Big Data Analytics on Object Stores: A Performance Study

Lukas Rupprecht[*]
Imperial College London
lr12@imperial.ac.uk

Rui Zhang
IBM Research - Almaden
ruiz@us.ibm.com

Dean Hildebrand
IBM Research - Almaden
dhildeb@us.ibm.com

***Abstract*—Object stores provide a highly scalable and cheap storage solution due to their key-value store semantics and commodity-hardware based deployment. This makes them an attractive option for archiving large amounts of data that are produced in science and industry. To analyze that data, advanced analytics such as MapReduce can be used. However, copying the data from the object store into the distributed file system that the analytics system requires is complex and time-intensive. Hence, running analytics directly on object stores greatly improves usability and performance. In this work, we study this combination and identify common problems.**

I. Introduction

As more and more data are produced by science and industry, the need for storage systems that can cope with these volumes in an efficient and cheap way is increasing. Object stores such as Dynamo [1], Haystack [2], or Swift [3] are a prominent solution that meet these requirements. Their simple key-value store semantics, decentralized meta-data management, and weaker consistency guarantees allow for very high scalability. Storage can be easily expanded by adding more disk devices to the system and as disks are usually commodity hardware, adding additional storage is cheap. Availability and reliability is achieved by n-way replication.

Data analytics frameworks such as MapReduce [4] have been widely adopted for analyzing large amounts of data. In general, data are pulled from a distributed file system (DFS) (e.g. HDFS [5]), processed in parallel on different machines, and finally, results are written back to the DFS. These file systems provide functionality such as strong consistency and directories but this limits their scalability as they require centralized meta-data servers and transaction semantics. Hence, they are less suitable for large-scale archival storage. On the other hand, data that is archived in an object store still provides value and hence, can benefit from analytics.

A common solution to providing analytics on archived data, i.e. to make the archive *active*, is to copy data between the analytics cluster and the object store. This is costly and adds additional complexity. A more intuitive solution is to directly run the analytics on the archived data which prevents the extra copying step. In this work we study the implications of such a setup. As there are fundamental differences between a DFS and an object store in how data is managed, running analytics on top of an object store poses some problems.

We identify these problems by running Hadoop MapReduce [6] on top of the OpenStack Swift [3] object store using a recently developed connector [7] and comparing its performance to a standard setup with MapReduce running on top of HDFS. We use a set of micro- and macro-benchmarks to analyze the performance of file system operations and full analytics jobs. Our initial results show that job completion time of MapReduce jobs decreases when executed on Swift. This decrease in performance is due to the consistent hashing based distribution and location of objects in object stores. We furthermore explore whether this mechanism to store and locate data can yield performance advantages in certain scenarios but find that this is not the case. We are not aware of any similar study.

[*]Work done during internship at IBM Research.

II. Testbed and Benchmarks

Our testbed comprises a cluster of three machines that run Hadoop 1.2.1 and Swift 1.13.1 with one master that runs the Hadoop JobTracker and NameNode, and the Swift proxy node and two workers that run the Hadoop TaskTrackers and DataNodes, and the Swift storage nodes. We turned off replication to ensure that mappers for the same data partition are always launched on the same machine for HDFS and Swift.

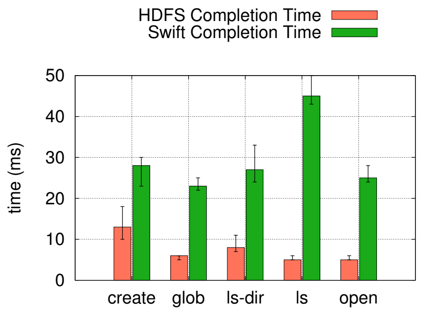
The micro benchmarks compare the execution time of file system operations such as `ls` or `open` on HDFS and Swift, executed via the Hadoop API. The macro benchmark measures the completion time of MapReduce jobs on both storage systems. We run a wordcount job with input data generated using the HiBench data generator [8]. Data is distributed across the workers in the same way for HDFS and Swift.

Hadoop accesses data from the underlying storage system by providing a `FileSystem` interface. To analyze the differences in job completion time in more detail when changing the interface implementation, we instrumented the code to measure the execution times of calls to `FileSystem` methods. This allows us to identify the specific methods that take longer/are faster for the Swift file system implementation.

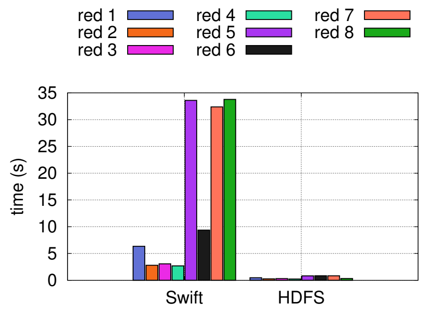
III. HDFS-Swift Comparison

We start our comparison with the results from the micro benchmarks. Figure 1a shows the execution times for the four file system calls `create`, `glob`, `ls` (file and directory) and `open`. Every call was issued from the master node.

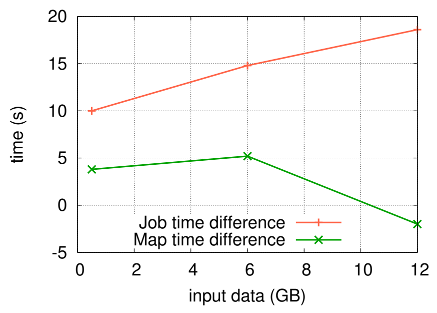
The data shows the median and the 10th and 90th percentiles from 150 runs. The first run is excluded from all bars as for Swift, the first call always involves authenticating at the object store which roughly takes 1s. After removing this outlier,



(a) Micro-benchmark results



(b) Macro-benchmark results



(c) Scaling input data size

Fig. 1: Benchmarking results

we observe that Swift is up to 9× slower than HDFS. This comes from the extra HTTP communication that is necessary to process a Swift request. In Swift, a call to the file system has to be translated into the appropriate HTTP request and sent to Swift's RESTful API while HDFS can use socket-based RPC. We also see that the `ls` operation has the highest overhead. To provide high scalability, Swift stores object meta-data with the object rather than at a central location. Hence, the `ls` call has to fetch the meta-data from the corresponding node while in HDFS, all meta-data is stored in the NameNode's memory.

Swift controls data distribution based on consistent hashing, i.e. a hash function is used to determine which node stores which object by hashing the object name. If a secure hash-function such as md5 or SHA-1 is used, its properties ensure that data is uniformly distributed across all nodes. Additionally, objects can be located completely decentralized and in constant time. Another interesting observation from Figure 1a is that this constant time lookup has no benefit. In our experiment, the accessed files in HDFS were stored under a 3-level path with 10^5 files at every level. However, as path elements are identified via binary search in HDFS, i.e. in logarithmic time, the lookup overhead was not noticeable.

Even though up to 9× is a significant overhead, the impact on MapReduce jobs is still negligible as the absolute numbers are only in the range of tens of milliseconds while jobs can run for several hours. However, interactive workloads might be affected and we plan to extend our study to include these.

Figure 1b shows the time that reducers spend in file system method calls during a job run with 12 GB input data. While for HDFS every reducer spends roughly 0.5s in such calls, the results for Swift vary between 3 to 30s. This overhead comes from the fact that Swift controls data distribution based on consistent hashing. In particular, we found that reducers make a `rename` call to rename the results file. Due to the consistent hashing, the mapping between object name and storage node is fixed so renaming an object always requires a data copy which in many cases is remote. In HDFS, renaming is a pure meta-data operation and can be done without touching the data.

This problem intensifies for larger input data sizes if we assume that larger input also produces larger output as is the case for jobs such as ETL, data decompression, or sorting. Figure 1c shows the difference in job completion time between HDFS and Swift for increasing input data and a constant number of mappers. While the overhead in the map phase stays constant, the overhead for the reduce phase increases linearly. For very large input or low latency jobs such as interactive queries, this overhead can become a major limiting factor.

IV. Related Work

MixApart [9] and MRAP [10] look at data analytics on different storage systems. MixApart deals with analytics on enterprise data and provides a file system that combines the requirements of analytics and enterprise storage to avoid copying. MRAP looks at data in HPC file systems which is analyzed using MapReduce. The authors reduce preprocessing and copying cost by extending the MapReduce framework. Neither MixApart nor MRAP considers object stores.

V. Conclusion and Future Work

We presented a performance study of MapReduce running on top of the Swift object store. While the results are still preliminary, we could identify problems for such a setup. A major problem is that objects cannot be renamed without a data copy. This can have significant impact on job completion time for large input data and latency sensitive jobs.

In the future, we plan to increase the scale of our setup and look at different workloads, the impact of replication, and the implications of object size to data locality.

References

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *SOSP*, 2007.

[2] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel *et al.*, "Finding a needle in haystack: Facebook's photo storage." in *OSDI*, 2010.

[3] "OpenStack Swift," http://docs.openstack.org/developer/swift/.

[4] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004.

[5] "Apache HDFS," http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[6] "Apache Hadoop MapReduce," http://hadoop.apache.org/.

[7] "Hadoop MapReduce - Swift Connector," http://goo.gl/mJIT7Y.

[8] "HiBench," https://github.com/intel-hadoop/HiBench.

[9] M. Mihailescu, G. Soundararajan, and C. Amza, "Mixapart: decoupled analytics for shared storage systems." in *FAST*, 2013.

[10] S. Sehrish, G. Mackey, J. Wang, and J. Bent, "Mrap: a novel mapreduce-based framework to support hpc analytics applications with access patterns," in *HPDC*, 2010.