Stocator: Providing High Performance and Fault Tolerance for Apache Spark over Object Storage

Gil Vernik*, Michael Factor*, Elliot K. Kolodner*, Pietro Michiardi†, Effi Ofer* and Francesco Pace†
*IBM Research, Haifa, Israel
†Data Science Department, Eurecom, Biot Sophia-Antipolis, France
Email: *{gilv,factor,kolodner,effio}@il.ibm.com, †{michiardi,pace}@eurecom.fr

***Abstract*—Until now object storage has not been a first-class citizen of the Apache Hadoop ecosystem including Apache Spark. Hadoop connectors to object storage have been based on file semantics, an impedance mismatch, which leads to low performance and the need for an additional consistent storage system to achieve fault tolerance. In particular, Hadoop depends on its underlying storage system and its associated connector for fault tolerance and allowing speculative execution. However, these characteristics are obtained through file operations that are not native for object storage, and are both costly and not atomic. As a result these connectors are not efficient and more importantly they cannot help with fault tolerance for object storage.**

We introduce Stocator, whose novel algorithm achieves both high performance and fault tolerance by taking advantage of object storage semantics. This greatly decreases the number of operations on object storage as well as enabling a much simpler approach to dealing with the eventually consistent semantics typical of object storage. We have implemented Stocator and shared it in open source. Performance testing with Apache Spark shows that it can be 18 times faster for write intensive workloads and can perform 30 times fewer operations on object storage than the legacy Hadoop connectors, reducing costs both for the client and the object storage service provider.

I. Introduction

Data is the natural resource of the 21st century. It is being produced at dizzying rates, e.g., for genomics, for Media and Entertainment, and for Internet of Things. This data increasingly resides in cloud object stores, such as Amazon S3 [1], Azure Blob storage [2], and IBM Cloud Object Storage [3], that are highly scalable distributed cloud storage systems offering high capacity and cost effective storage. But it is not enough just to store data; we also need to derive value from it, for example, through analytics engines such as Apache Hadoop [4] and Apache Spark [5]. However, these highly distributed analytics engines were originally designed to work on data stored in HDFS (Hadoop Distributed File System) where the storage and processing are co-located in the same server cluster. Moving data from object storage to HDFS in order to process it and then moving the results back to object storage for long term storage is inefficient. In this paper we present Stocator [6], a high performance storage connector, that enables Hadoop-based analytics engines to work directly on data stored in object storage systems. Here we focus on Spark however, our work can be extended to work with the other parts of the Hadoop ecosystem.

Until now Hadoop connectors to object storage, e.g., S3a [7] and the Hadoop Swift Connector [8], have been based on file semantics, a natural assumption given that their model of operation is based on the way that Hadoop interacts with its original storage system, HDFS [9]. However, treating object storage like a file system constitutes an impedance mismatch, which can lead to poor performance and incorrect execution. In particular, operations that are atomic for files may not be atomic for objects and operations that are inexpensive for files may not be inexpensive for objects, and vice versa. For example, to rename a directory in a file system requires a single atomic operation, whereas in object storage it requires copy and delete operations for each of the objects in the tree under the “virtual directory”[1].

We are not the first to recognize the poor performance of the object storage connectors. Others have tried to improve performance, by sacrificing speculative execution, and then writing objects directly to their final names, e.g., the DirectOutputCommitter [10] for Amazon S3, or by renaming Hadoop output objects to their final names when tasks complete (task commit) instead of waiting until the entire job completes (job commit) [11]. However, due to the impedance mismatch these attempts led to subtle failures.

Current connectors can also lead to failures and incorrect execution because the list operation on object storage containers/buckets is eventually consistent. EMRFS [12] from Amazon and S3mper [13] from Netflix overcome eventual consistency by storing file metadata in DynamoDB [14], an additional strongly consistent storage system separate from the object store. A similar feature called S3Guard [15] is being developed by the Hadoop open source community for the S3a connector. Solutions like these, which require multiple storage systems, are complex and can introduce issues of consistency between the stores. They also add cost since users must pay for the additional strongly consistent storage.

In this paper, we introduce Stocator, whose novel algorithms achieve both high performance and fault tolerance by taking advantage of object storage semantics. This greatly decreases the number of operations on object storage as well as enabling a much simpler approach to dealing with the eventually consistent semantics typical of object storage. We have implemented our connector for both the OpenStack Swift

This research received funding from the EU Horizon 2020 Research and Innovation Programme (grant agreement 644182-IOStack)

[1] Object stores emulate directories through hierarchical naming.

API [16] and the Amazon S3 API, and have shared it in open source [17]. We have compared its performance with the S3a and Hadoop Swift connectors over a range of workloads and found that it executes far less operations on the object store, in some cases as little as one thirtieth of the operations. Since the price for an object storage service typically includes charges based on the number of operations executed, this reduction in the number of operations lowers costs in addition to reducing the load on client software. It also reduces costs and load for the object storage provider since it can serve more clients with the same amount of processing power. Stocator also substantially increases performance for Spark workloads running over object storage, especially for write intensive workloads, where it is as much as 18 times faster.

In summary our contributions include:

- The design of a novel storage connector for Hadoop and Spark that leverages object storage semantics to provide high performance and correct execution in the face of faults and speculation.
- A solution that works correctly despite the eventually consistent semantics of object storage, yet without requiring additional strongly consistent storage.
- An implementation that has been contributed to open source.

Stocator is in production in the IBM Cloud and has enabled the SETI project to perform computationally intensive Spark workloads on multi-terabyte binary signal files [18].

The remainder of this paper is structured as follows. In Section II we present background on object storage and Apache Spark, we discuss related work and we motivate our work. In Section III we describe how Stocator works. In Section IV we present the methodology for our performance evaluation, including our experimental set up and a description of our workloads. In Section V we present a detailed evaluation of Stocator, comparing its performance with existing Hadoop object storage connectors, from the point of view of run time, number of operations and resource utilization. Finally in Section VI we conclude.

II. Background

We provide background material necessary for understanding the remainder of the paper. First, we describe object storage and then the background on Spark [5] and its implementation that have implications on the way that it uses object storage. Then we discuss related work and finally, we motivate the need for Stocator.

A. Cloud Object Storage

An object encapsulates data and metadata describing the object and its data. An entire object is created at once and cannot be updated in place, although the entire value of an object can be replaced. This simple object semantics enables the implementation of highly scalable, distributed and durable storage that can provide very large capacities at low cost. Object storage is typically accessed through RESTful HTTP, which is a good fit for cloud applications. It is ideal for storing unstructured data, e.g., video, images, backups and documents such as web pages and blogs. Examples of object storage systems include Amazon S3 [1], Azure Blob storage [2], OpenStack Swift [19] and IBM Cloud Object Storage [3].

Object storage has a shallow hierarchy. A storage account may contain one or more buckets or containers (hereafter we use the term container), where each container may contain many objects. Typically there is no hierarchy in a container, e.g., no containers within a container, although there is support for hierarchical naming. This is different than file systems where there is both hierarchy in the implementation as well as in naming.

Common operations on object storage include: *PUT Object*, which creates an object, with the name, data and metadata provided with the operation; *GET object*, which returns the data and metadata of an object; *HEAD Object*, which returns just the metadata of an object; *DELETE Object*, which deletes an object; *GET Container*, which lists the objects in a container; and *HEAD Container*, which returns the metadata of a container. Object creation is atomic, so that two simultaneous PUTs on the same name will create an object with the data of one PUT, but not some combination of the two. Object storage does not have an atomic rename operation; rename can be emulated non-atomically through COPY and DELETE.

In order to enable a highly distributed implementation the consistency semantics for object storage often includes some degree of *eventual consistency* [20]. Eventual consistency guarantees that if no new updates are made to a given data item, then eventually all accesses to that item will return the same value. There are various degrees of eventual consistency. An important aspect typical to most object stores concerns the listing of the objects in a container; the creation and deletion of an object may be eventually consistent with respect to the listing of its container. In particular, a container listing may not include a recently created object and may not exclude a recently deleted object.

B. Apache Spark

We describe Apache Spark's execution model and how it interacts with storage, pointing out some of the problems that arise when it works on data in object storage.

1) Spark execution model: The execution of a Spark application is orchestrated by the *driver*. The driver divides the application into *jobs* and jobs into *stages*. One stage does not begin execution until the previous stage has completed. Stages consists of *tasks*, where each task is totally independent of the other tasks in that stage, so that the tasks can be executed in parallel. The output of one stage is typically passed as the input to the next stage, so that a task reads its input from the output of the previous stage and/or from storage. Similarly, a task writes its output to the next stage and/or to storage. The driver creates worker processes called *executors* to which it assigns the execution of the tasks.

The execution of a task may fail. To overcome a failure the driver starts a new execution of the same task. The execution of a task may also be slow. Spark has an important feature to deal

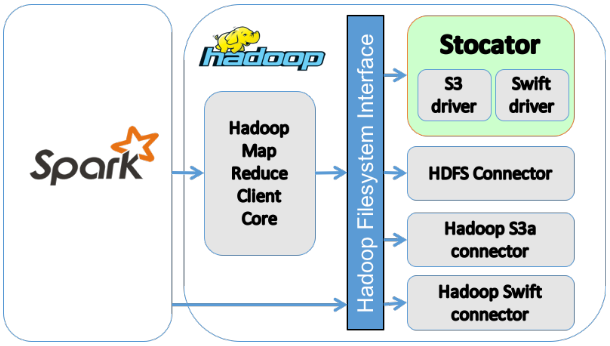

Fig. 1. Hadoop Storage Connectors

with slow execution called *speculation*, where it speculatively executes multiple executions of the same task in parallel. Speculation can cut down on the total elapsed time for a Spark application/job. Thus, a task may be executed multiple times due to a failure or speculation and each such *attempt* to execute a task is assigned a unique identifier, containing a job identifier, a task identifier and an execution attempt number.

2) Spark and its underlying storage: Spark interacts with its storage system through Hadoop [4], primarily through a component called the Hadoop Map Reduce Client Core (HMRCC) as shown in the diagram on the left side in Fig. 1. HMRCC interacts with its underlying storage through the Hadoop File System Interface. A connector that implements the interface must be implemented for each underlying storage system. For example, the Hadoop distribution includes a connector for HDFS, as well as an S3a connector for the Amazon S3 API and a Swift connector for the OpenStack Swift API.

A task writes output to storage through the Hadoop *FileOutputCommitter*. Since each task execution attempt needs to write an output file of the same name, Hadoop employs a rename strategy, where each execution attempt writes its own task temporary file. At *task commit*, the output committer renames the task temporary file to a job temporary file. Task commit is done by the executors, so it occurs in parallel. And then when all of the tasks of a job complete, the driver calls the output committer to do *job commit*, which renames the job temporary files to their final names. Job commit occurs in the driver after all of the tasks have committed and does not benefit from parallelism. This two stage strategy of task commit and then job commit ensures fault tolerance, i.e., that the output contains just a single complete output file for each task despite multiple executions due to failures and speculation.

Hadoop also writes a zero length object with the name _SUCCESS when a job completes successfully, so the case of incomplete results can easily by identified by the absence of a _SUCCESS object. This enables a new version of the file output committer algorithm (called version 2), where the task temporary files are renamed to their final names at task commit and job commit is largely reduced to the writing of the _SUCCESS object. However, as of Hadoop 2.7.3, this algorithm is not yet the default output committer.

Hadoop is highly distributed and thus it keeps its state in its storage system, e.g., HDFS or object storage. In particular, the output committer determines what temporary objects need to be renamed through “directory” listings, i.e., it lists the “directory” of the output dataset to find the “directory” and files holding task temporary and job temporary output. In object stores this is done through container listing operations. However, due to eventual consistency a container listing may not contain an object that was just successfully created, or it may still contain an object that was just successfully deleted. This can lead to situations where some of the legitimate output objects do not get renamed by the output committer, so that the output of the Spark/Hadoop job will be incomplete.

This danger is compounded when speculation is enabled, and thus, despite the benefits of speculation, Spark users are encouraged to run with it disabled. Furthermore, in order to avoid the dangers of eventual consistency entirely, Spark users are often encouraged to copy their input data to HDFS, run their Spark job over the data in HDFS, and then when it is complete, copy the output from HDFS back to object storage. Note, however, that this adds considerable overhead. Existing solutions to this problem require a consistent storage system in addition to object storage [12], [13], [21].

C. Related Work

There has been a variety of work, both from academia and industry, that target different aspects of analytics frameworks. The authors in [22]–[30], focus on analyzing the performance of analytics frameworks.

In particular Ousterhout et al. [22] use an ideal configuration (compute and data layer on the same Virtual Machine), with limited knowledge of the underlying storage system. With the help of an analysis performed on network, disk block time and percentages of resource utilization, this work states that the runtime of analytics applications is generally CPU-bound rather than I/O intensive. A recent work [31] shows that this is not always true; moving from a 1Gbps to a 10Gbps network can have a huge impact on the application runtime.

Albeit valid, all address a specific storage type, that is file storage. Object storage has not been the focus since it is not native to the ecosystems of analytics frameworks. Lately, the authors in [32]–[34] show the existence of an impedance mismatch between the analytics frameworks and object storage that imposes a toll on performance. Moreover, [32] shows that it is possible to improve performance by eliminating the impedance mismatch between the compute and storage layer, which can highly affect the run times of such applications, in particular, when using object storage (e.g., Openstack Swift [19], [35]). There has also been some work from industry and open source to improve this impedance mismatch. Databricks introduced the DirectOutputCommitter [10] for Amazon S3, but it failed to preserve the fault tolerance and speculation properties of the temporary file/rename paradigm. At the same time Hadoop developed version 2 of the FileOutputCommitter [11], which renames files when tasks complete instead of waiting for the completion (commit) of the entire job. However, this solution does not solve the entire problem.

```
val data = Array(1)
val distData = sc.parallelize(data)
val finalData = distData.coalesce(1)
finalData.saveAsTextFile("hdfs://res/data.txt")
```

Fig. 2. A Spark program that executes a single task that produces a single output object.

As mentioned in Section II-A, current connectors from the Hadoop community for the OpenStack Swift [8] and Amazon S3 [7] APIs, can also lead to failures and incorrect executions due to eventual consistency. Some work has been done to address this problem. EMRFS [12], [36] from Amazon and S3mper [13], [37] from Netflix overcome eventual consistency by storing file metadata in DynamoDB [14], an additional storage system separate from the object storage that is strongly consistent. A similar feature called S3Guard [15], [21] is being developed by the Hadoop open source community for the S3a connector. Solutions such as these that require multiple storage systems are complex and can introduce issues of consistency between the stores. They also add cost since users must pay for the additional strongly consistent storage. Our solution does not require any extra storage system.

Recently Databricks introduced a new commit protocol called DBIO [38] that removes the impedance mismatch and guarantees fault tolerance. This new transactional commit protocol provides strong guarantees in the face of various types of failures. Moreover, by enforcing correctness, it is able to provide safe task speculation, atomic file overwrite and consistency for Spark output. DBIO achieves similar objectives as our work, but the solution is proprietary, whereas we fully describe our work, put it in open source, and thoroughly analyze its performance.

D. Motivation

To motivate the need for Stocator we describe the sequence of interactions between Spark and its storage system for a program that executes a single task that produces a single output object as shown in Fig. 2.

At the beginning of a job, the Spark driver and executor recursively create the directories for the task temporary, job temporary and final output. Then, the task outputs the task temporary file. At task commit the executor lists the task temporary directory, and renames the file it finds to its job temporary name. At job commit the driver recursively lists the job temporary directories and renames the file it finds to its final names. Finally, the driver writes the _SUCCESS object.

When this same Spark program runs with the Hadoop Swift or S3a connectors, these file operations are translated to equivalent operations on objects in the object store. These connectors use PUT to create zero byte objects representing the directories, after first using HEAD to check if objects for the directories already exist. When listing the contents of a directory, these connectors descend the "directory tree" listing each directory. To rename objects these connectors use PUT or COPY to copy the object to its new name and then use DELETE on the object at the old name. All of the zero byte directory objects also need to be deleted. Overall the Hadoop Swift connector executes 48 REST operations and the S3a connector executes 117 operations. Table I shows the breakdown according to operation type.

TABLE I
BREAKDOWN OF REST OPERATIONS BY TYPE FOR THE SPARK PROGRAM THAT CREATES AN OUTPUT CONSISTING OF A SINGLE OBJECT.

	HEAD Object	PUT Object	COPY Object	DELETE Object	GET Cont.	Total
Hadoop-Swift	25	7	3	8	5	48
S3a	71	5	2	4	35	117
Stocator	4	3	–	–	1	8

In the next section we describe Stocator, which leverages object storage semantics to replace the temporary file/rename paradigm and takes advantage of hierarchal naming to avoid the creation of "directory" objects. For the Spark program in Fig. 2 Stocator executes just 8 REST operations: 3 PUT object, 4 HEAD object and 1 GET container.

III. STOCATOR LOGIC

The right side of Fig. 1 shows how Stocator fits underneath HMRCC; it implements the Hadoop Filesystem Interface just like the other storage connectors. Below we describe the basic Stocator protocol; and then how it streams data, deals with eventual consistency, and reduces operations on the read path. Finally we provide several examples of the protocol in action.

A. Basic Stocator protocol

The overall strategy used by Stocator to avoid rename is to write output objects directly to their final name and then to determine which objects actually belong to the output at the time that the output is read by its consumer, e.g., the next Spark job in a sequence of jobs. Stocator does this in a way that preserves the fault tolerance model of Spark/Hadoop and enables speculation. Below we describe the components of this strategy.

As described in Section II the driver orchestrates the execution of a Spark application. In particular, the driver is responsible for creating a "directory" to hold an application's output dataset. Stocator uses this "directory" as a marker to indicate that it wrote the output. In particular, Stocator writes a zero byte object with the name of the dataset and object metadata that indicates that the object was written by Stocator. All of the dataset's parts are stored hierarchically under this name.

Then when a Spark task asks to create a temporary object for its part through HMRCC, Stocator recognizes the pattern of the name and writes the object directly to its final name so it will not need to be renamed. If Spark executes a task multiple times due to failures, slow execution or speculative execution, each execution attempt is assigned a number. The Stocator object naming scheme includes this attempt number so that individual attempts can be distinguished.

Finally, when all tasks have completed successfully, Spark writes a _SUCCESS object through HMRCC; the presence of a _SUCCESS object means that there was a correct execution for each task and that there is an object for each part in the output. Notice that by avoiding rename, Stocator also avoids the need for list operations during task and job commit that may lead to incorrect results due to eventual consistency.

B. Alternatives for reading an input dataset

Stocator delays the determination of which parts belong to an output dataset until it reads the dataset as input. We consider two options.

The first option is simpler to implement since it can be done entirely in the implementation of Stocator. It depends on the assumption that Spark exhibits fail-stop behavior, i.e., that a Spark server executes correctly until it halts. After determining that the dataset was produced by Stocator through reading the metadata from the object written with the dataset's name, and checking that the _SUCCESS object exists, Stocator lists the object parts belonging to the dataset through a GET container operation. If there are objects in the list representing multiple execution attempts for same task, Stocator will choose the largest. Given the fail-stop assumption, the fact that all successful execution attempts write the same output, and that it is certain that at least one attempt succeeded (otherwise there would not be a _SUCCESS object), this is the correct choice.

At the completion of a Spark job, the second option includes the creation of a manifest inside the _SUCCESS object that contains a list of all the successful task execution attempts completed by the job. Now after determining that the dataset was produced by Stocator through reading the metadata from the object written with the dataset's name, and checking that the _SUCCESS object exists, Stocator reads the manifest of successful task execution attempts from the _SUCCESS object. Stocator uses the manifest to reconstruct the list of constituent object parts of the dataset. In particular, the construction of the object part names follows the same pattern used when the parts were written.

The benefit of the second option is that it solves the remaining eventual consistency issue by constructing the object names from the manifest rather than issuing a REST command to list the object parts, which may not return a correct result in the presence of eventual consistency. However, given that our primary target for Stocator is IBM Cloud Object Storage and that its container listing is immediately consistent with respect to the writing and deleting of objects, we have not had the need to implement this option.

C. Streaming of output

When Stocator outputs data it streams the data to the object store as the data is produced using chunked transfer encoding. Normally the total length of the object is one of the parameters of a PUT operation and thus needs to be known before starting the operation. Since Spark produces the data for an object on the fly and the final length of the data is not known until all of its data is produced, this would mean that Spark would need to store the entire object data prior to starting the PUT. To avoid running out of memory, a storage connector for Spark can store the object in the Spark server's local file system as the connector produces the object's content, and then read the object back from the file to do the PUT operation on the object store. Indeed this is what the default Hadoop Swift and S3a connectors do. Instead Stocator leverages HTTP chunked transfer encoding, which is supported by the Swift API. In chunked transfer encoding the object data is sent in chunks, the sender needs to know the length of each chunk, but it does not need to know the final length of the object content before starting the PUT operation. S3a has an optional feature, not activated by default, called fast upload, where it leverages the multi-part upload feature of the S3 API. This achieves a similar effect to chunked transfer encoding except that it uses more memory since the minimum part size for multi-part upload is larger than for chunked transfer.

```
val data = Array(1, 2, 3)
val distData = sc.parallelize(data)
distData.saveAsTextFile(”swift2d://res.sl/data.txt”)
```

Fig. 3. A Spark program where three tasks each write an object part.

D. Optimizing the read path

We describe several optimizations that Stocator uses to reduce the number of operations on the read path.

The first optimization can remove a HEAD operation that occurs just before a GET operation for the same object. In particular, the storage connector often reads the metadata of an object just before its data. Typically this is to check that the object exists and to obtain the size of the object. In file systems this is performed by two different operations. Accordingly a naive implementation for object storage would read object metadata through a HEAD operation, and then read the data of the object itself through a GET operation. However, object store GET operations also return the metadata of an object together with its data. In many of these cases Stocator is able to remove the HEAD operation, which can greatly reduce the overall number of operations invoked on the underlying object storage system.

A second optimization is caching the results of HEAD operations. A basic assumption of Spark is that the input is immutable. Thus, if a HEAD is called on the same input object multiple times, it should return the same result. Stocator uses a small cache to reduce these calls.

E. Examples

We show here some examples of Stocator at work. For simplicity we focus on Stocator's interaction with HMRCC to eliminate the rename paradigm and so we do not show all of the requests that HMRCC makes on Stocator, e.g., to create/delete "directories" and check their status.

Figure 3 shows a simple Spark program that will be executed by three tasks, each task writing its part to the output dataset called $data.txt$ in a container called res. The *swift2d:* prefix in the URI for the output dataset indicates that Stocator is to be used as the storage connector. Table II shows the

TABLE II
POSSIBLE OPERATIONS PERFORMED BY THE SPARK APPLICATION SHOWED IN FIG. 3

	Hadoop Map Reduce Client Core	Stocator
1	PUT /res/data.txt/_temporary/0/_temporary/attempt_201512062056_0000_m_000000_0/part-00000	PUT /res/data.txt/part-00000_attempt_201512062056_0000_m_000000_0
2	PUT /res/data.txt/_temporary/0/_temporary/attempt_201512062056_0000_m_000000_0/part-00001	PUT /res/data.txt/part-00001_attempt_201512062056_0000_m_000000_0
3	PUT /res/data.txt/_temporary/0/_temporary/attempt_201512062056_0000_m_000000_0/part-00002	PUT /res/data.txt/part-00002_attempt_201512062056_0000_m_000000_0
4	PUT /res/data.txt/_temporary/0/_temporary/attempt_201512062056_0000_m_000000_1/part-00002	PUT /res/data.txt/part-00002_attempt_201512062056_0000_m_000000_1
5	PUT /res/data.txt/_temporary/0/_temporary/attempt_201512062056_0000_m_000000_2/part-00002	PUT /res/data.txt/part-00002_attempt_201512062056_0000_m_000000_2
6	DELETE /res/data.txt/_temporary/0/_temporary/attempt_201512062056_0000_m_000000_0/part-00002	DELETE /res/data.txt/part-00002_attempt_201512062056_0000_m_000000_0
7	DELETE /res/data.txt/_temporary/0/_temporary/attempt_201512062056_0000_m_000000_2/part-00002	DELETE /res/data.txt/part-00002_attempt_201512062056_0000_m_000000_2
8	Task commits and job commit generate 2 pairs of COPY and DELETE for each successful attempt	No operations are performed here
9	PUT /res/data.txt/_SUCCESS	PUT /res/data.txt/_SUCCESS

operations that can be executed by our example in different situations.

Lines 1-3 and 8-9 are executed when each task runs exactly once and the program completes successfully. We show the requests that HMRCC generates; for each task it issues one request to create a temporary object and two requests to "rename" it (copy to a new name and delete the object at the former name). We see that Stocator intercepts the pattern for the temporary name that it receives from HMRCC, and creates the final names for the objects directly. At the end of the run Spark creates the _SUCCESS object.

Lines 1-5, instead, shows an execution where Spark decides to execute Task 2 three times, i.e., three attempts. This could be because the first and second attempts failed or due to speculation because they were slow. Notice that Stocator includes the attempt number as part of the name of the objects that it creates.

By adding lines 6-9 to the previous, we show what happens when Spark is able to clean up the results from the duplicate attempts to execute Task 2. In particular, Spark aborts attempts 0 and 2, and commits attempt 1. When Spark aborts attempts 0 and 2, HMRCC deletes their corresponding temporary objects. Stocator recognizes the pattern for the temporary objects and deletes the corresponding objects that it created.

If Spark is not able to clean up the results from the duplicate attempts to execute Task 2, we have lines 1-5 and 8-9. In particular, we see that Stocator created five object parts, one each for Tasks 0 and 1, and three for Task 2 due to its extra attempts. We assume as in the previous situation that it is attempt 1 for Task 2 that succeeded. Stocator recognizes this through the manifest stored in the _SUCCESS object.

IV. METHODOLOGY

We describe the experimental platform, deployment scenarios, workloads and performance metrics that we use to evaluate Stocator.

A. Experimental Platform

Our experimental infrastructure includes a Spark cluster, an IBM Cloud Object Storage (formerly Cleversafe) cluster, Keystone, and Graphite/Grafana. The Spark cluster consists of three bare metal servers. Each server has a dual Intel Xeon E52690 processor with 12 hyper-threaded 2.60 GHz cores (so 48 hyper-threaded cores per server), 256 GB memory, a 10 Gbps NIC and a 1 TB SATA disk. That means that the total parallelism of the Spark cluster is 144. We run 12 executors on each server; each executor gets 4 cores and 16 GB of memory. We use Spark submit to run the workloads and the driver runs on one of the Spark servers (always the same server). We use the standalone Spark cluster manager.

Our IBM Cloud Object Storage (COS) [39] cluster also runs on bare metal. It consists of two Accessers, front end servers that receive the REST commands and then orchestrate their execution across twelve Slicestors, which hold the storage. Each Accesser has two 10 Gbps NICs bonded to yield 20 Gbps. Each Slicestor has twelve 1 TB SATA disks for data. The Information Dispersal Algorithm (IDA) or erasure code is (12, 8, 10), which means that the erasure code splits the data into 12 parts, 8 parts are needed to read the data, and at least 10 parts need to be written for a write to complete. IBM COS exposes multiple object APIs; we use the Swift and S3 APIs.

We employ HAProxy for load balancing. It is installed on each of the Spark servers and configured in round-robin so that connections opened by a Spark server with the object storage alternate between Accessers. Each of the three Spark servers has a 10 Gbps NIC thus, the maximum network bandwidth between the Spark cluster and the COS cluster is 30 Gbps.

Keystone and Graphite/Grafana run on virtual machines. Keystone provides authentication/authorization for the Swift API. We collect monitoring data on Graphite and view it through Grafana to check that there are no unexpected bottlenecks during the performance runs. In particular we use the Spark monitoring interface and the collectd daemon to collect monitoring data from the Spark servers, and we use the Device API of IBM COS to collect monitoring data from the Accessers and the Slicestors.

B. Deployment scenarios

In our experiments, we compare Stocator with the Hadoop Swift and S3a connectors. By using different configurations of these two connectors, we define six scenarios: (i) Hadoop-Swift Base (**H-S Base**), (ii) S3a Base (**S3a Base**), (iii) Stocator Base (**Stocator**), (iv) Hadoop-Swift Commit V2 (**H-S Cv2**), (v) S3a Commit V2 (**S3a Cv2**) and (vi) S3a Commit V2 + Fast Upload (**S3a Cv2+FU**). These scenarios are split into 3 groups according to the optional optimization features that are active. The first group, with the suffix *Base*, uses connectors out of the box, meaning that no optional features are active. The

TABLE III
AVERAGE RUN TIME

	Read-Only 50GB	Read-Only 500GB	Teragen	Copy	Wordcount	Terasort	TPC-DS
Hadoop-Swift Base	37.80 ± 0.48	393.10 ± 0.92	624.60 ± 4.00	622.10 ± 13.52	244.10 ± 17.72	681.90 ± 6.10	**101.50 ± 1.50**
S3a Base	**33.30 ± 0.42**	254.80 ± 4.00	699.50 ± 8.40	705.10 ± 8.50	193.50 ± 1.80	746.00 ± 7.20	104.50 ± 2.20
Stocator	34.60 ± 0.56	**254.10 ± 5.12**	**38.80 ± 1.40**	**68.20 ± 0.80**	**106.60 ± 1.40**	**84.20 ± 2.04**	111.40 ± 1.68
Hadoop-Swift Cv2	37.10 ± 0.54	395.00 ± 0.80	171.30 ± 6.36	175.20 ± 6.40	166.90 ± 2.06	222.70 ± 7.30	102.30 ± 1.16
S3a Cv2	35.30 ± 0.70	255.10 ± 5.52	169.70 ± 4.64	185.40 ± 7.00	111.90 ± 2.08	221.90 ± 6.66	104.00 ± 2.20
S3a Cv2 + FU	35.20 ± 0.48	**254.20 ± 5.04**	56.80 ± 1.04	86.50 ± 1.00	112.00 ± 2.40	105.20 ± 3.28	103.10 ± 2.14

second group, with the suffix *Commit V2*, uses the version 2 of Hadoop FileOutputCommitter that reduces the number of copy operations on the object storage (as described in Section II). The last group, with the suffix *Commit V2 + Fast Upload*, uses both version 2 of Hadoop FileOutputCommitter and an optimization feature of S3a called S3AFastOutputStream that streams data to the object storage as it is produced (as described in Section III). We decided to compare Stocator to the **Base** scenarios, because the optional features are experimental and not always stable.

All experiments run on *Spark 2.0.1* with a patched [40] version of Hadoop 2.7.3. This patch allows us to use, for the S3a scenarios, *Amazon SDK version 1.11.53* instead of version 1.7.4. The Hadoop-Swift scenarios run with the default Hadoop-Swift connector that comes with Hadoop 2.7.3. Finally, the Stocator scenario runs with *stocator 1.0.8*.[2]

C. Benchmark and Workloads

To study the performance of our solution we use several workloads from popular benchmark suites that cover different kinds of applications. The workloads span from simple applications that target a single and specific feature of the connectors (micro benchmarks), to complex applications composed by several jobs (macro benchmarks).

The micro benchmarks include three applications: (i) Read-only, (ii) Write-only and (iii) Copy. The Read-only application reads two different text datasets, one whose size is 46.5 GB and the second 465.6 GB, and counts the number of lines in them. For the Write-only application we use the popular Teragen application, available in the Spark example suite, that only performs write operations, creating a dataset of 46.5 GB. The last application that we use for our micro benchmark set is what we call the Copy application; it copies the small dataset used by the Read-only application.

We also use three macro benchmarks. The first, Wordcount from Intel Hi-Bench [41], [42] test suite, is the "Hello World" application for parallel computing. It is a read-intensive workload, that reads an input 46.5 GB text file, computes the number of times each word occurs in the file and then writes a much smaller output file (1.3 MB) containing the word counts. The second macro benchmark, Terasort, is a popular application used to understand the performance of large scale computing frameworks like Spark and Hadoop. Its input dataset is the output of the Teragen application used in the micro benchmarks. The third macro benchmark,TPC-DS, is the Transaction Processing Performance Council's decision-support benchmark test [43], [44] implemented with DataBricks' Spark-Sql-Perf library [45]. It executes several complex queries on files stored in Parquet format [46]; the input dataset size is 50 GB, which is compressed to 13.8 GB when converted to Parquet. The query set that we use to perform our experiments is composed of the following 8 TPC-DS queries: q34, q43, q46, q59, q68, q73, q79 and ss_max. These are the queries from the *Impala* subset that work with the Hadoop-Swift connector. Stocator and S3a support all of the queries in the Impala subset.

The inputs and outputs for the Read-only, Copy, Wordcount, Teragen and Terasort benchmarks are divided into 128 MB objects. We also run Spark with a partition size of 128 MB.

D. Performance metrics

We evaluate the different connectors and scenarios by using metrics that target the various optimization features. As a general metric we use the total runtime of the application; this provides a quick overview of the performance of a specific scenario. To delve into the reason behind the performance we use two additional metrics. The first is the number of REST calls – and their type; with this metric we are able to understand the load on the object storage imposed by the connector. The second metric is the number of bytes read from, written to and copied in the object storage; this also help us to understand the load on the object storage imposed by the connectors.

V. EXPERIMENTAL EVALUATION

We now present a comparative analysis between the different scenarios that we defined in Section IV-B. We first show the benefit of Stocator through the average run time of the different workloads. Then we compare the number of REST operations issued by the Compute Layer toward the Object Storage and the relative cost for these operations charged by cloud object store services. Finally we compare the number of bytes transferred between the Compute Layer and the Object Storage.

A. Reduction in run time

For each workload we ran each scenario ten times. We report the average and standard deviation in Table III. The results shows that, when using a connector out of the box

[2] These were the latest official releases of these software components at the time of writing the paper.

TABLE IV
WORKLOAD SPEEDUPS WHEN USING STOCATOR

	Read-Only 50GB	Read-Only 500GB	Teragen	Copy	Wordcount	Terasort	TPC-DS
Hadoop-Swift Base	x1.09	x1.55	x16.09	x9.12	x2.29	x8.10	x0.91
S3a Base	x0.96	x1.00	x18.03	x10.33	x1.82	x8.86	x0.94
Stocator	x1	x1	x1	x1	x1	x1	x1
Hadoop-Swift Cv2	x1.07	x1.55	x4.41	x2.57	x1.57	x2.64	x0.92
S3a Cv2	x1.02	x1.00	x4.37	x2.72	x1.05	x2.64	x0.93
S3a Cv2 + FU	x1.02	x1.00	x1.46	x1.27	x1.05	x1.25	x0.93

TABLE V
RATIO OF REST CALLS COMPARED TO STOCATOR

	Read-Only 50GB	Read-Only 500GB	Teragen	Copy	Wordcount	Terasort	TPC-DS
Hadoop-Swift Base	x2.41	x2.92	x11.51	x9.18	x9.21	x8.94	x2.39
S3a Base	x1.71	x1.96	x33.74	x24.93	x25.35	x24.23	x2.40
Stocator	x1	x1	x1	x1	x1	x1	x1
Hadoop-Swift Cv2	x2.41	x2.92	x7.72	x6.55	x6.92	x6.29	x2.39
S3a Cv2	x1.71	x1.96	x21.15	x16.18	x16.44	x15.41	x2.40
S3a Cv2 + FU	x1.71	x1.96	x21.15	x16.18	x16.44	x15.41	x2.40

TABLE VI
FINANCIAL COST FOR REST CALLS COMPARED TO STOCATOR FOR IBM, AWS, GOOGLE AND AZURE INFRASTRUCTURE

	Read-Only 50GB	Read-Only 500GB	Teragen	Copy	Wordcount	Terasort	TPC-DS
Hadoop-Swift Base	x9.72	x13.67	x8.23	x8.60	x8.58	x8.57	x2.23
S3a Base	x1.63	x1.94	x27.82	x26.74	x26.84	x25.88	x2.25
Stocator	x1	x1	x1	1	x1	x1	x1
Hadoop-Swift Cv2	x9.72	x13.67	x5.24	x5.86	x5.85	x5.81	x2.23
S3a Cv2	x1.63	x1.94	x17.59	x17.29	x17.36	x16.40	x2.25
S3a Cv2 + FU	x1.63	x1.94	x17.55	x17.29	x17.34	x16.40	x2.25

and under workloads that perform write operations, Stocator performs much better than Hadoop-Swift and S3a. Only by activating and configuring optimization features provided by the Hadoop ecosystem, Hadoop-Swift and S3a manage to close the gap with Stocator, but they still fall behind.

Table IV shows the speedups that we obtain when using Stocator with respect to the other connectors. We see a relationship between Stocator performance and the workload; the more write operations performed, the greater the benefit obtained. On the one hand the write-only workloads, like Teragen, run 18 time faster with Stocator compared to the other out of the box connectors, 4 time faster when we enable FileOutputCommitter Version 2, and 1.5 times faster when we also add the S3AFastOutputStream feature. On the other hand, workloads more skewed toward read operations, like Wordcount, have lower speedups.

These results are possible thanks to the algorithm implemented in Stocator. Unlike the alternatives, Stocator removes the rename – and thus copy – operations completely. In contrast, the other connectors, even with FileOutputCommitter Version 2, must still rename each output object once, although the overhead of the remaining renames is partially masked since they are carried out by the executors in parallel.

Stocator performs slightly worse than S3a on two of the workloads that contain only read operations (no writes), Read-only 50 GB and TPC-DS, and virtually the same for the larger 500 GB Read-only workload. We have identified a small start-up cost that we have not yet removed from Stocator that can explain the difference between the results for the 50 GB and 500 GB Read-only workload.[3] As expected the results for the read-only workloads for S3a and Hadoop-Swift connectors are virtually the same with and without the FileOutputCommitter Version 2 and S3AFastOutputStream features; these features optimize the write path and do not affect the read path.

B. Reduction in the number of REST calls

Next we look at the number of REST operations executed by Spark in order to understand the load generated on the object storage infrastructure. Figure 4 shows that, in all the workloads, the scenario that uses Stocator achieves the lowest number of REST calls and thus the lowest load on the object storage.

When looking at Read-only with both 50 and 500 GB dataset, the scenario with Hadoop-Swift has the highest number of REST calls and more than double compared to the

[3] Since writing this paper we have removed start-up costs and improved the performance of the read-path of Stocator.

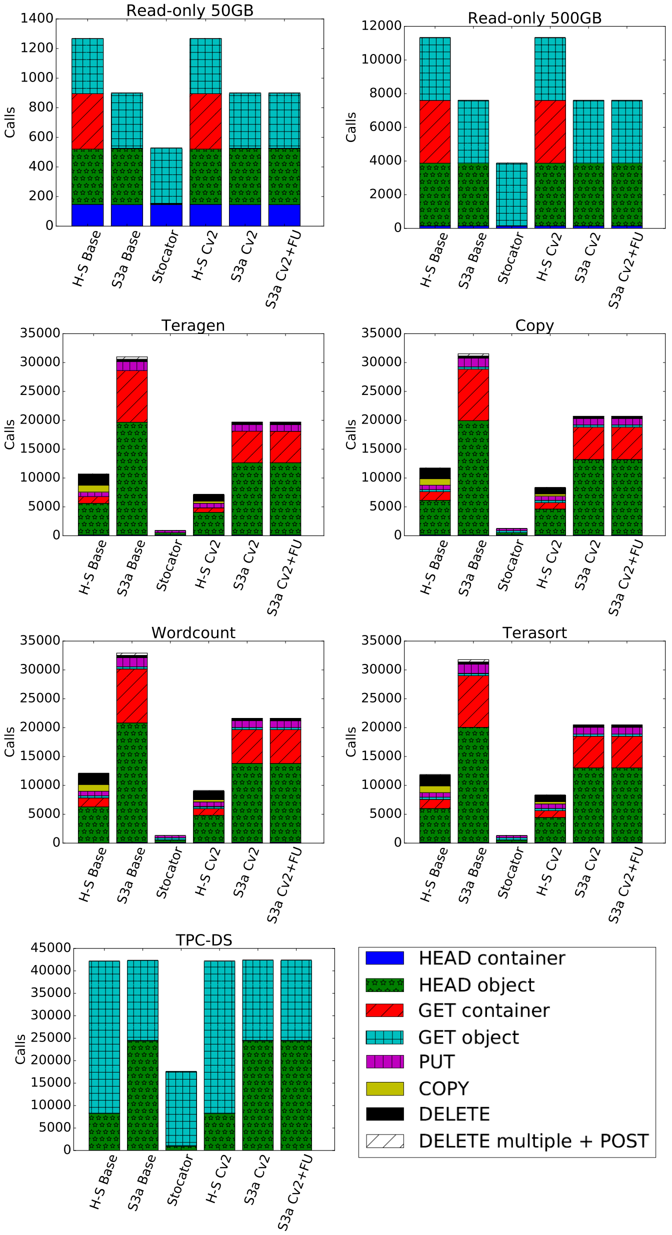

Fig. 4. Benchmarks REST calls comparison

scenario with Stocator. The Hadoop-Swift connector does many more GET calls on containers to list their contents. Compared to S3a, Stocator is optimized to reduce the number of HEAD calls on the objects. We see this consistently for all of the workloads.

In write-intensive workloads, Teragen and Copy, we see that the scenarios that use S3a as the connector have the highest number of REST calls while Stocator still has the lowest. Compared to Hadoop-Swift and Stocator, S3a performs many more HEAD calls for the objects and GET for the containers. Stocator also does not need to create temporary "directories" objects, thus uses far fewer HEAD requests, and does not need to DELETE objects; this is possible because our algorithm is conceived to avoid renaming objects after a task or job completes. Table V shows the number of REST calls that is possible to save by using Stocator. We observe that, for write-intensive workloads, Stocator issues 6 to 11 times less REST calls compared to Hadoop-Swift and 15 to 33 times less compared to S3a, depending on the optimization features.

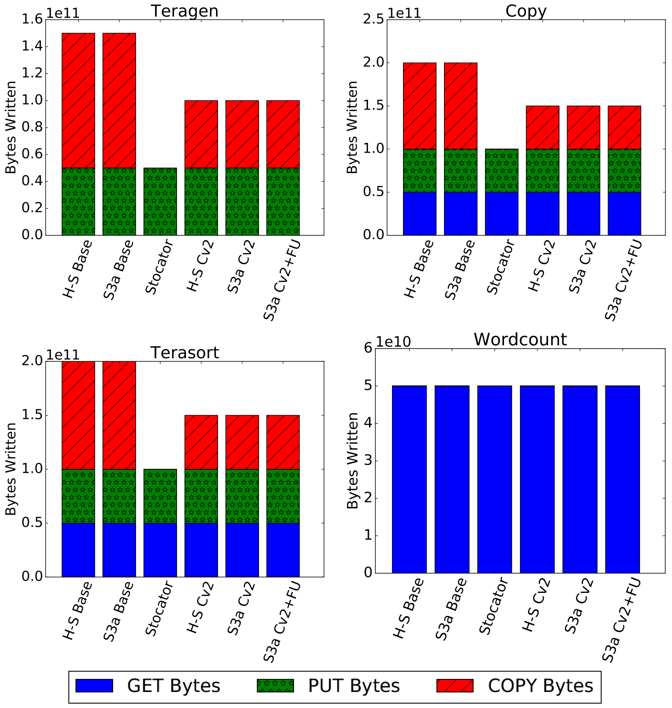

Fig. 5. Object Storage bytes read/written comparison

Having a low load on the Object Storage has advantages both for the data scientist and the storage providers. On the one hand, cloud providers will be able to serve a bigger pool of consumers and give them a better experience. On the other hand, since most public providers charge fees based on the number of operations performed on the storage tier, reducing the operations results in a lower cost for the data scientists. Table VI shows the relative costs for the REST operations. For the workloads with write (Teragen, Copy, Terasort and Wordcount) Stocator is 16 to 18 times less expensive than S3a run with FileOutputCommitter version 2, and 5 to 6 times less expensive than Hadoop-Swift. To calculate the cost ratio we used the pricing models of IBM [47], AWS [48], Google [49] and Azure [50]; given that the models are very similar we report the average price.

As an additional way of measuring the load on the object storage and confirming the fact that Stocator does not perform COPY (or DELETE) operations we present the number of bytes read and written to the object storage. From Fig. 5 we see that Stocator does not write more data than needed on the storage. In contrast we confirm that Hadoop-Swift and S3a base write each object three times – one from the PUT and two from the COPY – while Stocator only does it once. Only by enabling FileOutputCommitter Version 2 in Hadoop, it is possible to reduce the COPY operations to one, but this is still one more object copy compared to Stocator. We show only the workloads that have write operations since during a read-only workload, the number of bytes read from the object storage are identical for all of the connectors and scenarios (as we see from the Wordcount workload in Fig. 5 where the number of bytes written is very small). As expected the S3a scenario that uses the S3AFastOutputStream optimization gains no benefit with respect to the number of bytes written to the object storage.

VI. Conclusion and Future Work

We have presented a high performance object storage connector for Apache Spark called Stocator, which has been made available to the open source community [17]. Stocator overcomes the impedance mismatch of previous open source connectors with their storage, by leveraging object storage semantics rather than trying to treat object storage as a file system. In particular Stocator eliminates the rename paradigm without sacrificing fault tolerance or speculative execution. It also deals correctly with the eventually consistent semantics of object stores without the need to use an additional consistent storage system. Finally, Stocator leverages HTTP chunked transfer encoding to stream data as it is produced to object storage, thereby avoiding the need to first write output to local storage.

We have compared Stocator's performance with the Hadoop Swift and S3a connectors over a range of workloads and found that it executes far less operations on object storage, in some cases as little as one thirtieth. This reduces the load both for client software and the object storage service, as well as reducing costs for the client. Stocator also substantially increases the performance of Spark workloads, especially write intensive workloads, where it is as much as 18 times faster than alternatives.

In the future we plan to continue improving the read performance of Stocator and extending it to support additional elements of the Hadoop ecosystem such as MapReduce (primarily a matter of testing) and Hive. We also plan to continue extend our work on the impedance mismatch to enable additional data intensive frameworks such as those for deep learning to work efficiently with object storage.

References

[1] Amazon S3. https://aws.amazon.com/s3/.

[2] Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs/.

[3] IBM Cloud Object Storage. https://www.ibm.com/cloud-computing/products/storage/object-storage/cloud/.

[4] Apache Hadoop. http://hadoop.apache.org/.

[5] Apache Spark. http://spark.apache.org/.

[6] IBM Stocator Blog. http://www.spark.tc/stocator-the-fast-lane-connecting-object-stores-to-spark/.

[7] AWS SDK for Java. https://aws.amazon.com/sdk-for-java/.

[8] OpenStack Foundation. sahara-extra. https://github.com/openstack/sahara-extra/tree/master/hadoop-swiftfs.

[9] Apache Hadoop HDFS. https://hortonworks.com/apache/hdfs/.

[10] [SPARK-10063][SQL] Remove DirectParquetOutputCommitter #12229. https://github.com/apache/spark/pull/12229.

[11] Apache Hadoop JIRA. https://issues.apache.org/jira/browse/MAPREDUCE-6336.

[12] Amazon EMRFS Blog. https://aws.amazon.com/blogs/aws/emr-consistent-file-system/.

[13] Netflix S3mper Blog. http://techblog.netflix.com/2014/01/s3mper-consistency-in-cloud.html.

[14] Amazon DynamoDB. https://aws.amazon.com/dynamodb/.

[15] Hadoop S3Guard. http://www.slideshare.net/hortonworks/s3guard-whats-in-your-consistency-model.

[16] Swift API. https://developer.openstack.org/api-ref/object-storage/.

[17] IBM Stocator Source Code. https://github.com/SparkTC/stocator.

[18] G. Adam Cox. Simulating E.T.: Or how to insert individual files into object storage from within a map function in Apache Spark. Https://medium.com/ibm-watson-data-lab/simulating-e-t-e34f4fa7a4f0.

[19] OpenStack Swift. http://swift.openstack.org/.

[20] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009.

[21] Apache Hadoop S3Guard JIRA. https://issues.apache.org/jira/browse/HADOOP-13345.

[22] K. Ousterhout *et al.*, "Making sense of performance in data analytics frameworks," in *12th USENIX Symp. on NSDI*, 2015.

[23] G. Ananthanarayanan *et al.*, "Disk-locality in datacenter computing considered irrelevant." in *HotOS*, 2011.

[24] E. B. Nightingale *et al.*, "Flat datacenter storage," in *10th USENIX Symp. on OSDI*, 2012.

[25] J. Xie *et al.*, "Improving mapreduce performance through data placement in heterogeneous hadoop clusters," in *IEEE Intl Symp. on IPDPSW*, 2010.

[26] Z. Guo *et al.*, "Investigation of data locality and fairness in mapreduce," in *Proc. of 3rd Intl Workshop on MapReduce and its Applications Date*. ACM, 2012.

[27] K. Ranganathan *et al.*, "Decoupling computation and data scheduling in distributed data-intensive applications," in *11th IEEE Intl Symp. on HPDC*, 2002.

[28] G. Wang *et al.*, "A simulation approach to evaluating design decisions in mapreduce setups." in *MASCOTS*. Citeseer, 2009.

[29] R.-I. Roman *et al.*, "Understanding spark performance in hybrid and multi-site clouds," in *6th Intl Workshop on BDAC*, 2015.

[30] D. Venzano *et al.*, "A measurement study of data-intensive network traffic patterns in a private cloud," in *Proc. of 6th IEEE Intl Conf. on UCC*, 2013.

[31] A. Trivedi *et al.*, "On the [ir] relevance of network performance for data processing," *Network*, vol. 40, p. 60, 2016.

[32] F. Pace *et al.*, "Experimental performance evaluation of cloud-based analytics-as-a-service," in *9th IEEE Intl Conf. on Cloud Computing, CLOUD*, 2016. [Online]. Available: \url{http://dx.doi.org/10.1109/CLOUD.2016.0035}

[33] L. Rupprecht *et al.*, "Big data analytics on object stores: A performance study," *red*, vol. 30, p. 35, 2014.

[34] ——, "Swiftanalytics: Optimizing object storage for big data analytics," in *Cloud Engineering (IC2E), 2017 IEEE International Conference on*. IEEE, 2017, pp. 245–251.

[35] J. Arnold, *OpenStack Swift: Using, Administering, and Developing for Swift Object Storage*. O'Reilly Media, Inc., 2014.

[36] Amazon EMRFS. http://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-fs.html.

[37] Netflix S3mper. https://github.com/Netflix/s3mper.

[38] Databricks DBIO. https://databricks.com/blog/2017/05/31/transactional-writes-cloud-storage.html.

[39] J. K. Resch *et al.*, "Aont-rs: Blending security and performance in dispersed storage systems," in *Proc. of the 9th USENIX Conf. on FAST*, 2011.

[40] Apache Hadoop JIRA for Amazon Web Services. https://issues.apache.org/jira/browse/HADOOP-12269.

[41] Intel HiBench. https://github.com/intel-hadoop/HiBench.

[42] S. Huang *et al.*, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *36th ICDEW*, 2010.

[43] TPC-DS. http://www.tpc.org/tpcds/.

[44] R. O. Nambiar *et al.*, "The making of tpc-ds," in *Proc. of the 32nd Intl Conf. on VLDB Endowment*, 2006.

[45] DataBricks Spark SQL Performance Tests. https://github.com/databricks/spark-sql-perf.

[46] Apache Parquet. https://parquet.apache.org/.

[47] IBM REST calls cost. http://www-03.ibm.com/software/products/en/object-storage-public/#othertab2.

[48] AWS REST calls cost. https://aws.amazon.com/s3/pricing/.

[49] Google REST calls cost. https://cloud.google.com/storage/pricing.

[50] Azure REST calls cost. https://azure.microsoft.com/en-us/pricing/details/storage/blobs/.