Immediate data management for map/reduce applications

Thi-Thu-Lan Trieu

Intermediate Data Management for Map/Reduce Applications

Master Thesis

Thi-Thu-Lan TRIEU
thi-thu-lan.trieu@irisa.fr

Supervisors: **Luc Bougé, Gabriel Antoniu, Diana Moise**
Luc.Bouge@bretagne.ens-cachan, {Gabriel.Antoniu, Diana.Moise}@irisa.fr

ENS de Cachan, INRIA/IRISA, KerData Project-Team

June 4, 2010

Abstract

Map/Reduce is a popular programming model and an associated implementation for processing large data sets nowadays. This report aims to present the problem of managing intermediate data which is generated during Map/Reduce computations. We focus on the Hadoop Map/Reduce framework and two file systems, Hadoop Distributed File System and BlobSeer File System, used as storage backends of Hadoop Map/Reduce framework, which substitute for the original intermediate storage layer. As a result, our new design deals with data reliability and efficiency, thanks to BlobSeer which supports access concurrency. The prototype has been experimented on the Grid'5000 testbed, using up to 150 nodes.

Keywords: Map/Reduce, Hadoop Map/Reduce Framework, Hadoop Distributed File System, BlobSeer File System, Intermediate Data Management, Access Concurrency.

Contents

1 Introduction

1.1 Motivation

Data management capacity and capability have made significant progress lately. From local data management with low concurrency control, sequential access in the past, we have arrived now at distributed data management, parallel access and high concurrency. This is the result of the increasing demand of data processing, where performance is directly tied to the data management techniques employed. Nowhere is this progress illustrated more vividly as in the area of online search where an enormous amount of data is constantly collected, processed and managed by search engines. Without advanced data management, the ease with which we search for information and navigate the vast cyber world would not have been possible, let alone taken for granted as we often do today.

The challenges facing online search engine are actually twofold. Not only does it have to manage the huge amount of data efficiently, it must also do that at a reasonable cost to be commercially viable. The answer lies in the search engine's ability to reduce the amount of data to process via a technique pioneered and made famous by Google and Yahoo, Map/Reduce, which allows the search engine to express user queries in the form of simple computations hiding the messy details of parallelization, fault-tolerance, data distribution and load balancing. Consequently, this model becomes more and more popular as a solution for rapid implementation of distributed data-intensive applications. Furthermore, Hadoop Map/Reduce framework, known as an open source implementation of Map/Reduce, which facilitates ease of programming and high availability and reliability, has been successfully used to program data parallel applications.

The programming design of Map/Reduce is straight-forward, consisting of two stages: Map stage and Reduce stage in sequence. In this model, there are three types of data: input data, output data and intermediate data; intermediate data is produced as output from one stage and used as input for the next stage. In a traditional Map/Reduce framework's design, the most popular approach for intermediate data management relies on the Local File System. Failures are handled by the frameworks themselves without much assistance from the storage system. If and when a failure occurs, affected tasks are typically re-executed to regenerate intermediate data. Despite its significance, intermediate data management is still largely unexplored in dataflow programming frameworks such as Pig [4], whose infrastructure layer consists of a compiler that produces sequences of Map-Reduce programs, which are executed over Hadoop.

1.2 Quick Analysis

Let us now discuss the effect of failures on dataflow computations. Suppose we run a data flow computation using Pig which is compiled into a sequence of Map/Reduce jobs, and thus, consisting of multiple Map and Reduce stages. As illustrated in Figure 1, suppose that a failure occurs (e.g., due to a disk failure, a machine failure, etc.) on a node running task t at stage n. This failure will result in the loss of all the intermediate data from stage *1* to *(n - 1)* stored locally on the failed node because Pig (as well as other dataflow programming frameworks) stores intermediate data on the Local File System. When a failure occurs, Pig will reschedule the failed task t to a different node currently available for re-execution. Note

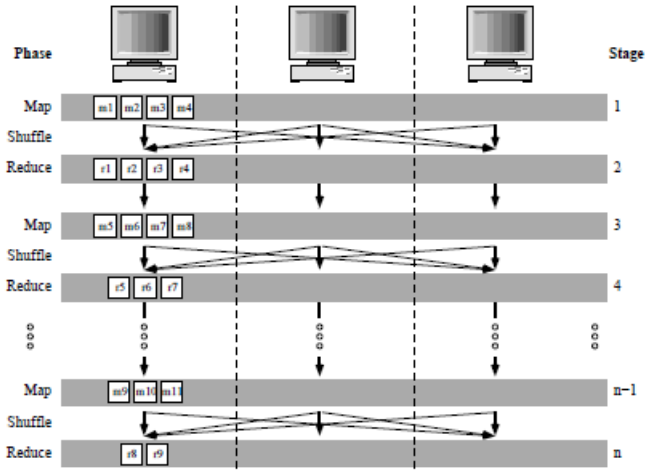

Figure 1: Pig Execution.

that the input of task t at stage n is generated by all the tasks in stage *(n - 1)* including the tasks that run on the failed node. Therefore, some portion of the input will be lost because of the failure, and thus, the re-execution of t cannot proceed immediately. As a result, the tasks running on the failed node will have to be re-executed in order to regenerate the lost portion of the input for task t.

In a similar manner, the failure at stage n will cause the re-execution of tasks run on the failed node in stage *(n - 2)*, and this cascades all the way back to stage 1. This phenomenon is called "cascaded re-execution" - generally defined as the situation that occurs when some tasks in every stage will have to be re-executed sequentially from the beginning to the current stage. Generally speaking, any dataflow framework with multiple stages will suffer from this problem at some point in time.

In the case of a single failure that occurs on the runtime of a Hadoop job with only two stages: Map and Reduce, one failure is supposed to appear at a random node immediately after the last Map task is completed. Since Hadoop's node failure detection timeout is 10 minutes by default, a single failure will cause an approximate 50% [10] increase in completion time. While this experiment shows only "cascaded re-execution" within a single stage, we believe that in dataflow computation with multi-stages, a few node failures will cause a greater increase in job completion time. When tasks are re-executed due to a failure, intermediate data may be read or generated multiple times causing the lifetime of the intermediate data to extend significantly. Taken together, failure will lead to an increase in overhead needed for generating, writing, reading, and storing intermediate data, eventually causing job completion time to increase.

Statistically speaking, Google reports an average of five worker deaths per Map/Reduce job in March 2006 [7], and at least one disk failure in every 6-hour Map/Reduce job with 4,000 machines [23]. Yahoo! reports their web graph generation (called WebMap) has grown to a chain of 100 Map/Reduce jobs [18]. In addition, many organizations such as Facebook and Last.fm have reported their usage of Map/Reduce and Pig to process hundreds of TBs

of data already with an increase of several TBs daily. It is safe to say that reliability and efficiency properties have become an appropriate attribute of intermediate data regardless of the failure types.

1.3 Proposal

Regarding the requirements needed to be satisfied by the storage layer for intermediate data, we propose a solution based on storing intermediate data in the Distributed File System (DFS). For this purpose, two file systems: Hadoop Distributed File System (HDFS) - as a primary storage system of Hadoop Map/Reduce framework and BlobSeer File System (BSFS) - the new substitute storage system which supports heavy access concurrency, versioning and fine-grain access, are used as substitute storage layer for intermediate data.

Besides, the new approach also promotes benefits in case of Map/Reduce applications by skipping the sort and shuffle phases required by the job. In this report, we have studied a new approach for intermediate data management, and also conducted our own contribution in analysis, design, implementation and experimentation.

The rest of the report is structured as follows. Section 2 provides an overview on the Map/Reduce programming model. It also introduces some storage systems like HDFS and BSFS. Then, our main contribution is described of designing new modification in Map/Reduce framework which provides intermediate data management in Section 3. In Section 4, we present the preliminary experiments of our system on the Grid'5000 testbed. Finally, Section 5 concludes the contribution of our work and the possibility of continuing what has been done.

2 Context: Map/Reduce Applications and Data Management

In this section, we introduce the Map/Reduce Applications framework, Hadoop Distributed File System (HDFS) and Blobseer File System (BSFS) as well. Furthermore, we present our proposal of modifying the Hadoop Map/Reduce framework.

2.1 Map/Reduce Application Framework

2.1.1 General Presentation about Map/Reduce Framework

With the rapid increase of Internet, Google has to implement hundreds of special-purpose computations that process large amounts of raw data such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Simply put, Map/Reduce is a programming model that enables easy development of scalable parallel applications to process vast amounts of data on large clusters of commodity machines [11]. Through a simple interface with two core functions, Map and Reduce, this model provides a platform for parallel execution of many real-world tasks such as data processing for search engines and machine learning. There are many implementations of Map/Reduce such as Map-Reduce-Merge [6], Map/Reduce for Multi-core and Multiprocessor systems [9] Google Map/Reduce [20] and Apache Hadoop [22].

The programming design of Map/Reduce is straight forward, consisting of two sequential phases: Map phase and Reduce phase. The Map/Reduce framework works exclusively on [key, value] ([k, v]) pairs. The mapping operation receives an input of [k, v] and subsequently produces a set of [k, v] pairs for further processing in the Reduce phase. Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer. In the Map phase, user defined operation is performed and results are collected; at the beginning of the next phase, Reduce, data is selectively truncated before the output data enters persistent storage.

The Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of splits. The input splits can be processed in parallel by different machines. Reduce invocations are distributed by partitioning the intermediate key space into *R* pieces using a partitioning function (e.g., hash(key) mod R). The number of partitions *(R)* and the partitioning function are specified by the user. In overall, a Map/Reduce job is a unit of work that the user wants to be performed: it consists of the input data, the Map/Reduce program, and configuration information. Figure 2 illustrates the overall flow of a Map/Reduce operation in implementation [8]. When the user program calls up the Map/Reduce function, the following sequence of actions occurs:

- The Map/Reduce library in the user program first splits the input into *M* pieces, typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up multiple copies of the program on a cluster of machines.
- One special copy of the program is named the Master that is responsible for assigning work to the rest, called Workers. There are *M* map tasks, and *R* reduce tasks to assign.

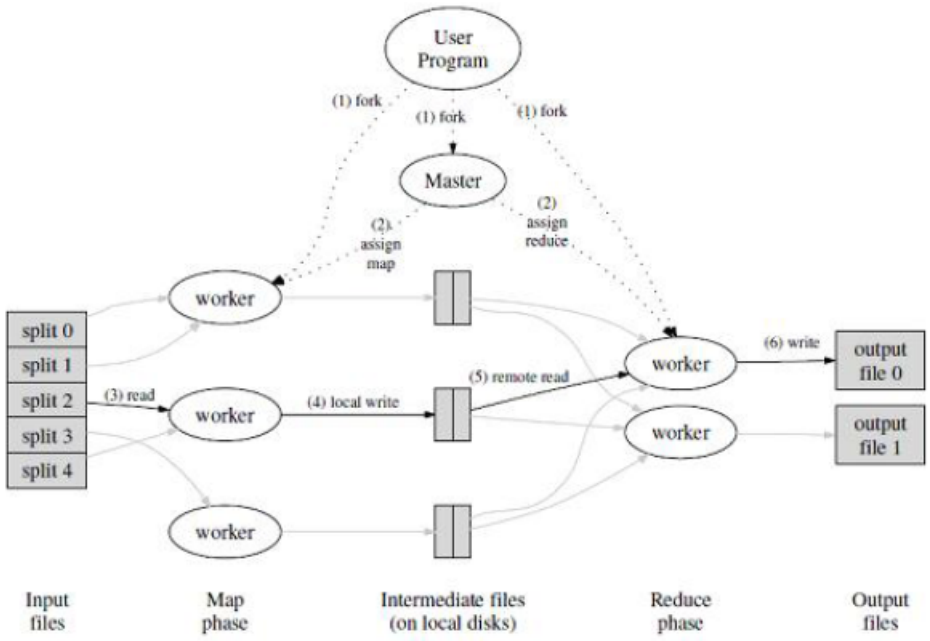

Figure 2: The Overall Flow of a Map/Reduce Operation in Implementation.

The Master picks idle Workers and assigns either a Map task or a Reduce task.

- A Worker that is assigned a Map task reads the content of the corresponding input split, parses key/value pairs out of the input data, and then passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory
- Periodically, the buffered pairs are written to local disk that are partitioned into *R* regions by the partitioning function. The locations of these buffered pairs on the local disk are sent to the Master who is responsible for forwarding these locations to the Reduce workers.
- When notified by the Master about these locations, the Reduce worker copies the data assigned to it from the local disks of the Map workers.
- The Reduce worker combs through the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.
- When all Map tasks and Reduce tasks have been completed, the Master wakes up the user program. After calling in the user program, the Map/Reduce returns back to the user code.

The output of a successful Map/Reduce execution comes in *R*- the number of Reduce tasks- output files (one per Reduce task).

Typically, users do not need to combine these R output files into one file - they often pass these files as input to another Map/Reduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

2.1.2 Hadoop Map/Reduce Framework

Map/Reduce is a new programming abstraction used by Google's search engines and other data intensive applications running in clusters. It attempts t5o ease the programming burden while managing and processing large data sets. Map/Reduce follows a SIMD or SPMD model as it runs single instruction in a single program on multiple large data sets.

Recently, an open source implementation of Map/Reduce, Hadoop [19] has also been successfully used to program data parallel applications. Conceptually, Hadoop can be used for data intensive scientific applications [5], because it facilitates ease of programming and high availability and reliability. Currently, whether it is feasible to implement scientific applications with Hadoop is still an open question.

To enable massively parallel data processing to a high degree over a large number of nodes, the storage layer needs to satisfy specific requirements.

- The storage layer is expected to provide efficient fine-grain access to files, because in Map/Reduce applications, the computations have to process huge amounts of small data records.
- In spite of heavy access concurrency to the same file, the storage layer must sustain a high throughput as thousands of clients access data simultaneously.
- Besides, the data file system needs to support data availability in the presence of frequent failures.
- Finally, one important requirement is its ability to expose an interface that enables the application to be data-location aware. It will be used by the scheduler to place computation tasks close to the data [12].

The Hadoop architecture consists of a namenode and a jobtracker, both of which are servers, and an 'N' number of servers that function as tasktrackers and datanodes (Figure 3). The namenode is responsible for managing all file system data within the Hadoop file system. It is also responsible for handling all read/write access to files as well as file replication. The datanodes service all read/write requests from clients based on received instructions from the namenode and are responsible for performing replication tasks and, more importantly, for storing the file system data.

The jobtracker is responsible for handling all jobs submitted by a client application. It makes all scheduling decisions and parallelizes the client application across the cluster. The jobtracker is also responsible for task resiliency in the cluster by monitoring all running tasks on the cluster and killing and restarting tasks that fail, hang or otherwise disappear from operation. The tasktracker is responsible for running the client application via instructions received from the jobtracker. The jobtracker and tasktrackers constitute the architecture for Map/Reduce programs to run on.

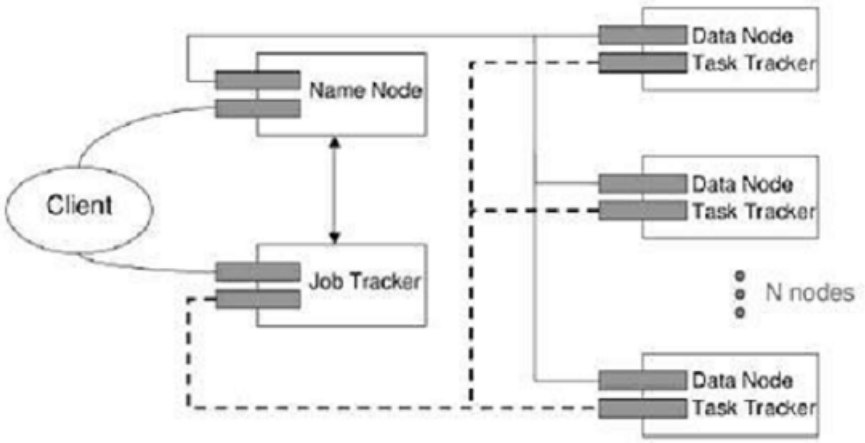

Figure 3: Hadoop Architecture.

Actually, Hadoop relies on the Hadoop Distributed File System (HDFS) as its primary storage system. A key component of Map/Reduce frameworks is their Distributed File System [21] that is built from scratch to provide high availability in face of component failures and to deliver a high performance.

2.1.3 Operation Flow of a Map/Reduce Application

There are two types of entities that participate in the job execution process: a jobtracker and a number of tasktrackers. The jobtracker coordinates all the jobs run on the system by scheduling tasks to run on tasktrackers. Tasktrackers run tasks and send progress reports to the jobtracker, which keeps a record of overall progress of each job. If a task fails, the jobtracker reschedule it on a different tasktracker.

The whole process of running a Map/Reduce job is illustrated in Figure 4 [5]. There are four independent entities:

- At client node, the client submits the Map/Reduce job.
- At jobtracker node, the jobtracker coordinates the job run. The jobtracker is a Java application whose main class is *JobTracker*.
- At tasktracker node, the tasktrackers run the tasks that the job has been split into. Tasktrackers are Java applications whose main class is *TaskTracker*.
- At shared file system, the Distributed File System is used for sharing job files between the other entities.

We describe operating interactions in detail between entities as follows:

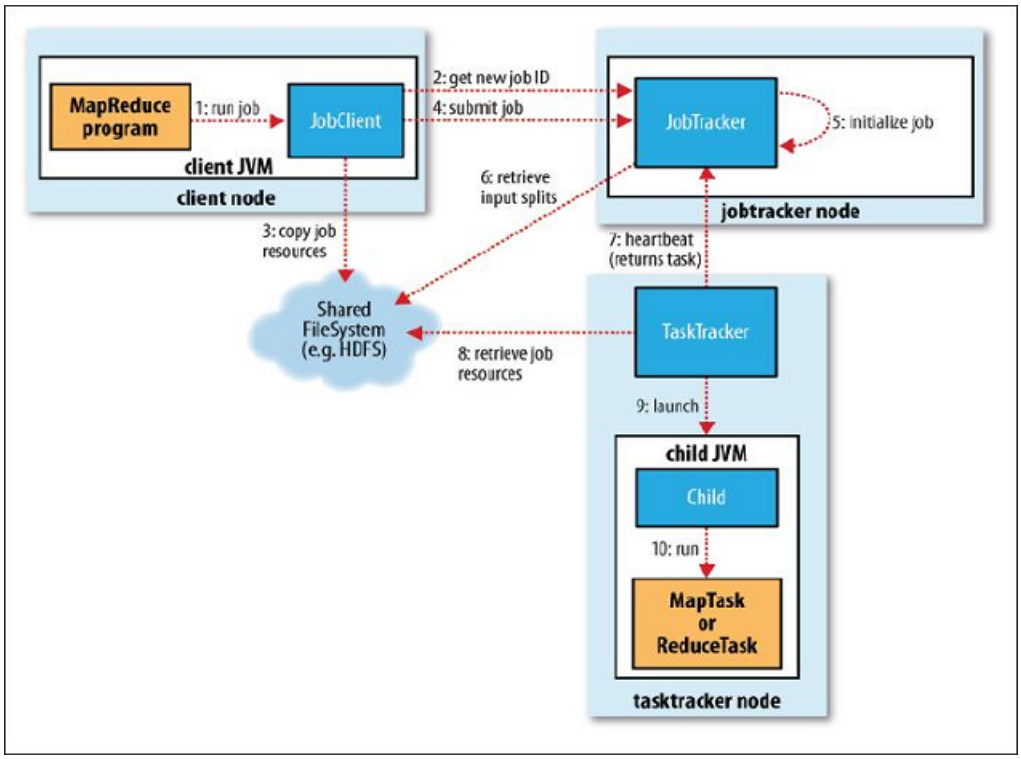

Figure 4: How Hadoop Map/Reduce Runs

Job Submission: The *runJob()* method on *JobClient* creates a new *JobCLient* instance and calls *summitJob()* on it (Step 1). Having submitted the job, *runJob()* polls the job's progress once a second, and reports the progress to the console if it has changed since the last report. When the job is complete, if it was successful, the job counters which show information of completed time, transferred data are displayed. Otherwise, the error that caused the job to fail is logged to the console.

The job submission process implemented as follows:

- Asks the jobtracker for the new job ID (Step 2).
- Checks the output specification of the job. For example, if the output directory has not been specified or it already exists, the job is not submitted.
- Computes the input splits for the job. If the splits cannot be computed, because the input paths do not exist, for example, the job is not submitted.
- Copies the resources needed to run the job, including the job JAR file, the configuration file and the computed input splits, to the jobtracker's file system in a directory named after the job ID (Step 3).
- Tells the jobtracker that the job is ready to execute (Step 4).

Job Initialization: When the jobtracker receives a call to its *submitJob()* method, it puts the job into an internal queue from where the job scheduler will pick it up and initialize it.

Initialization creates an object to represent the job being run, which encapsulates its tasks, and bookkeeping information to keep track of the tasks' status and progress (Step 5).

To create the list of tasks to run, the job scheduler first retrieves the input splits computed by the *JobClient* from the shared file system (Step 6). It then creates one Map task for each split. The number of Reduce tasks to create is determined by the *mapred.reduce.tasks* property in the *JobConf*, which is set by the *setNumReduceTasks()* method, and the scheduler simply creates this number of Reduce tasks to be run (for instance, to sort 1 PB, using 3658 nodes, took 975 minutes with 80,000 Map tasks and 20,000 Reduce tasks [17]). Tasks are given *IDs* at this point.

Task Assignment: Tasktrackers run a simple loop that periodically sends heartbeat method calls to the jobtracker which tell the jobtracker that a tasktracker is alive. Based on the heartbeat, a tasktracker will indicate whether it is ready to run a new task, and if it is, the jobtracker will allocate it a task, which it communicates to the tasktracker using the heartbeat return value (Step 7). Before it can choose a task for the tasktracker, the jobtracker must choose a job to select the task from. Having chosen a job, the jobtracker now chooses a task for the job.

Tasktrackers have a fixed number of slots for Map tasks and for Reduce tasks: for example, a tasktracker may be able to run two Map tasks and one Reduce tasks simultaneously. The default scheduler fills empty Map task slots before reduce task slots, so if the tasktracker has at least one empty Map task slot, the jobtracker will select a Map task; otherwise, it will select a Reduce task. To choose a Reduce task the jobtracker simply takes the next in its list of yet-to-be-run Reduce tasks, since there are no data locality considerations.

For a Map task, however, it takes into account of the tasktracker's network location and picks a task whose input split is as close as possible to the tasktracker. In the optimal case, the task is data-local, that is, running on the same node that the split resides on. Alternatively, the task may be rack-local: on the same rack, but not the same node, as the split. Otherwise, tasktracker retrieves data from a different rack from the one they are running on.

Task Execution: After the tasktracker has been assigned a task, it is run as follows.

First, it localizes the job JAR by copying it from the Distributed File System to the tasktracker's Local File System (Step 8).

Second, it creates a local working directory for the task, and un-jars the contents of the JAR into this directory.

Third, it creates an instance of *TaskRunner* to run the task. *TaskRunner* launches a new Java Virtual Machine (JVM) (Step 9) to run each task in (Step 10), so that any bugs in the user-defined Map and Reduce functions do not affect the tasktracker (by causing it to crash or hang, for example). It is however possible to reuse the JVM between tasks.

The child process communicates with its parent through the umbilical interface. This way it informs the parent of the task's progress every few seconds until the task is complete.

Progress and Status Updates: Map/Reduce jobs are long-running batch jobs, taking from minutes to hours to run, hence it is important for the user to get feedback on how the job is being progress. Status of a job and each of its tasks (e.g., running, successfully completed,

failed), the progress of Maps and Reduces, the values of the job's counters will be updated periodically.

Job Completion: When the jobtracker receives a notification that the last task for a job is complete, it changes the status for the job to "successful". Then, when the *JobClient* polls for status, it learns that the job has completed successfully, so it prints a message to tell the user, and then returns from the *runJob()* method.

Map/Reduce brings crucial benefits to large-scale data management which explain the increasing number of adaptation of the Map/Reduce framework in programming data-intensive application. Hadoop Map/Reduce is a framework designed for running applications on large clusters built of commodity hardware. The most promising implementation of Map/Reduce is the open-source platform Hadoop, which promises ease of programming, high reliability, and wide variety of Hadoop-based applications for different purposes. Hadoop [22] implements the computational paradigm Map/Reduce, and it provides a Distributed File System (HDFS) that stores data on the compute nodes, aiming to achieve a very high aggregate bandwidth across the cluster. Both Map/Reduce and the Distributed File System are designed so that node failures are automatically handled by the framework. We will consider Data Management in Hadoop in the next part.

2.1.4 Case Studies: Map/Reduce Applications

Distributed Grep Distributed Grep application is a very popular example to explain how Map/Reduce works; it extracts matching strings from text files and counts how many times they occurred.

The application runs two Map/Reduce phases in sequence to count how many times a matching string occurred. Each Mapper of the application takes a line as input and matches the user-provided regular expression against the line. It extracts all matching strings and emits (matching string, 1) pairs. Each Reducer sums the frequencies of each matching string. The output is sequence files containing the matching string and count. The reduce phase is optimized by running a combiner that sums the frequency of strings from map output. As a result it reduces the amount of data that needs to be shipped to a Reduce task.

Distributed Sort This is the trivial Map/Reduce program that does absolutely nothing other than use the framework to fragment and sort the input values. The Map function extracts sorting keys from a text line and emits the key and the original text line as the intermediate key/value pair. The Reduce function passes the intermediate key/value pair unchanged as the output key/value pair.

In a word, Distributed Sort application belongs to a kind of jobs that only need to run a filter on the input data, that means no sorting or shuffling are required by the job. No reduce function is needed since the map does all the file processing in parallel with no combine stage.

2.2 Data Management in Map/Reduce Applications

2.2.1 Hadoop Distributed File System

Hadoop supporting the Map/Reduce programming model relies on the Hadoop Distributed File System(HDFS) as its primary storage system. HDFS is a file system designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware. The features HDFS exhibits are detailed below:

Very large files: "Very large" in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size. There are Hadoop clusters running today that store petabytes of data.

Streaming data access: HDFS is built around the idea that the data processing patterns are write-once, read-many-times. A dataset is typically generated or copied from a source, then various analyzes are performed on that dataset over time. Each analysis will involve a large proportion, if not all, of the dataset, so the time to read the whole dataset is more important than the latency in reading the first record.

Commodity hardware: Hadoop does not require expensive, highly reliable hardware to run on. It is designed to run on clusters of commodity hardware (commonly available hardware available from multiple vendors) for which the chance of node failure across the cluster is high, at least for large clusters. HDFS is designed to carry on working without an interruption noticeable to the user in the face of such failure.

A HDFS cluster has two types of nodes operating in a master-worker pattern: a namenode (the master) and a number of datanodes (workers). The namenode manages the file system namespace. It maintains the file system tree and the metadata for all the files and directories in the tree. The namenode also knows the datanodes on which all the blocks for a given file are located, however, it does not store block locations persistently; this information is reconstructed by the namenode when the system starts by querying the datanodes about the data they store.

A client accesses the file system on behalf of the user by communicating with the namenode and datanodes. Figure 5 displays the process by which a client application accesses HDFS. A client application will first direct file queries to the namenode. The namenode then directs the file requests to the appropriate datanode(s) and the datanode(s) supplies the client application with the requested data. Also shown in Figure 5 is the replication of the file across servers in a rack and across multiple server racks. When Hadoop writes new data to its file system, it will try to apply some amount of data locality if possible. That is, as file chunks are written to datanodes across HDFS, the namenode will try to put at least one replicated chunk on the same server rack as the primary datanode, and another chunk on an adjacent datanode while also ensuring that no two replications of a chunk are stored on the same datanode.

In the event of hardware failure of a server, the namenode will take an active role in re-establishing the health of the cluster without the need for user intervention. It is the ability to automatically handle system failures and recover without disruption of service or user intervention that makes HDFS a valuable tool for efficient management of data intensive applications.

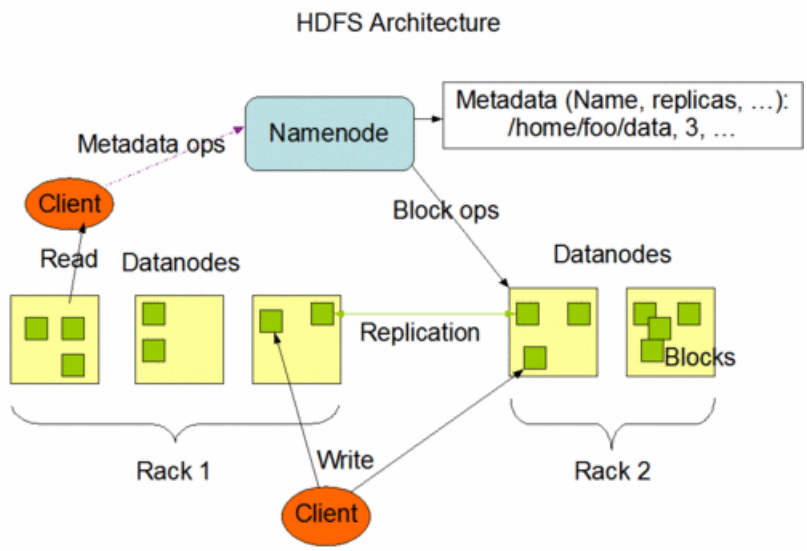

Figure 5: HDFS Architecture.

HDFS stores each file as a sequence of blocks; all blocks in a file system, except the last block, are the same size. Blocks belonging to a file are replicated for fault tolerance. The block size and replication factor are configured per file. Files in HDFS are write-once and have strictly one writer at any time. This assumption simplifies data coherency issues and enables high throughput data access. HDFS does not allow changes to a file once it is created, written and closed.

In HDFS, clients send write requests to a datanode only when they have data worth the chunk size 64 MB. A large chunk size brings some important benefits:

- First, it reduces clients' need to interact with the master because reads and writes on the same chunk require only one initial request to the master for chunk location information.
- Second, it reduces the size of metadata stored on the master. This allows to keep the metadata in memory which makes master operations fast.

Initially, clients buffer all write operations; for each file, the data to be written is collected in a temporary buffer in memory. If the file is closed when the buffer is not full, clients are forced to flush the buffer to the chunk's respective datanode.

In particular, HDFS exposes layout information to the Map/Reduce applications that uses it to schedule computation tasks to datanodes. Adhering to the principle of "moving computation is cheaper than moving data in massive data processing", HDFS offers a simple yet effective way to allow applications to stay close to where the data is located [1].

Some significant properties of HDFS are summarized in Table 6.

HDFS has some loose constraints in its design. It has some limitations in file access semantics, especially for concurrent access and file mutation. Furthermore, HDFS allows only one write-operation at a time and once written, data cannot be altered, overwritten or appended. HDFS has several optimization techniques to improve data throughput:

Deployment model	Compute and storage on one node that benefits to Hadoop/ Mapreduce model where computation is moved closer to the data
Concurrent writes	Not supported – allows only one writer at time
Small file operations	Not optimized for small files, but client-side buffering will only send a write to a server once
Append mode	Write once semantics that does not allow file rewrites or appends
Buffering	"Data staging" to buffer writes until the amount of data is worth the chunk size (64MB)
Data layout	Exposes mapping of chunk to data nodes to Hadoop applications
Replication	3 replicas of data using rack-aware replica placement policy
Compatibility	API designed for the requirements of data-intensive applications

Figure 6: HDFS Properties [21]

- HDFS employs a client side buffering mechanism for small read/write accesses. It prefetches data on reading, on writing. Besides, it delays committing data after the buffer has reach at least a full chunk size.
- Hadoop's job scheduler (the jobtracker) places computations as close as possible to the data. For this purpose, HDFS exposes the mapping of chunks over datanodes to the Hadoop framework.

2.2.2 BlobSeer: Using BlobSeer as Data Storage for Hadoop

In this part, we take into account and evaluate the properties and benefits offered by the Blobseer data management system for improving Hadoop Map/Reduce applications' storage performance.

BlobSeer Overview

BlobSeer is a binary large object (Blob) management service which manages massive data in a large-scale distributed context [2]. Its goal is to provide storage support for data-intensive applications [13]. It uses the concept of Blobs defined as huge storage objects of predefined, fixed sizes that are first allocated, then manipulated by reading and writing parts of them.

BlobSeer addresses the problem of storing and efficiently accessing very large, unstructured data objects in a distributed environment. Its main focus is on heavy access concurrency [14] where data is huge, mutable and potentially accessed by a very large number of concurrent, distributed processes. This kind of concurrent access is becoming more and more popular with scientific applications, multimedia processing, or astronomy over the recent years. BlobSeer is an efficient approach to accommodate huge Blobs (on the order of TBs) by splitting each Blob into small fixed-sized pages that are scattered across data providers. BlobSeer enables efficient fine-grained access to the Blob, without locking the

Blob itself. To deal with the mutable data problem, BlobSeer introduces an efficient versioning scheme which allows the client not only to roll back data changes at will, but also enables access to different versions of the Blob within the same computation.

Last but not least, the metadata management system is built on top of a distributed hash table (DHT), thus preventing the metadata servers from becoming performance bottlenecks [15].

Architecture overview

The system consists of distributed processes that communicate through remote procedure calls (RPCs). A physical node can run one or more processes and, at the same time, may play multiple roles from the ones mentioned below.

Clients: may issue Create, Write, Append and Read requests. There may be multiple concurrent clients. Their number may dynamically vary in time without notifying the system.

Data Providers: physically store and manage the pages generated by Write and Append requests. New data providers are free to join and leave the system in a dynamic way. In the context of Hadoop Map/Reduce, the nodes hosting data providers typically also act as computing elements. This enables them to benefit from the scheduling strategy of Hadoop, which aims at placing the computation as close as possible to the data.

Provider Manager: keeps information about the available data providers. When entering the system, each new joining provider registers with the provider manager. The provider manager tells the client to store the generated pages in the appropriate data providers according to a strategy aiming at global load balancing.

Metadata Providers: physically store the metadata, allowing clients to find the pages corresponding to the various Bob versions. Metadata providers may be distributed to allow an efficient concurrent access to metadata. The nodes hosting metadata providers may act as computing elements as well.

Version Manager: is the key actor of the system. It registers Blob update requests (Append and Write), assigning version numbers to each of them. The version manager eventually publishes these updates, guaranteeing total ordering and atomicity.

Versioning In BlobSeer, versioning [2] is a core feature. Not only does it enable rolling back data changes at will, but also cheap branching (possibly recursively), that is, the same computation may proceed independently on different versions of the blob. Versioning should obviously not significantly impact access performance to the object, given that objects are under constant heavy access concurrency. On the other hand, versioning leads to increased storage space usage and becomes a major concern when the data size is too large. Versioning efficiency thus refers to both access performance under heavy load and reasonably acceptable overhead of storage space.

BlobSeer splits a huge blob into small fixed-sized pages that are scattered across commodity data providers. Rather than updating the current pages, completely new pages are generated when clients request data modifications. The corresponding metadata is "weaved" with old metadata in such a way that it offers a complete virtual view of both the past version and the current version of the blob. Metadata is organized as

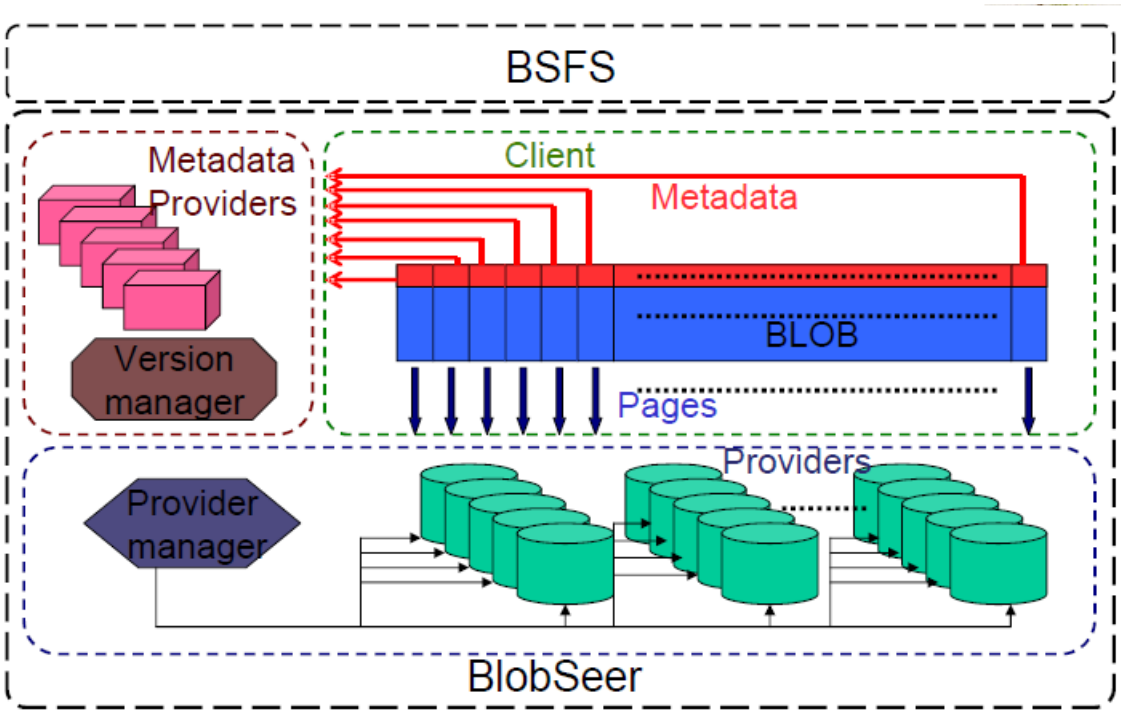

Figure 7: BSFS layer enables Hadoop to use BlobSeer as a storage backend.

a segment-tree like structure and also scattered across the system using a Distributed Hash Table (DHT). Distributing data and metadata not only enables more efficient performance through parallel, direct access I/O paths, but also favors efficient use of storage space: although a full virtual view of all past versions of the blob is offered, real space is consumed only by the newly generated pages.

Integrating BlobSeer with Hadoop

BlobSeer is a data management system with the goal of supporting efficient, fine-grain access to massive, distributed data accessed under heavy concurrency. Its features (built-in versioning, support for concurrent append operations) open the possibility for Hadoop Map/Reduce for further extend its functionalities such as bringing high throughput under heavy access concurrency in the reading, writing and appending to storage backend for Map/Reduce applications. Therefore using BlobSeer as a storage layer will overcome several limitations of Hadoop File System (as mentioned in Section 2.2.2) and bring efficiency to Map/Reduce applications.

In order to address the limitations of the default storage layer - Hadoop Distributed File System and to exploit Blobseer's features in the context of Map/Reduce applications, BlobSeer was integrated with Hadoop Map/Reduce by having BlobSeer act as a storage backend file system for Hadoop [16]. The integration was done by adding a new layer on top of the BlobSeer service, layer called the BlobSeer File System- BSFS (Figure 7).

The BSFS layer: This layer consists in a namespace manager which is not part of BlobSeer. The namespace manager maintains a file system namespace and maps files in the namespace to Blobs. The design is given to ensure the minimum interaction with

the namespace manager, in order to fully benefit from the decentralized metadata management scheme of BlobSeer.

Data prefetching: Caching mechanism was implemented for read/write operations in BSFS. The mechanism consists in prefetching a whole block when the requested data is not already cached and delaying committing writes until a whole block has been filled in the cache.

Affinity scheduling: BSFS returns a list of blocks that make up the requested range and the addresses of the physical nodes that store those blocks for a specified Blob id, version, offset and size. This new primitive makes Map/Reduce scheduler $data-location$ aware.

Experiments which were performed on the Grid'5000 [3], both with synthetic microbenchmarks and real Map/Reduce applications showed that BSFS has improved performance of Map/Reduce applications. In microbenchmarks experiences, BSFS is capable of delivering a higher throughput than HDFS, and sustaining it when the number of clients increases rapidly, due to the load balancing strategy BlobSeer applies when distributing the pages to providers. In real Map/Reduce applications as *Random Text Writer* and *Distributed Grep*, the results displayed that BSFS is able to finish the job faster than HDFS, which are consistent with the microbenchmarks results.

2.3 Failures with Map/Reduce Applications

In the real world, there are many reasons that prevent Map/Reduce applications to finish: a disk failure, a machine failure, a buggy code, etc. In overall, failures are classified into 3 types: Task failure, Tasktracker failure and Jobtracker failure [5].

Task Failure: Task failures are caused by many reasons.

Consider first the case of the child task failing. The most common way that this happens is when user code in the Map or Reduce task throws a runtime exception. If this happens the child JVM reports the error back to its parent tasktracker, before it exits. The error ultimately makes it into the user logs. The tasktracker marks the task attempt as failed, freeing up a slot to run another task.

Another failure mode is the sudden exit of the child JVM - perhaps there is a JVM bug that causes the JVM to exit for a particular set of circumstances exposed by the Map/Reduce user code. In this case, the tasktracker notices that the process has exited, and marks the attempt as failed.

Hanging tasks are dealt with differently. The tasktracker notices that it has not received a progress update for a while, and proceeds to mark the task as failed. The child JVM process will be automatically killed after this period. The timeout period after which tasks are considered to have failed is normally 10 minutes, but can be configured on a per-job basis.

When the jobtracker is notified of a task attempt that has failed (by the tasktracker's heartbeat call) it will reschedule the execution of the task. The jobtracker will try to avoid rescheduling the task on a tasktracker where it has previously failed. Furthermore, if a task fails more than four times, it will not be retried further.

For some applications, it is undesirable to abort the job if a few tasks fail, as it may be possible to use the results of the job despite some failures. In this case, the maximum percentage of tasks that are allowed to fail without triggering job failure can be set for the job.

Tasktracker Failure: Failure of a tasktracker is another failure mode. If a tasktracker fails by crashing, or running very slowly, it will stop sending heartbeats to the jobtracker (or send them very infrequently). The jobtracker will notice a tasktracker that has stopped sending heartbeats and remove it from its pool of tasktrackers to schedule tasks on. The jobtracker arranges for Map tasks that were run and completed successfully on that tasktracker to be rerun if they belong to incomplete jobs, since their intermediate output residing on the failed tasktracker's Local File System may not be accessible to the Reduce task. Any tasks in progress are also rescheduled. A tasktracker is blacklisted if the number of tasks that have failed on it is significantly higher than the average task failure rate on the cluster. Blacklisted tasktrackers can be restarted to remove them from the jobtracker's blacklist.

Jobtracker Failure: Failure of the jobtracker is the most serious failure mode. However, this failure mode has a low chance of occurring since the chance of a particular machine failing is low. It can be dealt with by running multiple jobtrackers, only one of which is the primary jobtracker at any time.

2.4 Description of Data Flow between Mappers and Reducers in Hadoop

Figure 8 describes the stages of a Map/Reduce program and detailed data flow of Map/Reduce more precisely.

Input Data: Input data is the input of Mapper residing in Distributed File System.

While this does not need to be the case, the input files typically reside in DFS. The format of these files is arbitrary; while line-based log files can be used, we could also use a binary format, multi-line input records, or anything else. It is typical for these input files to be very large - tens of gigabytes or more. When starting a Hadoop job, *FileInputFormat* is provided with a path containing files to read. The *FileInputFormat* will read all files in this directory. It then divides these files into one or more *InputSplits* each.

An *InputSplit* describes a unit of work that are required for a single Map task in a Map/Reduce program to be run. A Map/Reduce program applied to a data set, collectively referred to as a Job, is made up of several (possibly several hundred) tasks. Map tasks may involve reading a whole file; they often involve reading only part of a file. By default, the *FileInputFormat* and its descendants break a file up into 64 MB chunks (the same size as blocks in HDFS).

By processing a file in chunks, it allows several Map tasks to operate on a single file in parallel. If the file is very large, this can improve significantly performance through parallelism. Even more importantly, since the various blocks that make up the file may be spread across several different nodes in the cluster, it allows tasks to be scheduled

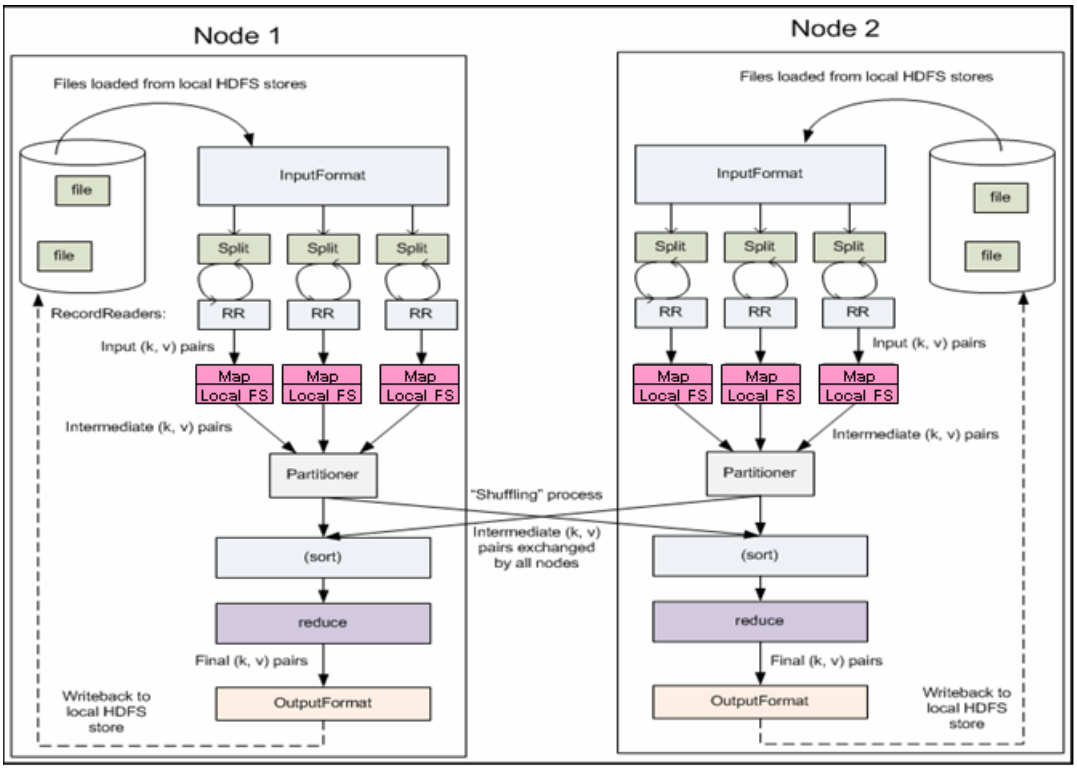

Figure 8: Detailed Hadoop Map/Reduce Data Flow.

on each of these different nodes; the individual blocks are thus all processed locally, instead of needing to be transferred from one node to another.

The *InputFormat* defines the list of tasks that make up the mapping phase; each task corresponds to a single input split. The tasks are then assigned to the nodes in the system; based on where the input file chunks are physically resident. An individual node may have several dozen tasks assigned to it. The node will begin working on the tasks, attempting to perform as many in parallel as it can.

Intermediate Data: Intermediate data, residing in Local File System, is the output of Mapper and becomes the input of Reducer.

The Mapper performs the user-defined work of the first phase of the Map/Reduce program. Given a key and a value, the *map()* method emits *(key, value)* pair(s) which are forwarded to the Reducers. A new instance of Mapper is instantiated in a separate Java process for each Map task (*InputSplit*) that makes up part of the total job input. The *OutputCollector* object has a method named *collect()* which will forward a *(key, value)* pair to the reduce phase of the job. The *Reporter* object provides information about the current task; its getInputSplit() method will return an object describing the current *InputSplit*. It also allows the Map task to provide additional information about its progress to the rest of the system. The *setStatus()* method allows you to emit a status message back to the user. The *incrCounter()* method allows you to increment shared performance counters. Each Mapper can increment the counters, and the jobtracker will collect the increments made by the different processes and aggregate them

for later retrieval when the job ends.

After the first Map tasks have completed, the nodes may still be performing several more Map tasks each. But they also begin exchanging the intermediate outputs, which are stored at Local File System, from the Map tasks to where they are required by the Reducers. This process of moving map outputs to the Reducers is known as shuffling. A different subset of the intermediate key space is assigned to each reduce node; these subsets (known as "partitions") are the inputs to the Reduce tasks. Each Map task may emit *(key, value)* pairs to any partition; all values for the same key are always reduced together regardless of which Mapper is its origin. Therefore, the map nodes must all agree on where to send the different pieces of the intermediate data. The *Partitioner* class determines which partition a given *(key, value)* pair will go to. The default partitioner computes a hash value for the key and assigns the partition based on this result.

Output Data: Output data is the output of Reducer residing in Distributed File System.

A Reducer instance is created for each Reduce task. For each key in the partition assigned to a Reducer, the Reducer's *reduce()* method is called once. This receives a key as well as an iterator over all the values associated with the key. The values associated with a key are returned by the iterator in an undefined order. Each Reducer writes a separate file in a common output directory. These files will typically be named *"part-nnnnn"*, where *"nnnnn"* is the partition id associated with the Reduce task.

Hadoop provides some *OutputFormat* instances to write to files. The basic (default) instance is *TextOutputFormat*, which writes *(key, value)* pairs on individual lines of a text file. A better intermediate format for use between Map/Reduce jobs is the *SequenceFileOutputFormat* which rapidly serializes arbitrary data types to the file; the corresponding *SequenceFileInputFormat* will deserialize the file into the same types and presents the data to the next Mapper in the same manner as it was emitted by the previous Reducer. The *NullOutputFormat* generates no output files and disregards any *(key, value)* pairs passed to it by the *OutputCollector*.

The output files written by the Reducers are then left in DFS (HDFS) for user's use, either by another Map/Reduce job or a separate program, for inspection.

2.5 Intermediate Data Management: Approach

In the real world, processes crash and machines fail. Thus, Map/Reduce systems need to have the ability to handle such failures and allow jobs to complete. In this report, our focus is on tasktrackers' failures. We consider dataflow programs (such as Pig) that consist of a sequence of Map and Reduce stages with the communication pattern that is either all-to-all (between a Map stage and the next Reduce stage) or one-to-one (between a Map stage and the next Reduce stage). The problem at hand is serious when one failure can lead to expensive cascaded re-execution of tasks where some tasks in every stage from the beginning have to be re-executed sequentially up to the stage where the failure happens (as shown in Section 1.2). Therefore, providing a method for handling intermediate data efficiently and reliably plays a key role in optimizing the execution of dataflow programs.

We also recognize the ability of Distributed File Systems such as Hadoop Distributed File System and BlobSeer File System to provide reliable support for data storage. In addition, the BlobSeer File System can offer the attainability of the efficiency property thanks to the support of heavy access concurrency, versioning and fine-grain access.

Based on the matching of the requirements for intermediate data and DFS's properties, we choose DFS for intermediate data storage. We have attempted to modify the Hadoop Map/Reduce framework so that the DFS stores the intermediate output of the Map phase.

Sometimes we only need to run a filter on the input data, so there is no sorting or shuffling required. In these cases, intermediate data becomes the final output. The new approach also offers an important benefit wherein the intermediate data is written directly to DFS instead of saving onto Local File System and then copied into DFS. For instance, in Distributed Sort application, the Reduce phase only copies the intermediate data to the DFS in the original Hadoop. In modified Hadoop, the Reduce phase is skipped (the number of Reducer is zero) since the intermediate data is already in the DFS.

3 Contribution: A New Approach for Intermediate Data of Map/Reduce Applications

3.1 Analysis

In order to use the Distributed File System for data intensive environments, we propose a modification of the Hadoop Map/Reduce framework to store the output from the Map phase and the input to the Reduce phase to the DFS.

3.1.1 General Idea

On the client side, Hadoop actually defines two interfaces to access the DFS, the command line and an abstract notion of file system. The command line is the simplest and we can do all of the usual file system operations such as reading file, creating directories, moving files, deleting data and listing directories. Besides, Hadoop has an abstract notion of file system that permits Hadoop Map/Reduce framework to access the storage layer. This interface also exposes the basic functions of a file system.

Applications use a Java API (*FileSystem* class) to perform file system operations. This *FileSystem* class is a Java abstract class *org.apache.hadoop.fs.FileSystem* which represents a file system in Hadoop and there are several concrete implementations as *LocalFileSystem* or *DistributedFileSystem*, etc. This API interface brings us a way to modify the Map/Reduce framework.

In summary, Hadoop Java API is the one that should be studied the most and then data operations in Map and Reduce phases will be made accordingly to analyze and design the modification.

3.1.2 Inner Operation of Hadoop Java API

In this part, we dig into the Hadoop's *FileSystem* class: the API for interacting with one of the Hadoop's file systems. We mainly focus on two operations: Read and Write.

Reading Data

Retrieve an instance

FileSystem is a general file system API, so the first step is to retrieve an instance for the file system we want to use. There are two static factory methods for getting a *FileSystem* instance:

```
public static FileSystem get(Configuration conf) throws IOException
public static FileSystem get(URI uri, Configuration conf) throws IOException
```

A *Configuration* object encapsulates a client or server's configuration, which is set using configuration files read from the classpath, such as "conf/core-site.xml". The first method returns the default file system (as specified in the file "conf/core-site.xml", or the default Local File System if not specified there).

The second uses the given URI's scheme and authority to determine the file system to use, falling back to the default file system if no scheme is specified in the given URI.

Get the input stream

With a *FileSystem* instance, we invoke an *open()* method to get the input stream for a file:

```
public FSDataInputStream open(Path f) throws IOException
public abstract FSDataInputStream open(Path f, int bufferSize) throws IOException
```

The *open()* method on *FileSystem* actually returns a *FSDataInputStream*. This class is a specialization of *java.io.DataInputStream* which supports random access which permits to read from any part of the stream.

Seek a position

The *Seekable* interface permits seeking to a position in the file, and a query method for the current offset from the start of the file *getPos()*:

```
public interface Seekable {
    void seek(long pos) throws IOException;
    long getPos() throws IOException;
    boolean seekToNewSource(long targetPos) throws IOException;
}
```

Calling *seek()* with a position that is greater than the length of the file will result in an *IOException*. Unlike the *skip()* method of *java.io.InputStream* which positions the stream at a point later than the current position, *seek()* can move to an arbitrary, absolute position in the file.

The *seekToNewSource()* method is not normally used by application writers. It attempts to find another copy of the data and seek to the offset targetPos in the new copy. This is used internally in DFS to provide a reliable input stream of data to the client in the face of datanode failure.

Read parts of a file

FSDataInputStream also implements the *PositionedReadable* interface for reading parts of a file at a given offset:

```
public interface PositionedReadable {
    public int read(long position, byte[] buffer, int offset, int length)
    throws IOException;
    public void readFully(long position, byte[] buffer, int offset, int length)
    throws IOException;
    public void readFully(long position, byte[] buffer) throws IOException;
}
```

The *read()* method reads up to length bytes from the given position in the file into the buffer at the given offset in the buffer. The return value is the number of bytes actually read: callers should check this value as it may be less than length. The *readFully()* methods will read length bytes into the buffer (or *buffer.length* bytes for the version that just takes a

byte-array buffer), unless the end of the file is reached, in which case an *EOF Exception* is thrown.

All of these methods preserve the current offset in the file and are thread-safe, so they provide a convenient way to access another part of the file-metadata perhaps-while reading the main body of the file. In fact, they are just implemented using the *Seekable* interface using the following pattern:

```
long oldPos = getPos();
try {
    seek(position);
    // read data
    finally
}
seek(oldPos);
```

Writing Data

Create a new file

The *FileSystem* class has a number of methods for creating a file. The simplest is the method that takes a *Path* object for the file to be created and returns an output stream to write to.

```
public FSDataOutputStream create(Path f) throws IOException
```

The *create()* methods create any parent directories of the file to be written that do not already exist. Though convenient, this behavior may be unexpected. The existence of the parent directory should be checked first by calling the *exists()* method to ensure that the parent directory does not exist.

Append an existing file

There is also an overloaded method for passing a callback interface, *Progressable*, so an application can be notified of the progress of the data being written to the datanodes:

```
package org.apache.hadoop.util;
public interface Progressable {
    public void progress();
}
```

As an alternative to creating a new file, an existing file can be appended using the *append()* method (there are also some other overloaded versions):

```
public FSDataOutputStream append(Path f) throws IOException}
```

The append operation allows a single writer to modify an already written file by opening it and writing data from the final offset in the file. With this API, applications that produce unbounded files, such as log files, can write to an existing file after a restart, for example. The append operation is optional and not implemented by all Hadoop file systems.

The *create()* method on *FileSystem* returns a *FSDataOutputStream*, which, like *FSDataInputStream*, has a method for querying the current position in the file:

```
package org.apache.hadoop.fs;
public class FSDataOutputStream extends DataOutputStream implements Syncable {
    public long getPos() throws IOException {
    // implementation elided}
    }
// implementation elided
}
```

3.2 Design

As presented in the Section 3.1, the Hadoop Map/Reduce framework accesses its storage backend (DFS) through a clean, specific Java API which exposes the basic operations of a file system: Read, Write, Append, etc. We modify Hadoop Map/Reduce framework so that DFS can store intermediate output from the Map phase by taking advantage of Hadoop Java API.

In this part, we describe what happens specifically:

In the Map side:

When the Map function starts producing output, it is not simply written to Distributed File System. Many steps are involved into this process.

Figure 9 illustrates what happens. Each Map task has a circular memory buffer that it writes the output to. The buffer is 100 MB by default, a size which can be tuned by changing the *io.sort.mb* property. When the contents of the buffer reaches a certain threshold size (*io.sort.spill.percent*, default 0.80) a background thread will start to spill the contents to DFS. Map outputs will continue to be written to the buffer while the spill takes place, but if the buffer fills up during this time, the map will block until the spill is complete.

Spills are written in round-robin fashion to the directories specified by the *mapred.local.dir* property, in a job-specific subdirectory. Before it writes to DFS, the thread first divides the data into partitions corresponding to the Reducers that they will ultimately be sent to. Within each partition, the background thread performs an in-memory sort by key, and if there is a combiner function, it is run on the output of the sort. Each time the memory buffer reaches the spill threshold, a new spill file is created, so after the Map task has written its last output record there could be several spill files.

Before the task is finished, the spill files are merged into a single partitioned and sorted output file. The configuration property *io.sort.factor* controls the maximum number of streams to merge at once; the default is 10. If a combiner function has been specified, and the number of spills is at least three (the value of the *min.num.spills.for.combine* property), then the combiner is run before the output file is written. Combiners may be run repeatedly over the input without affecting the final result.

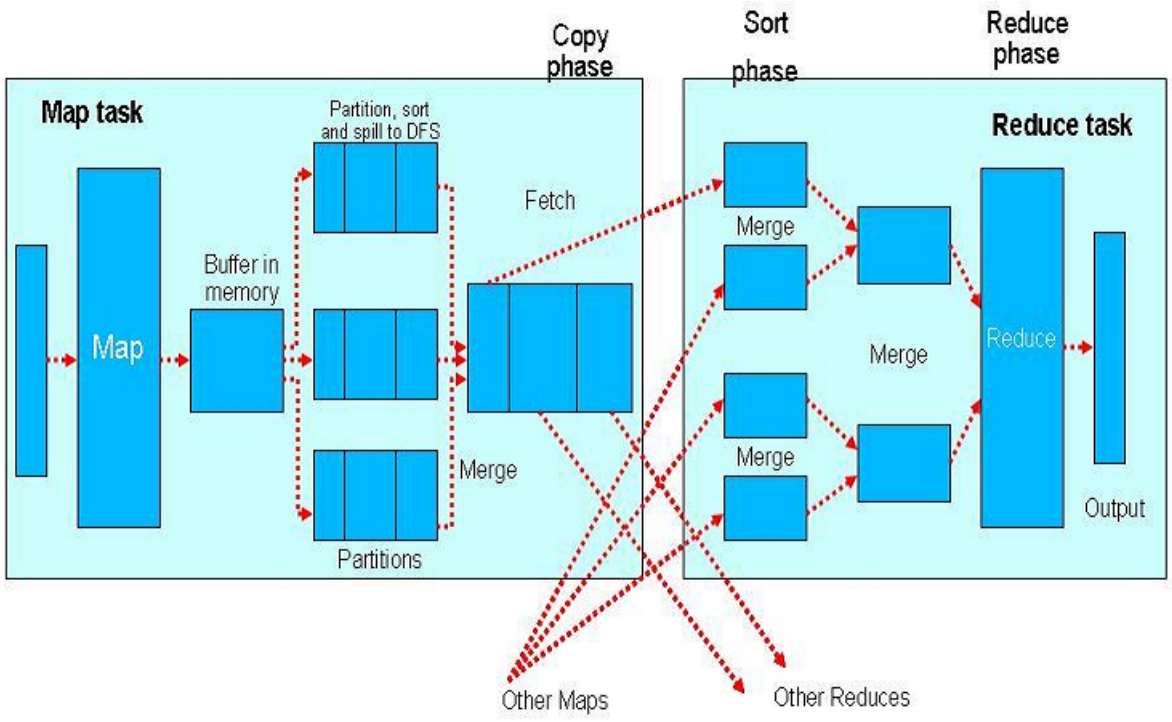

Figure 9: Data Interaction in Design.

In the Reduce side:

In the Reduce part of the process, the map output file is sitting on the DFS and is needed by the tasktracker that is about to run the Reduce task for the partition.

Furthermore, the Reduce task needs the map output for its particular partition from several Map tasks across the cluster. The Map tasks may finish at different times, so the Reduce task starts copying their outputs as soon as each completes. This is known as the copy phase of the Reduce task. The Reduce task has a small number of copier threads so that it can fetch map outputs in parallel. The default is five threads, but this number can be changed by setting the *mapred.reduce.parallel.copies* property.

The map outputs are copied to the reduce tasktracker's memory if they are small enough (the buffer's size is controlled by *mapred.job.shuffle.input.buffer.percent*, which specifies the proportion of the heap to use for this purpose); otherwise, they are copied to disk. When the in-memory buffer reaches a threshold size (controlled by *mapred.job.shuffle.merge.percent*), or reaches a threshold number of map outputs (*mapred.inmem.merge.threshold*), it is merged and spilled to disk. As the copies accumulate on disk, a background thread merges them into larger, sorted files. This saves some time merging later on.

When all the map outputs have been copied, the Reduce task moves into the sort phase (which should properly be called the merge phase, as the sorting was carried out on the map side), which merges the map outputs, maintaining their sort ordering. This is done in rounds. For example, if there were 50 map outputs, and the merge factor was 10 (the default, controlled by the *io.sort.factor property*, just like in the map's merge), then there would be 5 rounds. Each round would merge 10 files into one, so at the end there would be five

intermediate files. Rather than have a final round that merges these five files into a single sorted file, the merge saves a trip to disk by directly feeding the reduce function in what is the last phase: the Reduce phase. This final merge can come from a mixture of in-memory and on-disk segments.

During the Reduce phase the reduce function is invoked for each key in the sorted output. The output of this phase is written directly to the output file system, DFS.

3.3 Implementation

According to our design illustrated above, the work will put emphasis on certain classes of data that needs investigation:

InitTask(): creates:

HostMaps[ ] which contains a mapping from a host to file split id's that the host is holding on DFS.

JobInProgress also offers two methods for creating a Task instance to run, out of maps[] and reduces[].

MapTask: offers method run() that calls *MapRunner.run()*, which in turn calls the user-supplied *Mapper.map()*.

ReduceTask: offers *run()* that sorts input files using *SequenceFile.Sorter.sort()*, and then calls user-supplied *Reducer.reduce()*.

InputFormat: is an interface for instantiating file splits and the readers for Map task.

OutputFormat: is an interface for instantiating writer for consuming output of Reduce task.

FailedTask() can be directly called by *JobTracker* when a launching task is timed out without any heartbeat from tasktracker, or when a tasktracker is found to be lost.

TaskInProgress (TIP) represents a set of tasks for a given unique input, where input is a split for Map task or a partition for Reduce task.

TaskInProgress has two constructors, one that takes input *FileSplit* (for Map task), and another that takes partition number (for Reduce task). These two are invoked by *JobInProgress.initTask()*.

TaskInProgress contains:

RecentTasks: instance(s) of Task in flight for the given input.

GetTaskToRun() populates *RecentTasks* with either *MapTask* or *ReduceTask* instance and then returns the Task to caller. Task IDs are obtained from a field *usableTaskids*.

TaskStatuses: a set of *TaskStatus* instances that contains status per task which are keyed by task ID.

UpdateStatus(): puts latest status of a task into *TaskStatues* when a tasktracker emits heartbeat containing the task status.

Furthermore, we have TIP id which is set to *TaskInProgress.id* includes:

Map task id: task_ JobInProgress.uniqueString _m_ splitId _ i.

Reduce task id: task_ JobInProgress.uniqueString _r_ partitionId _ i
(i is 0 to max-1, where max is "TaskInProgress.Max_Task_Execs + Max_Task_Failures)".

JobInProgress: represents a job as it is being tracked by *JobTracker* which contains two sets of TaskInProgress instances:

TaskInProgress maps[ ]: an array holding one TIP per split.

TaskInProgress reduces[ ]: an array holding one TIP per partition.

JobTracker: is a daemon per pool that administers all aspects of activities. It keeps all the current jobs by containing instances of *JobInProgress* in jobs which map from jobid string to *JobInProgress*. Besides, *JobTracker* has a home dir called "jobTracker" on Local File System, which is value of *JobTracker.SubDir*. Under the dir, each *JobInProgress* copies two files from user-submitted location:

Job file: "jobTracker/" jobId ".xml"

Jar file: "jobTracker/" jobid ".jar"

TaskTracker: is a daemon process running on each of worker node. It contains instances of *TaskTracker.TaskInProgress*, which represents a single task as it is being tracked by *TaskTracker* daemon, in two fields:

RunningTasks: for subset of tasks that are being run.
A new instance of Task is obtained by calling *JobClient.pollForNewTasks*(), which will drive *JobTracker* to eventually call *TaskInProgress.getTaskToRun()*. Note again that an instance of *TaskTracker.TaskInProgress* is created, containing the new Task, then gets added into tasks and running tasks.

StartNewTask() takes the new Task then starts the actual sequence for running user code for the task. Eventually, the user code is executed in a child process.

TaskTracker has a home dir called "taskTracker" where each task creates a subdir:

Task dir: "taskTracker/" taskId

Under task dir, jobFile and jarFile are created as:

Job file: "taskTracker/" taskId "/job.xml"

Jar file: "taskTracker/" taskId "/job.jar"

Each task also has workdir at top level. Map output files go under this dir as:

Map output for a partition: taskId "/part−" partitionId ".out"
Above, taskId has "_m_" in it as its map task id.

When the map outputs for a partition is copied to the Reduce task, the local files are created as:

Reduce input from a split: taskId "/map_" splitId ".out"
Above, taskId has _r_ in it, as it is Reduce task id.

4 Experimental Evaluation

In this section, we carry out some experiments in order to evaluate our prototype. We used Yahoo!'s release of Hadoop v.0.20.0 (which essentially is the main release of Hadoop with some minor patches designed to enable Hadoop to run on the Yahoo! production cluster). We choose this release because it is freely available and enables us to experiment with a framework that is both stable and used in production on Yahoo!'s cluster.

4.1 Environmental Setup

Evaluations have been performed on the Grid'5000 testbed [3], a reconfigurable, controllable and monitorable experiment Grid platform gathering 9 sites geographically distributed in France. We used the clusters located in Orsay. Each experiment was carried out within a single such cluster. The nodes are outfitted with x86 64CPUs 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. A significant effort was invested in preparing for experimental setup, by defining an automated deployment process for the Hadoop framework both when using BlobSeer and HDFS as the storage backend.

Overview of the experiments: Our goal was to get a feeling of the impact of the modification at the application level. We had run two standard Map/Reduce applications from the Hadoop release, both with original Hadoop and modified Hadoop with both storage backends: BSFS and HDFS. We have evaluated the impact of using modified Hadoop instead of original Hadoop on the total job execution time. Note that Hadoop Map/Reduce applications run out of the box in an environment with modified Hadoop just like in the original, unmodified environment of original Hadoop. We have 4 scenarios to run applications:

The original Hadoop framework with HDFS as storage

The modified Hadoop framework with HDFS as storage

The modified Hadoop framework with BSFS as storage

The original Hadoop framework with BSFS as storage

The 4th possible scenario, with original Hadoop and BSFS was discussed in [16] and shows the gain that can be obtained when using BlobSeer as storage. Furthermore, this scenario is not relevant to our work, as we try to evaluate the impact of modifying Hadoop (the first 2 scenarios) and the benefits of using BSFS with the modified version of the Hadoop framework (the 3rd scenario).

To compare the performance in each scenario, we co-deploy a Hadoop tasktracker with a datanode in the case of HDFS (with a data provider in the case of BSFS) on the same physical machine, for a total of 150 machines. The other entities for Hadoop, HDFS (namenode, jobtracker) and for BSFS (version manager, provider manager, namespace manager) are deployed on separate dedicated nodes. For BlobSeer, 10 metadata providers are deployed on dedicated machines as well.

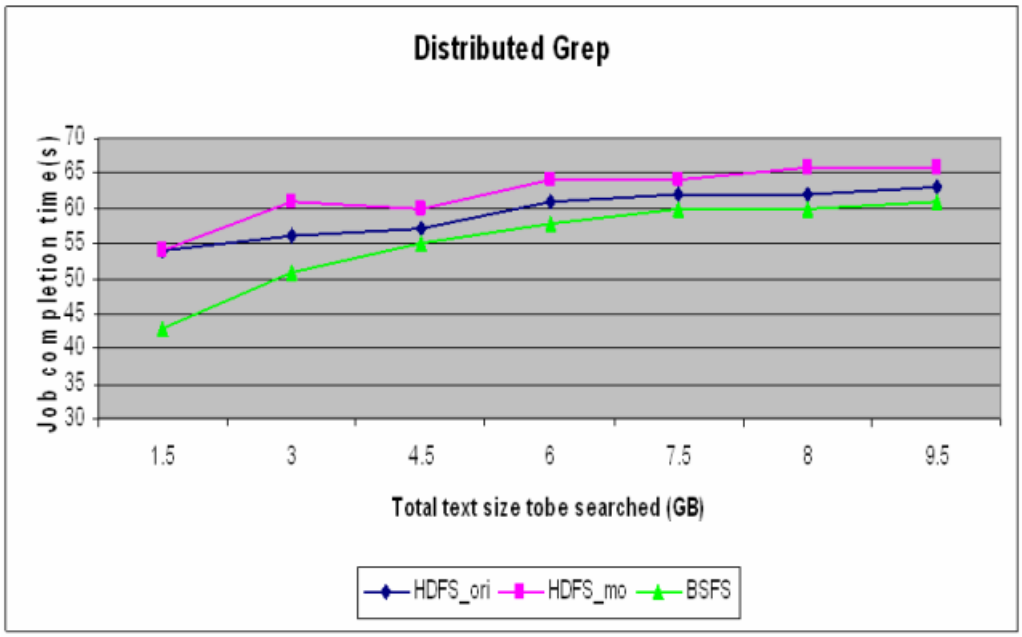

Figure 10: Performance of HDFS and BSFS with both Original Map/Reduce and Modified Map/Reduce in case of Distributed Grep Application.

4.2 Application Studies

In order to evaluate how our approach of storing intermediate data in DFS influences the performance of the Hadoop framework when running Map/Reduce applications, we chose two applications: *Distributed Grep* and *Distributed Sort* applications which are included in the contributions delivered with Yahoo!'s Hadoop release.

The first application we consider is *Distributed Grep*. It is representative of a distributed job where huge input data needs to be processed in order to obtain some statistics. The application scans a huge text input file for occurrences of a particular expression and counts the number of lines where the expression occurs. The Distributed Grep application runs two Map/Reduce phases in sequence. Mappers simply output the value of these counters, then the Reducers sum up the outputs of the Mappers to obtain the final result (as described in Section 2.1.4). The access pattern generated by this application corresponds to concurrent reads from the same shared file.

We first write a huge input file to HDFS and BSFS respectively. In the case of HDFS, the file is written from a node that is not collocated with a datanode, in order to avoid the scenario where HDFS writes all data blocks locally. This gives HDFS the chance to perform some load-balancing of data blocks. Then we run the Distributed Grep Map/Reduce application and measure the job completion time. We vary the size of the input data from 1.5 GB to 9.5 GB in increments of 1.5 GB. Since a Hadoop data block is 64MB large and since usually Hadoop assigns a single Mapper to process such a data block, this roughly corresponds to varying the number of concurrent Mappers from 24 to 144.

Results obtained are represented in Figure 10. The relative gain of the original Hadoop framework with HDFS as storage over the modified Hadoop framework with HDFS as stor-

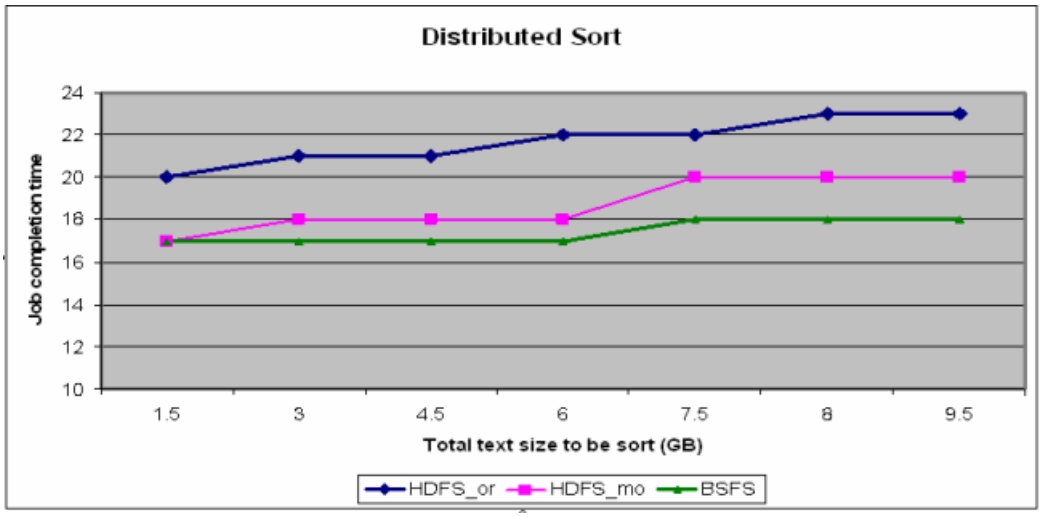

Figure 11: Performance of HDFS and BSFS with both Original Map/Reduce and Modified Map/Reduce in case of Distributed Sort Application.

age ranges from 1 % for 1.5 GB to 6 % for 9.5 GB. The reason that explains the difference is that storing intermediate data to DFS takes longer than to Local File System. As it can be also observed, the modified Hadoop framework with BSFS as storage outperforms the original Hadoop framework with HDFS as storage by 21 % and the gap decreases to 5 % for 9.5 GB. This is a direct consequence of how balanced is the block distribution for the input file. The superior load balancing strategy used by BlobSeer when writing the file has a positive impact on the performance of concurrent reads, whereas the HDFS suffers from the poor distribution of the file chunks.

The second application, *Distributed Sort*, uses the Map/Reduce framework to sort the input directory into the output directory. The Map function extracts sorting key from a text line and emits the key and the original text line as the intermediate key/value pair. The Reduce function passes the intermediate key/value pair unchanged as the output key/value pair. In a word, Distributed Sort application belongs to a kind of jobs that only need to run a filter on the input data that means no sorting or shuffling required by the job. No reduce function is needed since the map does all the file processing in parallel with no combine stage.

Running the Distributed Sort application involves deploying the Distributed File Systems (HDFS and BSFS) as well as original Hadoop Map/Reduce framework and modified Hadoop Map/Reduce framework. Similar to the Distributed Grep application above, an input file is written to HDFS and BSFS respectively and then we run the application and compare the job completion time in each scenario.

The data input size also varies from 1.5 GB to 9.5 GB with increments of 1.5 GB. The results displayed in Figure 11 show the completion time of the Distributed Sort application in all three scenarios previously described; the modified Hadoop framework with BSFS as storage outperforms the original Hadoop framework with HDFS as storage by 15 % and the gap increases to 22 % for 9.5 GB. We can also observe that the modified Hadoop framework with HDFS as storage outperforms the original Hadoop framework with HDFS as storage

with at least 9 % to 18 % for the tested range. The modified Hadoop Map/Reduce is in advantage compared to the original one, because the modified one saves intermediate data directly to DFS instead of first saving it to Local File System and after that, moving it to DFS like in the original Hadoop Map/Reduce framework. In case of applications like Distributed Sort with only map computations, the intermediate data is the output data as well.

5 Conclusion

Our focus for this report is the Map/Reduce programming model, which has recently emerged as a new trend in parallel processing technique to handle vast amount of data on large clusters of commodity servers. Efficiently supporting various cases of executions and applications requires that the framework executing them, as well as the data file system, are both extended with new properties. This work focuses on intermediate data management that can bring benefits at two levels. First, it supports properties of reliability and efficiency to intermediate data. Second, this also brings improvements in scenarios where the applications consist only of map phases.

5.1 Contribution

In this report we present a new approach that gives high priority to intermediate storage in dataflow program such as Map/Reduce. Our prototype was tested and experimented on the Grid'5000 testbed.

Preliminary results were encouraging as they suggest that our modification is capable of working under heavy concurrent access in a large-scale distributed environment. In comparison with the original Hadoop Map/Reduce, our prototype offers better performance as it provides not only greater reliability but also increased efficiency.

In order to implement the proposed design, we have conducted a detailed analysis of the Hadoop Map/Reduce framework and the HDFS and BlobSeer File System. This was not a simple job, however, due to the sheer scope of work. The Hadoop project is a complicated framework with 298MB of source code composed of 282 files for Map/Reduce framework, 115 files for HDFS and the BlobSeer File System has 100 source code files. For that reason, we chose to apply a spiral process in which each iteration requires analysis, design, implementation and experimentation in the Grid5000's testbed.

In addition, significant efforts were invested in the preparation of the experimental setup including defining an automated deployment process for both original Hadoop framework and the modified Hadoop framework, when using BlobSeer File System and HDFS as intermediate data storages. The deployment process involved:

- Generating configuration files, by assigning roles to each node, for both BSFS and HDFS.
- Launching the processes for each of the two file systems.
- Starting the Hadoop Map/Reduce framework.
- Generating the test file.
- Running Distributed Grep and Distributed Sort in each of the 3 scenarios.
- Collecting and parsing the results.

This process was repeated for various input data sizes. We had to overcome not trivial node management and configuration issues to reach this point.

5.2 Future Work

Using BSFS as an intermediate data storage layer, our improved Hadoop framework can be further optimized to address the case of re-running failed Mappers; in the situation where one Map task fails, the system can take full advantage of the stored data instead of rerunning the whole task. Supposed that on average, each Map task takes approximately 30 minutes to complete. If a task is 90 % completed when it fails, it will be wasteful if the jobtracker assigns another task to re-execute the job entirely from the beginning. By investigating ways to take advantage of the versioning property of BlobSeer, we can optimize the re-executed Map task to continue from the failure point instead of restarting the whole process. Understanding such a mechanism is a challenge for our future work in this area.

In addition, we also plan to expand our research to include experiments with Pig applications in order to assess the potential improvement in dataflow scenarios. Last but not least, we would like to compare our prototype with other approaches with respect to performance, scalability and usability.

References

[1] Dhruba Borthakur. HDFS Architecture. http://hadoop.apache.org/common/docs/r0.20.0/hdfs_design.pdf.

[2] Nicolae Bogdan, Antoniu Gabriel, and Bougé Luc. BlobSeer: How to enable efficient versioning for large object storage under heavy access concurrency. *EDBT/ICDT '09 Workshops*, pages 18–25, 2009.

[3] Raphael Bolze, Franck Cappello, Eddy Caron, Michel J. Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stéphane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quétier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *IJHPCA*, 20(4):481–494, 2006.

[4] Olston Christopher, Reed Benjamin, Srivastava Utkarsh, Kumar Ravi, and Tomkins Andrew. Pig Latin: a not-so-foreign language for data processing. *SIGMOD '08: The 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, 2008.

[5] Hadoop: The Definitive Guide. http://oreilly.com/catalog/9780596521974/index.html.

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

[7] Dean Jeffrey. Experiences with MapReduce, an abstraction for large-scale computation. *PACT '06: The 15th international conference on Parallel architectures and compilation techniques*, 2006.

[8] Dean Jeffrey and Ghemawat Sanjay. MapReduce: Simplified data processing on large clusters. *OSDI 2004*, pages 137–150, 2004.

[9] Kim Kiyoung, Jeon Kyungho, Han Hyuck, Kim Shin G., Jung Hyungsoo, and Yeom Heon Y. MRBench: A benchmark for MapReduce Framework. *ICPADS '08: The 2008 14th IEEE International Conference on Parallel and Distributed Systems*, pages 11–18, 2008.

[10] Steven Y. Ko, Imranul Hoque, Brian Cho, and Indranil Gupta. On availability of intermediate data in cloud computations. *The USENIX Workshop on Hot Topics in Operating Systems(HotOS)*, 2009.

[11] Grant Mackey, Saba Sehrish, Julio Lopez, John Bent, Salman Habib, and Jun Wang. Introducing Map-Reduce to High End Computing. *Petascale Data Storage Workshop at SC08*, 2008.

[12] Zaharia Matei, Konwinski Andy, Joseph Anthony D., Katz Randy, and Stoica Ion. Improving MapReduce performance in heterogeneous environments. *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[13] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. Distributed management of massive data: An efficient fine-grain data access scheme. *8th International Meeting on High Performance Computing for Computational Science VECPAR '08*, pages 532–543, 2008.

[14] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. Enabling lock-free concurrent fine-grain access to massive distributed data: Application to supernovae detection. *IEE International Conference on Cluster Computing 2008 Cluster '08*, pages 310–315, 2008.

[15] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. Enabling high data throughput in desktop grids through decentralized data and metadata management: The BlobSeer approach. *Euro-Par '09*, pages 404–416, 2009.

[16] Bogdan Nicolae, Diana Moise, Gabriel Antoniu, Luc Bougé, and Matthieu Dorier. BlobSeer: Bringing high throughput under heavy concurrency to Hadoop Map/Reduce applications. *24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[17] Owen OMalley and Arun C. Murthy. Winning a 60 Second Dash with a Yellow Elephant. http://sortbenchmark.org/Yahoo2009.pdf. April 2009.

[18] Hadoop Presentations. http://wiki.apache.org/hadoop/HadoopPresentations.

[19] The Hadoop Project. http://hadoop.apache.org/core.

[20] Ghemawat Sanjay, Gobioff Howard, and Leung Shun-Tak. The Google File System. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[21] Wwittawat Tantisiriroj, Swapnil Patil, and Garth Gibson. Data-intensive file systems for internet services: A rose by any other name ... *Technical Report CMU-PDL-08-114, Parallel Data Laboratory, CMU, Pittsburgh, PA*, October 2008.

[22] Hadoop wiki. http://wiki.apache.org/hadoop/FrontPage.

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