

Optimizing Intermediate Data Management in MapReduce Computations

Diana Moise

INRIA Rennes - Bretagne
Atlantique/IRISA
diana.moise@inria.fr

Thi-Thu-Lan Trieu

ENS Cachan, Brittany/IRISA
lan.trieu@irisa.fr

Gabriel Antoniu

INRIA Rennes - Bretagne
Atlantique/IRISA
gabriel.antoniu@inria.fr

Luc Bougé

ENS Cachan, Brittany/IRISA
luc.bouge@bretagne.ens-cachan.fr

ABSTRACT

Many cloud computations process large datasets. Programming paradigms have been proposed to design this type of applications, so as to take advantage of the huge processing and storage options the cloud holds, but at the same time, to provide the user with a clean and easy to use interface. Among these programming models, we consider the MapReduce paradigm and its reference implementation, the Hadoop framework. We focus on the aspect of intermediate data, that is data produced and transferred between the two stages of the computation (*map* and *reduce*). The goal of this paper is to propose a storage mechanism for intermediate data with the purpose of optimizing the execution of MapReduce applications in the presence of failures, while keeping the impact on the job completion time to the minimum. To meet this goal, we rely on a fault-tolerant, concurrency-optimized data storage layer based on the BlobSeer data management service. We modify the Hadoop MapReduce framework to store the intermediate data in this layer (acting as a BlobSeer-based distributed file system) rather than using the local storage of the mappers, as in the vanilla version of Hadoop. To validate this work, we perform experiments on a large number of nodes of the Grid'5000 testbed. We demonstrate that our approach not only provides for intermediate data availability in case of failures, but also efficiently handles read/write accesses so that the overall job completion time is substantially improved.

Categories and Subject Descriptors

H.3 [Information Storage and Retrieval]: Miscellaneous

Keywords

MapReduce, cloud computing, intermediate data management, Hadoop, HDFS, BlobSeer

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CloudCP '11 April 10, 2011 Salzburg, Austria
Copyright 2011 ACM 978-1-4503-0727-7/11/04 ...\$10.00.

1. INTRODUCTION

Cloud computing is a powerful new paradigm for managing resources, while offering scalable, highly-available services using a pay-per-use model. A large part of cloud-based applications are data-intensive; whether they are scientific applications or Internet services, the data volume they process is continuously growing. Programming paradigms have been proposed to design applications handling large datasets, so as to take advantage of the huge processing and storage options the cloud holds, but at the same time, to provide the user with a clean and easy to use interface.

The MapReduce [1] paradigm has recently been proposed as a solution for rapid implementation of distributed data-intensive applications. Ever since it was introduced by Google, the MapReduce programming model gained in popularity, thanks to its simple, yet versatile interface. The paradigm has also been implemented by the open source community through the Hadoop project, maintained by the Apache Foundation and supported by Yahoo!, and even by Google. The MapReduce paradigm has been recently introduced as a cloud service through Amazon's Elastic MapReduce [12] that basically offers the Hadoop implementation on top of the Elastic Compute Cloud infrastructure (EC2, [11]). A MapReduce computation takes a set of input key/value pairs, and produces a set of output key/value pairs. Implementing an application using MapReduce requires specifying two functions: 1) *map*, that processes a key/value pair to generate a set of intermediate key/value pairs; and 2) *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. All these aspects are handled transparently for the user. The concept proposed by MapReduce under a simplified interface is powerful enough to suit a wide range of applications.

Cloud computations, whether they are based on MapReduce, Dryad [5], Pig latin [10], and others, all have in common the existence of *intermediate data* that is data produced and transferred between stages of computation. The goal of this paper is to propose a storing mechanism for intermediate data with the purpose of optimizing the execution of MapReduce applications in the presence of failures, while keeping the impact on the job completion time to the minimum. Most approaches for storing intermediate data rely on writing the data to the local file system of the node generating it. However, by doing so, if the node storing the data

crashes, all intermediate data is lost; what the frameworks typically do, is to reschedule the failed task and produce the intermediate data all over again. This is a costly design choice when dealing with applications consisting of multiple computing stages; as shown in [7], a failure can lead to *cascaded re-execution*, which means that some tasks in all the stages up to the failed one have to be re-executed. This is inconceivable in large-scale environments where failures happen on a daily basis [7]. To store intermediate data in a way that is both efficient and provides data-availability, we rely on BlobSeer, a concurrency-optimized BLOB (Binary Large Object) management system. In previous work [8] we illustrated how BlobSeer could be used as a storage layer for MapReduce applications; we now investigate how BlobSeer can also be used for storing intermediate data. In section 2 we take a closer look at the properties of intermediate data and at the way it is handled in the Hadoop framework. Section 3 describes our approach to handling intermediate data in the Hadoop project and the steps we made in order to put it into practice. We validated our proposal through large-scale experiments, detailed in section 4.

2. 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. Intermediate data in the MapReduce context, takes the form of the key/value pairs generated by the map phase of the application. All intermediate values associated with the same intermediate key are grouped together and passed to the reduce function. 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.

2.1 Hadoop

The Hadoop project [2] provides, among others, an open-source implementation of Google's MapReduce model through the Hadoop MapReduce framework [3]. The framework was designed to work on clusters of commodity hardware; the cluster nodes play the role of several entities: 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 split into chunks of equal size, that are stored in a distributed file system across the cluster. Each tasktracker executing a map task is assigned a chunk of the input file; after all the maps have finished, the tasktrackers execute the reduce function on the map outputs.

The Hadoop Distributed File System (HDFS) [4] is part of the Hadoop project. In HDFS a file is split into 64 MB chunks that are placed on storage nodes called *datanodes*. A centralized *namenode* is responsible for keeping the file

metadata and the chunk location. As it was designed for specific workloads, HDFS is not POSIX compliant and has special semantics regarding write operations: HDFS does not support concurrent writes to the same file; moreover, once a file is created, written and closed, the data cannot be overwritten or appended to. Some optimization techniques are employed by HDFS in order to improve the over-all throughput. Client-side buffering is used for small I/O operations and consists in prefetching a whole data chunk when a read request has been issued for a block belonging to that chunk, and in buffering all write operations until the data reaches the size of a chunk (64MB). Another mechanism in HDFS is to expose the data layout to the Hadoop scheduler (the jobtracker). This is done to compensate for the policy of randomly distributing chunks to datanodes, which leads to unbalanced data layout. The jobtracker will use the information about data distribution to place the computation as close as possible to the needed data.

2.2 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, thus 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 several 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 they are told to delete them by the jobtracker, which is after the job has completed. The output of the reduce phase is written to a distributed file system (by default, HDFS).

2.3 Effect of failures

The importance of intermediate data management is best illustrated when considering failures. 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. Whereas 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 output to disk, and when the reducers are copying the map outputs to their local file system. In both cases, a mapper node failure leads to the map output (partially or completely generated) being lost; consequently, the reducers are not able to transfer the data and proceed further in the computation. The policy Hadoop uses in this situations, is to restart the execution of the failed mapper on another node; when the jobtracker becomes aware of a tasktracker

failure, it simply reschedules the map task. This approach however, implies re-generating the intermediate data, which is an overhead that translates into additional runtime.

3. OUR APPROACH

In order to address the issues described in section 2.3, 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 choose the DFS that fits the specific features of intermediate data and also optimally satisfies the availability and efficiency requirements. In this work, we rely on BlobSeer, a concurrency-optimized data management system, to deal with the problem of efficiently and reliably handling intermediate data in MapReduce computations.

3.1 BlobSeer - overview

BlobSeer [8] is a data-management service that aims at providing efficient storage for data-intensive applications. BlobSeer uses the concept of *BLOBs* (binary large objects) as an abstraction for data; a BLOB is a large sequence of bytes (its size can reach the order of TB), uniquely identified by a key assigned by the BlobSeer system. Each BLOB is split into even-sized blocks, called pages; in BlobSeer, the page is the data-management unit, and its size can be configured for each BLOB. BlobSeer provides an interface that enables the user to create a BLOB, to read/write a range of bytes given by offset and size from/to a BLOB and to append a number of bytes to an existing BLOB. In BlobSeer, data is never overwritten: each write or append operation generates a new version of the BLOB; this snapshot becomes the latest version of that BLOB, while the past versions can still be accessed by specifying their respective version numbers. BlobSeer's architecture comprises several entities. The *providers* store the pages, as assigned by the *provider manager*; the distribution of pages to providers aims at achieving load-balancing. The information concerning the location of the pages for each BLOB version is kept in a Distributed HashTable, managed by several *metadata providers*. Versions are assigned by a centralized *version manager*, which is also responsible for ensuring consistency when concurrent writes to the same BLOB are issued.

3.2 Using BlobSeer as storage for intermediate data

Our approach aims at using BlobSeer as storage layer for the intermediate data generated by MapReduce applications. We focus on evaluating our approach within the Hadoop project, which implies first, allowing the intermediate data to be stored in a DFS at the level of the Hadoop MapReduce framework, and second, building a Hadoop file system interface on top of BlobSeer. We further present in detail these two steps.

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; figure 1(a) illustrates this process. 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 first divided into partitions corresponding to the reducers that will process them; the data within each partition is then sorted by key. The content of the buffer is 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 output file. Each partition in this file is copied by its assigned reducer over HTTP; a reduce task starts copying in parallel the map outputs it needs to process, as soon as they become available. As figure 1(b) shows, the map outputs are copied to a memory buffer which is 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 not relevant to the focus of this work.

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 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; on the reducer side, we adjusted the code to read the outputs it needs from the files in the DFS, and into the memory buffer. These modifications were possible also because we stored each map output file in the DFS by preserving the path and the name under which the file was stored on disk, in the original version of Hadoop. When a tasktracker starts running the reduce function, the jobtracker will send the list of file names in the DFS, instead of the mappers that store the outputs, as it was the case for the unmodified Hadoop framework. Next, instead of copying the files from the mappers' local filesystem, the tasktracker starts reading data from the specified files stored in the DFS.

Integrating BlobSeer with Hadoop.

The features BlobSeer exhibits meet the storage needs of MapReduce 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 layer consists in a centralized *namespace manager*, which is responsible for maintaining a file system namespace, and for mapping files to BLOBs. We also implemented the optimization techniques employed by HDFS, described in section 2.1. More details about how we integrated BlobSeer with Hadoop can be found in [9].

4. EXPERIMENTAL EVALUATION

In order to evaluate the benefits of using BlobSeer as storage for intermediate data in Hadoop, 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 involves direct accesses to the file system, through

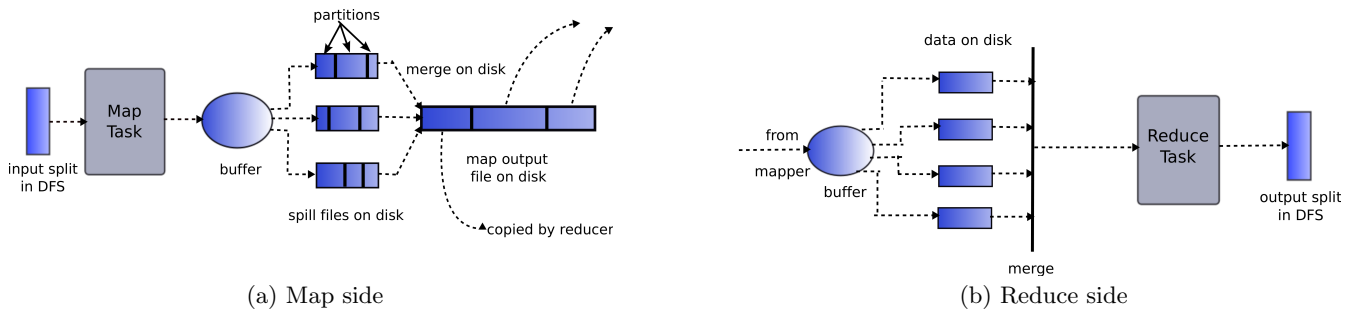


Figure 1: Intermediate data in the original Hadoop MapReduce framework

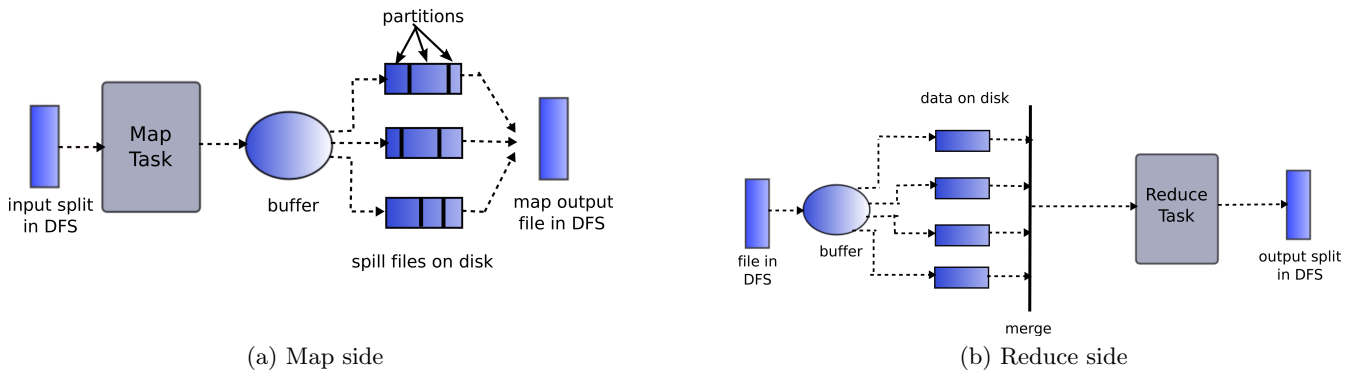


Figure 2: Intermediate data in the modified Hadoop MapReduce framework

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, accessing the storage layer indirectly, through the MapReduce framework. 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 having BSFS act as storage layer, is accomplished through a performance comparison of HDFS and BSFS in both the microbenchmarks and the execution of real MapReduce applications. The environmental setup as well as the experiments and the obtained results are further presented.

4.1 Environmental setup

The experiments were carried out on the Grid'5000 [6] experimental platform. Grid'5000 provides a testbed for supporting experiments distributed at large scales, while offering a high degree of flexibility with respect to the resources it holds. The Grid'5000 infrastructure consists of more than 20 clusters geographically distributed over 9 sites throughout the French territory. 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. Intracluster 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.

4.2 Microbenchmarks

The goal of the microbenchmarks is to evaluate the through-

put achieved by BSFS and HDFS when multiple, concurrent clients access the file systems, under several test scenarios. The scenarios we chose simulate the access patterns exhibited while generating and processing intermediate data in our modified version of the Hadoop MapReduce framework. The microbenchmarks were performed using 270 nodes, on which 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 are 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. For each microbenchmark we measured the average throughput achieved when multiple concurrent clients perform the same set of operations on the file systems. 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 and the average results are discussed below.

Concurrent writers, each writing to a different file.

In this test scenario, we start an increasing number of clients that write to HDFS/BSFS concurrently. Each client writes a 1 GB file sequentially in blocks of 64 MB. This pattern of concurrent clients writing to different files reproduces the last step in our modified "map" phase, when the mappers write their output to the DFS; each mapper writes its sorted and partitioned output to a unique file in the DFS. Figure 3 shows the write performance of both HDFS and BSFS.

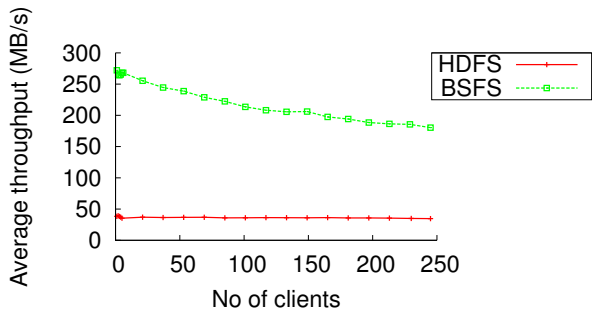


Figure 3: Performance of HDFS and BSFS when concurrent clients write different files

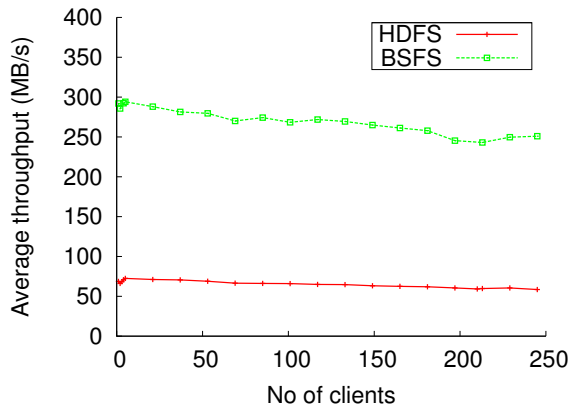


Figure 4: Performance of HDFS and BSFS when concurrent clients read from the same file

The write average throughput delivered by HDFS is almost constant, due to HDFS’s replica placement policy: when a client writes on a machine where a datanode was started, the first copy (and the only one, in this case) is written locally. 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 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. The file is created so that the chunks are distributed among the datanodes/providers for both HDFS and BSFS. This test case simulates the beginning of the “reduce” phase when the tasktrackers read the outputs belonging to their assigned partitions; these outputs are spread over several files stored in the DFS. As figure 4 shows, BSFS’s throughput is significantly higher, which is a consequence of BlobSeer’s data distribution scheme: the shared file is uniformly striped among the providers, using a round robin pattern. On the other hand, 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 re-

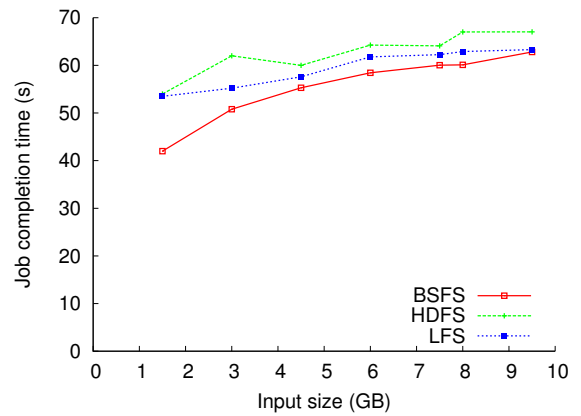


Figure 5: Distributed grep

quests.

4.3 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; the deployment configuration for HDFS and BSFS is similar to the one described in 4.2, with the remark that for BlobSeer we deployed 10 metadata providers instead of 20 (as the number of providers to handle is halved). 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 (original Hadoop framework), HDFS and BSFS (our modified version of Hadoop). 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 supplied by the user and the output 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

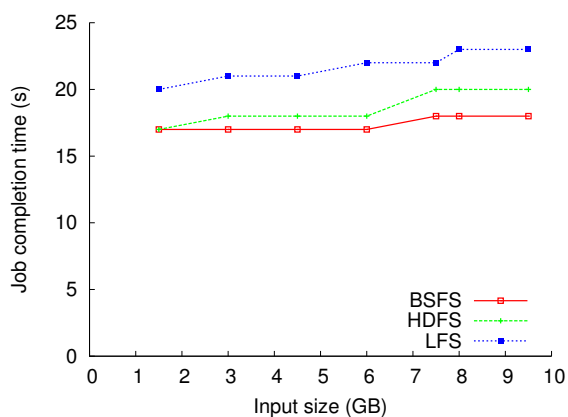


Figure 6: Distributed sort

chunks spread across the datanodes/providers. The Hadoop jobtracker starts a mapper to process each chunk from the input file, consequently the number of mappers to produce intermediate data ranges between 24 and 152 mappers; the amount of intermediate data written by the mappers and read by the reducers is small, in the case of the grep application. As can be seen in figure 5, 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 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 text line and emits the key and the original text line 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 6 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 in-

put 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.

5. CONCLUSION

As cloud computations are becoming more complex and the datasets they process are continuously growing, special care must be given to every aspect of the underlying framework. In this paper, we address the problem of efficiently storing intermediate data generated by MapReduce applications, even in the presence of failures. We proposed to modify the Hadoop MapReduce framework that stores the intermediate data in the distributed file system (DFS) acting as storage for the user data (input and output files). We tested our approach with 2 distributed file systems: HDFS and our BlobSeer-based BSFS; our experiments showed that the modified version of Hadoop with BSFS as storage layer for intermediate data, is able to complete MapReduce applications faster, while providing data availability in case of mapper failure. Moreover, our approach of storing the intermediate data in the DFS, proved to be highly gainful in the case of MapReduce application that have a trivial “reduce” phase, as the intermediate data is also the final output. In case of failures, we believe that resuming the map computation from where the failure took place, instead of re-executing the failed task and producing the intermediate data again, saves a significant amount of time when running multi-stage applications. As future direction, we plan to improve the jobtracker’s scheduling policy to incorporate these actions; a validating scenario for this optimization would be running pipeline MapReduce applications (through Pig) in the presence of failures.

6. REFERENCES

- [1] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [2] The Apache Hadoop Project. <http://www.hadoop.org>.
- [3] The Hadoop Map/Reduce Framework. <http://hadoop.apache.org/mapreduce/>.
- [4] HDFS. The Hadoop Distributed File System. http://hadoop.apache.org/common/docs/r0.20.1/hdfs_design.html.
- [5] M. Isard, M. Budiou, Y. Yu, A. Birrell, and al. Dryad: distributed data-parallel programs from sequential building blocks. In *Procs of the 2nd ACM SIGOPS/EuroSys 2007, EuroSys '07*, pages 59–72, New York, NY, USA, 2007. ACM.
- [6] Y. Jégou, S. Lantéri, M. Leduc, and al. Grid’5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, November 2006.
- [7] S. Y. Ko, I. Hoque, B. Cho, and al. On availability of intermediate data in cloud computations. *Procs of the USENIX Workshop on Hot Topics in Operating Systems*, 2009.
- [8] B. Nicolae, G. Antoniu, L. Bougé, and al. BlobSeer: Next generation data management for large scale infrastructures. *J. Parallel Distrib. Comput.*, Aug 2010.
- [9] B. Nicolae, D. Moise, G. Antoniu, and al. BlobSeer: Bringing high throughput under heavy concurrency to Hadoop Map/Reduce applications. In *Procs of the 24th IPDPS 2010*, 2010. In press.
- [10] C. Olston, B. Reed, U. Srivastava, and al. Pig latin: a not-so-foreign language for data processing. In *Procs of the*

2008 ACM SIGMOD, pages 1099–1110, New York, NY, USA, 2008. ACM.

- [11] Amazon Elastic Compute Cloud (EC2).
<http://aws.amazon.com/ec2/>.
- [12] Amazon Elastic MapReduce.
<http://aws.amazon.com/elasticmapreduce/>.