Performance Evaluation for BlobSeer and Hadoop using Machine Learning Algorithms

Elena Burceanu, Irina Presa
Automatic Control and Computers Faculty
Politehnica University of Bucharest
Emails: {elena.burceanu, irina.presa}@gmail.com

***Abstract*—Hadoop is a software framework based on the Map-Reduce programming model. It relies on the Hadoop Distributed File System (HDFS) as its primary storage system. In order to improve the data-intensive Map-Reduce applications the original HDFS layer of Hadoop can be substituted with a new, concurrency-optimized data storage layer based on the BlobSeer data management service. The project aims to evaluate and compare the performance of Hadoop and BlobSeer filesystems. This will imply implementing different machine learning algorithms (e.g. Neural Networks) using the map reduce paradigm over each of the above distributed filesystems.**

***Index Terms*—Hadoop, BlobSeer, Map-Reduce, Cluster, Machine Learning Algorithms**

I. Introduction

A. Context and motivation of the project

As the virtual storage expands, the need of processing large data sets across clusters of computer is more overwhelming each day. Hadoop is with one step ahead of others, as an open source solution. Many big names in industry developed proprietary software starting with this solution, but also contributed to the open source branch development. Some only use it, some builds soft on top of it, some implements solutions for its scheduler and some for its filesystem.

Other file systems integrated with Hadoop, except for HDFS are Amazon S3, CloudStore, FTP, read-only HTTP and HTTPS, each optimized in a direction (or more). One is Blobseer, which was developed in an academical environment.

Because it was designed as a large scale distributed filesystem, it is normal for Blobseer to aim to integrate itself with a dedicated framework, such as a Map-Reduce one. Map-Reduce proposes to explicitly exploit parallelism at data level, by forcing the user to design the application according to a predefined model. The model is inspired by the 'map' and 'reduce' primitives. These were commonly used in functional programming, but in the case of Map-Reduce frameworks, they haven't kept their original significance. Several attempts (and achievements) were done to integrate Blobseer with Hadoop, but unfortunately, very poorly documented. Next logical step after integrating it, was to test it's performance, by comparing it with HDFS, on some specific Map-Reduce algorithms.

B. Contributions

We contribute to this project by trying to find out how the performance for machine learning algorithms (so, a class of algorithms) is influenced by the filesystem's implementation and also to compare the results for HDFS and BSFS. With this in mind, we have deployed Mahout over Blobseer. Mahout is a strong tool built on top of Hadoop, a set of machine learning algorithms written considering the Map-Reduce paradigm.

II. Related work

There is a large team of professors, PhDs and masters students that worked at the Blobseer project [1]. They have run a complex deployment in order to see how their filesystem scale in a map-reduce framework. Experiments were done over Yahoo release of Hadoop v0.20. A set of benchmarks that write, read and append data to files through Hadoop's file system API were implemented and throughput was measured as more and more concurrent clients accessed the system. In order to compare with the Blobseer performance, the BSFS replaced HDFS and the same Hadoop map-reduce framework was used for testing. This was possible due to the Blobseer java interface.

The tests were done over 270 machines from the same cluster of Grid'5000, with x86_64 CPUs and having 2-4GB RAM of memory. Intracluster bandwidth was 1 Gbit/s. The control of the clients was directly implemented.

Benchmarks:

- A single process writes a huge distributed file.
- Concurrent readers read different parts of the same huge file.
- Concurrent writers append data to the same huge file.

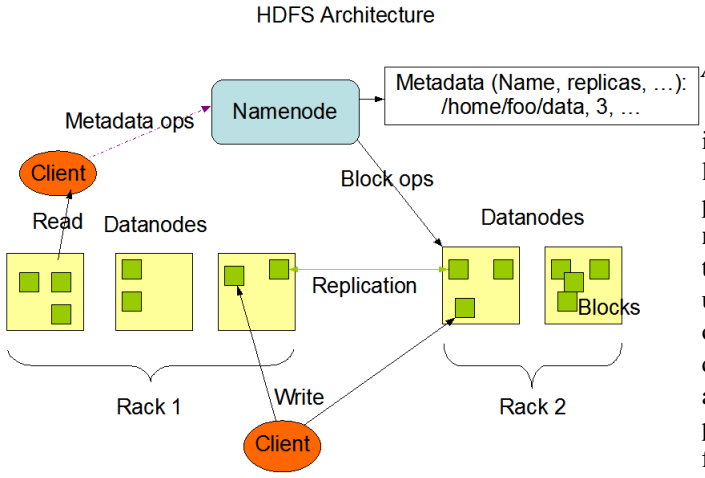
Low level tests enabled the precise control of the clients (as number and action), in order to extract the previously list access patterns. High level tests were done with real-world Map-Reduce applications: **random text writer** (many tasks, each writing large amount of data, with no interaction between tasks - concurrent, massively parallel writes), **distributed grep** (huge data input that needs to be processed in order to obtain output - counting the number of lines that matches an expression - concurrent reads from a shared file) and **sort** (concurrent reads from the same file and concurrent writes to different files).

The measured **metrics** were: the throughput when a single client perform and then the throughput per client, as the number is gradually increased. So, for N concurrent clients, first deploy on HDFS and BSFS and run the benchmark.

III. Architecture

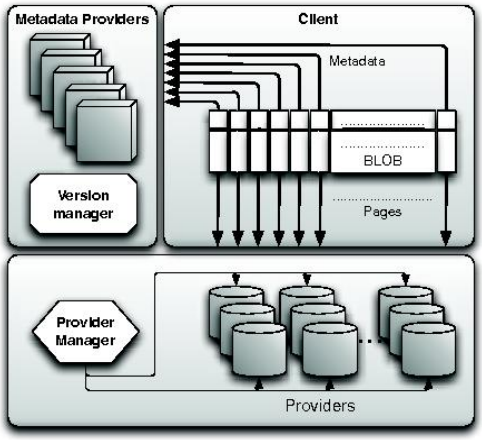
A. Hadoop

HDFS has a master/slave architecture. The master is the NameNode, that manages the filesystem namespace (maps blocks to DataNodes) and regulates clients access to files (opening, closing, and renaming files and directories). The slaves are DataNodes and manage storage (block creation, deletion and replication upon instruction from the NameNode). A file is split into one or more blocks and those are stored in a set of DataNodes. Because there is only one NameNode in a cluster, the architecture is simplified. User data never flows through the NameNode. It is the arbitrator and repository for all HDFS metadata.



B. Blobseer

Data providers (provider/provider) 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.



The **provider manager** (pmanager/pmanager) 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 (provider/sdht) physically store the metadata, allowing clients to find the pages corresponding to the various BLOB versions. Metadata providers may be distributed to allow an efficient concurrent access to metadata.

The **version manager** (vmanager/vmanager) is the key actor of the system. It registers update requests (APPEND and WRITE), assigning BLOB version numbers to each of them. The version manager eventually publishes these updates, guaranteeing total ordering and atomicity

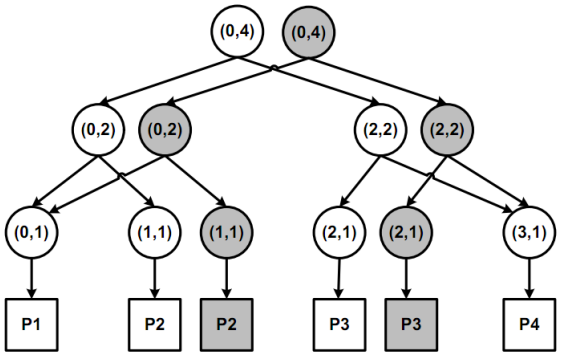
IV. Implementation

A. Used technologies

1) Apache Hadoop: The Apache Hadoop software library is a framework that facilitates the distributed processing of large data sets across clusters of computers using a simple programming model. Hadoop consists of two key services: **reliable data storage** using the Hadoop Distributed File System (HDFS) and **high-performance parallel data processing** using MapReduce. Hadoop, is fault-tolerant, you can add or remove servers in a Hadoop cluster at will, the system detects and compensates for hardware or system problems on any server. It can deliver data and can run large-scale, high-performance processing jobs, in spite of system changes or failures. Some names that uses Hadoop: Facebook, Yahoo, AOL, Adobe, Amazon, Twitter, Google, IBM, Microsoft, Last.fm, LinkedIn.

2) Blobseer: Core principles of Blobseer are: Organizing data as BLOBs (**Binary Large OBjects**) Treat data as a set of huge unstructured data, so that a BLOB can reach 1TB. This gives **scalability** since the data can grow as fast as needed without affecting the performance (the program should manage only the offset of the in the blob). This is effective because Blobseer supports an efficient fine-grain access to the BLOBs, for a large number of concurrent processes.

Modify a file in Blobseer:



Inner nodes labeled with the range they cover represent **metadata** and leaves identify the **pages**. Each node is stored

as a (key, value) pair, where key includes the **blob id**, the **version number** and the **covered range** delimited by an **offset** and a **size**, while value identifies the left and right child. Given an initial blob made up of four pages, a subsequent write generates pages 2 and 3 and also new metadata nodes represented in gray. These new gray nodes are interleaved with the white nodes corresponding to the unmodified first page and forth page respectively. Concurrent writes and appends are also allowed as they can send data to the storage space providers independently of each other. **Synchronization** takes place only at the version manager level where the updates are serialized and assigned version numbers in an incremental fashion. Then, support for metadata enables generating metadata in parallel too with the assumption that all concurrent updates will be generated in the future.

Data striping is the operation of splitting BLOBs into chunks that are distributed on the nodes that provide storage space. At this level the space from storage providers is configurable, so that various objectives could be reached: low energy consumption, high data localization, etc. The size of the chunks is dynamically adjustable, so that based on system loading or other heuristics, the way the computation is partitioned and scheduled can be optimized.

The minus of **versioning** is that it needs extra space, but the space becomes cheaper and cheaper, so versioning has the potential to bring high benefits for a low price: enhanced data access parallelism (clients access different versions of the file), overlapped data acquisition with data processing (one step older data can be processed, while the new ones are acquired, with no synchronization), asynchronous operations (read-write can be concurrent on different versions of the files), differential updates (the updates are kept as difference between versions).

3) Apache Mahout: The Apache Mahout is a scalable machine learning library for map-reduce, built on top of Hadoop. There are four main directions that Mahout supports: **recommendation mining** (takes users' behavior and from that tries to find items users might like), **clustering** (takes text documents and groups them into groups of topically related documents), **classification** (learns from existing categorized documents what documents of a specific category look like and is able to assign labels of the correct category to the documents), **frequent item set mining** (takes a set of item groups (terms in a query session, shopping cart content) and identifies, which individual items usually appear together). Core algorithms for clustering, classification and batch based collaborative filtering are implemented on top of Apache Hadoop using the Map-Reduce paradigm.

4) Map-Reduce: Map-Reduce paradigm can be resumed in two steps:

Map step - The master node takes the input, partitions it up into smaller sub-problems, and distributes them to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes the smaller problem, and passes the answer back to its master node.

Reduce step - The master node then collects the answers to all the sub-problems and combines them in some way to form the output (the answer to the problem it was originally trying to solve). Map-Reduce allows for distributed processing of the map and reduction operations. Provided each mapping operation is independent of the others, all maps can be performed in parallel (though in practice it is limited by the number of independent data sources and/or the number of CPUs near each source).

B. Our Deployment Architecture evolution:

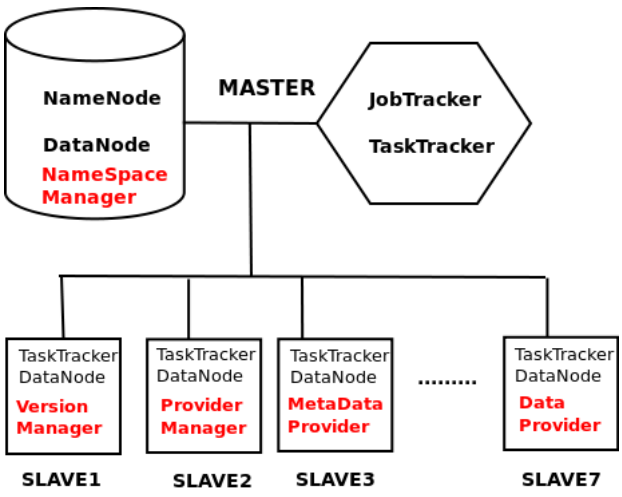
Significant effort was invested in preparing the experimental setup. We had to overcome nontrivial node management and configuration issues to reach this objective. Hadoop and Blobseer are available for both Windows and Linux. We have chosen Ubuntu 10.10 as operating system for the virtual machines. First we have deployed Hadoop, Blobseer and Mahout on localhost, then we have extended to 4 and 8 nodes. The architecture evolved as it follows:

On a single node:

- Deploy Hadoop, with namenode and datanode functionalities.
- Add the Mahout library over Hadoop and prepare its test datasets (20 newspapers, wikipedia, netflix).
- Install Blobseer on localhost (not connected with Hadoop). Components of Blobseer are: 2 managers (version, provider), metadata provider and data node.
- Add a fifth component to Blobseer, the Namespace Manager in order to replace the Hadoop filesystem.
- Integrate Blobseer in Hadoop.

Multiple nodes (4 and 8):

- Multiply the previously configured virtual machine on the CS grid: fep.grid.pub.ro.
- From Hadoop's point of view there are: 1 master-namenode and 3 slaves-datanodes (1 master and 7 slaves).
- From Blobseer's point of view there are: 1 node (keeps the administrative part) with manager (version, provider and namespace) and metadata provider roles and 3 nodes as data provider (one node for each manager (3) + 1 metadata provider + 4 data provider).



C. Configuration Steps:

Cluster Configuration [9] [10]

1) Create a big hda in LustreFS (should be large enough for Hadoop, Blobseer, Mahout and the datasets).
```
qemu-img create -f qcow2 hda.qcow2 200G
```
2) Boot the virtual machine from an Ubuntu iso mounted on cdrom and pick a queue from the cluster (qstat -f).
```
apprun.sh kvm --queue all.q@opteron
-wn08.grid.pub.ro --vmname master
--cpu 8 --memory 8G --hda hda.qcow2
--cdrom ubuntu.iso  --status stat.txt
--mac aa:bb:01:12:34:12 --vncport 7
--master --boot d
```
3) Launch the virtual machine.
```
apprun.sh kvm --queue all.q@opteron
-wn08.grid.pub.ro --vmname master
--cpu 8 --memory 8G --hda hda.qcow2
--mac aa:bb:01:12:34:12 --vncport 7
--master
```
4) Test if your machine is running (qstat).
5) Attach to the virtual machine.
```
vncviewer all.q@opteron...grid.pub.ro
ssh user@ip
```
6) In order to create the slave copies, use the same hda - uses copy on write, so no need to replicate the entire hda, since most of the configurations are identical.
```
apprun.sh kvm --queue all.q@opteron
-wn08.grid.pub.ro --vmname slave
--cpu 8 --memory 8G --hda hda.qcow2
--mac aa:bb:01:12:34:12 --vncport 7
--slave
```

Hadoop Configuration

1) For a single node deployment, follow this tutorial [5].
2) For cluster deployment use this tutorial [4].
3) *Possible problem:* If the datanode doesn't start, remove all the content from /tmp (*.pid and hadoop-*) you can configure the the path of hadoop tmp files from hadoop-dir/conf/core-sites.xml, by adding a property named 'hadoop.tmp.dir' with the value of the new path '/home/user/tmp'.
4) *Possible problem:*If the namenode doesn't start, hadoop namenode -format.

Blobseer Configuration

1) Download the sources from github [3].
2) Install and deploy on localhost from this tutorial [1] (use the configuration file blobseer_dir/test/test.cfg from the sources, not from the tutorial) (*Possible problem:* Pay attention to Boost version, should be between 0.40 and 1.47).
3) *Possible problem:* If you have problemes with the space from blobs (cannot append to blob), try modifying in the provider configuration, 'space = 1024' in the configuration file (test.cfg).
4) Download sources for Hadoop integration (namespace manager) [2].
   Here starts the main part of the configuration:
5) Copy bshadoop in Hadoop's filesystem sources (fs/bsfs).
6) Create directory named blobseer in Hadoop's core and copy the java interface of blobseer here.
7) Build Hadoop using ant.
8) Copy libblobseer.so and libblobseer-java.so in the architecture dependent folder (Linux-arch/) of Hadoop.
9) Override Hadoop's core-site.xml with bshadoop/core-site.xml.
10) Start the namespace-node (the one needed for integrating with Hadoop).
11) Build the Blobseer's java namespace manager. Link it to filesystem client and to the java Blobseer's interface. Then run ant.
    Hint: Run commands detached from the user/terminal with nohup bash_cmd.
12) While running the namespace manager, test it by executing bin/hadoop dfs -mkdir /newdir or bin/hadoop dfs -ls.

Mahout Configuration

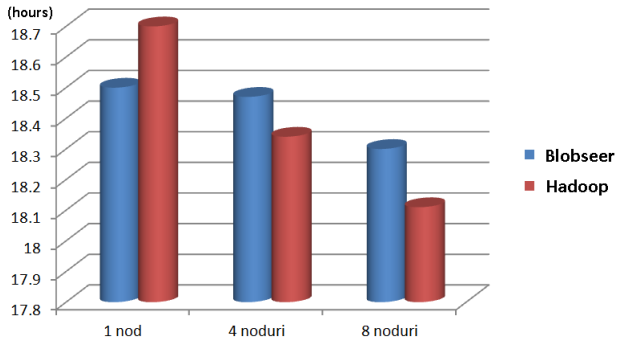
1) To install it, follow this tutorial [6].
2) To start a classification example, read from here [7].

V. Scenarios and Results

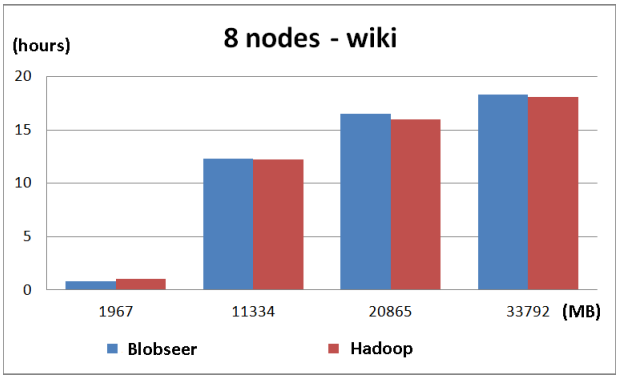
For our first tests we have used different machine learning algorithms from Mahout (e.g. k-means, naive bayes) or classical map-reduce problems, such as Word Count or Grep. But for most of them we only had small data sets, so the results weren't quite relevant.

More accurate results were obtained after applying Naive Bayes classification algorithm over a big dataset (33GB) consisting in a Wikipedia articles database [8]. First, the test splits the input into chunks. Further, the data is grouped by the country of origin of each article. Finally, the classifier is trained to predict what country an unseen article should be categorized into. The algorithm was also a good choice regarding the intensive usage of the filesystem: doing lots of readings of the input data for the map operations and writing back the results in the reduce phases.

The image bellow illustrates a comparison between Hadoop and BlobSeer after running the Naive Bayes algorithm on different cluster configurations: single node and master/slave deployments with 4 and 8 nodes.



The second image shows the results of running the same algorithm as above, on a 8 nodes master/slaves deployment, this time alternating the size of the input data set.



VI. Possible Sources of Errors

While setting up the testing environments, we've encountered various factors that could have affected our results.

1) Isolate file system operations
   In order to obtain a better comparison between Hadoop and BlobSeer filesystems, we need to isolate and closely monitor the filesystem operations. In this case, the cpu intensive jobs wouldn't come with relevant information for our tests. For example, for more accurate results, it would be better to only watch for the read operations of the map phases, and the write operations in the final reduce phase.
2) Small datasets
   We also need to use larger data sets in order to increase the number of file system accesses.
3) Cluster loading and intracluster bandwidth
   A very important factor that can affect the test results would be the cluster's state. Causes like the cluster's load or bandwidth can have a strong influence on the communication between the master node and the slaves. For example, after testing in different times of the day on the same data sets, we obtained substantial differences between the results.
4) Suboptimal Blobseer Configuration
   Another problem we've encountered while deploying the master/slave nodes was finding the optimal configuration for BlobSeer. As opposed to Hadoop, a highly used and documented system, BlobSeer is rather a newer system which lacks proper documentation. For example, adding an extra Metadata Provider node would have increased the speed for finding the requested data, while adding the overhead of a new node. Also, adding another version manager would have improved the concurrent coordination of the filesystem's operations - e.g. writes and appends. Such decisions might have increased the overall performance of the BlobSeer deployment.
5) The relevance of the testing algorithms
   The algorithms chosen for the testing scenarios also pay an important role on the relevance of the results. We've tested using several machine learning algorithms that were provided by Mahout library and also with some simple algorithms from the Hadoop framework. From among them we had to select only the ones that were intensively using the file system, e.g. lots of read/write operations.

VII. Conclusion and Further Work

So far, from our test results, Hadoop seems to have a better performance. But given the facts that the performance differences were quite small, the weak configuration of BlobSeer and all the possible error sources described above, the configurations of the deployed systems might still be improved in order to obtain more relevant results.

For our future tests, we plan to:

- Expand our distributed environments by adding more processing nodes.
- Use larger data sets as testing input.
- Closely monitorize the filesystem operations. One possible method to achieve this would be using the MonALISA Distributed Monitoring Framework.

Acknowledgment

The authors would like to thank the cluster team for their technical support.

References

[1] Blobseer configuration. http://blobseer.gforge.inria.fr/doku.php.
[2] Blobseer integration with hadoop module. https://github.com/acarpena/blobseer/tree/4416a8c141cf03fc80a0c9bc7980ea72dcc58d8a/contrib/bshadoop.
[3] Blobseer sources. https://github.com/acarpena/blobseer/.
[4] Hadoop cluster deployment tutorial. https://ncit-cluster.grid.pub.ro/trac/PP2010/wiki/Multi-Node.
[5] Hadoop one node deployment tutorial. https://ncit-cluster.grid.pub.ro/trac/PP2010/wiki/Hadoop.
[6] Mahout. https://cwiki.apache.org/MAHOUT/buildingmahout.html.
[7] Mahout runclasification example. https://cwiki.apache.org/MAHOUT/twenty-newsgroups.html.
[8] Mahout wikipedia classification example. https://cwiki.apache.org/MAHOUT/wikipedia-bayes-example.html.
[9] Manual for configure virtual machines on cluster. http://cluster.grid.pub.ro/images/clusterguide-v3.1.pdf .
[10] Virtual machines configuration scripts. http://swarm.cs.pub.ro/ irinap/cluster-kvm/start_machines.sh.