On the reduce side, the process is shown on Figure 6.1(b). It consists of several steps:

- Each partition in this map output file is copied by its assigned reducer over HTTP. A reduce task starts copying in parallel all the map output files it needs to process, as soon as they become available.
- The map output data is copied to a memory buffer which is then merged and spilled to disk whenever it reaches a threshold size.
- As the spills accumulate on disk, the tasktracker merges them into larger, sorted files. The merging process is done in stages, for efficiency reasons. We do not go into details

concerning this aspect, as it is a complex process which is not relevant to the focus of the work presented in this chapter.

- The tasktracker applies the "reduce" function on each (key, value) pair in the sorted files that are yielded by the merging phase. The output data is written to the underlying DFS, in the output directory specified by the job.

We modified the Hadoop MapReduce framework to store the intermediate data generated by the mappers and processed by the reducers, in the distributed file system (DFS) used as storage backend. Figure 6.2 describes the changes we made:

- We modified the mapper code to write the output to the DFS, after all the spill files are merged and sorted. When the mapper finishes the processing, it combines all the spills accumulated on disk, and creates a single sorted file that is written directly to the DFS.

- On the reducer side, we adjusted the code to read the output data it needs from the files in the DFS into the memory buffer. Each reducer reads the data in the partitions they have to process, from all the map output files in the DFS, as soon as they are ready.

These modifications were possible also because we kept some useful naming and communication mechanisms from the original Hadoop version.

- In the original Hadoop framework, each map task is given a unique identifier. The output of each mapper is stored on disk under a naming scheme that uses this identifier. In our modified Hadoop, we store each map output file into the DFS by preserving the path and the name under which the file was stored on disk, in the original version of Hadoop.

- The original jobtracker - tasktracker communication protocol requires the jobtracker to send the reducer the list of addresses of the nodes that store the map output data. We modified this communication layer to send the list of file names in the DFS, instead of network addresses. Next, instead of copying the files from the mappers' local filesystem, the tasktracker starts reading data from the specified files stored in the DFS.

Summary

The work presented here brings a twofold contribution. First, we investigate the features of intermediate data in MapReduce computations and we propose a new approach for storing this kind of data in a DFS. In this manner, we avoid the re-execution of tasks in case of failures that lead to data loss. Second, we consider the BSFS file system as storage for intermediate data. It is shown to be suitable for the requirements of intermediate data: availability and high I/O access. This work was carried out in collaboration with Thi-Thu-Lan Trieu during her master internship at INRIA Rennes.

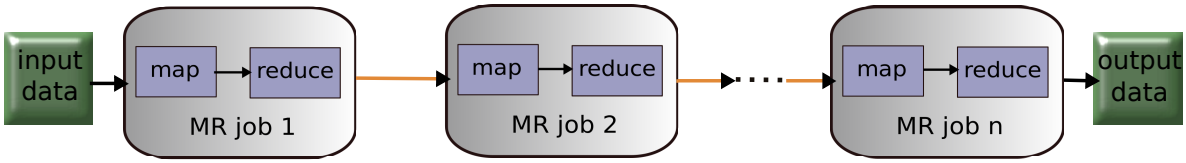


Figure 6.3: A pipeline MapReduce application.

6.3 Intermediate data generated between jobs of a pipeline application

6.3.1 Pipeline MapReduce applications

Many of the computations that fit the MapReduce model, cannot be expressed as a single MapReduce execution, but require a more complex design. These applications that consist of multiple MapReduce jobs that are chained into a long-running execution, are called *pipeline MapReduce applications*. Figure 6.3 illustrates the data flow between the jobs in the pipeline. Each stage in the pipeline is a MapReduce job (with 2 phases, "map" and "reduce"), and the output data produced by one stage is fed as input data to the next stage in the pipeline. Usually, pipeline MapReduce applications are long-running tasks that generate large amounts of intermediate data (the data produced between stages).

However, there are only few scenarios in which users directly design their application as a pipeline of MapReduce jobs. Most of the use cases of MapReduce pipelines come from applications that *translate* into a chain of MapReduce jobs. One of the drawbacks of the extreme simplicity of the MapReduce model is that it cannot be straightforwardly used in more complex scenarios. For instance, in order to use MapReduce for higher-level computations (for example, the operations performed in the database domain) one has to deal with issues like multi-stage execution plan, branching data-flows, etc. The trend of using MapReduce for database-like operations led to the development of high-level query languages that are executed as MapReduce jobs, such as Hive [71], Pig [64], and Sawzall [66].

Pig is a distributed infrastructure for performing high-level analysis on large data sets. The Pig platform consists of a high-level query language called *PigLatin* and the framework for running computations expressed in PigLatin. The PigLatin language is considered to be a compromise between SQL and MapReduce. Programs written in PigLatin comprise SQL-like high-level constructs for manipulating data that are interleaved with MapReduce-style processing.

The Pig framework compiles these programs into a pipeline of MapReduce jobs that are executed within the Hadoop MapReduce environment. Pig takes care of generating the execution plan (the pipeline of jobs), of submitting the jobs to Hadoop and finally, of monitoring and reporting the status of the workflow. Since it uses Hadoop as execution environment, Pig inherits Hadoop's properties, such as scalability and fault tolerance.

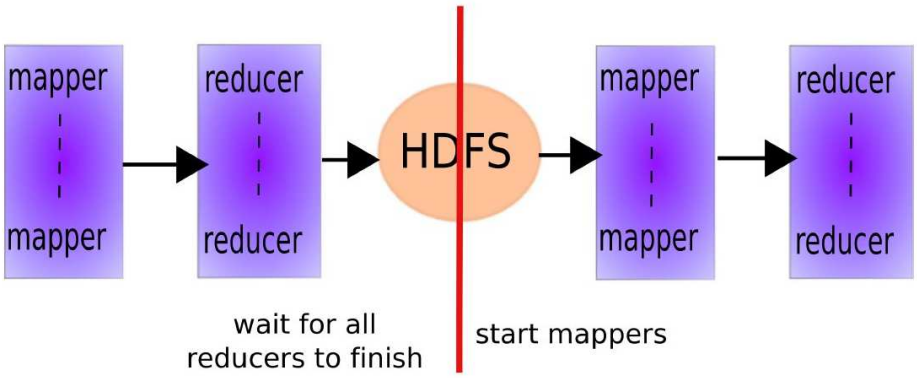


Figure 6.4: Executing a pipeline MapReduce application with Hadoop. The execution of each job is started only when the previous job has completed.

6.3.2 Introducing dynamic scheduling of map tasks in Hadoop

In a pipeline of MapReduce applications, the intermediate data generated between the stages represent the output data of one stage and the input data for the next stage. The intermediate data is produced by one job and consumed by the next job in the pipeline. When running this kind of applications in a dedicated framework, the intermediate data is usually stored in the distributed file system that also stores the user input data and the output result. This approach ensures intermediate data availability, and thus, provides fault tolerance, a very important factor when executing pipeline applications.

However, using MapReduce frameworks to execute pipeline applications raises performance issues, since MapReduce frameworks are not optimized for the specific features of intermediate data. The main performance issue comes from the fact that the jobs in the pipeline have to be executed sequentially. Figure 6.4 shows the process of executing pipeline applications with Hadoop. Considering the case of Hadoop, and the fact that intermediate data is stored in HDFS, a job cannot start until all the input data it processes has been generated by the job in the previous stage of the pipeline. Consequently, Hadoop runs only one job at a time, which results in inefficient cluster utilization and basically, a waste of resources.

As shown on Figure 6.4, each job in the pipeline consists in a "map" and a "reduce" phase. The "map" computation is executed by Hadoop tasktrackers only when all the data it processes is available in the underlying DFS. Thus, the mappers are scheduled to run only after all the reducers from the preceding job have completed their execution. This scenario is also representative for a Pig processing: the jobs in the logical plan generated by the Pig framework are submitted to Hadoop sequentially. In consequence, at each step of the pipeline, at most the "map" and "reduce" tasks of the same job are being executed (Figure 6.5). Running the mappers and the reducers of a single job involves only a part of the nodes in the Hadoop cluster. The rest of the computational and storage cluster capabilities remains idle.

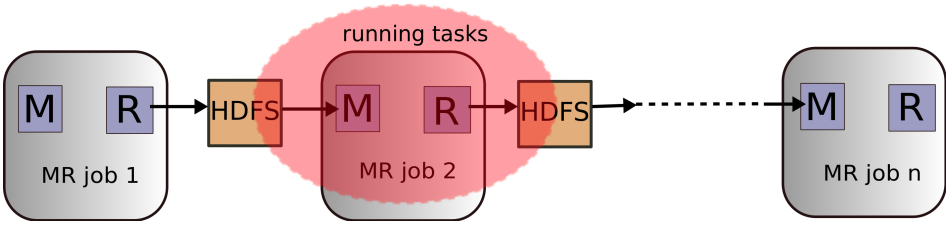


Figure 6.5: Sequential execution of jobs in a pipeline application.

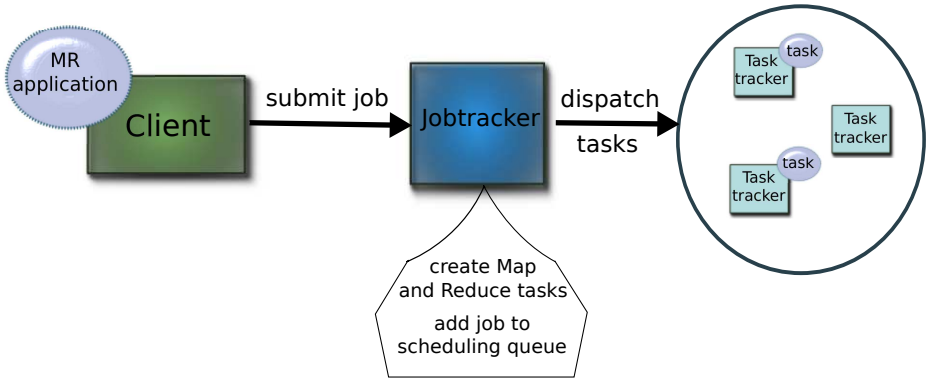


Figure 6.6: Job submission process in Hadoop.

6.3.3 Our approach

In order to speed-up the execution of pipeline MapReduce applications, and also to improve cluster utilization, we propose an optimized Hadoop MapReduce framework, in which the scheduling is done in a *dynamic* manner. For a better understanding of our approach, we first detail the process through which "map" and "reduce" tasks are created and scheduled in the original Hadoop MapReduce framework.

Figure 6.6 displays the job submission process:

- the user specifies the "map" and "reduce" computations of the application it wants to execute;
- the Hadoop client generates all the job-related information (input and output directories, data placement, etc.) and then submits the job for execution to the jobtracker;
- the jobtracker creates the list of tasks for the submitted job. The number of "map" tasks is equal to the number of chunks in the input data, while the number of "reduce" tasks is computed by taking into account various factors, such as the cluster capacity, the user specification, etc.
- the list of tasks is added to the *job queue* that the jobtracker holds;
- the jobtracker makes use of the job queue to schedule jobs for execution on tasktrackers. The tasks of the currently scheduled job are dispatched to tasktrackers.

In the Hadoop MapReduce framework, the "map" and "reduce" tasks are created by the jobtracker when the job is submitted for execution. When they are created, the "map" tasks

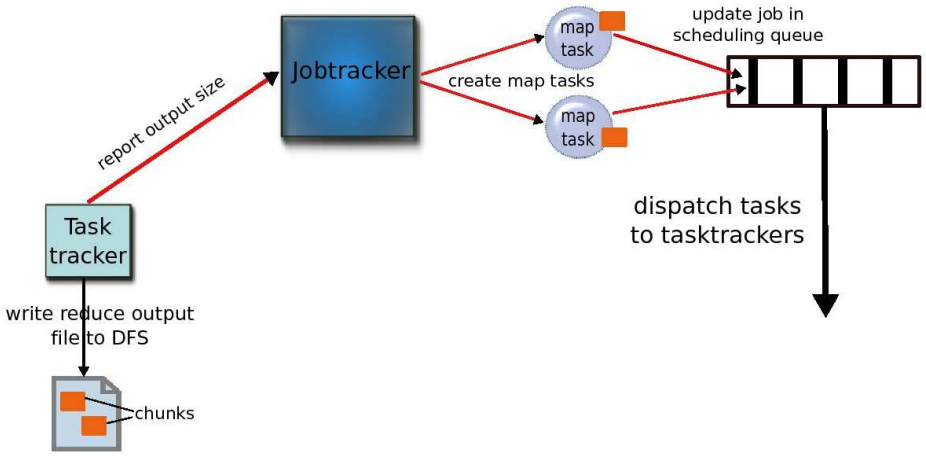


Figure 6.7: Dynamic creation of "map" tasks.

require to know the location of the chunks they will work on. In the context of multiple jobs executed in a pipeline, the jobs are submitted to the jobtracker sequentially, as the chunk-location information is available only when the previous job completes.

Our approach is based on the remark that a "map" task is created for a single input chunk. It only needs to be aware of this very chunk location. Furthermore, when it is created, the only information that the "map" task requires, is the list of nodes that store the data in its associated chunk. We modified the Hadoop MapReduce framework to create "map" tasks *dynamically*, that is, as soon as a chunk is available for processing. This approach can bring substantial benefits to the execution of pipeline MapReduce applications. Since the execution of a job can start as soon as the first chunk of data is generated by the previous job, the total runtime is significantly reduced. Additionally, the tasks belonging to several jobs in the pipeline can be executed at the same time, which leads to a more efficient cluster utilization.

The modifications and extensions of the Hadoop MapReduce framework that we propose, are further presented.

Job-submission process

Client side. On the client side, we modified the submission process between the Hadoop client and the jobtracker. Instead of waiting for the execution to complete, the client launches a *job monitor* that reports the execution progress to the user. With this approach, a pipeline MapReduce application employs a single Hadoop client to run the application. The client submits all the jobs in the pipeline from the beginning, instead of submitting them sequentially. For an application like the one shown on Figure 6.3, the modified Hadoop client submits the whole set of jobs job_1...job_n for execution.

Jobtracker side. The job-submission protocol is similar to the one displayed on Figure 6.6. However, at submission time, only input data for job_1 is available in the DFS. Regarding job_2...job_n, the input data has to be generated throughout the pipeline. Thus, the jobtracker creates the set of "map" and "reduce" tasks only for job_1. For the rest of

Example 4 Report output size (**on tasktracker**)

```
1: procedure CommitTask
2:     (size, files) ← tasktracker.writeReduceOutputData()
3:     jobtracker.transmitOutputInfo(size, files)
4: end procedure
```

the jobs, the jobtracker creates only "reduce" tasks, while "map" tasks will be created along the pipeline, as the data is being generated.

Job scheduling

Tasktracker side: For a job_i in the pipeline, the data produced by the job's "reduce" phase ($reduce_i$) represents the input data of job_{i+1}'s "map" task (map_{i+1}). When $reduce_i$ is completed, the tasktracker writes the output data to the backend storage. We modified the tasktracker code to notify the jobtracker whenever it successfully completes the execution of a "reduce" function: the tasktracker informs the jobtracker about the size of the data produced by the "reduce" task.

Jobtracker side: In our modified framework, the jobtracker keeps track of the output data generated by reducers in the DFS. This information is important for the scheduling of the jobs in the pipeline, as the output directory of job_i is the input directory of job_{i+1}. Each time data is produced in job_i's output directory, the jobtracker checks to see if it can create new "map" tasks for job_{i+1}. If the data accumulated in job_{i+1}'s input directory is at least of the size of a chunk, the jobtracker creates "map" tasks for the newly generated data. For each new chunk, the jobtracker creates a "map" task to process it. All the "map" tasks are added to the scheduling queue and then dispatched to idle tasktrackers for execution.

Figure 6.7 summarizes these interactions. The arrows in red mark additional steps introduced in our modified framework.

The modifications on the tasktracker side are described in Algorithm 4. We extended the code with a primitive that sends to the jobtracker the information about the "reduce" output data: the files written to the DFS and the total size of the data.

Algorithm 5 shows the process of updating a job with information received from tasktrackers. The algorithm is integrated in the jobtracker code, mainly in the scheduling phase. The jobtracker also plays the role of *task scheduler*. It keeps a list of data written to the input directory of each job. For each received update, the jobtracker checks if the data in the job's input directory reaches at least a chunk in size (64 MB default). If it is the case, "map" tasks will be created, one per each new data chunk. Otherwise, the job's information is stored for subsequent processing.

The mechanism of creating "map" tasks is presented in Algorithm 6, executed by the jobtracker, and integrated into the job code. We extended the code so that each job holds the list of files that were generated so far in the job's input directory. When the jobtracker computes that at least a chunk of input data has been generated, new "map" tasks are created for the job. The data in the files is split into chunks. A "map" task is created for each

Example 5 Update job (**on jobtracker**)

```
1:  procedure TRANSMITOUTPUTINFO(size, files)
2:      invoke updateJob(size, files) on taskscheduler
3:  end procedure
4:  procedure UPDATEJOB(size, files)
5:      for all job ∈ jobQueue do
6:          dir ← job.getInputDirectory()
7:          if dir = getDirectory(files) then
8:              if writtenBytes.contains(dir) = False then
9:                  writtenBytes.put(dir, size)
10:             else
11:                 allBytes ← writtenBytes.get(dir)
12:                 writtenBytes.put(dir, allBytes + size)
13:             end if
14:             allBytes ← writtenBytes.get(dir)
15:             if allBytes ≥ CHUNK_SIZE then
16:                 b ← job.createMapsForSplits(files)
17:                 writtenBytes.put(dir, allBytes − b)
18:             else
19:                 job.addToPending(files)
20:             end if
21:         end if
22:     end for
23: end procedure
```

chunk and the newly launched tasks are added to the scheduling queue. The jobtracker also informs the "reduce" tasks that the number of "map" tasks has changed. The reducers need to be aware of the number of mappers of the same job, as they have to transfer their assigned part of the output data from all the mappers to their local disk.

Summary

Section 6.3 of this chapter addresses a special class of MapReduce applications, i.e., applications that consist of multiple jobs executed in a pipeline. In this context, we propose several optimizations in the Hadoop MapReduce framework in order to improve its performance when executing pipelines. Our proposal consists mainly of a new mechanism for creating tasks along the pipeline, as soon as their input data become available. This dynamic task scheduling leads to an improved performance of the framework, in terms of job completion time. In addition, our approach ensures a more efficient cluster utilization, with respect to the amount of resources that are involved in the computation.

Example 6 Create map tasks (**on job**)

```
1:  procedure ADDTOPENDING(files)
2:      pendingFiles.addAll(files)
3:  end procedure
4:  function CREATEMAPSFORSPLITS(files) returns splitBytes
5:      pendingFiles.addAll(files)
6:      splits ← getSplits(pendingFiles)
7:      pendingFiles.clear()
8:      newSplits ← splits.length
9:      jobtracker.addWaitingMaps(newSplits)
10:     for i ∈ [1..newSplits] do
11:         maps[numMapTasks + i] ← newMapTask(splits[i])
12:     end for
13:     numMapTasks ← numMapTasks + newSplits
14:     notifyAllReduceTasks(numMapTasks)
15:     for all s ∈ splits do
16:         splitBytes ← splitBytes + s.getLength()
17:     end for
18:     return splitBytes
19: end function
```

Chapter 7

Enabling and Leveraging the Append Operation in Hadoop

Contents

In a previous chapter we presented BSFS - a distributed file system able to deliver high throughput under heavy concurrency, when used as a storage layer for MapReduce applications. In this chapter, we show how BSFS can enable extensions to the de facto MapReduce implementation - Hadoop. The focus of this work is on introducing support for the *append* operation in Hadoop. We first investigate the benefits of providing support for append in scenarios that involve MapReduce applications, but also in other contexts where the append operation is needed at the file-system level. The second part of the chapter is dedicated to presenting the extensions we made in Hadoop in order to take advantage of the append operation.

7.1 Motivation

The storage layer is a key component of MapReduce frameworks. As both the input data and the output data produced by the reduce function are stored by this layer (typically a distributed file system), its design and functionalities influence the overall performance. MapReduce applications typically process data consisting of up to billions of small records (of the order of KB), hence scalability is critical in this context.

One important aspect of scalability regards the number of files that need to be managed by the file system. Acquiring and storing large data sets of the order of hundreds of TB and beyond, using KB-sized files incurs a significant metadata overhead. This approach is both unmanageable and inefficient. This issue, known as the "file-count problem", has been acknowledged as a major source of inefficiency for large-scale settings of distributed file systems. To take a representative example, according to its designers, Google File System is facing this problem and therefore is likely to undergo substantial design changes in the near future [57].

To handle such very large data sets of small pieces of data without having to manage very large sets of small files, a better approach consists in packing these pieces of data together into huge files (e.g., gathering hundreds of GB or TB of data). Consequently, massively parallel data generation leads to a large number of processes *appending* records to a huge, shared file. This is why we believe that providing an efficient support for the append operation under heavy concurrency will be increasingly important in the forthcoming years in the context of data-intensive applications.

The work presented in this chapter focuses on the problem of efficiently supporting the append operation to huge shared files in large-scale distributed infrastructures under heavy concurrency. This problem is timely and particularly relevant to today's emerging MapReduce frameworks like Hadoop. In the context of massively parallel MapReduce applications, enabling efficient concurrent append operations to shared files within the MapReduce framework brings two main benefits. First, the number of files (and the associated overhead related to file management) can be substantially reduced. Second, application programming also gets simpler: data do not need to be explicitly managed as a set of distributed chunks and the MapReduce tasks can simply access data within globally shared files.

7.2 The need for the append operation in MapReduce frameworks

MapReduce applications do not require the append operation to be supported by the distributed file system used as underlying storage. However, many benefits can be drawn from this functionality. These advantages are briefly discussed below, together with the status for this feature in two of the file systems developed to support data-intensive applications.

7.2.1 Potential benefits of the append operation

MapReduce data processing applications are not the only class of applications that may potentially benefit from an efficient support of the append operation in a file system. In the Google File System, record append is heavily used in the context of applications following the multiple-producer/single-consumer model; this kind of applications usually consists in

processing data that is collected from multiple sources. Record append is useful in this scenario, as it allows the data to be collected by multiple parallel processes into a single file.

As far as HDFS is concerned, supporting append can enable applications that require a more elaborate API, to use the file system as storage back-end. An example of such application is HBase [6], an open-source project from Hadoop, designed after Google's BigTable system [23], with the purpose of providing distributed, column-oriented storage of large amounts of structured data, on top of HDFS. HBase keeps its transaction log in main memory and periodically, flushes it to HDFS; if a crash occurs, HBase can recover its previous state by going through the transaction log. However, although the transaction log can be opened for reading, after recovery, HBase will write its updates to a different file in HDFS. Supporting appends can enable HBase, as well as other database applications, to manage their ever-expanding transaction log as a single huge file, stored in HDFS.

At the level of the Hadoop Map/Reduce framework, a single output file can be generated in the "reduce" phase, instead of having each reducer writing its output to a different file. In a scenario with multiple Map/Reduce applications that can be executed in pipeline, the framework can rely on append to significantly improve execution time, by allowing readers to work in parallel with appenders: applications that generate the data can append it concurrently to shared files, while at the same time, applications that process the data can read it from those files.

7.2.2 Append status in Google File System

The Google File System (GFS) [40] was developed with the goal of accommodating the storage needs of the applications Google runs on a daily basis. Hence, GFS is optimized for access patterns involving huge files that are mostly appended to, and then read from. Since applications that exhibit these types of access patterns were targeted, supporting append was a critical functionality and thus was implemented right from the beginning.

The append operation in GFS is called *record append*. Its purpose is to enable multiple clients to append data to the same file concurrently. By simply using the write operation provided by GFS, this scenario with multiple clients simultaneously modifying the same file can lead to inconsistent data. When writing data, the client has to specify also the offset where the data must be written. Concurrent clients writing data to the same *region* (part of a chunk) may lead to a interweaving of data fragments belonging to different clients. Due to the way it is implemented, the *record append* operation prevents this from happening.

Record append was introduced with the purpose of ensuring *atomicity* in terms of data contiguousness. When using record append, the clients supply only the data to be appended and GFS ensures that the data will be appended to the file as a continuous sequence of bytes. The offset the data is appended at is chosen by GFS and is returned to the client issuing the append.

7.2.3 Append status in HDFS

The Hadoop Distributed File System was developed with the initial purpose of supporting applications that follow the Map/Reduce programming paradigm. These applications process files that comply with the write-once, read-many-times model; HDFS features and

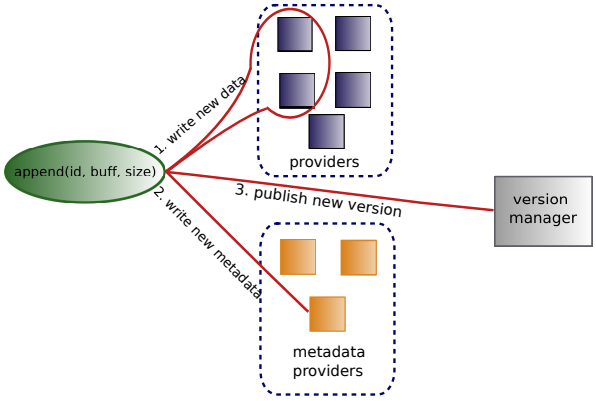


Figure 7.1: The steps of appending in BlobSeer.

semantics were designed to suit this model. However, the growing popularity of HDFS, as well as the variety and the increasing number of applications that can be modeled using the Map/Reduce paradigm, led to extending HDFS with more functionalities. One of these required functionalities is the support for append operations.

In early versions of HDFS, files were immutable once closed. They were visible in the file system namespace only after a successful close operation. Implementing append in HDFS required substantial modifications to the whole framework. One of them concerned the fact that blocks were not longer immutable; once append was enabled, the last block of a file became mutable, which rose the problem of how to detect obsolete versions of that block. This situation is possible for instance, when a datanode holding a replica of the block dies while data is appended to the block. The solution was to add a generation stamp to each block - an incrementing integer that records the version of a particular block.

However, shortly after being introduced, append support was disabled, because all the changes it involves are still an open issue. Currently, the append operation is defined in the file system interface of Hadoop, but its support in HDFS is disabled for normal users.

7.3 Introducing support for the append operation in Hadoop

In order to provide support for the append operation in Hadoop, we rely on BlobSeer, the versioning-based, concurrency-optimized BLOB (Binary Large Object) management system described in Chapter 4. In Chapter 5 we presented how BlobSeer could be used as a storage substrate providing the same interface as Hadoop's default file system (HDFS), thanks to a file system layer (BSFS) built on top of BlobSeer. In this chapter, we show how BSFS can be integrated into Hadoop, to allow Hadoop's Map/Reduce applications to benefit from BlobSeer's efficient support for concurrent data access to shared data.

7.3.1 BlobSeer: efficient support for the append operation

In BlobSeer, append is implemented as a special case of the write operation, in which the offset is implicitly assumed to be the size of the latest version of the BLOB. For an append operation, the user supplies as input the id of the BLOB that is appended to, the data to be

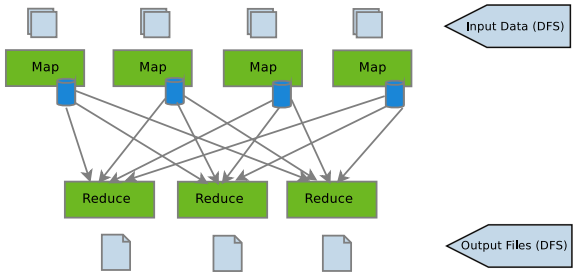


Figure 7.2: Original Hadoop framework: each reducer writes to a separate file.

stored (a buffer holding the data and the data size); after the append is finished, the user receives the number of the version this update generates. The input data is split into pages that are then written in parallel to a list of providers retrieved from the provider manager. When all the pages are successfully written to the providers, the version manager assigns a number to the newly generated BLOB version.

The design concepts BlobSeer uses enable a high degree of parallelism, especially where updates are concerned. BlobSeer relies on a versioning-based concurrency control algorithm that maximizes the number of operations performed in parallel in the system. This is done by avoiding synchronization as much as possible, both at the data and metadata levels. An update to a BLOB is done in two phases. First, any writer or appender writes its new data blocks, by storing the differential patch. Then, in a second phase, the version number is allocated and the new metadata referring to these blocks are generated. These steps are shown in Figure 7.1. Since each writer or appender generates new data/metadata and never modifies existing data/metadata, readers are completely decoupled from writers/appenders, as they always access immutable snapshots. A reader can thus access data and metadata in a fully parallel fashion with respect to writers and appenders (and of course, with respect to other readers).

To sum up, in BlobSeer, multiple clients can append their data in a fully parallel manner, by asynchronously storing the pages on providers; synchronization is required only when writing the metadata, but this overhead is low, as shown in [61].

7.3.2 How BlobSeer enables appends in Hadoop

Our approach aims at enabling Map/Reduce applications to benefit from the append operation BlobSeer provides, and consists of two steps: using append at the level of the Hadoop framework, and supporting append at the level of the distributed file system that acts as storage layer.

Modifying Hadoop to use appends

In the original Hadoop Map/Reduce framework (Figure 7.2), when a tasktracker executes the "reduce" function specified by the user, the output is written to a temporary file; each temporary file has a unique name, so that each reducer writes to a distinct file. When the "reduce" phase is completed, each reducer renames the temporary file to the final output

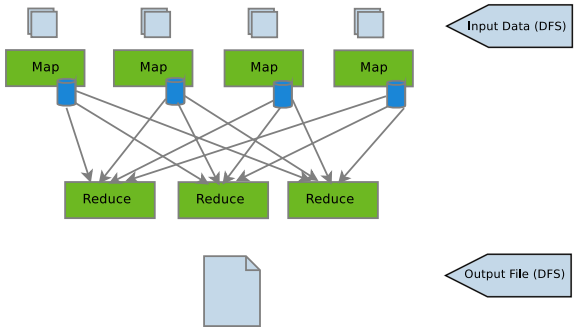


Figure 7.3: Modified Hadoop framework: all the reducers append to the same file.

directory, specified by the user. The final result obtained by running the Map/Reduce application, consists of multiple parts, one part per reducer. We modified the reducer code to append the output it produces to a single file, instead of writing it to distinct files (Figure 7.3). The name of the output file can be specified from the beginning when executing the MapReduce job.

Having all the reducers append to the same file, impacts on both the application running on top of the framework, and the file system storing the data. An application consisting of multiple Map/Reduce instances that can be executed in pipeline, is able to complete substantially faster by running in parallel "map" and "reduce" phases from different stages. Mappers from one stage of the pipeline open the input file for reading in order to process the data, while reducers from the previous stage can still generate the data and append it to the same file. The append operation reduces the number of files to be stored in the distributed file system that serves as storage for the application executed by the Hadoop framework. This impacts on the namespace management, by considerably reducing the metadata associated to files.

Supporting appends at the file-system level

The features BlobSeer exhibits meet the storage needs of Map/Reduce applications. In order to enable BlobSeer to be used as a file system within the Hadoop framework, we added an additional layer on top of the BlobSeer service, layer that we called the *BlobSeer File System - BSFS*. This work was described in Chapter 5.

As previously shown, the Hadoop Map/Reduce framework accesses the storage layer through an interface that exposes the basic functions of a file system. The append operation is available in the interface (but is not implemented in the latest Hadoop release available): we could thus implement it using the primitives provided by BlobSeer. Performing an append to an existing file is translated into two operations: appending the data to the corresponding BLOB, and updating the size of the file at the level of the *namespace manager* of BSFS.

Supporting append enables applications like HBase to directly use the file system to store their logs as a single file that can be read from and appended to at the same time.

7.4 Summary

The Map/Reduce programming model initially emerged in the Internet services community, but its simple yet versatile interface led to an increasing number of applications that are modeled using this paradigm. Efficiently supporting various types of applications requires that the framework executing them, as well as the distributed file system that acts as backend storage, are extended with new functionalities. The work presented in this chapter focuses on the append operation as a functionality that can bring benefits at two levels. First, introducing append support at the level of the file system may be a feature useful for some applications (not necessarily belonging to the Map/Reduce class). For instance, an application may need to manage a log that is simultaneously and continuously being read from/appended to. We describe how our BlobSeer-based file system (BSFS) offers support for the append operation and, moreover, it is able to deliver high throughput when multiple clients concurrently append data to the same file. Second, since append is supported by the file system, we have modified the Hadoop Map/Reduce framework to take advantage of this functionality. In our modified Hadoop framework, the reducers append their data to a single file, instead of writing it to a separate file, as it was done in the original version of Hadoop. The advantage is obvious in terms of simplicity: at the end of the computation, data is already available in a single logical file (the distribution of the file chunks is transparently handled by BlobSeer). This file is ready to use for any subsequent processing. No extra application logic is needed for subsequent processing, in contrast to the original Hadoop, which has to explicitly handle a (potentially large) group of (thousands of) files.

Based on the use of BSFS as a storage layer, our improved Hadoop framework can further be optimized for the case of MapReduce applications that are executed in pipeline. For this type of applications, the mappers and the reducers belonging to distinct stages of the pipeline, can concurrently be executed: the reducers generate the data and append it to a file that is at the same time, read and processed by the mappers. This scenario can be efficiently supported by BSFS, since the impact of concurrent readers and appenders on each other is low.

Part III

Implementation Details

Chapter 8

Implementation details

Contents

In this chapter we present the implementation of the contributions we proposed in the previous chapters of this manuscript. First, we describe the implementation of the BSFS file system and its interconnection with Hadoop and BlobSeer. Second, we focus on the extensions and modifications we carried out within the Hadoop MapReduce framework to enhance it with the aforementioned features. The last section of this chapter is dedicated to the deployment tools we developed to allow us to evaluate our work.

8.1 Designing BSFS

The BSFS file system presented in Chapter 5 consists of a layer added on top of BlobSeer. This layer that implements the Hadoop file-system interface. Since the Hadoop project is written in Java, and BlobSeer provides bindings for several programming languages (including Java), the BSFS layer is implemented using Java as well. In this section, we focus on the implementation of BSFS, and we elaborate on specific issues for each module of the implementation.

Figure 8.1 illustrates the modules that constitute the BSFS layer. There are two main components in the BSFS layer: the *namespace manager* and the *file-system client*.The namespace manager is a centralized standalone entity. It is responsible for managing the file-system

metadata. On the other hand, the file-system client is integrated with the Hadoop code. Its main role being to enable BlobSeer to be used as a file system. Each Hadoop client launches a BSFS file-system client to interact with the centralized namespace manager and to perform file-system operations over the BLOBs stored in BlobSeer. We further detail the modules in each of the two components.

The file-system client

This component resides on each node that acts as a BSFS user, including Hadoop's mappers and reducers. Each file-system client interacts with the namespace manager on one side, and acts as a BlobSeer client, on the other side. To enable all these interactions we developed several modules:

- The `BlobSeerFileSystem` module implements the file-system interface through which Hadoop MapReduce accesses the storage backend. This interface specifies basic file-system operations for accessing files and directories. The module processes two types of user requests: file-system namespace inquiries, and file access operations. The first class of requests involves only accessing file metadata, whereas the second type entails additional steps of direct access to BLOBs.
- The `BSFSClient` module handles the communication with the namespace manager. This module uses a Java socket to send user requests over the network. The reply is then interpreted by the `BlobSeerFileSystem` module and returned to the user.
- The `CachedInputStream` and `CachedOutputStream` modules implement buffered handlers for I/O operations. These handlers contain an `ObjectHandler` that is used to perform operations on BLOBs, and a BLOB id that links a file to its corresponding BLOB. Both streaming modules use Java's NIO byte buffer to implement caching. For performance reasons, we chose to use direct byte buffers, as the Java virtual machine handles them more efficiently: the JVM will make a best effort to perform native I/O operations directly upon it. That is, it will attempt to avoid copying the buffer content to (or from) an intermediate buffer before (or after) each invocation of one of the underlying operating system native I/O operations.

The namespace manager

The role of this centralized component is to manage the file-system namespace and to keep the mapping of files to BLOBs. The modules in this component's implementation are further detailed:

- The `Manager` module handles the communication with the clients. Its implementation is based on the Java New I/O (NIO) API, more specifically, the NIO channels and selectors that enable efficient handling of multiple connections in an asynchronous manner. The manager is basically a NIO server that uses a dedicated selector thread to process client requests. Whenever it receives a request on an opened socket, the manager adds the request to a `Requests queue`. When the request has been processed, the manager sends the reply to the corresponding socket.

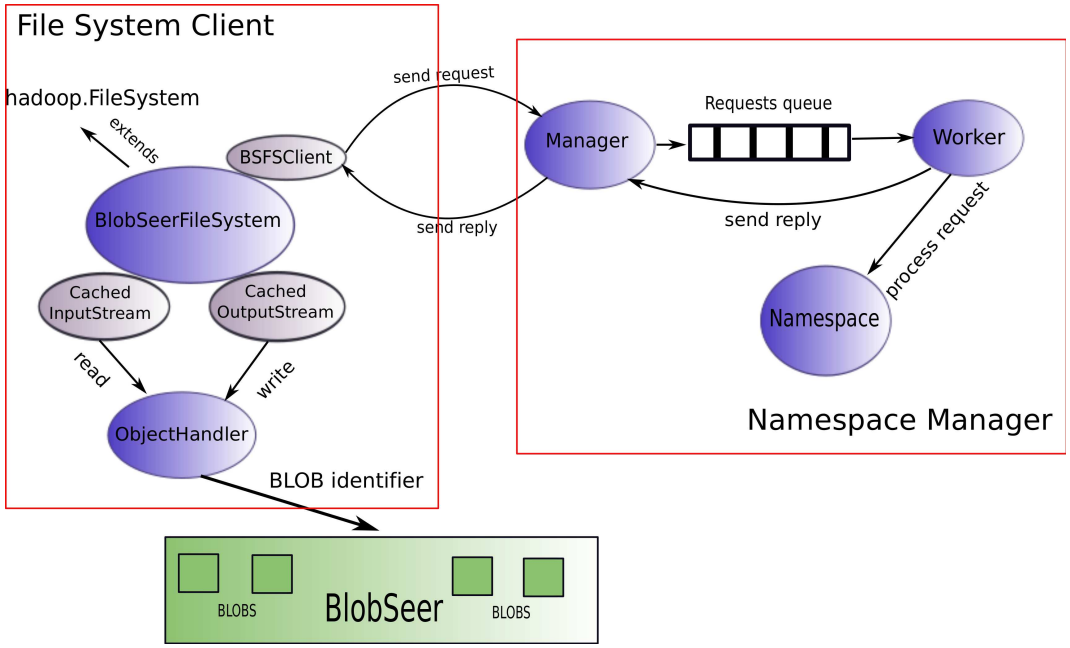


Figure 8.1: BSFS implementation.

- The `Worker` thread extracts requests from the queue and then handles them sequentially.
- The `Namespace` module is the entity that does the actual metadata management, whereas the other modules implement the communication layer. The namespace module stores the metadata related to files (file properties, mapping to BLOBs) and also the hierarchy of directories and files. The requests extracted by the worker from the queue are processed by the namespace thread that updates the metadata accordingly and then generates the reply.

8.2 Extensions to Hadoop

Chapters 6 and 7 describe several extensions and features we introduced in the Hadoop MapReduce framework. In order to implement these features, a lot of effort was invested in studying and understanding the design of Hadoop's components and the interactions between them. This section details the implementation issues related to the extensions we brought to Hadoop.

8.2.1 Efficient intermediate data management in Hadoop

We addressed the problem of efficiently managing intermediate data in MapReduce computations in two cases: when the intermediate data is generated within the same job and when it is produced between the stages of a pipeline application. In the first scenario, our proposal of storing intermediate data in a DFS involves modifying both the mapper and the reducer

code of Hadoop. Section 6.2 lists the modifications on the mapper and reducer side. The implementation uses Hadoop's mechanisms and interfaces.

In the context of pipeline MapReduce applications, introducing a new mechanism for creating and scheduling "map" tasks required a more complex and substantial implementation phase. To better illustrate the extensions of our modified framework, we further present class diagrams summarizing the interactions between the entities in both original and modified Hadoop. To implement the pipeline-optimization algorithm we proposed in Section 6.3, we extended and modified the Hadoop code in three of the interactions and communication protocols, which we detail below.

Tasktracker - Task communication protocol

As Figure 8.2 shows, in the original Hadoop, the tasktracker communicates with the task it is currently running, through an interface, `TaskUmbilicalProtocol`. When the task is launched in a JVM on the tasktracker, the `run` method of either the "map" or "reduce" task is executed. Upon completion, the data generated by the task is committed as output data by invoking the `commitTask` on the `FileOutputCommitter`. In the case of a "reduce" task, this method moves the task's output data from the task's temporary directory in the DFS, to the final output directory. Figure 8.2 also illustrates the heartbeat mechanism through which the tasktracker and the jobtracker interact. Every (configurable) amount of time, the tasktracker sends a status message to the jobtracker, indicating if it still has available slots for running new tasks. The jobtracker replies with a list of actions that basically represent new tasks scheduled for running on the tasktracker.

The first step towards enabling dynamic creation of "map" tasks is to keep track of the amount of data produced by each "reduce" task. To meet this goal, we extended the Hadoop interfaces, as shown on Figure 8.3. We introduced a new class that manages information about the output data produced by the reducers: the size of the data and the directory containing the data. We added new methods in both the tasktracker - task interface and the jobtracker - tasktracker communication interface to allow the propagation of this information from the `FileOutputCommitter` to the jobtracker.

Jobtracker - Jobscheduler interaction

Hadoop's jobtracker is a complex component that implements communication interfaces and also runs the scheduler of the submitted jobs. Figure 8.4 presents the flow of interactions that constitute the scheduling process. The default task scheduler is the `JobQueueTaskScheduler` that manages a queue of `JobInProgress` objects. The job queue is updated by the `Jobtracker` as the jobs are submitted for execution, and it is processed in sequential order. Whenever a heartbeat message is received from a tasktracker with available slots, the jobtracker invokes the `assignTasks` method on the `JobQueueTaskScheduler`. This method searches for the first active (running) job in the queue and then tries to find a "map" task of that job, that can be run locally on the given tasktracker. If it fails to find a local "map" task, a remote task is selected. The scheduler selects as many tasks to run as the number of available slots on the tasktracker. The list of tasks to launch is then transmitted by the jobtracker over the network.

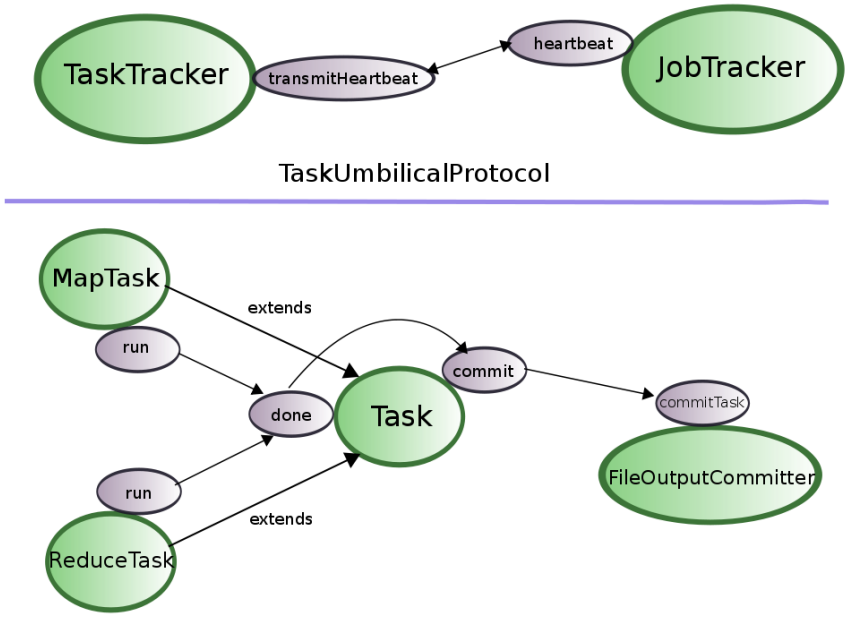


Figure 8.2: Original Hadoop: Tasktracker - Task interactions.

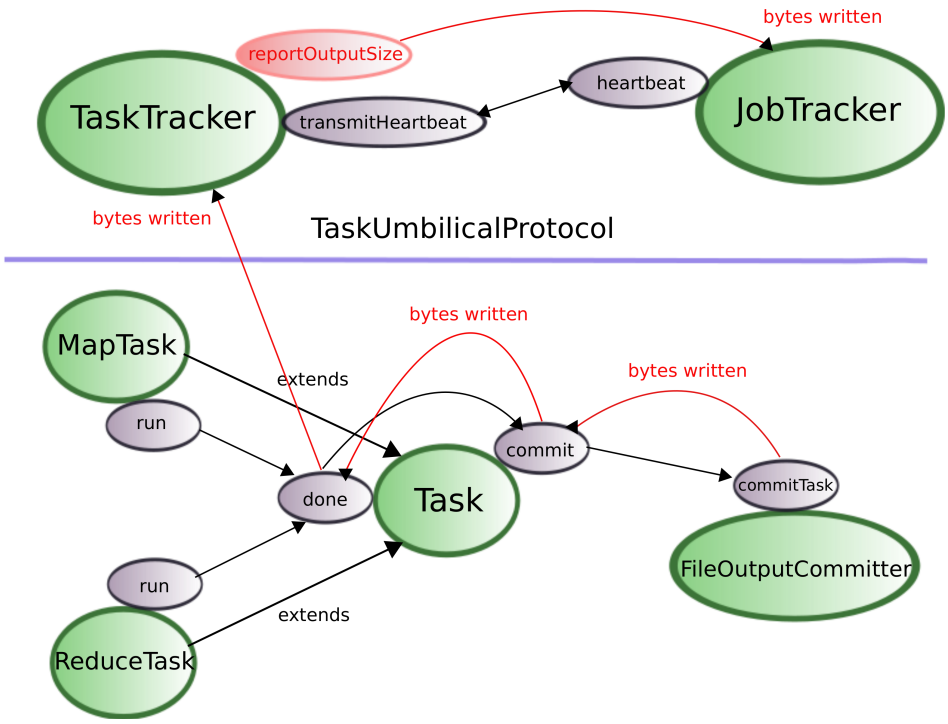


Figure 8.3: Modified Hadoop: Tasktracker - Task interactions.

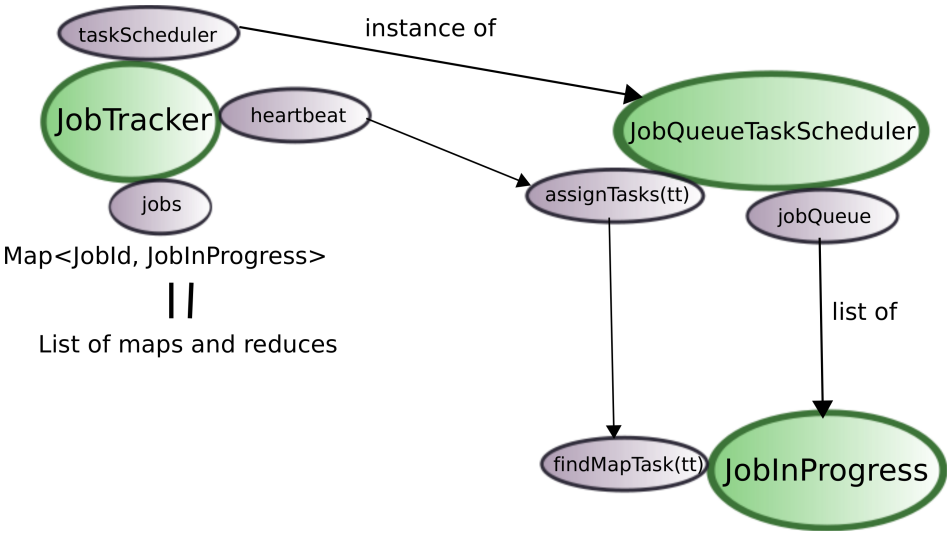


Figure 8.4: Original Hadoop: Jobtracker - Jobscheduler interactions.

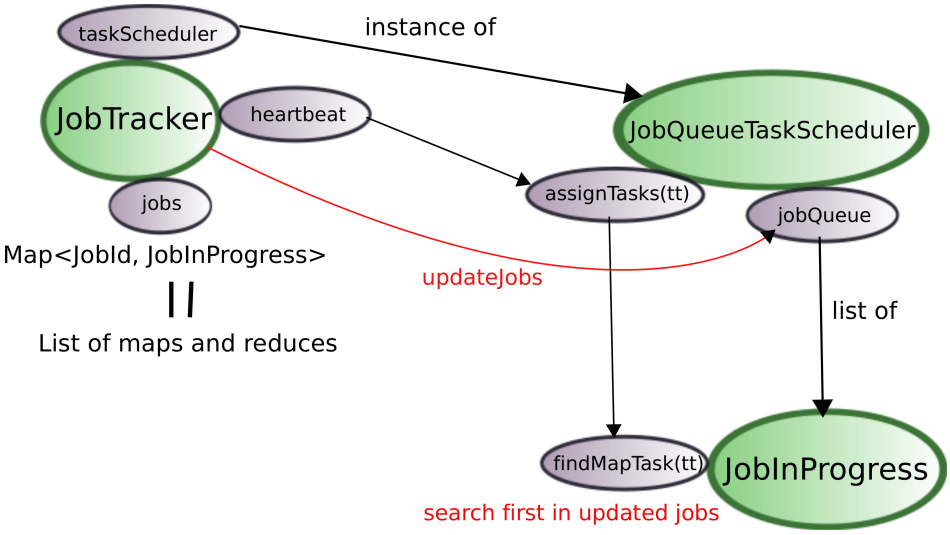


Figure 8.5: Modified Hadoop: Jobtracker - Jobscheduler interactions.

In order to keep track of the size of data generated by each stage of the pipeline, we extended the `JobInProgress` class to hold a list of files written so far in the job input directory. This list is updated by the jobtracker whenever a notification is received from a reducer that produced output into the DFS. Moreover, the jobtracker checks if the data accumulated in the job input directory accounts for a chunk size. If it is the case, the `JobInProgress` is updated with new "map" tasks created to process the newly generated chunks. The jobtracker also updates all the data structures it holds and notifies all the entities that make use of the number of "map" tasks. Figure 8.5 summarizes those extensions. The code snippet below illustrates the job scheduler modifications.

```
1  public void updateJob(CommitInfo c, long blk)
2   {
3           Collection<JobInProgress> jobQueue = jobQueueJobInProgressListener.getJobQueue();
4           synchronized (jobQueue) {
5                   for (JobInProgress job : jobQueue) {
6                           JobConf conf = job.getJobConf();
7                           String jod = conf.get("mapred.input.dir");
8                           ArrayList<String> files = c.getJobOutputDir();
9                           if (files == null)
10                                  return;
11                          if (files.size() == 0)
12                                  return;
13                          String dir = new Path(files.get(0)).getParent().toString();
14                          if (jod.compareTo(dir) == 0) {
15                                  if (!writtenBytes.containsKey(dir))
16                                          writtenBytes.put(dir, c.getSize());
17                                  else {
18                                          long l = writtenBytes.get(dir);
19                                          writtenBytes.put(dir, l + c.getSize());
20                                  }
21                                  long l = writtenBytes.get(dir);
22                                  if (l >= blk)
23                                          try {
24                                                  long rez = job.createMapForSplit(files);
25                                                  writtenBytes.put(dir, l − rez);
26                                          } catch(Exception e) {
27                                                  e.printStackTrace();
28                                          }
29                                  else
30                                          job.addToPending(files);
31                          }
32                  }
33          }
34  }
35
36  public long createMapForSplit(ArrayList<String> files) throws IOException
37  {
38          pendingFiles.addAll(files);
39
40          InputSplit[] splits = null;
41          try {
42                  splits = conf.getInputFormat().getSplits(conf, conf.getNumMapTasks(), pendingFiles);
43          }
44          catch(Exception e) {
45                  e.printStackTrace();
46          }
```

```
47              pendingFiles.clear();
48              if (splits == null)
49                      return 0;
50
51              int newSplits = splits.length;
52              jobtracker.getInstrumentation().addWaitingMaps(getJobID(), newSplits);
53
54              TaskInProgress[] newMaps = new TaskInProgress[numMapTasks];
55              System.arraycopy(maps, 0, newMaps, 0, maps.length);
56              numMapTasks += newSplits;
57              maps = new TaskInProgress[numMapTasks];
58              System.arraycopy(newMaps, 0, maps, 0, newMaps.length);
59
60              String jobFile = profile.getJobFile();
61
62              RawSplit[] rw = new RawSplit[newSplits];
63
64              DataOutputBuffer buffer = new DataOutputBuffer();
65              for(int i = 0; i < newSplits; i++) {
66                      RawSplit rawSplit = new RawSplit();
67                      rawSplit.setClassName(splits[i].getClass().getName());
68                      buffer.reset();
69                      splits[i].write(buffer);
70                      rawSplit.setDataLength(splits[i].getLength());
71                      rawSplit.setBytes(buffer.getData(), 0, buffer.getLength());
72                      rawSplit.setLocations(splits[i].getLocations());
73                      maps[newMaps.length + i] = new TaskInProgress(jobId, jobFile, rawSplit,
74                              jobtracker, conf, this, newMaps.length + i);
75                  rw[i] = rawSplit;
76                }
77
78              if (newSplits > 0) {
79                      Map<Node, List<TaskInProgress>> m = createCache(rw, maxLevel);
80                      if (m != null)
81                              if (nonRunningMapCache == null)
82                                      nonRunningMapCache = m;
83                              else
84                                      nonRunningMapCache.putAll(m);
85                }
86
87              long splitBytes = 0;
88
89              for(InputSplit split: splits) {
90                      splitBytes += split.getLength();
91              }
92
93              for (TaskInProgress t : nonRunningReduces) {
94                      t.updateNumTasks(numMapTasks);
95              }
96              return splitBytes;
97      }
```

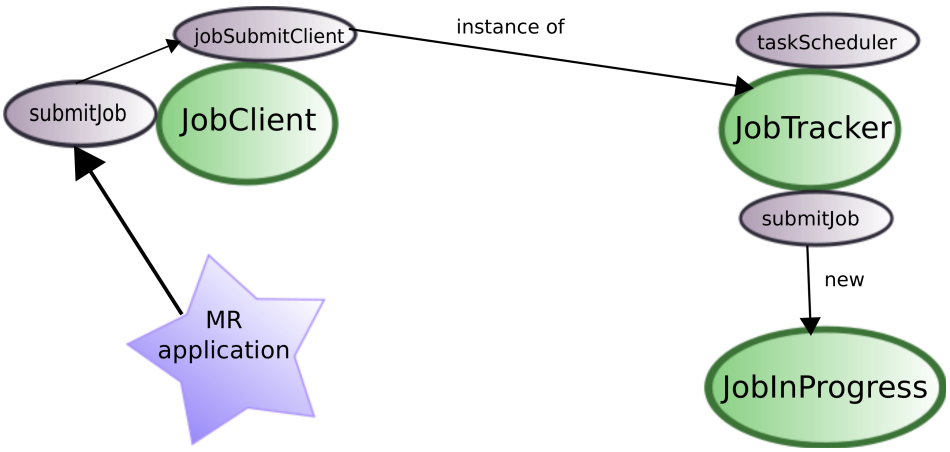


Figure 8.6: Original Hadoop: Job-submission process.

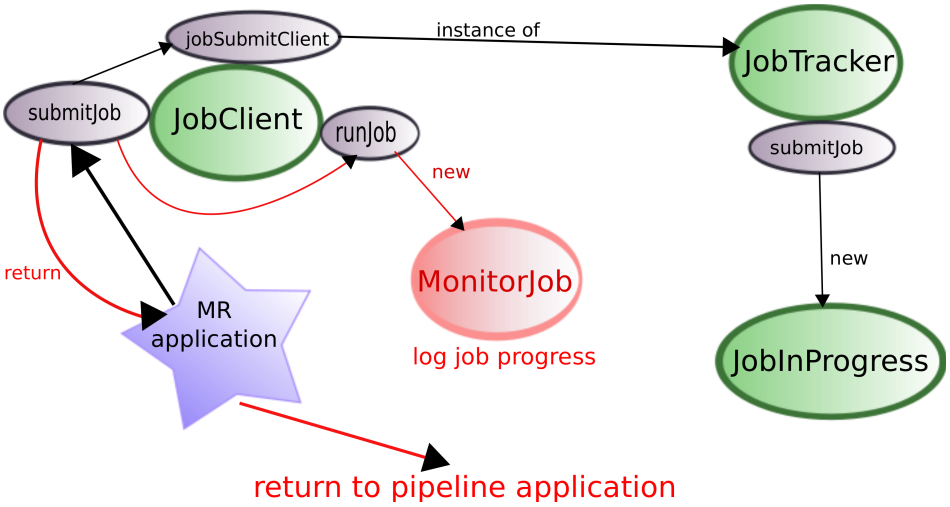


Figure 8.7: Modified Hadoop: Job-submission process.

Job-submission process

The job-submission process in Hadoop is handled by the JobClient class. The user calls the submitJob method of the JobClient that copies all the application-related metadata (job jar, configuration properties, etc.) from the user's local machine to the DFS. The client also employs a communication interface with the jobtracker through the jobSubmitClient. Through this interface, the jobtracker becomes aware of the newly submitted job and thus is able to update its internal structures accordingly. A new JobInProgress object is created which holds lists of "map" and "reduce" tasks. The "reduce" tasks do not hold any meaningful information, as they will be updated as they are scheduled to run. On the other hand, the "map" tasks are assigned to process one or several data chunks and are aware of their data location since their creation time. The job with its internal structures is added to the scheduling queue and its tasks will be executed as soon as the jobtracker receives heartbeat messages from tasktrackers with available slots. Calling the submitJob on the JobClient is a blocking process that will return to the user only when the job is completed. The job-submission protocol is described on Figure 8.6.

For our pipeline-optimized MapReduce framework, the job-submission process requires that all the jobs are submitted at once. Thus, we modified the Hadoop client so that invoking the submitJob does not block the whole submission process. In order to keep track of each job execution status, we introduced a new entity MonitorJob that is responsible for monitoring job progress and reporting the final result. The modified submitJob method creates a new MonitorJob instance for each submitted job in the pipeline. On the jobtracker's side, the job-submission protocol consists in creating a JobInProgress instance and adding it to the scheduler's queue. However, in the modified Hadoop version, the newly created job has an empty list of "map" tasks, as those tasks will be created at a later point in the pipeline. The modifications we made are shown on Figure 8.7.

The following is a code snippet showing the modified job submission process on the client side.

```
1  public class MonitorJob extends Thread
2    {
3      JobConf conf;
4      RunningJob job;
5      String fileName;
6      public MonitorJob(JobConf c, RunningJob j, String file)
7      {
8              conf = c;
9              job = j;
10             fileName = file;
11     }
12     public boolean monitorAndPrintJob() throws IOException, InterruptedException
13     {
14             File outFile = new File(fileName);
15             DataOutputStream dos = new DataOutputStream(new FileOutputStream(outFile));
16             String lastReport = null;
17             TaskStatusFilter filter;
18             filter = getTaskOutputFilter(conf);
19             JobID jobId = job.getID();
20             dos.writeBytes("Running␣job:␣" + jobId + "\n");
21             //LOG.info("Running job: " + jobId);
22             int eventCounter = 0;
```

```
23          boolean profiling = conf.getProfileEnabled();
24          Configuration.IntegerRanges mapRanges = conf.getProfileTaskRange(true);
25          Configuration.IntegerRanges reduceRanges = conf.getProfileTaskRange(false);
26          while (!job.isComplete()) {
27                  if (!report.equals(lastReport)) {
28
29                          dos.writeBytes(report + "\n");
30
31                          // LOG.info(report);
32                          lastReport = report;
33                  }
34                  TaskCompletionEvent[] events = job.getTaskCompletionEvents(eventCounter);
35                  eventCounter += events.length;
36                  for(TaskCompletionEvent event : events){
37                          TaskCompletionEvent.Status status = event.getTaskStatus();
38                          if (profiling && (status == TaskCompletionEvent.Status.SUCCEEDED ||
39                              status == TaskCompletionEvent.Status.FAILED) &&
40                                (event.isMap ? mapRanges : reduceRanges).isIncluded(event.idWithinJob())) {
41                                  downloadProfile(event);
42                          }
43                          ...........
44                  }
45          dos.writeBytes("Job␣complete:␣" + jobId + "\n");
46          //LOG.info("Job complete: " + jobId);
47          job.getCounters().log(LOG);
48          dos.close();
49          return job.isSuccessful();
50        }
51        public void run()
52        {
53          try {
54              if (!monitorAndPrintJob()) {
55                  throw new IOException("Job␣failed!");
56              }
57            } catch (Exception ie) {
58              Thread.currentThread().interrupt();
59            }
60         }
61  }
62
63  public static RunningJob runJob(JobConf job) throws IOException {
64    JobClient jc = new JobClient(job);
65    RunningJob rj = jc.submitJob(job);
66    MonitorJob mj = jc.new MonitorJob(job, rj, "logs/" + rj.getID() + ".log");
67    mj.start();
68    return rj;
69  }
```

8.2.2 Introducing the append operation

Enabling the support for the append operation in Hadoop requires first implementing append at the file-system level, and second, exploiting this feature in the MapReduce layer. As detailed in Chapter 7, BlobSeer provides efficient support for concurrently appending to files, even in presence of a high degree of concurrency. Implementing support for append in BSFS simply consists in mapping the append operation specified in the file-system inter-

face to the append operation on the corresponding BLOB in BlobSeer. An additional step is required at the level of BSFS, to update the file metadata to the new size after the append is successfully completed.

The second step of taking advantage of the append feature in the MapReduce context, is to use the operation at the level of the framework, i.e. when the reducers write their output data to the DFS. In the original Hadoop, the output data of the reducers is committed as final result in the DFS, by the FileOutputCommitter. This class simply moves the data produced by the reducer in its assigned temporary directory in the DFS, to the job output directory indicated by the user. The temporary directory may contain several files that are copied as such to the final directory. In consequence, the final result is split into multiple files, each representing a part of the application's output data.

The approach we propose in Chapter 7 consists in having all the reducers append data to a single file, instead of writing the data to many different files in the DFS. To this end, we modified the FileOutputCommitter class to invoke the append operation provided by the file-system interface, on the shared output file in the DFS. Thereby, running the MapReduce application results in a single output file stored in the DFS.

8.3 Automatic deployment tools

Apart from the implementation stage, a significant amount of effort was invested in building deployment tools for evaluation purposes. Given the complexity of the frameworks involved in our experiments that target a large-scale environment, we developed a scripting framework to automate and facilitate the evaluation phase. The scripts automatically deploy HDFS, BlobSeer, BSFS and the Hadoop MapReduce framework. After this stage is completed, the scripts launch tests and then collect the results.

Our deployment tools were designed such that they provide us with a framework that is *configurable*, enabling fine-tuning and straightforward customization of the deployment setup. Another important feature that is required from the testing framework is to ensure that the experiments can be performed *automatically* and thus, *repeatable*.

By the means of these tools, we were able to easily deploy and test all the systems required by our evaluation setup, on a platform consisting of hundreds of machines.

Part IV

Evaluation

Chapter 9

Evaluating BSFS as backend storage for MapReduce applications

Contents

CHAPTER 5 illustrates the benefits of using BlobSeer in the MapReduce context, and the integration with the Hadoop framework. In this chapter, we evaluate the impact of the BlobSeer File System by performing experiments both with synthetic microbenchmarks and with real MapReduce applications. The microbenchmarks consist of processes that access the storage layer directly using the file system interface, whereas MapReduce applications access the storage layer through the MapReduce framework.

9.1 Environmental setup

The experiments were carried out on the Orsay cluster of the Grid'5000 platform. Both the microbenchmarks and the MapReduce applications were performed using 270 nodes, on which we deployed both BSFS and HDFS. The nodes are connected through a 10 Gbit Ethernet network emulated over Myrinet, with a measured bandwidth for end-to-end TCP sockets of 527 MB/s. We selected the Orsay cluster for this set of experiments, as it provides us with a large, homogeneous environment for deploying our frameworks and performing accurate measurements.

For HDFS, we deployed the namenode on a dedicated machine and the datanodes on the remaining nodes (one entity per machine). For BSFS, we deployed one version manager,

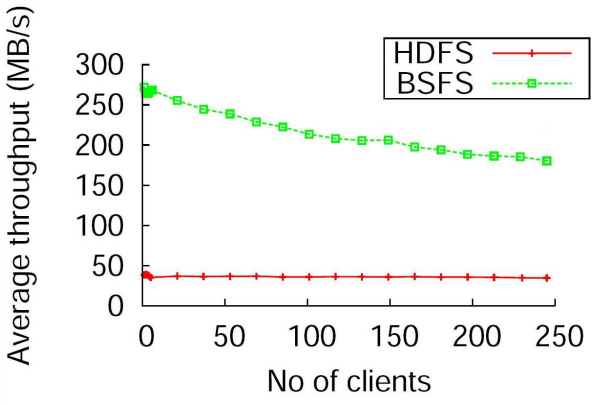


Figure 9.1: Performance of HDFS and BSFS when concurrent clients write to different files.

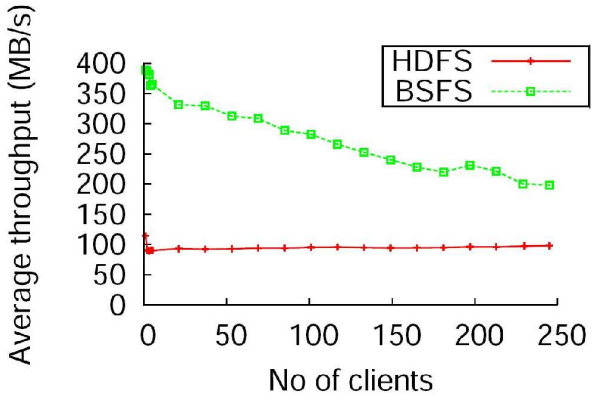


Figure 9.2: Performance of HDFS and BSFS when concurrent clients read different files.

one provider manager, one node for the namespace manager and 20 metadata providers. The remaining nodes were used as data providers. As HDFS handles data in 64 MB chunks, we set the page size to 64 MB as well at the level of BlobSeer.

9.2 Microbenchmarks

The goal of the microbenchmarks is to evaluate the throughput achieved by BSFS and HDFS when multiple, concurrent clients access the file systems, under several test scenarios. The scenarios we chose are common access patterns in MapReduce applications. For each microbenchmark we measure the average throughput achieved when multiple concurrent clients perform the same set of operations on the file system. The clients are launched simultaneously on the same machines as the datanodes (data providers, respectively). The number of concurrent clients ranges from 1 to 246. Each test is executed 5 times, for each set of clients.

Concurrent writers, each writing to a different file. In this test scenario, we start N clients that write to HDFS/BSFS concurrently. Each client writes a 1 GB file sequentially in chunks

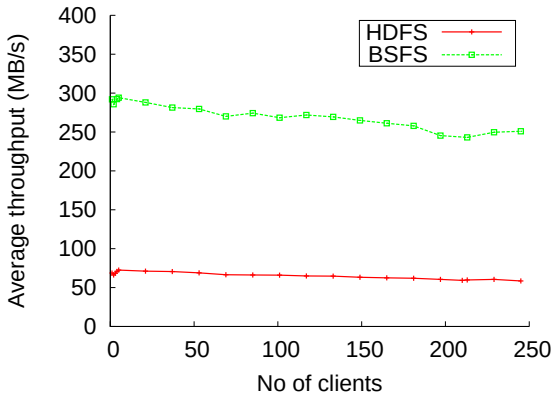


Figure 9.3: Performance of HDFS and BSFS when concurrent clients read different parts from the same file.

of 64 MB. This microbenchmark reproduces a pattern corresponding to a typical "reduce" phase of a MapReduce application, when the reduce tasks all generate the data which they write to different output files. Figure 9.1 shows the write performance of both HDFS and BSFS.
As expected, when a client writes on a machine where a datanode was started, HDFS's policy of writing the first copy (and the only one, in this case) locally, leads to a constant average throughput. BSFS achieves a significantly higher throughput than HDFS, which is a result of the balanced, round-robin block distribution strategy used by BlobSeer. A high throughput is sustained by BSFS even when the number of concurrent clients increases.

Concurrent readers, each reading from different files. In this experiment, N concurrent clients read each a different 1 GB file, sequentially in chunks of 64 MB. This test scenario with multiple concurrent readers, each processing a large file, corresponds to the "map" phase of a MapReduce application, when the mappers read the input files in order to parse the $(key, value)$ pairs.
As shown by the previous test, in the case of HDFS, writing a file on a datanode is performed locally. Thus, reading the file on the same datanode is also be performed *locally*. As far as BSFS is concerned, reading a file is always performed *remotely*, because the file is spread over several providers. In order to achieve a proper comparison, we configured HDFS so that the clients read files *remotely*. This is done by letting the files be stored by datanodes not co-deployed with them.
The average throughputs delivered by HDFS and BSFS in this test case are shown in Figure 9.2. Although HDFS reads remotely, the chunks read by a client are all stored by the same datanode. Since the reading is done sequentially, the datanode will serve the read the requests one at a time. For this reason, HDFS is able to maintain a constant throughput even when dealing with a large number of clients. In contrast, in BSFS, a provider has to serve read requests arriving concurrently from multiple clients. Although BSFS performs significantly better than HDFS, there is a decrease in the average throughput when the number of clients increases.

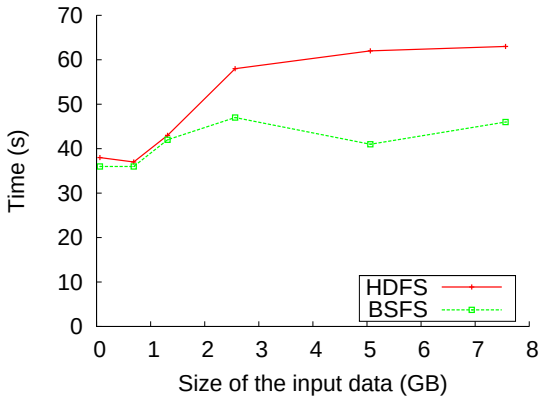


Figure 9.4: Sort - Job completion time.

Concurrent readers, each reading from the same file. This microbenchmark tests the performance of the file systems when concurrent clients read different (non-overlapping) parts from the same file. Each client reads a 64 MB chunk, starting from a unique offset in the shared file. For a configuration of N clients, the input shared file is $N \times 64$ MB of size. The file is created so that the N chunks are distributed among the datanodes/providers for both HDFS and BSFS.
The obtained results are shown on Figure 9.3. In BlobSeer, the shared file is uniformly striped among the providers, using a round-robin pattern. The average throughput delivered by BSFS is thus high. In contrast, HDFS uses a random data-layout policy which leads to load imbalance for datanodes when processing read requests: some of the datanodes get saturated with client requests.

9.3 Experiments with real MapReduce applications

We compared the performance of HDFS and BSFS when being used by the Hadoop framework to execute several MapReduce applications: *sort* and *grep*.

Sort is a standard MapReduce application that sorts key-value pairs. The key is represented by the first 10 bytes from each record, while the value is the remaining 100 bytes. This application is read intensive in the *map* phase and it generates a write-intensive workload in the *reduce* phase. The access patterns exhibited by this application are thus *concurrent reads from the same file* and *concurrent writes to different files*.

In addition to the deployment of HDFS and BSFS, the environmental setup in which this application was run also includes the entities belonging to the Hadoop framework: the jobtracker, deployed on a dedicated node, and the tasktrackers, co-deployed with the datanodes/providers. The input file processed by the application is stored in 64 MB chunks spread across the datanodes/providers. The Hadoop jobtracker starts a mapper to process each chunk from the input file. The input data was generated so as to vary the number of mappers from 1 to 121. This corresponds to an input file whose size varies from 64 MB to 8 GB. For each of these input files, we measured the job completion time when HDFS and BSFS are respectively used as storage layers.

Figure 9.4 displays the time needed by the application to complete, when increasing the size of the input file. When using BSFS as a storage layer, the Hadoop framework manages to finish the job faster than when using HDFS. These results are consistent with the ones delivered by the microbenchmarks. However, the impact of the average throughput when accessing a file in the file system is less visible in these results, as the job completion time includes not only file access time, but also the computation time and the I/O transfer time.

Chapter 10

Evaluating our approach for intermediate data management

Contents

THIS chapter is dedicated to the evaluation of the mechanisms we propose for managing intermediate data generated when executing MapReduce applications. In Chapter 6, we identify two types of intermediate data, according to the context in which it appears: between the "map" and "reduce" phases of the same MapReduce application, and between the jobs of a pipeline application. We present the validation of our respective contributions in those two contexts.

10.1 Intermediate data generated inside a job

In order to evaluate the benefits of using BlobSeer as storage for intermediate data in Hadoop, we performed experiments with two MapReduce applications in various scenarios. The purpose of our experiments is twofold: measure the impact of storing the intermediate data in a DFS, and also assess the benefits of using BSFS as the backend DFS. The first part of our goal is achieved by running real MapReduce applications through the Hadoop framework, with both the original and modified versions. Evaluating the gains of using BSFS as

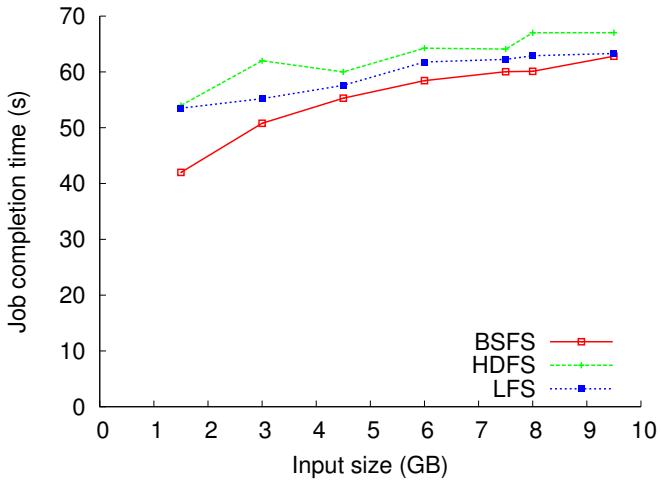


Figure 10.1: Distributed grep.

storage layer, is accomplished through a performance comparison of HDFS and BSFS when running MapReduce applications. The environmental setup as well as the experiments and the obtained results are further presented.

10.1.1 Environmental setup

The experiments were carried out on the Grid'5000 experimental platform. For our series of experiments we used the nodes in the Orsay cluster, with x86_64 CPUs and 2 GB of RAM for each node. Intra-cluster bandwidth is 10 Gbit/s provided by a Ethernet network emulated over Myrinet, with a measured bandwidth for end-to-end TCP sockets of 527 MB/s.

10.1.2 Experiments with MapReduce applications

Running real MapReduce applications involves deploying the distributed file systems (HDFS and BSFS) as well as the Hadoop MapReduce framework. The environmental setup consists of 170 nodes. For HDFS, we deploy the namenode on a dedicated, single machine and the datanodes on the rest of the nodes. For BSFS, each centralized entity (version manager, provider manager and namespace manager) is deployed on a single machine each, while the rest of the nodes are used for launching 10 metadata providers and data providers. In addition to deploying the file systems, one dedicated machine acted as the jobtracker, while the tasktrackers were co-deployed with the datanodes/providers.

We executed through the Hadoop framework 2 standard MapReduce applications: *sort* and *distributed grep*, as they are representative for workloads commonly encountered in the data-intensive community, and are often used for benchmarking purposes. For each of these applications, we measured the job completion time in 3 scenarios, corresponding to the 3 file systems that can be used for storing intermediate data:

- the local filesystem (LFS) of the mappers (as it is stored in the original Hadoop framework)

- HDFS (using our modified version of Hadoop allowing to store data in a DFS)
- BSFS (modified Hadoop framework)

Note that for the latter 2 test cases, the same DFS is used for storing both intermediate data and application-specific data (input and output files), whereas the first scenario is run with the original Hadoop framework with its default storage backend (HDFS). By comparing the original Hadoop framework with the modified one, both using HDFS as underlying storage, we analyze the impact of our approach and try to identify the class of MapReduce applications that could benefit from it. On the other hand, we also evaluate HDFS and BSFS when they are used for storing intermediate data, in addition to storing the input data supplied by the user and the output data generated by the application.

Distributed grep

This application is a distributed job that scans a huge text input file in order to find occurrences of a particular expression. The map function in this case, counts the number of times the expression appears and the reduce function sums up these counters and outputs the final result. We measured the job completion time in all 3 scenarios, when varying the input text to be scanned from 1.5 GB to 9.5 GB. The input file processed by the application is stored in 64 MB chunks spread across the datanodes/providers. The Hadoop jobtracker starts a mapper to process each chunk of the input file. Consequently the number of mappers to produce intermediate data ranges from 24 to 152 mappers. The amount of intermediate data written by the mappers and read by the reducers remains small, in the case of the grep application.

As can be seen on Figure 10.1, storing the intermediate data in HDFS leads to the highest execution time, while using BSFS for that purpose is the fastest of the 3 scenarios. The difference of runtime in the first 2 test cases (LFS and HDFS) is very small (of a few seconds) because of the fact that HDFS writes locally (data written on a datanode is stored on that datanode), which means that writing to HDFS a small amount of data is practically equivalent to writing to the local filesystem of the datanode/tasktracker. There is however, a small overhead when testing with HDFS, inferred by namespace management. As the microbenchmarks in Chapter 9 showed, BSFS delivers higher write/read throughput, therefore when used with our modified Hadoop framework, the job finishes faster. Again, since the generated intermediate data is small, the runtime accounts mostly for computation time, rather than I/O operations.

Distributed sort

The sort benchmark is a standard MapReduce application that sorts key/value pairs. The data generated for this test, consists of records each holding a key of 10 bytes, and a value of the remaining 100 bytes. The input data was generated so as to vary the number of mappers from 24 to 152. This corresponds to an input file whose size varies from 1.5 GB to 9.5 GB. For each of these input files, we measured the job completion time in the 3 scenarios previously described. The map function for this application extracts the 10-byte sorting key from each input record and emits the key and the value as the intermediate key/value pair. The reduce function is trivial in this case, as it simply passes the intermediate key/value pair unchanged as the output key/value pair; these final pairs are written to the DFS.

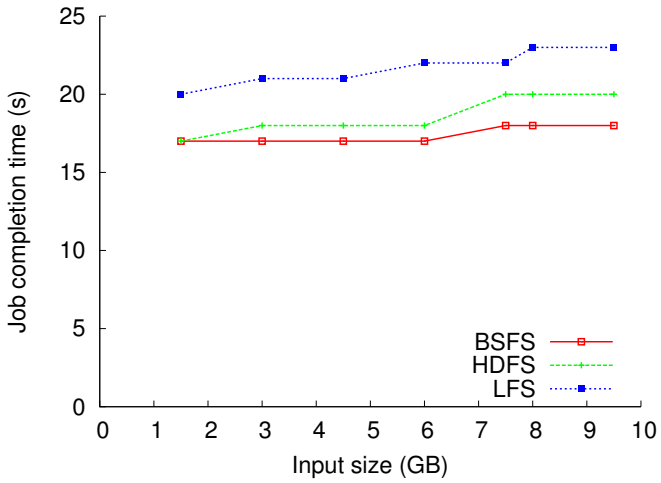


Figure 10.2: Distributed sort.

Figure 10.2 displays the time needed by the application to complete, when increasing the size of the input file. For this kind of applications with a trivial "reduce" phase, consisting in copying the map outputs to the DFS, our approach of storing the intermediate data in the DFS, allows us to run the MapReduce job without the "reduce" phase. In the case of applications that only process the input data, without any further aggregation, the intermediate data is the final output data; for this reason, the modified Hadoop framework completes the sorting job significantly faster both when running with HDFS and BSFS as storage.

10.2 Intermediate data generated between the jobs of a pipeline

In Chapter 6 we proposed an algorithm for dynamically creating "map" tasks that enables the Hadoop MapReduce framework to efficiently execute pipeline applications. In this chapter, we validate the proposed approach through a series of experiments that compare the original Hadoop framework with our modified version, when running pipeline MapReduce applications.

10.2.1 Environmental setup

The experimental platform comprises nodes from the Orsay cluster of the Grid'5000. Intra-cluster communication is done through a 1 Gbps Ethernet network. We performed an initial test at a small scale, i.e., 20 nodes, in order to assess the impact of our approach. The second set of tests involves 50 nodes belonging to the Orsay cluster.

10.2.2 Microbenchmarks

The evaluation presented here focuses on assessing the performance gains of the optimized MapReduce framework we propose, over the original one. To this end, we developed a benchmark that creates a pipeline of n MapReduce jobs and submits them to Hadoop for

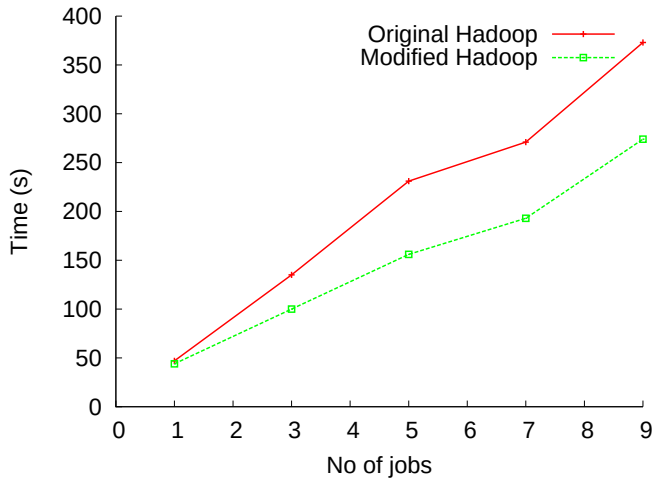


Figure 10.3: Completion time for short-running pipeline applications.

execution. Each job in the pipeline simulates a load that parses key-value pairs from the input data and outputs 90 % of them as final result. In this manner, we manage to obtain a long-running application that generates a large amount of data, allowing our dynamic scheduling mechanism to optimize the execution of the pipeline. The computation itself is not relevant in this case, as our goal is to create a scenario in which enough data chunks are generated along the pipeline so that "map" tasks can be dynamically created. We run this type of application first with the original Hadoop framework, then with our optimized version of Hadoop. In both cases, we measure the pipeline completion-time and compare the results.

In a first set of experiments, we run the benchmark in a small setup involving 20 nodes, on top of which HDFS and Hadoop MapReduce are deployed as follows: a dedicated machine is allocated for each centralized entity (namenode, jobtracker), a node serves as the Hadoop client that submits the jobs, and the rest of 17 nodes represent both datanodes and tasktrackers. At each step, we keep the same deployment setup and we increase the number of jobs in the pipeline to be executed. The first test consists in running a single job, while the last one runs a pipeline of 9 MapReduce jobs.

The application's input data, i.e., job_1's input data, consists of 5 data chunks (a total of 320 MB). Job_i keeps 90 % of the input data it received from job_{i-1}. In the case of the 9-job pipeline, this data-generation mechanism leads to a total of 2 GB of data produced throughout the pipeline, out of which 1.6 GB account for intermediate data.

Figure 10.3 shows the execution time of pipeline applications consisting of an increasing number of jobs (from 1 to 9), in two scenarios: when running on top of the original Hadoop, and with the pipeline-optimized version we proposed. In the first case, the client sequentially submits the jobs in the pipeline to Hadoop's jobtracker, i.e., waits for the completion of job_i before submitting job_{i+1}. When using our version of Hadoop, the client submits all the jobs in the pipeline from the beginning, and then waits for the completion of the whole application. As expected, the completion time in both cases increases proportionally to the number of jobs to be executed. However, our framework manages to run the jobs faster, as it creates and schedules "map" tasks as soon as a chunk of data is generated during the execu-

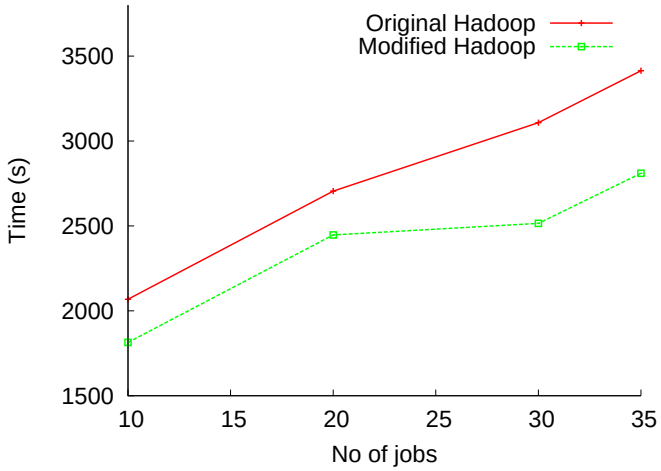


Figure 10.4: Completion time for long-running pipeline applications.

tion. This mechanism speeds-up the execution of the entire pipeline, and also exhibits a more efficient cluster utilization. Compared to the original Hadoop, we obtain a performance gain between 26 % and 32 %.

The first experiment we presented was focused on pipeline applications that consist of a small up to a medium number of jobs (1 to 9). Due to the long-running nature of pipeline applications and considering the significant size of the intermediate data our benchmark generates, we performed experiments with larger applications and larger datasets in a different setup, including 50 nodes. HDFS and Hadoop MapReduce are deployed as for the previous experiment, employing thus 47 tasktrackers. The size of the input data for each pipeline application amounts to 2.4 GB (40 data chunks). We vary the number of jobs to be executed in each pipeline, from 10 to 35. For the longest-running application, the generated data add up to a total of 24.4 GB.

The results for this setup are displayed on Figure 10.4. Consistently with the previous results, our approach proves to be more efficient for long-running applications as well. The performance gains vary between 9 % and 19 % in this scenario. The benefits of our optimized framework have a smaller impact in this case, because of the data size involved in the experiment. Since more chunks are used as input, and substantially more chunks are being generated throughout the pipeline, a large part of the tasktrackers is involved in the current computation, leaving a smaller number of resources available for dynamically running created "map" tasks.

In the context of pipeline applications, the number of nodes involved in the Hadoop deployment can have a substantial impact on completion time. Furthermore, considering our approach of dynamic scheduling "map" tasks, the scale of the deployment is an important factor to take into account. Thus, we performed an experiment in which we vary the number of nodes employed by the Hadoop framework. At each step, we increase the number of nodes used for the deployment, such that the number of tasktrackers that execute "map" and "reduce" tasks is varied from 10 to 45. In each setup, we run the aforementioned benchmark with a fixed number of 7 jobs in the pipeline. The input data is also fixed, consisting of 25 chunks of data, i.e., 1.5 GB.

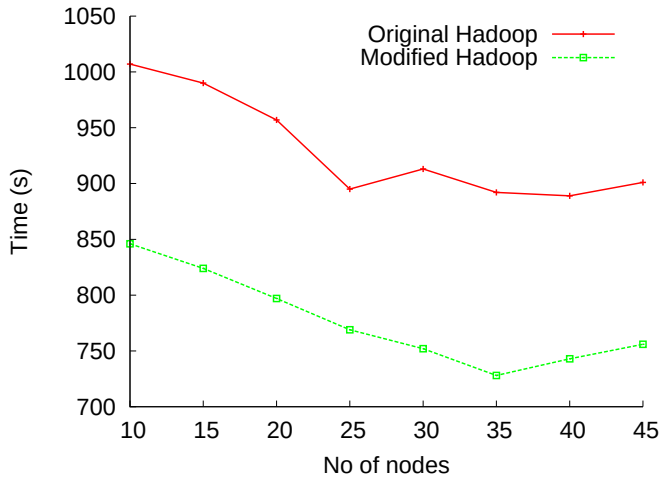


Figure 10.5: Impact of deployment setup on performance.

Figure 10.5 shows the completion time of the 7-job pipeline when running with both original Hadoop and modified Hadoop, while increasing the deployment setup. As the previous experiments also showed, our improved framework manages to execute the jobs faster than the original Hadoop. In both cases, as more nodes are added to the deployment, the application is executed faster, as more tasktrackers can be used for running the jobs. However, increasing the number of nodes yields performance gains up to a point, which corresponds to 25 tasktrackers for the original Hadoop. This number is strongly related to the number of chunks in the input data, since the jobtracker schedules a tasktracker to run the "map" computation on each chunk. For the modified Hadoop, the point after which expanding the deployment does not prove to be profitable any longer, is higher than for the original Hadoop. The reason for this behavior lies in the scheduling approach of both frameworks: in original Hadoop, the scheduling of jobs is done sequentially, while in modified Hadoop, the "map" tasks of each job are scheduled as soon as the data is generated. The completion time starts to increase for both frameworks after a certain point, as the overhead of launching and managing a larger number of tasktrackers overcomes the advantage of having more nodes for running the application.

Chapter 11

Evaluating the benefits of the append operation

Contents

In Chapter 7 we extended the Hadoop MapReduce framework to make use of the *append* operation supported at the file-system level. In this chapter, we evaluate the benefits that can be achieved through an efficient support for the append operation. To test the append functionality, we performed two types of experiments: at the level of the file system, and at the level of the Hadoop framework. The first type of experiments involve direct accesses to the file system, through the interface it exposes; we will further refer to these tests as *microbenchmarks*. The second class of experiments consists in running MapReduce applications, and thus in indirectly accessing the storage layer, through the MapReduce framework. The environmental setup, as well as the experiments and the obtained results, are further presented.

11.1 Environmental setup

The experiments were performed on the Grid'5000 testbed, more precisely, on the nodes of the Orsay cluster. These nodes are outfitted with dual-core x86_64 CPUs and 2 GB of RAM. Intra-cluster bandwidth is 10 Gbit/s provided by a Ethernet network emulated over Myrinet, with a measured bandwidth for end-to-end TCP sockets of 527 MB/s. Both the microbenchmarks and the MapReduce applications were performed using 270 nodes, on which

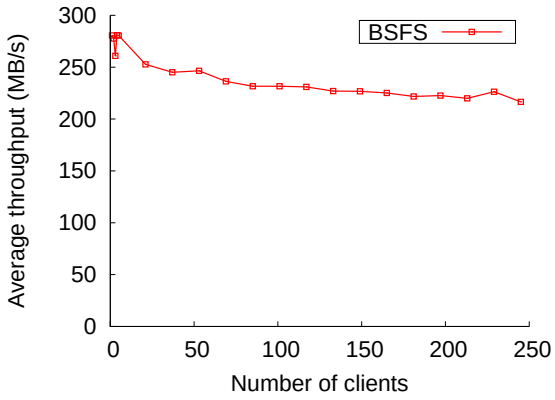


Figure 11.1: Performance of BSFS when concurrent clients append data to the same file.

we deployed both BSFS and HDFS. For HDFS, we deployed the namenode on a dedicated machine and the datanodes on the remaining nodes (one entity per machine). For BSFS, we deployed one version manager, one provider manager, one node for the namespace manager and 20 metadata providers. The remaining nodes were used as data providers. As HDFS handles data in 64 MB chunks, we also set the page size at the level of BlobSeer to 64 MB, to enable a fair comparison.

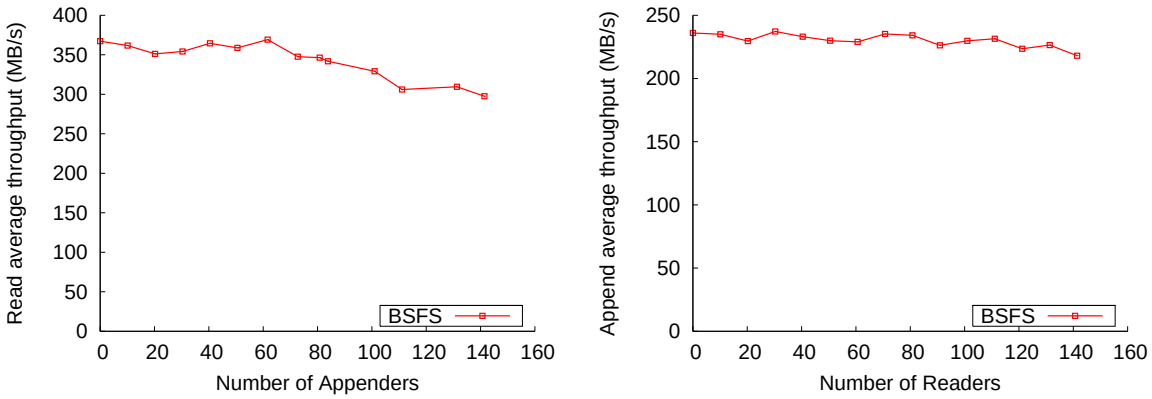
11.2 Microbenchmarks

The goal of the microbenchmarks is to evaluate the throughput achieved by BSFS when multiple, concurrent clients access the file systems, under several test scenarios. The scenarios we chose involve the append operation and represent access patterns exhibited by the MapReduce applications described in Chapter 7. For each microbenchmark we measure the average throughput achieved when multiple concurrent clients perform the same set of operations on the file system. The clients are simultaneously launched on the same machines as the datanodes (data providers, respectively). The number of concurrent clients ranges from 1 to 246. Each test is executed 5 times, for each set of clients.

The microbenchmarks were performed only for BSFS; since the append operation is not supported by HDFS, no comparison between HDFS and BSFS is possible.

Concurrent appends to the same file

In this test case, N concurrent clients append each a 64 MB chunk to the same file. The results are displayed on Figure 11.1. They show that BSFS maintains a good throughput as the number of appenders increases. This scenario illustrates the data access pattern exhibited by the modified Hadoop framework, in which all the reducers append their outputs to the same file, instead of creating many output files as it is done in the original version of Hadoop.



(a) Impact of concurrent appends on concurrent reads from the same file.

(b) Impact of concurrent reads on concurrent appends to the same file.

Figure 11.2: Concurrent readers and appenders.

Concurrent reads and appends to the same file

This access pattern with concurrent clients reading and appending to the same file corresponds to the case of MapReduce applications that can be executed in pipeline: the mappers of one application can read the data for processing, while the reducers of an application belonging to previous stages of the pipeline, can generate the data.

The test shown in Figure 11.2(a) assesses the performance of concurrent read operations from a shared file, when they are executed simultaneously with multiple appends to the same file. The test consists in deploying 100 readers and measuring the average throughput of the read operations for a number of concurrent appenders that ranges between 0 (only readers) and 140. Each reader processes 10 chunks of 64 MB and each appender writes 16 such chunks to the shared file. Each client processes disjoint regions of the file. The obtained results show that the average throughput of BSFS reads is sustained even when the same file is accessed by multiple concurrent appenders. As a consequence of the versioning-based concurrency control in BlobSeer, the appenders work on their own version of the file, and thus do not interfere with the older versions accessed by read operations.

Concurrent appenders maintain their throughput as well, when the number of concurrent readers from a shared file increases, as can be seen on Figure 11.2(b). In this experiment, we fixed the number of appenders to 100 and varied the number of readers accessing the same file from 0 to 140. Both readers and appenders access 10 chunks of 64 MB.

11.3 Application study

In order to evaluate how supporting the append operation influences the performance of the Hadoop framework when running a MapReduce application, we chose the *data join* application that is included in the contributions delivered with Yahoo!'s Hadoop release. The *data join* application is similar to the *outer join* operation from the database context. *Data join* takes as input two files consisting of key/value pairs, and merges them based on the keys from the first file that appear in the second file as well. The generated output consists

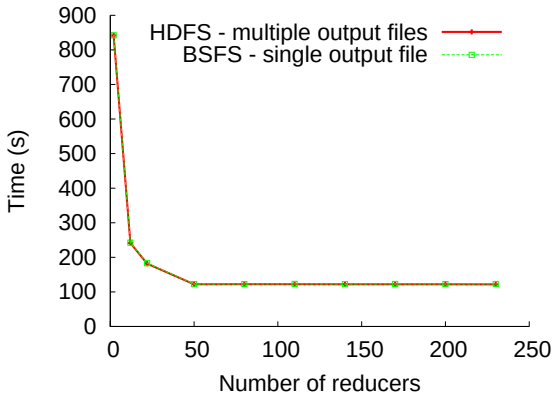


Figure 11.3: Completion time of the *data join* application when varying the number of reducers.

of 3 columns: the key from the first file and the values associated to the key in each of the files. If a key in the first file appears more than once in either one of the two files, the output will contain all the possible combinations. The keys that appear only in the first file are not included in the output.

Running the *data join* application involves deploying the distributed file systems (HDFS and BSFS) as well as the Hadoop MapReduce framework. The environmental setup is similar to the one described in 11.1; one dedicated machine acted as the jobtracker, while the tasktrackers were co-deployed with the datanodes/providers.

The input data consists of two files of 320 MB each; the input files contain key/value pairs extracted from the datasets made public by Last.fm. For the experiments, we kept the input data fixed, and we varied the number of reducers from 1 to 230. Since the Hadoop framework starts a mapper to process each input chunk, 10 concurrent mappers will perform the "map" phase of the application. The join operation performed on the input files, generates 6.3 GB of output data, written concurrently by the reducers, to the distributed file system. In this environmental setup, we ran the *data join* application in two scenarios:

The original Hadoop framework with HDFS as storage. With this setup, the number of output files is equal to the number of reducers. In the original Hadoop framework, each reducer writes its output to a different file in HDFS. Thus, the access pattern generated in the "reduce" phase corresponds to concurrent writes to different files.

The modified Hadoop framework with BSFS as storage. BSFS supports concurrent appends to the same file. By modifying the Hadoop framework to append the data generated by each reducer instead of writing it to a separate file, a single output file can be obtained by running the *data join* application in this context, without any intermediate step. The reducers act as concurrent appenders to the same file.

The results displayed in figure 11.3 show the completion time of the *data join* application in both of the scenarios previously described. BSFS finishes the job in approximately the same amount of time as HDFS, and moreover, it produces a single output file, by supporting the append operation. The completion time in both scenarios remains constant even

when the number of reducers increases, because *data join* a computation-intensive application, and most of the time is spent on searching and matching keys in the "map" phase, and on combining key-value pairs in the "reduce" phase.

Chapter 12

Evaluating the Cost of Running MapReduce Applications in the Cloud

Contents

In this chapter, we provide a cost evaluation of running MapReduce applications in the Cloud, by looking into several aspects: the overhead incurred by executing the job on the Cloud, compared to executing it on a Grid, the actual costs of renting Cloud resources, and also, the impact of the storage system used as backend by MapReduce applications. To evaluate all these factors, we run three MapReduce applications with the Hadoop framework in two environments: first on clusters belonging to the Grid'5000 [47] platform, then in a Nimbus [50] Cloud deployed on the Grid'5000 testbed. We then consider the payment scheme used by Amazon for the rental of their Cloud resources and we compute the cost of using our Nimbus deployment for running MapReduce applications. The evaluation presented in this chapter has been carried out in collaboration with Alexandra Carpen-Amarie. A summary of this work as well as additional results can be found in her thesis manuscript.

12.1 Motivation

Our goal is to assess and understand the overhead generated by the execution of a MapReduce computation in a Cloud. More specifically, there are two aspects that we address: we first analyze the *virtualization overhead* and the associated trade-offs when moving MapReduce applications to the Cloud; second, we examine several *storage options* that can be used as backend for MapReduce frameworks in the Cloud and their impact on the overall cost.

Virtualization overhead

One of the first aspects to consider when porting an application to the Cloud regards the potential benefits and gains, if any. Moreover, it is important to be able to assess if those benefits are worth the costs. Apart from the obvious advantages of the Cloud (huge processing and storage capabilities), there are some application-related issues to take into account when choosing the right environment for running that application. In most cases, there is a compromise to make between the *cost* of executing the application in the Cloud, and the gains expected to be obtained. Therefore, it is highly important to understand the requirements and features of MapReduce applications, in order to be able to tune the Cloud environment in an optimal manner, so that the right balance between cost and performance is struck.

The main advantage of using *virtualization* is that one can create a homogeneous environment comprising a substantial number of machines by using a considerably lesser number of physical machines. In this work, we consider various MapReduce applications, with the goal of assessing the impact of replacing the typical MapReduce execution environment, i.e. a physical cluster, with a virtualized one, i.e. Cloud resources.

Impact of storage on performance

Storage is another factor that may substantially impact the cost of running an application in the Cloud. It is therefore interesting to investigate the behavior of several distributed file systems as storage backend for MapReduce applications in the Cloud. Choosing the file system that suits the application best, becomes a crucial factor when considering the *pay-per-use* Cloud model. We provide a performance comparison of two distributed file systems when employed as storage for MapReduce applications. We also assess the performance of the BSFS file system presented in Chapter 5, when running in a virtualized environment.

12.2 Computational and cost model

For our cost-evaluation of running MapReduce applications in the Cloud, we chose the Amazon services as the basic model. Amazon's EC2 infrastructure as a service (IaaS) Cloud is the most widely-used and feature-rich commercial Cloud. The storage system introduced by Amazon, S3 [4], proposes a simple access interface that has become the IaaS standard for data transfers in and out of the Cloud.

Amazon EC2 allows users to rent compute or storage resources, in order to run their own applications. Typically, users first choose the type of virtual machine (VM) that suits their needs (application requirements, budget, etc.) and then boot the VM on multiple Amazon

resources, thus creating what is referred to as *instances* of that VM. Users are charged on a pay-per-use model that involves 3 types of costs:

Computational cost (CPU cost), that accounts for the VM type, the number of required instances and their use in EC2.

Data storage costs involve charges for persistently storing input and output data for the executed applications. We only focus on saving data directly into S3 objects, since existing storage alternatives, such as EBS [1] volumes, eventually rely on S3 for backup storage and introduce additional costs.

Data transfer charges include costs for moving data into and out of the Cloud. Data transfers between instances are free of charge, as well as transfers between S3 and the rented EC2 VMs.

To evaluate the *computational cost* of our experiments, we consider the *c1.medium* Amazon image type, as it meets two requirements: first, it is equivalent to the physical nodes we used when measuring the overhead of moving applications to the Cloud. Second, in [49], the authors show that the *c1.medium* image is the most cost-effective Amazon instance. The *c1.medium* instance is charged $0.19 per hour in the EU Amazon region and it features 1.7 GB of memory, 2 virtual cores and 350 GB of instance storage.

One important parameter that influences the cost-analysis of a Cloud application is the granularity at which the Cloud provider charges for resources. In the case of Amazon EC2, the rented instances are charged by the hour, assumption that may conceal the differences between storage backends or the benefits of adding resources to improve the runtime performance, when the execution lasts less than one hour. To better characterize the costs associated with our experiments, we assume per-second charges in our cost model, by dividing the hourly prices in Amazon EC2 by 3600.

As for the storage costs, we consider Amazon S3 charges for the EU region, i.e., $0.140 per GB for the first TB per month. Regarding the transfer costs, Amazon charges only for data transfers out of the Cloud, that is $0.12 per GB for data downloaded from the Cloud (download is free for less than 1 GB of output data). Some applications may need to persistently store all the input and output data in S3, as input data-sets may be processed several times by the application, and output results may be further refined. Besides storage costs, Amazon S3 also charges for HTTP requests, as follows: the price for PUT, COPY, POST, or LIST requests is $0.01 per 1,000 requests and GET requests are charged $0.01 per 10,000 requests.

12.3 Execution environment

To analyze the performance and costs of MapReduce applications, we performed experiments on two different platforms. To evaluate the pure performance of MapReduce access patterns, we relied on the Grid'5000 testbed that provides the typical execution environment for MapReduce. Then, we deployed the same MapReduce tests in an IaaS Cloud, namely the Nimbus [50, 9] open-source Cloud toolkit, deployed on top of the Grid'5000 testbed. As a MapReduce framework for running our computations, we chose Hadoop, the widely used open-source implementation of Google's MapReduce model.

Experimental platform

For both of our experimental environments (a physical cluster and the Cloud) we employed the clusters in Grid'5000. The first set of experiments aims to run MapReduce applications in a typical cluster environment. For this setup, we selected the Grid'5000 cluster in Orsay, i.e., 275 nodes outfitted with dual-core x86_64 CPUs and 2 GB of RAM. Intra-cluster communication is done through a 1 Gbps Ethernet network.

The second type of environment is Cloud-oriented: it was achieved by first deploying the Nimbus Cloud toolkit on top of physical nodes, and then by deploying VMs inside the obtained Nimbus Cloud. For these experiments, we used 130 nodes belonging to the Rennes site, and a VM type with features similar to the ones exhibited by the nodes from the first setup. Thus, we deployed VMs with 2 cores and 2 GB of RAM each.

In both setups, we created and deployed an execution environment for Hadoop comprising a dedicated node/VM for the jobtracker and another one for the namenode, while the rest of the nodes/VMs served as both datanodes and tasktrackers.

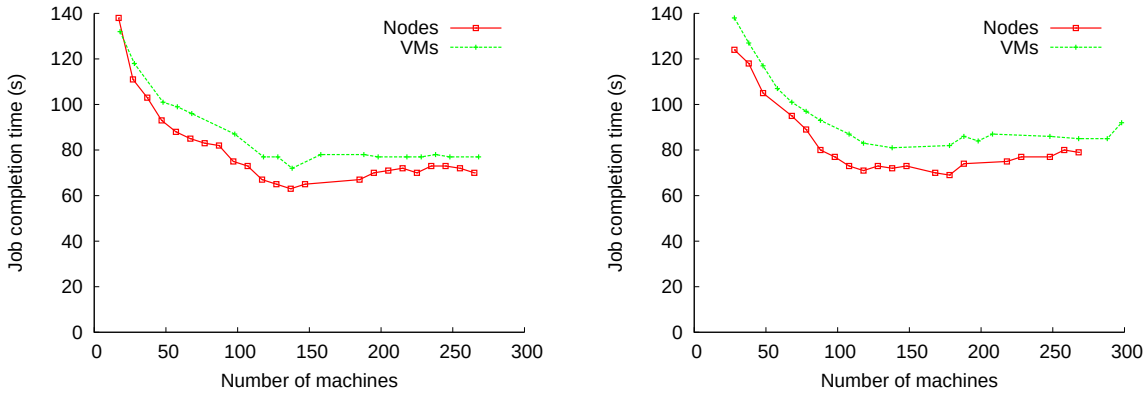
12.4 Results

This sections describes the experiments we performed in order to achieve the goals presented in section 12.1. We further discuss the obtained results, with respect to the factors we aim to evaluate.

Virtualization overhead

In order to assess the virtualization overhead, we compare Hadoop's performance when running in the two experimental environments previously described (grid cluster versus Cloud). In both scenarios, we run the same tests that imply measuring the runtime of the *grep* and *sort* applications. A test consists in deploying both HDFS and the Hadoop MapReduce framework on a number of nodes/VMs and then running the two MapReduce applications. For each test, we fix the input data size to 18.8 GB residing in HDFS. The number of nodes/VMs on top of which Hadoop is deployed, ranges from 20 to 280. By executing the same workload and keeping the same setup when running Hadoop on the Grid and the in the Cloud, we manage to achieve a fair comparison between the two environments, and thus, to evaluate the virtualization overhead.

Figure 12.1 shows the completion time of *grep* and *sort* when Hadoop runs on physical and then on virtual machines. The results show that the job completion time decreases in both environments and for both applications, as the deployment platform increases, i.e. more nodes/VMs are used. However, this gain in performance stabilizes when a certain number of machines is reached. In our case, for the *grep* application (Figure 12.1(a)) the execution time does not improve further when using more than 150 machines for the deployment. The explanation for this behavior comes from the size of the input data and the scheduling policy of Hadoop: the jobtracker launches a mapper to process each chunk of input data. Considering the fact that we process 300 chunks of input data (18.8 GB) and that by default, Hadoop executes 2 mappers per node/VM, the optimal number for running this workload is 150 machines. After this point, performance is almost the same. As *grep* is only



(a) Distributed Grep. (b) Distributed Sort.

Figure 12.1: Grid versus Cloud: completion time for Hadoop applications.

read-intensive and produces very little output data, the completion time accounts mostly for computation time.

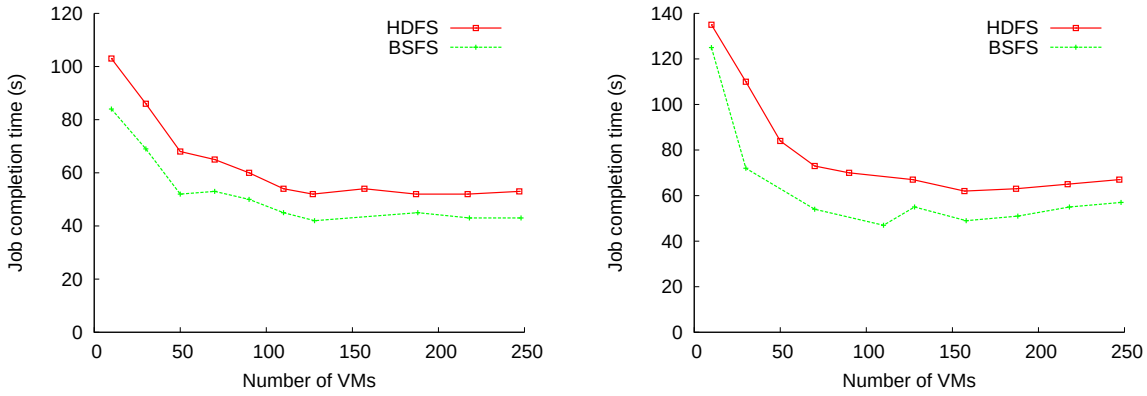
For the *sort* application, the optimal number of machines involved in the deployment is around 100, since the completion time accounts not only for computation, but also for writing the output data. *Sort* generates the same amount of output data as the given input data, thus a considerably large amount of time is spent in writing the final result to HDFS.

These tests also help us assess the overhead of porting the *grep* and *sort* applications to the Cloud. As the results show, the virtualization overhead is negligible especially when considering the major benefit provided by the Cloud through virtualization. With a much smaller number of nodes, we managed to create inside the Cloud the same setup as the testing environment provided by a large number of physical machines in the Grid (double, in our case).

Performance analysis

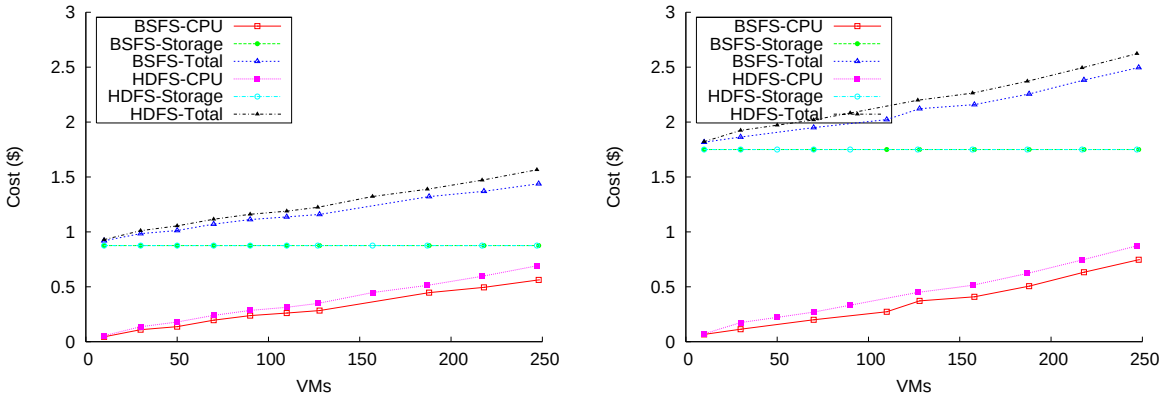
In a second set of experiments we evaluated the performance of two storage options for the *grep* and *sort* applications, by measuring their completion time in a Cloud environment. We increased the number of VMs comprising the virtual Hadoop cluster from 10 to 250. For each Hadoop backend, we repeated the experiments and we measured the application runtime. The input data amounted to 100 chunks, i.e., 6.25 GB, for each of the two applications. As storage backend, we evaluated the default storage option of Hadoop, HDFS, and the BSFS system we developed and presented in Chapter 5.

The results in Figure 12.2 show the execution runtime improves when more VMs are added to the cluster. This trend however becomes less steep when the number of VMs reaches 50 and thus the framework deploys a number of mappers equal to the number of input data chunks. In both experiments, the BSFS backend performs slightly better than HDFS, result consistent with our previous evaluations in local clusters. This improvement suggests that BSFS is able to deliver similar performance in both Cloud and Grid environments and is thus suitable for read-intensive workloads, such as *grep*, as well as for write-intensive ones, as shown by the *sort* results.



(a) Distributed Grep. (b) Distributed Sort.

Figure 12.2: Completion time when running Hadoop in the Cloud.



(a) Distributed Grep. (b) Distributed Sort.

Figure 12.3: Cost evaluation.

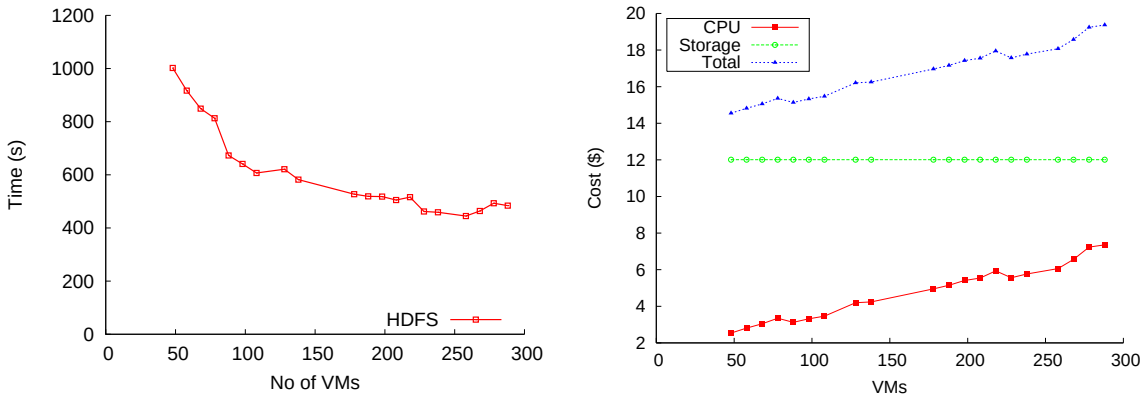
12.5 Cost evaluation

In this section, we compute various types of costs associated to running three MapReduce applications in the Cloud. The costs are calculated according to the model described in section 12.2.

Grep and Sort

We first evaluate the cost of running *grep* and *sort* applications in the Cloud. Since data transfers between S3 and the Cloud environment are free, we analyze only two of the types of costs detailed in section 12.2, namely the cost of computation and the cost of storing the data in S3.

Figure 12.3(a) shows the costs of running the *grep* application as a function of the number of virtual machines that act as Hadoop nodes, for the same input size as in the previous



(a) Completion time.

(b) Cost evaluation.

Figure 12.4: Pipeline MapReduce application.

experiment. The computational cost is computed as the number of VMs $\times$ the execution time $\times$ the cost of the VM instance per second. It increases as more virtual machines are provisioned, since the cost of deploying additional VMs outweighs the performance gain caused by the decreasing execution time.

The data-storage costs do not vary with the number of deployed VMs, since the input and output data are the same at each deployment setup. Both *grep* and *sort* process the same input data, that is 100 chunks of text files. While the result yielded by *grep* has a negligible size, the amount of data generated by *sort* is similar to its input. Figure 12.3 shows the cost estimations for running both applications in a Cloud environment. As expected, since running the applications with a BSFS backend is more efficient in terms of performance, the same behavior is exhibited for the computational cost, which directly depends on the time spent in computation. The storage cost, however, is independent of the used backend and it significantly impacts the total cost for both applications.

Pipeline MapReduce applications

To illustrate the characteristics and requirements of pipeline MapReduce applications, we developed a synthetic test which we then executed in a virtualized environment. Our synthetic application consists of 10 MapReduce jobs that are chained into a pipeline and executed with Hadoop using HDFS as storage. The computation performed by each job in the pipeline is trivial, as the "map " and "reduce" functions simply output key-value pairs. However, for our experiment, the computation itself is not relevant, as our goal is to execute a long-running pipeline application that generates large amounts of data. The input data consists of 200 chunks accounting for 12.5 GB. Each job in the pipeline parses key-value pairs from the input data and outputs 90% of them. This leads to a total amount of data to be stored in HDFS (input, intermediate and output data) of 85.8 GB.

Figure 12.4(a) shows the runtime for the 10-job pipeline, while the number of VMs ranges between 48 and 288. As expected, the pipeline is completed faster, as more machines are employed by Hadoop. At some point, the time spent in reading and writing a considerably large data size overcomes the advantage of having more VMs added to the deployment, and

thus, the completion time ceases to decrease. The costs associated with this application are displayed on Figure 12.4(b). The CPU cost is computed as previously mentioned for the *grep* and *sort* applications. However, for this scenario, the CPU cost is far higher and has a significantly greater impact on the total cost, as a pipeline MapReduce application is usually a long-running job. We also assess the cost of storing all the data generated throughout the pipeline, in S3. This scenario in which data is stored in S3 and then transferred to HDFS before performing the computation, also represents the typical usage of EC2. Since transferring data between S3 and VMs is free of charge, the total cost accounts only for computation and storage.

12.6 Related Work

Several studies have investigated the performance of various Cloud platforms and the costs and benefits of running scientific applications in such environments. Most evaluations focused on the Amazon's EC2 Cloud, as it has become the most popular Infrastructure-as-a-Service (IaaS) platform and has imposed its specific cost model to the Cloud computing community. In [73, 32, 44], the authors explored the trade-offs of running high-performance applications on EC2, showing that the Cloud environments introduce a significant overhead for parallel applications compared to local clusters. The cost of using HPC Cloud resources is discussed in [22], where the authors introduced a cost model for local resources and compared the computational cost of jobs against a Cloud environment. However, this work includes only a benchmark-based performance evaluation and no specific type of application is considered.

Recent works have focused on loosely-coupled applications, such as [30, 49, 19], in which the authors conducted a cost analysis of running scientific workflows in Cloud environments. They considered the performance penalties introduced by Cloud frameworks and evaluated computational and storage costs through simulations and experiments on EC2 and a local HPC cluster. More in-depth studies have investigated data storage in Clouds, evaluating the Amazon S3 service through data-intensive benchmarks [65]. Moreover, Paper [48] evaluated several file systems as Cloud storage backends for workflow applications, emphasizing running times and costs for each backend. The work in [53] conducted a comparative evaluation of Cloud platforms against Desktop Grids. They examined performance and cost issues for specific volunteer-computing applications and discussed hybrid approaches designed to improve cost effectiveness.

In [42], the authors introduced the AzureMapReduce platform and conducted a performance comparison of several commercial MapReduce implementations in Cloud environments. The analysis included scalability tests and cost estimations on two MapReduce applications.

Part V

Conclusions and future work

Chapter 13

Conclusions

DATA-INTENSIVE applications are nowadays, widely used in various domains to extract and process information, to design complex systems, to perform simulations of real models, etc. These applications exhibit challenging requirements in terms of both storage and computation. In the context of data-intensive applications, we focus on the MapReduce programming paradigm and its implementations. Introduced by Google, the MapReduce abstraction has revolutionized the data-intensive community and has rapidly spread to various research and production areas. An open-source implementation of Google's abstraction was provided by Yahoo! through the Hadoop project. This framework is considered the reference MapReduce implementation and is currently heavily used for various purposes and on several infrastructures.

The second part of the context in which this work was carried out, concerns *large-scale infrastructures*. As a validation for the contribution proposed in this thesis manuscript, we targeted two environments that are most-commonly used for executing data-intensive applications. The first platform that we focused on is the Grid, a well-established approach to distributed computing. In a second phase, we considered another environment for our work, Cloud computing, a recently-emerged model with a continuously-growing popularity.

In this manuscript, we addressed the problem of efficiently managing data processed and produced by MapReduce applications, on infrastructures distributed at large scales. The contribution of our work can be organized in three main research direction:

Designing a concurrency-optimized file system for MapReduce frameworks. We brought a first contribution at the level of the storage layer for MapReduce applications. In order to propose an efficient solution to the challenges raised by storing data processed by MapReduce computations, we first investigated the specific features of MapReduce applications that have an impact on the storage backend. We also discussed several file systems belonging to various communities, that are successfully used as storage for frameworks executing MapReduce jobs. Starting from the characteristics of

MapReduce applications and the limitations of existing storage solutions, we proposed a *concurrency-optimized file system for MapReduce frameworks*. Our solution is based on the BlobSeer data-management system. We built the *BlobSeer File System (BSFS)* with the goal of providing high throughput under heavy concurrency to MapReduce applications. BSFS can be used as a stand-alone file system that can be directly accessed through a file-system interface. We further integrated BSFS with the Hadoop MapReduce framework, the reference MapReduce open-source implementation.

To validate our proposal, we performed intensive experiments on the Grid'5000 platform. We thus showed that the proposed storage layer, BSFS, manages to deliver and sustain a significantly higher throughput than the default storage backend of Hadoop, HDFS. The various experiments we performed exhibit high concurrency to shared files and were carried out at large scales, on clusters of up to 300 machines. We developed a set of synthetic benchmarks to directly compare BSFS with HDFS when the file systems are accessed in various patterns. At the level of the Hadoop MapReduce framework, we evaluated the benefits that BSFS brings to real MapReduce applications. We observed that BSFS is able to complete the job's execution in substantially less amount of time than HDFS.

Optimizing intermediate data management in MapReduce computations. A second part of our contribution was dedicated to the management of a special type of data produced by MapReduce computations, i.e., *intermediate data*. We studied the characteristics of intermediate data in general, and we discussed the way it is handled in MapReduce frameworks. Our work addressed intermediate data at two levels: inside the same MapReduce job, and during the execution of pipeline applications.

We focused first on efficiently managing intermediate data generated between the "map" and "reduce" phases of MapReduce computations. In this context, we proposed to store the intermediate data in the distributed file system used as underlying backend. In this direction, we investigated the features of intermediate data in MapReduce computations and we proposed a new approach consisting in storing this kind of data in a DFS. The major benefit of this approach is better illustrated when considering failures. Existing MapReduce frameworks store intermediate data on nodes local disk. In case of failures, intermediate data produced by mappers can no longer be retrieved and processed further by reducers. the solution of most frameworks is to reschedule the failed tasks and to re-generate all the intermediate data that was lost because of failures. This solution is costly in terms of additional execution time. With our approach of storing intermediate data in a DFS, we avoid the re-execution of tasks in case of failures that lead to data loss. As storage for intermediate data, we considered BSFS as being a suitable candidate for providing for the requirements of intermediate data: availability and high I/O access. The tests we performed in this context, measured the impact of using a DFS as storage for intermediate data instead of the local-disk approach. We then assessed the performance of BSFS and HDFS when serving as storage for intermediate data produced by several MapReduce applications.

We then considered another type of intermediate data that appears in the context of *pipeline MapReduce applications*. In order to speed-up the execution of pipeline MapReduce applications (applications that consist of multiple jobs executed in a pipeline) and also, to improve cluster utilization, we proposed an optimized Hadoop MapReduce

framework, in which the scheduling is done in a dynamic manner. We introduced several optimizations in the Hadoop MapReduce framework in order to improve its performance when executing pipelines. Our proposal consisted mainly in a new mechanism for creating tasks along the pipeline, as soon as the tasks' input data becomes available. As our evaluation showed, this dynamic task scheduling leads to an improved performance of the framework, in terms of job completion time. In addition, our approach ensures a more efficient cluster utilizations, with respect to the amount of resources that are involved in the computation.

Enabling and leveraging the append operation in Hadoop. In a third contribution of our work, we investigated ways to instrument the features of BSFS in order to enable extensions to the de facto MapReduce implementation, Hadoop. Starting from the limitations of existing approaches, we identified the append operation as having valuable potential not only in the MapReduce context. We provided several examples of scenarios where the append operation is needed at the file-system level. Tests at the file-system level showed that BSFS provides efficient support for the append operation by delivering high throughput when multiple clients concurrently append data to the same file. Since append is supported by the file system, we modified the Hadoop MapReduce framework to take advantage of this functionality. In our modified Hadoop framework, the reducers append their data to a single file, instead of writing it to a separate file, as it was done in the original version of Hadoop. The benefits of our approach are obvious in terms of simplicity: at the end of the computation, data is already available in a single logical file. This file is ready to use for any subsequent processing. No extra application logic is needed for subsequent processing, in contrast to the original Hadoop, which has to explicitly handle a (potentially large) group of files.

The implementation and evaluation of the aforementioned contributions required a considerable amount of effort. The main challenges that we had to overcome arose from several factors: the complexity of the Hadoop project that we had to study in order to validate our work and to introduce several extensions, the targeted platforms on which we experimented, the Grid and the Cloud, that both involved a substantial scripting phase, the difficulties entailed by building an experimental environment consisting of various, complex systems.

In our experiments, we aimed at measuring the *scalability* of our approaches. We also focused on observing how our solutions behaved in stress conditions of *high concurrency*. We started by thoroughly testing the proposed solutions on Grid'5000 and then we addressed the Cloud environment, with the help of the Nimbus toolkit. In the Cloud context, we proposed a cost-evaluation of MapReduce applications, in terms of computation and storage. By considering the existing storage options, we provided a performance evaluation of running MapReduce jobs in the Cloud. We also modeled the costs of executing different types of MapReduce applications when considering Amazon ECs costs.

Chapter 14

Future work

THE work presented in this manuscript comprises several proposals that aim at optimizing data management in MapReduce computations. To this end, we designed and implemented systems and extensions that were then validated on Grid and Cloud infrastructures. During the achievement of this work, several open-issues emerged, issues that require a more profound investigation and that can lead to further optimizations and extensions. We detail below several research directions brought forth by this work.

Exposing and leveraging versioning at the file-system level. The versioning mechanism employed by BlobSeer could be exposed by BSFS's interface. Applications accessing data stored in BSFS can be re-designed to take advantage of versioning in their workflow. An interesting example refers to pipeline MapReduce applications that could instrument versioning to enable parallel execution of stages.

Optimizing Hadoop's scheduling policies in case of failures. The contribution we presented related to intermediate data management can be employed by Hadoop's jobtracker to deal with failures in an efficient way. Since the intermediate data is stored in the underlying DFS, when failures occur, the application can carry on with a normal execution, instead of repeating the failed computations. This involves extensions to Hadoop's scheduling mechanisms, to permit computations to resume on other nodes, starting from the intermediate data generated up to the point of failure. This aspect needs careful investigation of the scheduling policy Hadoop uses, as well as a validation phase that involves injecting failures and triggering the job-recovery process.

Experiments with a wider range of MapReduce applications. The features supported by BSFS can bring substantial benefits to other domains that use MapReduce processing. For instance, MapReduce-based image processing is a challenging area that crunches huge amounts of data to execute complex computations. Image-processing applications can benefit from the high throughput BSFS delivers.

Integration with other data-processing frameworks. Both BSFS and our contribution in the context of pipeline applications can be validated with higher-level frameworks, such as Pig and HBase.

Optimizing MapReduce execution in the Cloud. Various Cloud models imply different interfaces, features and costs. By looking into each of the aspects that are specific to different types of Clouds, we can tune both the application and the storage so that an optimal environment is delivered. Another future direction that is worth considering is supplying mechanisms that enable a cost-effective execution of MapReduce applications in the Cloud. Techniques such as performance models, dynamic tuning of storage in compliance with applications needs, etc., are tools we can utilize in this direction.

Improving proposed systems. Further improvements can be performed for each of the contributions. For example, the BSFS's namespace manager is currently centralized. A distributed approach would be welcomed at this level, as it would enable better-scaling capabilities. Also at the level of BSFS, we are considering several chunk-allocation strategies, in addition to the round-robin policy proposed by BlobSeer. For instance, a location-aware approach like the one employed by HDFS could prove to be beneficial in deployment models where clients are co-located with storage nodes.

Bibliography

[1] Amazon Elastic Block Store (EBS). http://aws.amazon.com/ebs/.

[2] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/.

[3] Amazon Elastic MapReduce. http://aws.amazon.com/elasticmapreduce/.

[4] Amazon Simple Storage Service (S3). http://aws.amazon.com/s3/.

[5] Cloudstore distributed file system (formerly, Kosmos file system). http://kosmosfs.sourceforge.net/index.html.

[6] HBase. The Hadoop Database. http://hadoop.apache.org/hbase/.

[7] HDFS. The Hadoop Distributed File System. http://hadoop.apache.org/common/docs/r0.20.1/hdfs_design.html.

[8] The Kadeploy project. http://kadeploy.imag.fr/.

[9] The Nimbus project. http://www.nimbusproject.org/.

[10] The OAR project. http://oar.imag.fr/.

[11] Parallel Virtual File System, Version 2. http://pvfs2.org/.

[12] PoweredBy Hadoop. http://wiki.apache.org/hadoop/PoweredBy.

[13] Science Clouds. http://www.scienceclouds.org/.

[14] Sorting 1PB with MapReduce. http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html.

[15] The Apache Hadoop Project. http://www.hadoop.org.

[16] The Hadoop MapReduce Framework. http://hadoop.apache.org/mapreduce/.

[17] The Windows Azure Platform. http://www.microsoft.com/windowsazure/.

[18] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[19] G. Bruce Berriman, Ewa Deelman, Gideon Juve, Moira Regelson, and Peter Plavchan. The application of cloud computing to astronomy: A study of cost and performance. *CoRR*, abs/1010.4813:1–7, 2010. informal publication.

[20] Roy Bragg. Cloud computing: When computers really rule. *Tech News World, Electronic Magazine, available at http://www.technewsworld.com/story/63954.html.*, July 2008.

[21] Rajkumar Buyya, Chee S. Yeo, and Srikumar Venugopal. Market-oriented cloud computing: Vision, hype, and reality for delivering IT services as computing utilities. In *Department of Computer Science and Software Engineering (CSSE), The University of Melbourne, Australia. He*, pages 10–1016, 2008.

[22] Adam G. Carlyle, Stephen L. Harrell, and Preston M. Smith. Cost-effective HPC: The community or the cloud? In *Proceedings of the 2010 IEEE 2nd International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 169–176, Washington, DC, USA, 2010. IEEE Computer Society.

[23] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[24] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The Data Grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23:187–200, 1999.

[25] Benoit Claudel, Guillaume Huard, and Olivier Richard. TakTuk, adaptive deployment of remote executions. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pages 91–100, New York, NY, USA, 2009. ACM.

[26] Gianmarco De Francisci Morales, Aristides Gionis, and Mauro Sozio. Social content matching in MapReduce. *Proceedings of VLDB Endowment*, 4:460–469, April 2011.

[27] Jeff Dean. Designs, Lessons and Advice from Building Large Distributed Systems, Keynote. Keynote at The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS), Big Sky, MT, October 2009.

[28] Jeffrey Dean. Experiences with MapReduce, an abstraction for large-scale computation. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 1–1, New York, NY, USA, 2006. ACM.

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

[30] Ewa Deelman, Gurmeet Singh, Miron Livny, Bruce Berriman, and John Good. The cost of doing science on the cloud: the Montage example. In *Supercomputing'08*, SC '08, pages 50:1–50:12, Piscataway, NJ, USA, 2008. IEEE Press.

[31] Thomas A. DeFanti, Ian Foster, Michael E. Papka, Rick Stevens, and Tim Kuhfuss. Overview of the I-WAY: wide-area visual supercomputing. *International Journal of Supercomputer Applications and High Performance Computing*, 10(2/3):123–131, Summer/Fall 1996.

[32] Constantinos Evangelinos and Chris N. Hill. Cloud Computing for parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2. Cloud Computing and Its Applications, October 2008.

[33] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, 2002.

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

[35] Ian Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779*, pages 2–13, 2005.

[36] Ian Foster, Jonathan Geisler, Bill Nickless, Warren Smith, and Steven Tuecke. Software infrastructure for the i-way high-performance distributed computing experiment. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, HPDC '96, pages 562–, Washington, DC, USA, 1996. IEEE Computer Society.

[37] Ian Foster and Carl Kesselman, editors. *The Grid: blueprint for a new computing infrastructure.* Morgan Kaufmann, San Francisco, CA, USA, January 1998.

[38] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: enabling scalable virtual organizations. *Supercomputer Applications*, 15(3):200–222, March 2001.

[39] Yasser Ganjisaffar, Thomas Debeauvais, Sara Javanmardi, Rich Caruana, and Cristina Videira Lopes. Distributed tuning of machine learning algorithms using MapReduce clusters. In *Proceedings of the 3rd Workshop on Large Scale Data Mining: Theory and Applications*, LDMTA '11, pages 2:1–2:8, New York, NY, USA, 2011. ACM.

[40] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *SIGOPS - Operating Systems Review*, 37(5):29–43, 2003.

[41] Galen Gruman and Eric Knorr. What cloud computing really means. *InfoWorld, Electronic Magazine, available at http://www.infoworld.com/article/08/04/07/15FE-cloudcomputing-reality_1.html.*, April 2008.

[42] Thilina Gunarathne, Tak-Lon Wu, Judy Qiu, and Geoffrey Fox. MapReduce in the Clouds for Science. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 565–572, Washington, DC, USA, 2010. IEEE Computer Society.

[43] Brian Hayes. Cloud computing. *Communications of the ACM*, (7):9–11, July 2008.

[44] Zach Hill and Marty Humphrey. A quantitative analysis of high performance computing with Amazon's EC2 infrastructure: The death of the local cluster? In *Grid Computing, 2009 10th IEEE/ACM International Conference on*, pages 26–33. IEEE, October 2009.

[45] Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data management in an international Data Grid Project. pages 77–90. Springer-Verlag, 2000.

[46] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[47] Yvon Jégou, Stephane Lantéri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Touche Iréa. Grid'5000: a large scale and highly reconfigurable experimental Grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, November 2006.

[48] Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, Benjamin P. Berman, Bruce Berriman, and Phil Maechling. Data Sharing Options for Scientific Workflows on Amazon EC2. In *Supercomputing'10*, SC '10, pages 1–9, Washington, DC, USA, 2010. IEEE Computer Society.

[49] Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, Bruce Berriman, Benjamin P. Berman, and Phil Maechling. Scientific Workflow Applications on Amazon EC2. *2009 5th IEEE International Conference on EScience Workshops*, pages 59–66, 2010.

[50] Kate Keahey and Tim Freeman. Science clouds: Early experiences in cloud computing for scientific applications. In *CCA '08: Cloud Computing and Its Applications*, Chicago, IL, USA, 2008.

[51] Kiyoung Kim, Kyungho Jeon, Hyuck Han, Shin-Gyu Kim, Hyungsoo Jung, and Heon Y. Yeom. MRBench: A Benchmark for MapReduce Framework. In *Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*, pages 11–18, Washington, DC, USA, 2008. IEEE Computer Society.

[52] Steven Y. Ko, Imranul Hoque, Brian Cho, and Indranil Gupta. On availability of intermediate data in cloud computations. In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS'09, pages 6–6, Berkeley, CA, USA, 2009. USENIX Association.

[53] Derrick Kondo, Bahman Javadi, Paul Malecot, Franck Cappello, and David P. Anderson. Cost-benefit analysis of cloud computing versus desktop grids. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[54] Alexander Lenk, Markus Klems, Jens Nimis, Stefan Tai, and Thomas Sandholm. What's inside the cloud? an architectural map of the cloud landscape. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, CLOUD '09, pages 23–31, Washington, DC, USA, 2009. IEEE Computer Society.

[55] Cecchi Marco, Capannini Fabio, Dorigo Alvise, Ghiselli Antonia, Giacomini Francesco, Maraschini Alessandro, Marzolla Moreno, Monforte Salvatore, Pacini Fabrizio, Petronzio Luca, and Prelz Francesco. The gLite workload management system. In

GPC '09: Proceedings of the 4th International Conference on Advances in Grid and Pervasive Computing, pages 256–268, Berlin, Heidelberg, 2009. Springer-Verlag.

[56] Andrea Matsunaga, Mauricio Tsugawa, and Jose Fortes. CloudBLAST: Combining MapReduce and virtualization on distributed resources for bioinformatics applications. *eScience, IEEE International Conference on*, 0:222–229, 2008.

[57] Kirk McKusick and Sean Quinlan. Gfs: Evolution on fast-forward. *Communications of the ACM*, 53(3):42–49, 2010.

[58] Rohith K. Menon, Goutham P. Bhat, and Michael C. Schatz. Rapid parallel genome indexing with MapReduce. In *Proceedings of the 2nd international workshop on MapReduce and its applications*, MapReduce '11, pages 51–58, New York, NY, USA, 2011. ACM.

[59] Rafael Moreno-Vozmediano, Ruben S. Montero, and Ignacio M. Llorente. Elastic management of cluster-based services in the cloud. In *ACDC '09: Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 19–24, New York, NY, USA, 2009. ACM.

[60] Andrew Y. Ng, Gary Bradski, Cheng-Tao Chu, Kunle Olukotun, Sang Kyun Kim, Yi-An Lin, and YuanYuan Yu. MapReduce for machine learning on multicore. In *NIPS*, December 2006. Selected for Oral Presentation.

[61] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. BlobSeer: How to enable efficient versioning for large object storage under heavy access concurrency. In *Data Management in Peer-to-Peer Systems*, St-Petersburg, Russia, 2009. Workshop held within the scope of the EDBT/ICDT 2009 joint conference.

[62] Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, Diana Moise, and Alexandra Carpen-Amarie. BlobSeer: Next-generation data management for large scale infrastructures. *Journal of Parallel and Distributed Computing*, 71:169–184, February 2011.

[63] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[64] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD*, pages 1099–1110, New York, NY, USA, June 2008. ACM.

[65] Mayur R. Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon S3 for science grids: a viable solution? In *Proceedings of the 2008 international workshop on Data-aware distributed computing*, DADC '08, pages 55–64, New York, NY, USA, 2008. ACM.

[66] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal*, 13:277–298, October 2005.

[67] Mathilde Romberg. The UNICORE grid infrastructure. *Scientific Programming*, 10(2):149–157, 2002.

[68] Dan Sanderson. *Programming Google App Engine: Build and Run Scalable Web Apps on Google's Infrastructure*. O'Reilly Media, Inc., 2009.

[69] Michael C. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[70] Frank B. Schmuck and Roger L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 231–244. USENIX Association, 2002.

[71] Ashish Thusoo, Joydeep S. Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a MapReduce framework. In *Proceedings of the 35th conference on Very Large Databases (VLDB'09)*, pages 1626–1629, 2009.

[72] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Computer Communication Review*, 39:50–55, December 2008.

[73] Edward Walker. Benchmarking Amazon EC2 for high-performance scientific computing. *LOGIN*, 33(5):18–23, October 2008.

[74] Guanying Wang, Ali R. Butt, Prashant Pandey, and Karan Gupta. A simulation approach to evaluating design decisions in MapReduce setups. In *MASCOTS*, pages 1–11. IEEE, 2009.

[75] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.

[76] Keith Wiley, Andrew Connolly, Jeffrey P. Gardner, Simon Krughof, Magdalena Balazinska, Bill Howe, YongChul Kwon, and Yingyi Bu. Astronomy in the cloud: Using MapReduce for image coaddition. *CoRR*, abs/1010.1015, 2010.

[77] Garth Gibson Wittawat Tantisiriroj, Swapnil Patil. Data-intensive file systems for internet services: A rose by any other name... Technical Report UCB/EECS-2008-99, Parallel Data Laboratory, October 2008.

[78] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1029–1040, New York, NY, USA, 2007. ACM.

[79] Changxi Zheng, Guobin Shen, Shipeng Li, and Scott Shenker. Distributed Segment Tree: Support range query and cover query over DHT. In *Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS)*, 2006.