Improving the Hadoop Map/Reduce Framework to Support Concurrent Appends through the BlobSeer BLOB management system

Diana Moise, Gabriel Antoniu, Luc Bougé

Improving the Hadoop Map/Reduce Framework to Support Concurrent Appends through the BlobSeer BLOB management system

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

Hadoop is a reference software framework supporting the Map/Reduce programming model. It relies on the Hadoop Distributed File System (HDFS) as its primary storage system. Although HDFS does not offer support for concurrently appending data to existing files, we argue that Map/Reduce applications as well as other classes of applications can benefit from such a functionality. We provide support for concurrent appends by building a concurrency-optimized data storage layer based on the BlobSeer data management service. Moreover, we modify the Hadoop Map/Reduce framework to use the append operation in the "reduce" phase of the application. To validate this work, we perform experiments on a large number of nodes of the Grid'5000 testbed. We demonstrate that massively concurrent append and read operations have a low impact on each other. Besides, measurements with an application available with Hadoop show that the support for concurrent appends to shared file is introduced with no extra cost, whereas the number of files managed by the Map/Reduced framework is substantially reduced.

Keywords

Map/Reduce, data-intensive applications, scalable data management, append, concurrency, Hadoop, HDFS, BlobSeer

1. INTRODUCTION

More and more applications today generate and handle very large volumes of data on a regular basis. Governmental and commercial statistics, climate modeling, cosmology, genetics, bio-informatics, high-energy physics are just a few examples of fields where it becomes crucial to efficiently manipulate massive data, which are typically *shared* at the global scale. Data volumes of applications in such fields are expected to double every two years over the next decade and further. With this continuing data explosion, it is necessary to store and process data efficiently by leveraging the huge computing power that is today available.

The Map/Reduce [2] paradigm has recently been proposed and is now being used by large Internet service providers to perform computations on massive amounts of data. After having been strongly promoted by Google, it 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. This model is currently getting more and more popular as a solution for rapid implementation of distributed data-intensive applications. Its growing popularity is explained by its simplicity. A Map/Reduce computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the Map/Reduce library expresses the computation through 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 Map/Reduce under a simplified interface is powerful enough to suit a wide range of applications.

The Map/Reduce paradigm has been recently introduced in the cloud computing context through Amazon's Elastic MapReduce [16] that offers Map/Reduce as a service on the Elastic Compute Cloud infrastructure (EC2, [15]). Other directions focused on adapting existing file systems belonging to the HPC community, file systems like PVFS (Parallel Virtual File System) [1] and GPFS(General Parallel File System) [13], to fit the needs of Map/Reduce applications. Some efforts aim at improving the Map/Reduce framework according to the context it is used in; in [14] an improvement in the scheduling algorithm of Hadoop is proposed in order to efficiently employ the framework in heterogeneous environments.

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

One important aspect of scalability regards the number of files that need to be managed by the file system. Acquiring and storing large data sets of the order of hundreds of TB and beyond using KB-sized files is unmanageable and inefficient, due to the overhead incurred by the file system for file metadata management. This issue, known as the "file-count problem", has been acknowledged as a major source of inefficiency for large-scale settings of distributed file systems. To take a representative example, according to its designers, Google File System is facing this problem and therefore is likely to undergo substantial design changes in the near future [9].

Instead of managing very large sets of small files, a better approach for handling such very large data sets of small pieces of data consists in packing these pieces of data together into huge files (e.g. gathering hundreds of GB or TB of data). Consequently, massively parallel data generation leads to a large number of processes appending records to a huge, shared file. This is why we believe that providing an efficient support for the append operation under heavy concurrency will be increasingly important in the forecoming years in the context of data-intensive applications.

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

To enable such a feature, we rely on the approach proposed by the BlobSeer [10] versioning-based, concurrency-optimized BLOB (Binary Large Object) management system. In previous work [12] we presented how BlobSeer could be used as a storage substrate providing the same interface as Hadoop's default file system (HDFS), thanks to a file system layer (BSFS) built on top of BlobSeer. We now make a step further and integrate BSFS into Hadoop, to allow Hadoop's Map/Reduce applications to benefit from BlobSeer's efficient support for concurrent data access to shared data. In Section 2 we explain with more details the need for an efficient support to the append operation and we briefly describe the state-of-the art with respect to this feature. We then describe the steps we made in order to enable the use of appends by the Hadoop framework, thanks to the use of BlobSeer (Section 3). To validate the approach, experiments with a few microbenchmarks (presented in Section 4) demonstrate that concurrent append and read operations on the same shared file have a low impact on each other. Then, experiments with a real Map/Reduce application (available with Hadoop) show that BlobSeer enables the management of concurrent appends to shared files with no overhead with respect to the setting where the original Hadoop generates many small files. The advantage is obvious in terms of simplicity: at the end of the computation data is already available in a single logical file, ready to use for any subsequent processing, whereas in the original Hadoop framework data are scattered in many separated files across the distributed nodes and extra application logic is needed to handle this group of files for subsequent processing.

2. THE NEED FOR THE APPEND OPERATION IN MAP/REDUCE FRAMEWORKS

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

2.1 Append: motivation and state of the art

Potential benefits of the append operation. Map/Reduce data processing applications are not the only class of applications that may potentially benefit from an efficient support of the append operation in a file system. In Google File System, record append is heavily used in the context of applications following the multiple-producer/single-consumer model. Whereas HDFS is concerned, supporting append can enable applications that require a stronger API, to use the file system as storage back-end. An example of such application is HBase [6], an open-source project from Hadoop, designed after Google's BigTable system, with the purpose of providing distributed, column-oriented storage of large amounts of structured data, on top of HDFS. HBase keeps its transaction log in main memory and periodically, flushes it to HDFS; if a crash occurs, HBase can recover its previous state by going through the transaction log. However, although the transaction log can be opened for reading, after recovery, HBase will write its updates to a different file in HDFS. Supporting appends can enable HBase, as well as other database applications, to keep their ever-expanding transaction log as a single huge file, stored in HDFS. At the level of the Hadoop Map/Reduce framework, a single output file can be generated in the "reduce" phase, instead of having each reducer writing its output to a different file. In a scenario with multiple Map/Reduce applications that can be executed in pipeline, the framework can rely on append to significantly improve execution time, by allowing readers to work in parallel with appenders: applications that generate the data can append it concurrently to shared files, while at the same time, applications that process the data can read it from those files.

Append: status in Google File System. To meet its storage needs, Google introduced the Google File System (GFS) [3], a distributed file system that supports large-scale data processing on commodity hardware. GFS is optimized for access patterns involving huge files that are mostly appended to, and then read from; since applications that exhibit these types of access patterns were targeted, supporting append was a critical functionality and thus was implemented from the beginning. The append operation is called *record append* and is implemented in such way as to guarantee atomicity;

its purpose is to enable multiple clients to append data to the same file concurrently. The clients supply only the data to be appended and GFS ensures that the data will be appended to the file as a continuous sequence of bytes; the offset the data is appended at, is chosen by GFS, and is returned to the client issuing the append.

Append: status in HDFS. The Hadoop Distributed File System was developed with the initial purpose of supporting applications that follow the Map/Reduce programming paradigm. These applications process files that comply with the write-once-read-many model; HDFS's features and semantics were designed to suit this model. However, the growing popularity of HDFS, as well as the variety and the increasing number of applications that can be modeled using the Map/Reduce paradigm, led to the need of extending HDFS with more functionalities. One of these required functionalities is the support for append operations. In early versions of HDFS, files were immutable once closed. They were visible in the file system namespace only after a successful close operation. Implementing append in HDFS required substantial modifications to the whole framework; shortly after being introduced, append support was disabled, because all the changes it involves are still an open issue.

2.2 Hadoop Map/Reduce and HDFS

Hadoop's [5] implementation of Map/Reduce follows the Google model: it provides an open-source implementation of Google's Map/Reduce model. The framework consists of a single master *jobtracker*, and multiple slave *tasktrackers*, one per node. A Map/Reduce job is split into a set of tasks, which are executed by the tasktrackers, as assigned by the jobtracker. The input data is also split into chunks of equal size, that are stored in a distributed file system across the cluster. First, the map tasks are run, each processing a chunk of the input file, by applying the map function defined by the user, and generating a list of key-value pairs. After all the maps have finished, the tasktrackers execute the reduce function on the map outputs.

The Hadoop Distributed File System (HDFS) [7] is part of the Hadoop project [4]. HDFS uses the same design concepts as GFS: 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. When accessing a file, a client first contacts the master to get the datanodes that store the required chunks; all file I/O operations are then performed through a direct interaction between the client and the datanodes.

Like most file systems developed by the Internet services community, HDFS is optimized for specific workloads and has different semantics than the POSIX compliant file systems. HDFS does not support concurrent writes to the same file; moreover, once a file is created, written and closed, the data cannot be overwritten or appended to. HDFS is not optimized for small I/O operations, however it uses client side buffering to improve the throughput. Clients buffer all write operations until the data reaches the size of a chunk (64MB). HDFS also implements readahead buffering: when HDFS receives a read request for a small block, it prefetches the entire chunk that contains the required block. Another technique HDFS uses to achieve an overall high throughput, is to expose the data layout to the Hadoop scheduler (the jobtracker). When distributing the chunks among datanodes, HDFS picks random servers to store the data, which will often lead to a layout that is not load balanced. To make up for this, the scheduler will try to place the computation as close as possible to the needed data; HDFS provides the information about the location of each chunk, and the jobtracker will use it to execute tasks on datanodes in such way as to achieve load balancing across all nodes.

3. INTRODUCING SUPPORT FOR THE APPEND OPERATION IN HADOOP

3.1 Background: the BlobSeer approach

3.1.1 BlobSeer: overview

BlobSeer [10] 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 is 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. BlobSeer implements fault tolerance through page-level replication and offers persistence through a BerkleyDB layer. Results in [11] show that BlobSeer is able to sustain high throughput under heavy access concurrency, for various access patterns.

3.1.2 BlobSeer: optimizations for append

In BlobSeer, append is implemented as a special case of the write operation, in which the offset is implicitly assumed to be the size of the latest version of the BLOB. For an append operation, the user supplies the data to be stored and receives the number of the version this update generates. The input data is split into pages that are then written in parallel to a list of providers retrieved from the provider manager. When all the pages are successfully written to the providers, the version manager assigns a number to the newly generated BLOB version. The design concepts BlobSeer uses enable a high degree of parallelism, especially where updates are concerned. Multiple clients can append their data in a fully parallel manner, by asynchronously storing the pages on providers; synchronization is required only when writing the metadata, but this overhead is low, as shown in [10].

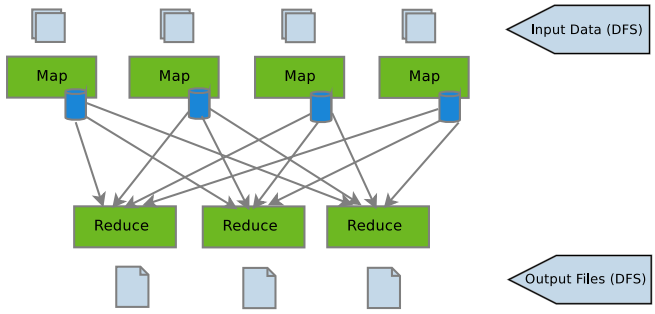

Figure 1: Original Hadoop framework: each reducer writes to a separate file

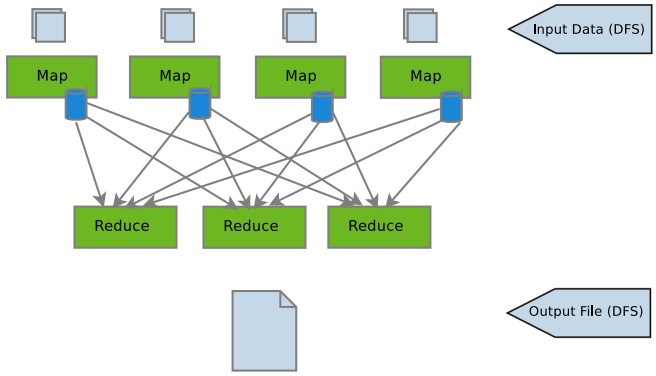

Figure 2: Modified Hadoop framework: all the reducers append to the same file

3.2 How BlobSeer enables appends in Hadoop

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

Modifying Hadoop to use appends. In the original Hadoop Map/Reduce framework (figure 1), when a tasktracker executes the "reduce" function specified by the user, the output is written to a temporary file; each temporary file has a unique name, so that each reducer writes to a distinct file. When the "reduce" phase is completed, each reducer renames the temporary file to the final output directory, specified by the user. The final result obtained by running the Map/Reduce application, consists of multiple parts, one part per reducer. We modified the reducer code to append the output it produces to a single file, instead of writing it to a distinct file (figure 2). Having all the reducers append to the same file, impacts on both the application running on top of the framework, and the file system storing the data. An application consisting of multiple Map/Reduce instances that can be executed in pipeline, is able to complete substantially faster by running in parallel "map" and "reduce" phases from different stages. Mappers from one stage of the pipeline open the input file for reading in order to process the data, while reducers from the previous stage can still generate the data and append it to the same file. The append operation reduces the number of files to be stored in the distributed file system that serves as storage for the application executed by the Hadoop framework. This impacts on the namespace management, by considerably reducing the metadata associated to files.

Supporting appends at the file system level. The features BlobSeer exhibits meet the storage needs of Map/Reduce applications. In order to enable BlobSeer to be used as a file system within the Hadoop framework, we added an additional layer on top of the BlobSeer service, layer that we called the *BlobSeer File System - BSFS*. This 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 a caching mechanism for read/write operations, as Map/Reduce applications usually process data in small records (4KB, whereas Hadoop is concerned). This mechanism prefetches a whole block when the requested data is not already cached, and delays committing writes until a whole block has been filled in the cache. To make the Map/Reduce scheduler data-location aware, we extended BlobSeer with a new primitive, that exposes the pages distribution to providers. More details about how we integrated BlobSeer with Hadoop can be found in [12].

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

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

4. EXPERIMENTAL EVALUATION

To test the append functionality, we performed two types of experiments: at the level of the file system and at the level of the Hadoop framework. The first type of experiments involve direct accesses to the file system, through the interface it exposes; we will further refer to these tests as microbenchmarks. The second class of experiments consist in running Map/Reduce applications and thus, accessing the storage layer indirectly, through the Map/Reduce framework. The environmental setup as well as the experiments and the obtained results are further presented.

4.1 Environmental setup

The experiments were performed on the Grid'5000 [8] testbed, a large-scale experimental grid platform, with an infrastructure geographically distributed on 9 different sites in France. Users of the Grid'5000 platform are offered a high degree of flexibility with respect to the resources they request. The tools Grid'5000 offers allow the users to reconfigure and adjust resources and environments to their needs, and also to monitor and control their experiments. For this series of experiments we use the nodes of the Orsay cluster. Both the microbenchmarks and the Map/Reduce applications were performed using 270 nodes, on which we

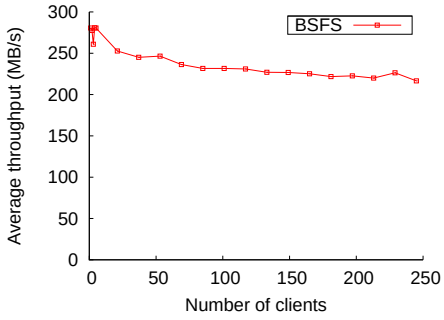

Figure 3: Performance of BSFS when concurrent clients append data to the same file

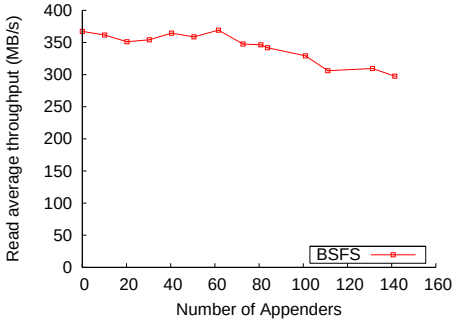

Figure 4: Impact of concurrent appends on concurrent reads from the same file

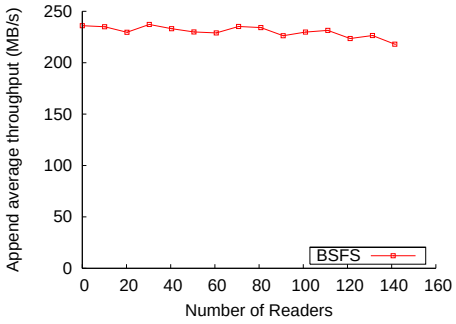

Figure 5: Impact of concurrent reads on concurrent appends to the same file

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.

4.2 Microbenchmarks

The goal of the microbenchmarks is to evaluate the throughput achieved by BSFS when multiple, concurrent clients access the file systems, under several test scenarios. The scenarios we chose involve the append operation and represent access patterns exhibited by the Map/Reduce applications described in section 2. For each microbenchmark we measure the average throughput achieved when multiple concurrent clients perform the same set of operations on the file system. The clients are launched simultaneously on the same machines as the datanodes (data providers, respectively). The number of concurrent clients ranges from 1 to 246. Each test is executed 5 times, for each set of clients.

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

Concurrent appends to the same file. In this test case, N concurrent clients append each a 64 MB chunk to the same file. The results are displayed on Figure 3. They show that BSFS maintains a good throughput as the number of appenders increases. This scenario illustrates the data access pattern exhibited by the modified Hadoop framework, in which all the reducers append their outputs to the same file, instead of creating many output files as it is done in the original version of Hadoop.

Concurrent reads and appends to the same file. The test shown in Figure 4 assesses the performance of concurrent read operations from a shared file, when they are executed simultaneously with multiple appends to the same file. The test consists in deploying 100 readers and measuring the average throughput of the read operations for a number of concurrent appenders that ranges between 0 (only readers) and 140. Each reader processes 10 chunks of 64 MB and each appender writes 16 such chunks to the shared file. Each client processes disjoint regions of the file. The obtained results show that the average throughput of BSFS reads is sustained even when the same file is accessed by multiple concurrent appenders. As a consequence of the versioning-based concurrency control in BlobSeer, the appenders work on their own version of the file, and thus do not interfere with the older versions accessed by read operations.

Concurrent appenders maintain their throughput as well, when the number of concurrent readers from a shared file increases, as can be seen on Figure 5. In this experiment, we fixed the number of appenders to 100 and varied the number of readers accessing the same file from 0 to 140. Both readers and appenders access 10 chunks of 64 MB.

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

4.3 Application study: data join

In order to evaluate how supporting the append operation influences the performance of the Hadoop framework when running a Map/Reduce application, we chose the *data join* application that is included in the contributions delivered with Yahoo!'s Hadoop release. The *data join* application is

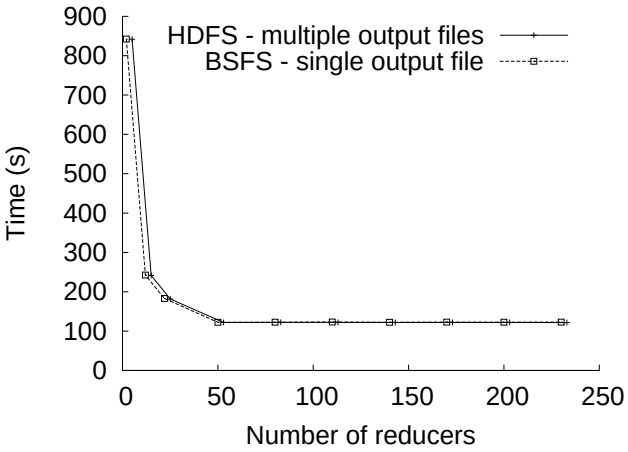

Figure 6: Completion time of the *data join* application when varying the number of reducers

similar to the *outer join* operation from the database context. *Data join* takes as input two files consisting of key - value pairs, and merges them based on the keys from the first file that appear in the second file as well. The generated output consists of 3 columns: the key from the first file and the two values associated to the key in each of the files. If a key in the first file appears more than once in either one of the two files, the output will contain all the possible combinations. The keys that appear only in the first file are not included in the output.

Running the *data join* application involves deploying the distributed file systems (HDFS and BSFS) as well as the Hadoop Map/Reduce framework. The environmental setup is similar to the one described in 4.1; one dedicated machine acted as the jobtracker, while the tasktrackers were co-deployed with the datanodes/providers.

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

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

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

The results displayed in figure 6 show the completion time of the *data join* application in both of the scenarios previously described. BSFS finishes the job in approximately the same amount of time as HDFS, and moreover, it produces a single output file, by supporting the append operation. The completion time in both scenarios remains constant even when the number of reducers increases, because *data join* a computation-intensive application, and most of the time is spent on searching and matching keys in the "map" phase, and on combining key-value pairs in the "reduce" phase.

5. CONCLUSION

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

Based on the use of BSFS as a storage layer, our improved Hadoop framework can further be optimized for the case of Map/Reduce applications that are executed in pipeline. For this type of applications, the mappers and the reducers belonging to distinct stages of the pipeline, can concurrently be executed: the reducers generate the data and append it to a file that is at the same time, read and processed by the mappers. As suggested by our microbenchmarks, the impact on concurrent reads and appends on each other is low. As future work, we plan to validate the append operation implemented both by the file system and in the Hadoop framework, by testing such a scenario in which sev-

eral Map/Reduce applications execute in a pipeline.

6. REFERENCES

[1] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

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

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *SIGOPS - Operating Systems Review*, 37(5):29–43, 2003.

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

[5] The Hadoop Map/Reduce Framework. `http://hadoop.apache.org/mapreduce/`.

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

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

[8] Y. Jégou, S. Lantéri, J. Leduc, M. Noredine, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and T. Iréa. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, November 2006.

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

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

[11] B. Nicolae, G. Antoniu, and L. Bougé. Enabling high data throughput in desktop grids through decentralized data and metadata management: The blobseer approach. In *Proc. 15th International Euro-Par Conference on Parallel Processing (Euro-Par '09)*, volume 5704 of *Lect. Notes in Comp. Science*, pages 404–416, Delft, The Netherlands, 2009. Springer-Verlag.

[12] B. Nicolae, D. Moise, G. Antoniu, L. Bougé, and M. Dorier. BlobSeer: Bringing high throughput under heavy concurrency to Hadoop Map/Reduce applications. In *Proc.24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*, 2010. In press.

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

[14] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. Technical Report UCB/EECS-2008-99, EECS Department, University of California, Berkeley, Aug 2008.

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

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