Big Data Platforms

Apache Spark and Object Storage

Roy Ludan & Yarden Fogel
Roee.ludan@post.idc.ac.il
yarden.fogel@post.idc.ac.il

Submission Date: 16 February, 2022

Table of Contents

Abstract

Apache Spark is displaying inefficiencies when using the FileOutputCommitter algorithm. This is done when it persists RDDs in object storage. As fault tolerance algorithms were initially designed to work with the HDFS file systems, a need for a more efficient interaction with object storage is needed as the world migrates from file systems to cloud-based object storage systems. One of the main inefficiencies is the inability to execute atomic rename operations in object stores.

"Most Apache Spark users overlook the choice of an S3 committer (a protocol used by Spark when writing output results to S3), because it is quite complex and documentation about it is scarce. This choice has a major impact on performance whenever you write data to S3. On average, a large portion of Spark jobs are spent writing to S3, so choosing the right S3 committer is important for AWS Spark users".

Motivation and background

Definitions

MapReduce is a programming model for handling big data, enabling scalability with an algorithm that runs on a distributed cluster of machines in parallel, splitting the data and then re-aggregating to a consolidated final output for the user. MapReduce is relatively simple to use and enables faster, flexible, and more scalable performance for processing big data.

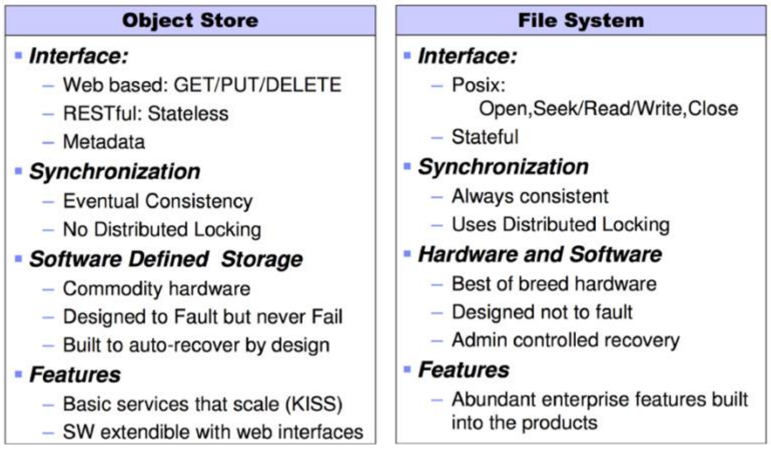
Object storage: The defining characteristic of object storage is that it treats data as objects, instead of file systems which treat data as files in a directory/folder/file hierarchy. It typically (but not exclusively) is synonymous with cloud object storage, largely because object storage is the go-to storage mechanism for the cloud, and it enables massive amounts of data to be stored in a distributed scaled cloud infrastructure, instead of on local or distributed file systems on actual machines/computers. An object stored in object storage contains the data, some metadata, and an identifier.

HDFS is a distributed file system that handles large data sets running on commodity hardware. It is used to scale a single Apache Hadoop cluster to hundreds (and even thousands) of nodes. HDFS is one of the major components of Apache Hadoop, the others being MapReduce and YARN.

Object storage vs. HDFS

Object storage and HDFS have many notable differences, none more relevant to this project than their differences in consistency and atomicity. In terms of their different semantics, on file systems, writing/creating a file or object is not an atomic operation (O(1)), but renaming is atomic (which can be a critical element of functions like MapReduce), and in object storage the opposite is true, where creating an object is atomic but renaming is not as it requires some version of copying the data and then deleting, which increases run-time, cost, and vulnerability.

In terms of consistency, object stores are often eventual consistent, so it can take time for changes to be visible to all users and interfaces, whereas file systems always consistent.
As a relatively recent development (2021), one possible exception to this is that S3 is now strongly consistent, obviating the need for S3Guard or other backup storage paradigms to ensure stronger consistency overall.
There are other important differences between object storage and HDFS, like hardware, fault tolerance (including failures & recovery), features and interfaces/APIs[1].



Object Store

- ***Interface:***
  - Web based: GET/PUT/DELETE
  - RESTful: Stateless
  - Metadata
- ***Synchronization***
  - Eventual Consistency
  - No Distributed Locking
- ***Software Defined Storage***
  - Commodity hardware
  - Designed to Fault but never Fail
  - Built to auto-recover by design
- ***Features***
  - Basic services that scale (KISS)
  - SW extendible with web interfaces

File System

- ***Interface:***
  - Posix: Open,Seek/Read/Write,Close
  - Stateful
- ***Synchronization***
  - Always consistent
  - Uses Distributed Locking
- ***Hardware and Software***
  - Best of breed hardware
  - Designed not to fault
  - Admin controlled recovery
- ***Features***
  - Abundant enterprise features built into the products

Figure 1: Big Data Platform Lecture slides (Gil Vernik)

[1] Of course there are also critical differences between object storage and HDFS in terms of scalability, cost/price, performance, security, elasticity, etc, but we chose to focus here mostly on characteristics that affect the performance of object storage vs. HDFS when Spark interacts with them for a MapReduce job to keep it pertinent to the subject matter we're studying in this project

MapReduce in HDFS vs. in object storage

Object storage is comparably faster, cheaper, and more scalable when compared to HDFS but the major difference when it comes to MapReduce that compromises the performance, run-time, and fault tolerance of object storage is the renaming operation. Since rename is not an option on object storage, and instead the alternative is to copy and delete, this operation causes a bottleneck and inefficiency that will be explored in this project.

As an example, if you have a folder with a million files, in a traditional file system, when you would rename the folder – it'll instantly has a new name, and it will still contain the million files inside, all with a run-time complexity of O(1).
When copy and delete are run in object storage - those million files need to be copied, and then deleted, so the "renaming" process has a runtime complexity of O(2 million).
The result doesn't conclude only of a longer run-time but also creates 2 million opportunities for a fault, which then cascades into a whole new set of issues and disadvantages.

However, with the improvements in S3A committers, including Stocator, the two Netflix staging committers, the Magic committer, and the zero-rename committer, among others, renaming can either be altogether avoided or significantly improved, and in other cases using a database like mySQL as a "middle-man" for naming only, allowing for atomic renaming.

These innovations have really closed the performance and fault tolerance gap between HDFS and object storage for MapReduce.

Data partitions in the distributed data systems

Data partitioning is the process of splitting data into multiple chunks, and assigning those chunks to different nodes across the cluster, so that workloads of read and write operations are properly distributed.

This partitioning enables scalability because we can add more nodes to the cluster as data requirements grow and the new nodes hold partitions of new incremental data. There are also various strategies for partitioning, which are needed to ensure that the way data is split and allocated across the cluster remains well-balanced as the amount of data changes and grows.

Various examples of partitioning methods include: key range and key hash partitioning and consistent hashing, each with their own limitations and advantages[2].

MapReduce partitioning is the process of translating the map output (key,value) pairs to a set of (key,value) pairs to supply the reducers as their input. The partition step also determines what (key,value) format to hand off to the reducers, and which reducers are assigned each (key,value) pair, based on the hash value of the keys. There is one partition (sometimes referred to as an "executor") for each reducer.

[2] though in our view consistent hashing seems to be the most elegant and evolved strategy

Why fault tolerance is important when persisting distributed datasets

Fault tolerance is the ability to overcome faults and/or failures of individual nodes.
When a fault tolerance system is functioning properly and robustly it maintains the availability and reliability of big data processing systems. Proper fault tolerance is therefore a major part in big data systems.

Redundancy is one of the key approaches to addressing fault tolerance, on both the data side via copies and backups, and on the process execution side with speculative execution. Saadoon et al. summarizes the state of fault tolerance in big data systems in his paper along with some of the associated challenges encountered and solutions available.

To cite one statistic from this paper: "*a cluster of 1000 super-machines that on average fail only once every THIRTY years is expected to experience one failure EVERY day*". From this we learn that those faults and failures are unavoidable with big data processed across large, distributed clusters, and if failures are inevitable then the best thing is to optimize how to withstand them.

Speed, efficiency, cost-optimization and security are all important motivations for big data end-users, but "behind the curtains" what happens with a storage provider's fault-tolerance mechanisms often contributes meaningfully to the ultimate success of that platform.

That is why fault-tolerance is so critically important, and why the decision about the inevitable trade-offs users have to make when performing an analysis, or developers have to make when locking in product and infrastructure architecture and design, carries so much weight and receives so much attention and mindshare.

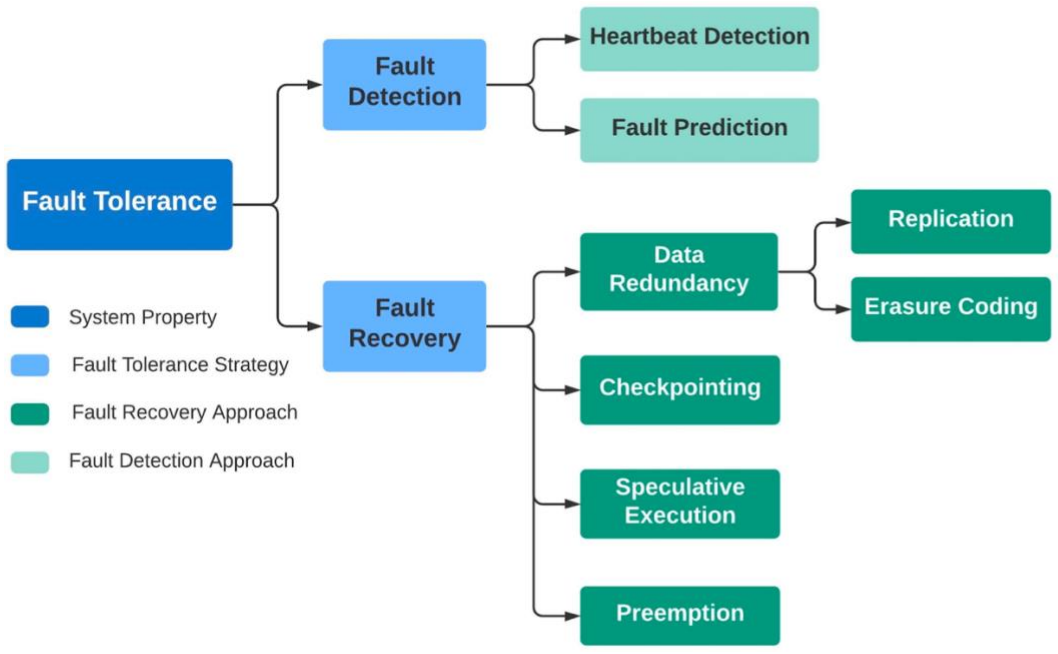


Figure 2: Various Approaches to Fault Tolerance

How Apache Spark persists its RDDs

RDD persistence is the process of saving the result for evaluation and analysis, and Spark has an optimization technique to do this. The result is saved as an intermediate file or result so that it can be used for further analysis and processing, and doing this also saves on overhead and run-time complexity.

Apache Spark achieves this and persists its RDDs using either the cache() or persist() methods, with the difference between the two being the storage level used :

- Cache: the default is memory_only
- Persist: has more variety in storage level employed

The main benefit of persisting RDDs is efficiency gains in terms of cost and time. Spark in particular uses an optimization technique known as "lazy evaluation" where execution of operations is not instant, but rather only occur when an action is "triggered" and the driver makes a request. This process reduces the number of queries/calls and improves run-time and cost, and is said by some to be "the true genius of Spark".

FileOutputCommitter version 1 vs. version 2

Version 1

Application Master (AM) performs mergePaths() after all reducer jobs are completed. If there are many reducers, AM will wait until all reducers are finished and then use a single thread to merge all output files. So it's not built for scaling large jobs and larger datasets.

Version 1 was designed to address failures and AM restarts, and does not log in-progress renaming. Major drawback is cost of time to commit because it needs to recursively list all files in all directories and then rename them, and this is all performed sequentially. Also major vulnerability is at scale with large datasets where it cannot recover from failure during job abort and needs to re-execute the entire query.

The longer the task, the higher the risk of failure and it also takes more time and is more complex to re-run the query when recovery is needed.

Version 2

The v2 algorithm directly commits output of tasks into the destination directory and handles mergePaths() differently than v1
Each reducer will perform mergePaths() concurrently to move output files into final output directories, which can save the AM lots of run-time in the job commit.

However, by listing all children of a source directory and recursively calling mergePaths() on them, it does rename on all the individual files, instead of on directories, which adds an O(data) cost to the mergePaths() run-time. Generally speaking, for both v1 and v2, most operations can run in O(1) on a genuine filesystem, but approximate O(data) or O(files) on object stores.

V2 was designed to support parallel writers to the output directory and reduces the number of list operations, copies and deletes since those are all handled during the task commit. It also saves time by reducing pauses at the end of work since job commit is reduced to just cleanup and creating the _SUCCESS file.

The v2 algorithm slows down more when there are directories to commit since it lists those files recursively, though much of that slowdown can be compensated in the overall job due to the parallel workflow across the cluster. Lastly, the v2 algorithm is also weaker overall in fault tolerance vs. v1 since task commit is not atomic and will be harder to recover from task failure during commit.

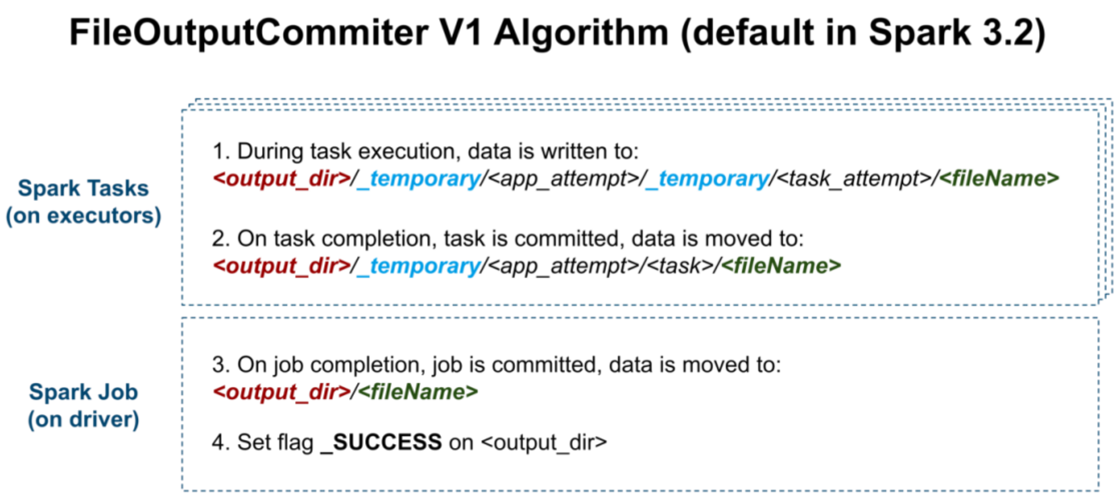

FileOutputCommiter V1 Algorithm (default in Spark 3.2)
Spark Tasks (on executors)
1. During task execution, data is written to: <output_dir>/_temporary/<app_attempt>/_temporary/<task_attempt>/<fileName>
2. On task completion, task is committed, data is moved to: <output_dir>/_temporary/<app_attempt>/<task>/<fileName>
Spark Job (on driver)
3. On job completion, job is committed, data is moved to: <output_dir>/<fileName>
4. Set flag _SUCCESS on <output_dir>

Figure 3: How default FileOutputCommitter Works

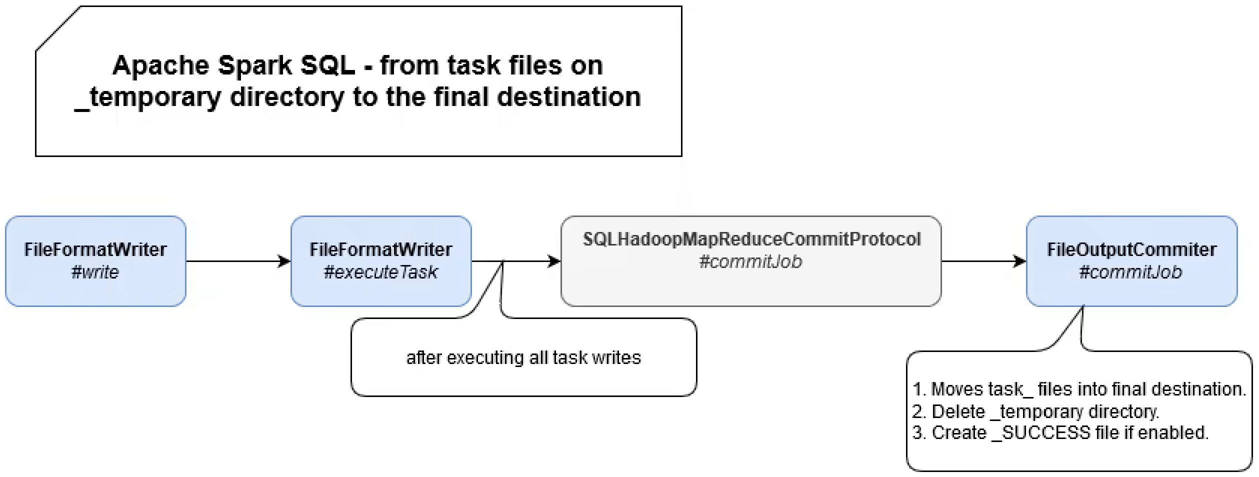

Apache Spark SQL - from task files on _temporary directory to the final destination
FileFormatWriter #write
FileFormatWriter #executeTask
SQLHadoopMapReduceCommitProtocol #commitJob
FileOutputCommiter #commitJob
after executing all task writes
1. Moves task_ files into final destination.
2. Delete _temporary directory.
3. Create _SUCCESS file if enabled.

Figure 4: Spark Job Confirmation in FileOutputCommitter

Stocator and object storage

Why Apache Spark is not efficient when persisting its RDDs in object storage compared to HDFS?

We will list a slew of inefficiencies and shortcomings of Apache Spark when it uses the default FileOutputCommitter to persist RDDs in the object store, though the most prevalent one is renaming, which goes from being an atomic runtime-complexity of O(1) operation in HDFS to being non-atomic and ~O(data) when interacting with object storage using the default committer

Fault tolerance was designed with file systems and in the Hadoop context more specifically, HDFS, in mind, and was not designed to take into account the different semantics of object storage

Operations, most notably renaming, go from being atomic in HDFS to being non-atomic in object storage. The results are slower run-time, more complexity, and compromised fault tolerance. These issues all get worse the bigger the data, so scaling is a major issue, which the object store is meant to enable, not disable.
Bigger data means longer run-time, and longer run-time means higher risk of failures, and higher complexity and time to do recovery. For example, if recovery can only be achieved by a complete job restart, that's obviously a longer more complex process the larger the dataset

How Stocator works?

Stocator's algorithm achieves many of its performance improvements by avoiding rename, and secondarily by employing transfer encoding to split data into chunks. Stocator avoids renaming by writing output objects directly to their final name/directory. Stocator flags the pattern of Spark requesting to create temporary objects as a sign of task success and then takes those objects and writes them directly to the final output.

By avoiding renaming, Stocator also obviates the need to run list operations during commits, which is a key source of bottleneck and potential errors in an eventually consistent object store. Stocator also utilizes chunked transfer encoding to split the data into chunks and addresses what would otherwise be a need to know the total length of the object and store the entire object data before starting a PUT operation.

How Stocator achieves fault tolerance when persisting distributed datasets?

Stocator achieves fault tolerance by having a naming scheme that includes an attempt number, which enables for individual attempts to be differentiated. This is needed because a task may be run multiple times for several reasons - including failure, bottlenecks / slowdowns, and/or speculation.

Stocator's naming scheme allows for maintaining the ability to perform speculative execution, which is critical to achieve fault tolerance, and would otherwise not be possible when renaming is avoided. Speculative execution is particularly important when dealing with "stragglers"[3].

Stocator vs. other committers, pros & cons

Stocator vs. default S3a committer

In the research paper "Stocator: A High-Performance Object Store Connector for Spark" (Gil Vernik et al.) a comparison has been made between Stocator committer and the default S3a committer. The paper shares the number of REST calls made by each committer during an experiment. The experiment runs a Spark program that creates an output consisting of a single object.

In that experiment Stocator made a mere amount of 8 REST calls as compared to 117 that the S3a committer has made, a whopping 174% decrease. "*Compared to Hadoop-swift and Stocator S3a performs many more HEAD calls for the objects and GET for the containers*".

Stocator has outperformed the S3a committer by almost every parameter (except for a runtime experiment for a read-only scenario) tested[4] sometimes by a factor of 30.

Stocator vs. Hadoop-Swift committer

In the experiment mentioned above the Hadoop-Swift made a total of 48 REST calls with a difference of 142% from Stocator's 8 REST calls.

Stocator has outperformed the Hadoop-Swift committer by almost every parameter (except for the industry standard "TPC-DS" benchmark test).

[3] a form of edge case that can cause faults, as the system is only as strong as its weakest link

[4] Mainly REST calls and memory allocation

Our approach

Our prototype demonstrates how an object-storage client can enable an atomic rename to objects given the assumption that all CRUD operations to an object must pass through the client.
The client uses a MySQL database as a catalog for all objects stored in the service and stores (in a single table) only the object's key and alt-key.
The alt-key's responsibility is to refer 'get_object' calls to the newly created object thus providing an atomic rename operation.

Let's review some of the methods implemented in the prototype:

1. Create Object:
   a. Stores the file system's key in the DB
   b. Stores the file system's object in the Object storage

2. Rename Object:
   a. Searches for the object's key in the DB
   b. Adds an alt-key in the object's record in the DB
   c. Copies the object to the new key in Object storage
   d. Removes the original Object from storage
   e. Removes the original record from the DB

3. Create directory:
   a. Creates a new object in the storage with a trailing '/'
   b. Stores a new key with trailing '/' in the DB

4. Rename directory:
   a. Searches for the objects residing 'inside' the directory in the DB
   b. Adds an alt-key in the object's record
   c. Adds a record for each object in the DB
   d. Copies the objects to the destination directory in Object storage
   e. Removes the source keys in the DB
   f. Removes the source keys in Object storage

Pros and cons of using our prototype

Pros:

1. A simple, atomic, solution to object and directory rename in object storage service
2. Ability to run aggregate operations on object's meta-data (e.g., count the number of objects in a bucket or in a directory) without reaching the object storage service directly
3. Very small memory footprint as the client does not store any data
4. Can support up to 2^{48} different object keys (MySQL max records limit)
5. Supports any Cloud Object Storage service by implementing the CloudObjectStorageInterface interface

Cons:

1. get_object operation becomes slower as now instead of just downloading the file you must search for it in the DB first
2. As the state of the Object storage is managed by the MySQL DB, we are limited by the maximum number of connections that the DB can withstand simultaneously (250 connections)
3. Single point of failure: as all operations must pass through the client - if it fails it will leave the Object storage and the DB in an unknown state
4. If the number of objects created will pass the 2^{48} limit – the client will fail

With added capabilities such as: backup, sync operation between the DB and object storage, multi-table use and transaction support we'd highly recommend a cloud provider to internally use a mechanism such as our prototype for the **sole purpose** of atomic renaming.

Prototype

Our prototype is available at the following URL:
https://github.com/trickstyler/ApacheSparkAndObjectStoragePrototype

After cloning, please install all dependencies by running the following command:

```
pip install -r requirements.txt
```

use a MySQL DB version 8.0.28 or later with credentials of:

```
username: root
password: administrator
```

Next steps

To improve the functionality of our prototype and bring it to general availability
We'd like to conduct the following operations:

1. Add support for transactional operations
2. Add support for concurrent operations by locking rows/columns in the DB
3. Add support for REST calls on the existing operations
4. Add meta-data to the table schema
5. Implement a committer that will use our prototype with Apache spark
6. Test and benchmark our committer

Conclusion

In this project we have researched the relationship of Apache Spark with Object storage services. We have covered the difference between Hadoop file-system, object storage and traditional file-systems. We presented known problems and broad solutions to deal with these problems and we've built a prototype to simulate an object storage client with a capability of an atomic rename. We'd highly recommend cloud storage providers to use our prototype for the sole purpose of renaming objects and directories, but not for other CRUD operations.

Bibliography

1. https://hadoop.apache.org/docs/r3.1.1/hadoop-aws/tools/hadoopaws/Committer_architecture.html
2. [1709.01812] Stocator: A High Performance Object Store Connector for Spark
3. Storing data w Spark on the cloud, HDFS vs. S3 - Storing Apache Hadoop Data on the Cloud - HDFS vs. S3 | Integrate.io
4. Hadoop FileOutputCommitter - hadoop/FileOutputCommitter.java at trunk · apache/hadoop · GitHub
5. Improvement in S3 magic committer with Spark 3.2 release in Oct-2021 blogpost, January 2022 - Improve Apache Spark performance with the S3 magic committer - The Spot by NetApp Blog
6. Improve Apache Spark performance with the S3 magic committer - The Spot by NetApp Blog
7. Apache Spark Success Anatomy - Apache Spark's _SUCESS anatomy on waitingforcode.com
8. Zero-Rename Committer, Steve Loughran & Ryan Blue, May-2021 - Releases · steveloughran/zero-rename-committer · GitHub
9. Apache Hadoop Amazon Web Services support – Committing work to S3 with the S3A Committers
10. Spark Integration with Cloud Providers - Integration with Cloud Infrastructures - Spark 3.0.0-preview Documentation
11. https://www.one-tab.com/page/I24iuQliSfSXs3o0cP7_DQ
12. https://www.one-tab.com/page/pkgf01GdRl2FnzWZxU3ifA
13. Hadoop - Object Stores vs. File Systems - https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/filesystem/introduction.html#Object_Stores_vs._Filesystems
14. Apache Spark 3.2 Release: Main Features and What's New for Spark-on-Kubernetes - Data Mechanics Blog
15. Connecting to SQL Databases for Data Scientists, Analysts, and Engineers | by Matt Gray | Towards Data Science
16. Improving Apache Spark with S3 - Ryan Blue
17. MapReduce - Wikipedia
18. MapReduce 101: What It Is & How to Get Started | Talend
19. What is Apache MapReduce? | IBM
20. Hadoop - MapReduce
21. Object storage - Wikipedia
22. What is Object Storage and Why Should You Care? - Cloudian
23. What Is Object Storage? - Object vs. File vs. Block | NetApp
24. Apache Hadoop 3.3.1 – Introduction
25. Hadoop S3 Comparison: 7 Critical Differences - Learn | Hevo
26. Top 5 Reasons for Choosing S3 over HDFS - The Databricks Blog
27. S3 vs HDFS - Comparing Technologies in the Big Data Ecosystem | USEReady
28. HDFS vs Cloud-based Object storage(S3) - Blog | luminousmen
29. Data Lakes: From HDFS To S3 (& NFS) In 15 Years
30. Storing Apache Hadoop Data on the Cloud - HDFS vs. S3 | Integrate.io
31. Performance comparison between MinIO and HDFS for MapReduce Workloads
32. Partitioning: The Magic Recipe For Distributed Systems | by Pranay Kumar Chaudhary.
33. Engineering dependability and fault tolerance in a distributed system | Ably Blog: Data in Motion
34. Fault tolerance in big data storage and processing systems: A review on challenges and solutions - ScienceDirect
35. CAP Theorem Explained
36. RDD Persistence and Caching Mechanism in Apache Spark - DataFlair.
37. What is lazy evaluation in Spark? - DataFlair
38. What is the difference between mapreduce.fileoutputcommitter.algorithm.version=1 and 2 | Open Knowledge Base
39. [#MAPREDUCE-7341] Add a task-manifest output committer for Azure and GCS - ASF JIRA
40. Object Storage: Everything You Need to Know - LakeFS
41. Introducing the S3A Committers - Hortonworks Data Platform
42. [#HADOOP-17833] Improve Magic Committer Performance - ASF JIRA
43. https://www.ibm.com/topics/hdfs#:~:text=HDFS%20is%20a%20distributed%20file,others%20being%20MapReduce%20and%20YARN
44. https://hadoop.apache.org/docs/r3.0.3/hadoop-aws/tools/hadoop-aws/s3guard.html