Large-Scale Distributed Storage for Highly Concurrent MapReduce Applications

Diana Moise, Gabriel Antoniu, Luc Bougé

Large-Scale Distributed Storage for Highly Concurrent MapReduce Applications

Diana Moise
INRIA
IRISA
Rennes, France
diana.moise@inria.fr

Advisor: Gabriel Antoniu
INRIA Rennes
IRISA
Rennes, France
gabriel.antoniu@inria.fr

Advisor: Luc Bougé
ENS Cachan, Brittany
IRISA
Rennes, France
luc.bouge@bretagne.ens-cachan.fr

I. Introduction

A large part of today's most popular applications are data-intensive. Whether they are scientific applications or Internet services, the data volume they process is continuously growing. Two main aspects arise when trying to accomodate the size of the data: processing the computation in a manner that is efficient both in terms of resources and time, and providing storage capable to deal with the requirements of data-intensive applications. Since the input data is large, the computation, which is, in most cases straightforward, is distributed across hundreds or thousands of machines; thus, the application is split into tasks that run in parallel on different machines, tasks that will need to access the data in a highly concurrent manner.

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

Specialized abstractions like Google's MapReduce [1] and Pig-Latin [2] were developed to efficiently manage the workloads of data-intensive applications. These models propose high-level data processing frameworks intended to hide the details of parallelization from the user. Such frameworks rely on storing huge objects and target high performance by optimizing the parallel execution of the computation.

This PhD research focuses on providing large-scale distributed storage for highly concurrent MapReduce applications.

II. MapReduce: the Hadoop implementation

This section briefly presents the concepts MapReduce relies on, and provides an overview of the file systems that can support MapReduce applications.

A. MapReduce

Google's MapReduce is a parallel programming paradigm successfully used by large Internet service providers to perform computations on massive amounts of data. A computation takes a set of input key-value pairs, and produces a set of output key-value pairs. The user of the MapReduce library expresses the computation as two functions: *map*, that processes a key-value pair to generate a set of intermediate key-value pairs, and *reduce*, that merges all intermediate values associated with the same intermediate key. The framework takes care of splitting the input data, scheduling the jobs' component tasks, monitoring them and re-executing the failed ones. Hadoop's [3] implementation of MapReduce follows the Google model. The framework consists of a single master *jobtracker*, and multiple slave *tasktrackers*, one per node. A MapReduce job is split into a set of taks, which are executed by the tasktrackers, as assigned by the jobtracker. The input data is also split into chunks of equal size, that are stored in a distributed file system across the cluster. First, the map tasks are run, each processing a chunk of the input file, by applying the map function defined by the user, and generating a list of key-value pairs. After all the maps have finished, the tasktrackers execute the reduce function on the map outputs.

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

B. Impact on the storage layer

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

One of the optimization techniques the MapReduce framework employs, is to ship the computation to nodes that store the input data; the goal is to minimize data transfers between nodes. For this reason, the storage layer must be able to provide the information about the location of the data.

Another feature of the storage layer, that can bring further enhancements in the MapReduce framework, is the ability to support versioning. This functionality can be of great importance, considering the size of the stored data; versioning enables easy roll-back to previous snapshots of the data.

C. Distributed File Systems for MapReduce Applications

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

To meet its storage needs, Google introduced the Google File System (GFS) [4], a distributed file system that supports large-scale data processing on commodity hardware. In GFS, a file is split into 64 MB chunks that are placed on storage nodes, called *chunkservers*. A centralized *master server* is responsible for keeping the file metadata and the chunk location. When accessing a file, a client first contacts the master to get the chunkservers that store the required chunks; all file I/O operations are then performed through a direct interaction between the client and the chunkservers. GFS is optimized for access patterns involving huge files that are mostly appended to, and then read from. Fault tolerance is ensured through chunk replication and data checksumming. GFS also implements cheap snapshotting and branching.

The Hadoop Distributed File System (HDFS) [5] is part of the Hadoop project, that provides an open-source implementation of Google's MapReduce model. HDFS uses the same design concepts as GFS: servers called *datanodes* are responsible for storing data, while the *namenode* takes care of the file system namespace and the data location. 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.

Amazon released Elastic MapReduce [6], a web service that enables users to easily and cost-effectively process large amounts of data. The service consists in a hosted Hadoop framework running on Amazon's Elastic Compute Cloud (EC2). Amazon's Simple Storage Service (S3) [7] serves as storage layer for Hadoop. The S3 file system stores files as objects, using the key-value abstraction: the filename is used as the key, whereas the file content is the value. Files can be created, listed, and retrieved using either a REST-style HTTP interface or a SOAP interface. S3's design aims to provide scalability, high availability and low latency at comodity costs.

Distributed file systems belonging to the HPC community are also good candidates for supporting data-intensive workloads. Such file systems that were adapted to fit the needs of MapReduce applications, are PVFS (Parallel Virtual File System) [8] and GPFS(General Parallel File System) [9]. The integration of GPFS with the Hadoop framework involves solving two main issues. Firstly, GPFS supports a maximal block size of 16 MB, whereas Hadoop often makes use of data in 64 MB chunks; IBM solved this issue by adding a new concept, *metablocks*, that meant keeping the small block size (512 KB-2 MB), but changing the block allocation policy, so that contiguous blocks are placed on the same node. The second problem concerns the data layout the Hadoop's jobtracker must be aware of. Since GPFS exposes a POSIX interface, this aspect was solved by introducing a new GPFS function.

In order to be used as a storage back-end for Hadoop, some functionalities were added to PVFS, through an additional layer built on top of it: readahead buffering, data layout exposure and replication emulation.

III. Our approach

The purpose of this PhD is to provide efficient storage for the MapReduce framework and the applications it was designed for. The research conducted so far, concerned the storage layer this type of applications require. To meet these requirements we rely on BlobSeer, a system for managing massive data in a large-scale distributed context.

A. BlobSeer

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 persistency through a BerkleyDB layer. Results in [11] show that BlobSeer is able to sustain high throughput under heavy access concurrency, for various access patterns.

B. BlobSeer as a file system for 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 a caching mechanism for read/write operations, as MapReduce 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 MapReduce scheduler data-location aware, we extended BlobSeer with a new primitive, that exposes the pages distribution to providers.

IV. Preliminary results

To evaluate the benefits of using BlobSeer as the storage backend for MapReduce applications we used Hadoop - Yahoo!'s implementation of the MapReduce framework. We substituted the original data storage layer of Hadoop, the *Hadoop Distributed File System - HDFS* with our BlobSeer-based file system - BSFS. To measure the impact of our approach, we performed experiments both with synthetic microbenchmarks and real MapReduce applications. The microbenchmarks are tests that directly access the storage layer, by using the file system interface it provides, while real MapReduce applications access the storage layer through the MapReduce framework. The obtained results are described in the following sections.

A. Grid'5000

The experiments were performed on the Grid'5000 [12] 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.

B. Microbenchmarks

We evaluated the throughput achieved by BSFS and HDFS when the distributed file system is accessed by multiple, concurrent clients. The scenarios we chose are common access patterns in MapReduce applications:

- Clients concurrently reading from different files.
- Clients concurrently reading non-overlapping parts of the same huge file; these two scenarios correspond to the Map Phase of a MapReduce application, when the map tasks process the input files.
- Clients concurrently writing to different files; this test case reproduces a pattern corresponding to a typical Reduce Phase of a MapReduce application, when the reduce tasks all generate and write to different output files.

The microbenchmarks were performed in an environmental setup consisting of 270 nodes, on which we deployed both BSFS and HDFS. The number of concurrent clients accessing the file systems, ranged from 1 to 250; considering that each client processed a 1GB file, the amount of data reached hundreds of GB.

The results showed that BSFS is capable to deliver a higher throughput than HDFS, and to sustain it when the number of clients significantly increases. This is due mainly to the load-balancing strategy BlobSeer applies when distributing the pages to providers. HDFS employs a different policy when allocating chunks to datanodes; the first replica of a chunk is always written locally; for fault tolerance, the second replica is stored on a datanode in the same rack as the first replica, and the third copy is sent to a datanode belonging to a different rack (randomly chosen).

C. *Real MapReduce applications*

We also measured the impact of using BSFS instead of HDFS when being accessed through the Hadoop framework, by real MapReduce applications. We chose to test with two applications: *Random Text Writer*, which generates a huge sequence of random sentences formed from a list of predefined words, and *Distributed Grep*, which scans huge input data to find ocurrences of particular expressions. Random text writer exhibits an access pattern corresponding to concurrent massively parallel writes to different files, whereas distributed grep generates an access pattern of concurrent reads from the same huge file. For these applications, we measured the job completion time when the Hadoop framework uses both BSFS and HDFS as storage back-ends. BSFS was able to finish the job faster than HDFS, results that are consistent with the microbenchmarks we performed.

V. Future work

We consider the MapReduce paradigm and the specific features of the applications that use this programming model. So far, we focused on the storage layer and on improving its throughput under heavy concurrency and when used in specific access patterns.

As future directions, we plan to explore ways of adding functionalities to the storage layer of the MapReduce framework, as well as adjusting the framework itself to take advantage of such functionalities. Appending data concurrently to the same file, is a functionality which can be useful for MapReduce applications, as well as for many data-intensive worloads. For instance, copying a large file can be done by multiple clients, which read different parts of the file and then append the data in parallel. Furthermore, this functionality enables the MapReduce workers to write the reduce output to the same file, instead of creating several output files, as it is currently done in Hadoop.

Another issue we plan to tackle is to analyze how versioning can be integrated in the MapReduce framework. A storage layer that supports versioning enables complex MapReduce workflows to run in parallel, on different snapshots of the same original dataset.

References

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 1099–1110.

[3] "The Apache Hadoop Project," http://www.hadoop.org.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *SIGOPS - Operating Systems Review*, vol. 37, no. 5, pp. 29–43, 2003.

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

[6] "Amazon Elastic Map Reduce," http://aws.amazon.com/elasticmapreduce/.

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

[8] PVFS, "Parallel virtual file system, version 2," http://pvfs2.org/.

[9] 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*. USENIX Association, 2002, pp. 231–244. [Online]. Available: http://portal.acm.org/citation.cfm?id=651312

[10] B. Nicolae, G. Antoniu, and L. Bougé, "BlobSeer: How to enable efficient versioning for large object storage under heavy access concurrency," in *Proc. 2nd Workshop on Data Management in Peer-to-Peer Systems (DAMAP'2009)*, Saint Petersburg, Russia, Mar. 2009, held in conjunction with EDBT'2009. [Online]. Available: http://hal.archives-ouvertes.fr/inria-00382354/en/

[11] ——, "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)*, ser. Lect. Notes in Comp. Science, vol. 5704. Delft, The Netherlands: Springer-Verlag, 2009, pp. 404–416. [Online]. Available: http://hal.archives-ouvertes.fr/inria-00410956/en/

[12] 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*, vol. 20, no. 4, pp. 481–494, November 2006.