SQL Based Atomic Renaming for Apache Spark

Big Data Platforms 2022 Final Project

Reichman University

By

Omri Newman - 204866586

Guy Taggar - 206260762

Introduction:

Fault tolerant algorithms are paramount to the widespread success of programming models like Hadoop MapReduce and Apache Spark. Behind the scenes these architectures use file committers to ensure persistence in distributed datasets. Two main approaches are used to manage the associated data with these techniques—Hadoop distributed file system (HDFS) and object storage—and each has a drastically different way of renaming files.

When a Spark job is executed it creates temporary files to keep track of its work before completing said job. All that is left to do therefore is simply rename the file, but this is not as easy as it seems. Within filesystems renaming a file is an atomic O(1) task, but this is not the case for object storage due to its logical structure. An inefficient and expensive solution then is to copy the file, remove it, and upload again under a different name which is an O(data) task.

Cloud based object storage has become the de facto approach for managing backend solutions in today's data driven world. This increasing popularity amongst organizations to depend on object storage necessitates a more precise solution to the renaming problem.

Motivation and Background:

MapReduce is a programming model which changed the Big Data paradigm. MapReduce engines take a set of input key/value pairs and produce an output set of key/value pairs. The map task produces an intermediate set of key/value pairs which are then fed into a reduce phase which aggregates the set by merging appropriate values. Distributing a single task over multiple machines and aggregating their results back to a single target machine fundamentally changed the way companies process data. These large sets of data had straightforward computations, but distributing across hundreds or thousands of machines requires a complex scheduling system to handle parallelization and machine failures. The MapReduce programming model takes care of these details by partitioning the data appropriately, scheduling execution tasks across machines, and even managing the inter-machine communication. This allows users with no experience in distributed systems to enjoy the computational power of several cheap machines instead of relying on a single top tier machine.

Object storage is a cloud based data storage architecture that manages data as objects, where each object contains both metadata and the data itself. This architecture is capable of storing massive amounts of unstructured data, with additional data-management functions and a programmable interface. Object storage is fault tolerant, meaning it is designed to operate during failures. HDFS on the other hand is a traditional data storage architecture that supports hierarchical organization of files, using POSIX operations like open, close, read, write, and search. Some main differences between HDFS and object storage include consistency, atomicity, and latency. HDFS is consistent while object storage is eventually consistent, which means it may take time for object creation, deletion, and updating to become visible to all users. HDFS supports atomic operations while object storage does not. Lastly, object storage has more I/O latency when compared to HDFS since the data is stored in cloud based data centers as opposed to being stored on a private network.

HDFS maintains data locality so accessing data is faster than relying on network connection for object storage, which has its own limitations and dependencies. This translates to slower MapReduce execution. Object storage is generally cheaper and self extendable which is an important consideration when working with big data, thus it is impossible to overload storage capacities that might terminate the MapReduce process. MapReduce relies on many file operations such as creating temporary files. And HDFS performs file operations faster since it is itself a filesystem. Object storage will perform worse in this case because it handles file operations less efficiently when compared to HDFS.

By nature of the MapReduce programming model which splits data across numerous machines, there is an emphasis on how to properly partition data. There are three separate data partitioning methods— horizontal, vertical, and functional partitioning—where each method treats the split of data differently. These various data partitioning schemes can improve database availability, scalability, security, and query processing performance.

Distributed datasets as opposed to classic dataframes have elements of data strung out across multiple machines. Persistent storage is any storage device that retains data after power to the device is shut off. And fault tolerance is a system's ability to continue operating despite machine failure amongst its components. Fault tolerance therefore is an important characteristic for any persistent distributed data set, where access to the data must be prioritized even in the case of machine failure. Apache Spark uses resilient distributed datasets (RDDs), and it ensures persistence by applying lazy evaluations where applicable to achieve fault tolerance. RDDs are parallel data structures that let users explicitly persist intermediate results in memory and also have control over partitioning to optimize data placement. This helps reduce the number of overall queries which optimizes performance. Spark maintains a record of which operation is being called while the lazy evaluations do not load data until necessary, meaning operations are not immediately executed once loaded into the RDD.

FileOutputCommitters are used to manage files by listing directories and renaming their contents into the final destination when tasks and jobs are committed. There are two main versions (V1 and V2) of this committer and they differ in the way they handle merge paths. V1 merges paths once all reducers are done with their jobs, while V2 merges paths concurrently for each reducer. By doing this, V2 renames all individual files instead of directories which can be costly on a normal filesystem. Furthermore, V1 can be slower at appending data to existing datasets and V2 remedies this issue.

The Renaming Problem:

The backbone of any Spark job relies on renaming numerous files as they change from temporary to final names. In local filesystems like HDFS, renaming is an atomic O(1) operation. Unfortunately object storage does not allow this kind of renaming due to its logical structure. Doing so on object stores is an O(data) operation since data is copied, deleted, and reuploaded under a different name.

Stocator and Object Storage:

Various object store connectors for Spark have been introduced in recent years to solve the renaming problem. IBMs Stocator is one connector that aims to fix this prevalent issue.

Stocator recognizes the name/write patterns when Spark creates temporary objects, and instead of renaming temporary outputs, it simply writes to the final directory. Stocator uses HDFS implementations in order to recognize these patterns. Apart from this, Stocator also supports GetFileStatus calls on the original path which returns information on the file at the final destination. Stocator also enables speculation, which means to run the same task on multiple machines in parallel. This, along with preserving the Spark and Hadoop models achieves fault tolerance.

Stocator is one of many committers compatible with object storage. Apart from the standard FileOutputCommitter, there are also the DirectOutputCommitter, Staging Committer, and the Magic Committer. Stocator behaves differently from these committers, but at the end of the day they all achieve the same task: to write the output of mappers into the file system such that they are ready for the reducers.

The DirectOutputCommitter implements zero renaming by returning the destination directory as the task working directory. Working with the same underlying directory as the destination means there is no need to move or rename task outputs on successful commits. This basically guarantees persistence on object storage. However this implies there are no additional commit or abort tasks once a job has been instantiated, so there is no way to handle failures.

The Staging Committer stages all generated outputs to a local filesystem of worker nodes and uploads data once the task is complete. Outputs are written to temporary local directories, and are only uploaded to object storage once committed, essentially delaying or staging the commit process. Under the hood, it uses FileOutputCommitter Version 1 to manage commit and abort files. Relying on a local directory such as HDFS for temporary writes of a given task means the benefits of HDFS apply here. All these tasks are persisted to a consistent filesystem. Added overhead however for uploading each file at the end of every task can compound and quickly become an expensive O(data) operation.

The Magic Committer quickly pushes data to object storage, depending on the object store client to recognise some outputs as special, potentially delaying the write to final destination. This postpones the completion of writes of all files while the final destination is altered. There is no staging, so data is directly uploaded to object storage and it stores a list of pending commits. The create call must recognize whether the file being created is to be delayed or not which is complex and can be costly. Moreover, files containing temporary information could be mistaken for actual data. But the magic committers ability to discern between file types is a strong benefit which outweighs the associated costs of the method.

Worse performance on object storage compounded with the use of FileOutputCommitter will hinder the persistence of RDDs. Object storage tends not to perform as well as HDFS due to higher latency and lower data throughout. HDFS stores and processes data on the same machines, giving it a leg up on performance. Using FileOutputCommitter therefore is less efficient on object storage as multiple jobs must be managed through the workflow. As a result, the output of work may not be immediately visible to a follow-up query, directly affecting consistency and persistence.

Our Approach:

Due to the costly overhead incurred by Spark jobs when initiating tasks with regards to rename operations, we decided to construct a prototype which mimics classic file system POSIX operations. Using Python and SQLite as our main tools, we define a methodology which acts as a catalog and represents files on object storage and supports atomic renaming.

Our approach was to create a ready made platform for integrating with object storage. Users must simply utilize their specific cloud providers API and download/upload their data using standard HTTP methods such as GET and PUT. The main intention of this approach is to satisfy atomic renaming which does not exist on object storage, but other benefits exist as well. Namely, users save on communicating with the cloud by avoiding superfluous calls to object storage. On the other hand, introducing SQL databases increases the chances of dependency issues arising in your workflow. Relying on additional storage through local SQL databases defeats the whole point of object storage, in that it provides cheap storage in a self-extendable way. There's a tradeoff here; relying on a cloud provider for all your storage needs and dealing with expensive rename operations, or paying for additional local storage while adding more to your workflow in order to achieve atomic rename operations.

In general we would advise cloud providers not to provide additional databases for atomic renaming since it would go against their business model. These providers have every incentive for users to accrue more operations on their platforms. It is worth mentioning however that if one cloud provider does go out of their way to provide additional resources for atomic renaming, then users will have more of an incentive to work with that specific provider.

Prototype:

Our prototype uses Python and SQLite to support artificial renaming on top of basic file operations. To do this, we define a Python class—ExtendedObjectStorage—that serves as a catalog for object storage.

As a relational database, SQL relies on primary keys within its hierarchical structure. Our prototype constructs pairs of directory and file names and treats them as a primary key within the database. This ensures that two separate files cannot have identical file names under the same directory, which is a basic property across well established filesystems.

All of the operations were designed to raise dedicated exception errors when prompted with illegal operations, an example of which is attempting to list a directory that does not exist.

Each directory used in the workflow is treated as a separate SQL table. Deleting directories translates to dropping SQL tables, and listing directories translates to SELECT * FROM dir. These equate to methods which exist in object storage while supporting POSIX style operations.

We included an automatic timestamp method which keeps track of any changes to the file and records them in the database. This automatic timestamp is relevant because of how Spark reconciles machine failures during its jobs. In the case of machine failure, a new task is instantiated on a new machine and the first one to complete said task is used in the final directory, hence it is crucial to keep track of time.

So how does this prototype work in the context of Spark? By design our platform works exceptionally well with renaming operations, hence it is ready made for Spark usage which is normally heavily affected by the renaming problem on object storage. Spark renames files to keep track of its tasks. And Spark jobs can keep track of this renaming using our SQL based method.

Next Steps:

Natural next steps include connecting the platform with object storage and in particular with Apache Spark. While connecting the platform to Spark requires modifying Spark code itself, an easy solution exists for object storage integration using the Lithops package (https://github.com/lithops-cloud/lithops), which consists of all APIs for featured cloud providers. Including Lithops as a one-stop-shop within our prototype will make it even easier for users to integrate their object stores.

Conclusion:

If you are looking for a way to save on expensive renaming operations on the cloud and are willing to allocate more database resources for this task, then our SQL based renaming for Apache Spark is right for you. It offers artificial atomic renaming, Apache Spark compatibility, and POSIX style file operations, all while easily connecting to object storage. The code base is located on the following GitHub repository:
https://github.com/GuyPago/Big_Data_Platform/tree/master/Final%20Project

References:

Storing Apache Hadoop Data on the Cloud - HDFS vs. S3 | Integrate.io

Performance comparison between MinIO and HDFS for MapReduce Workloads

Data Partitioning: System Design Concept

Lazy Evaluation in Apache Spark – A Quick guide - DataFlair .

Apache Hadoop Amazon Web Services support – Committing work to S3 with the S3A Committers

Improve Apache Spark performance with the S3 magic committer - The Spot by NetApp Blog

Stocator – IBM Developer

Object Storage versus Block Storage: Understanding the Technology Differences | Druva

[1709.01812] Stocator: A High Performance Object Store Connector for Spark

What Is Persistent Storage? How Does It Work? | NetApp

Apache Hadoop

What is the difference between mapreduce.fileoutputcommitter.algorithm.version=1 and 2 | Open Knowledge Base

Spark 2.0.0 cluster takes a long time to append data | Databricks on AWS

https://www.educba.com/hadoop-vs-spark/