HORIZON 2020 FRAMEWORK PROGRAMME

IOStack

(H2020-644182)

Software-Defined Storage for Big Data on top of the OpenStack platform

D4.3 Summary and demonstration of results

Due date of deliverable: 31-12-2017
Actual submission date: 04-01-2018

Start date of project: 01-01-2015 Duration: 36 months

Summary of the document

Document Type	Deliverable
Dissemination level	Public
State	v5.0
Number of pages	43
WP/Task related to this document	WP4
WP/Task responsible	IBM Israel
Leader	Yosef Moatti (IBM)
Technical Manager	Paula Ta-Shma (IBM)
Quality Manager	Francesco Pace (EUR)
Author(s)	Yosef Moatti, Paula Ta-Shma, Guy Gerson-Golan, Francesco Pace, Gil Vernik, Michael Factor, Eliot Kolodner, Effi Ofer, Pietro Michiardi
Partner(s) Contributing	Raul Gracia Tinedo
Document ID	IOStack_D4.3_Public.pdf
Abstract	This deliverable reviews the implementation of the SDS services for analytics in IOStack in 2017. In a first part, we review the design and implementation of the Pluggable Spark Filters technology, which enables the acceleration of Spark Big Data analytics on data stored in object stores through a-priori filtering of objects, followed by a review and discussion of the experimental evaluation that we performed on a large cluster. In the second part of this document, we review the design and results of the stocator connector which very much enhances the access of Object Stores from Apache Spark.
Keywords	analytics, spark, csv, parquet, SQL, multi-dimensional indexing, swift, COS, Spark connector, Object Store

History of changes

Version	Date	Author	Summary of changes
1.0	22-10-2017	Yosef Moatti, Paula Ta-Shma, Guy Gerson-Golan, Francesco Pace, Gil Vernik, Michael Factor, Eliot Kolodner, Effi Ofer, Pietro Michiardi	Initial version
2.0	04-12-2017	Yosef Moatti, Paula Ta-Shma, Guy Gerson-Golan, Tal Ariel	First full version - addresses Raul's comments on initial version
3.0	21-12-2017	Yosef Moatti, Paula Ta-Shma, Guy Gerson-Golan, Tal Ariel	Addresses almost all of Raul's remarks
4.0	24-12-2017	Yosef Moatti, Paula Ta-Shma, Guy Gerson-Golan, Francesco Pace	Addresses Raul's remarks for his second full review
5.0	04-01-2018	Yosef Moatti, Paula Ta-Shma, Gal Lushi	Add advanced experimental results

Table of Contents

1 Executive summary

In this deliverable, we detail the achievements of WP4 related activities during 2017: Pluggable Spark SQL Filters (developed in 2017) and the Stocator technology developed from the onset of the IOStack project until 2017.

In the first part of this document, we present Pluggable Spark SQL Filters and how we applied this technology to the GridPocket use case. Similarly to the Spark SQL Pushdown research, developed in 2015-2016 and which was presented in [1], the Pluggable Spark SQL Filters research strives to improve analytics performance of Apache Spark jobs which operate on big datasets in Object Stores. Both efforts aim at reducing the amount of data sent from Object Storage to Spark. Besides this common high-level goal, the path that we pursued this past year is quite different: instead of using active storage to reduce the amount of data sent, we use indexing techniques, more specifically data skipping techniques to avoid touching objects which are not relevant to a query.

In the second part of this document, we present Stocator, a high-performance object store connector for Apache Spark, that takes advantage of object store semantics. Previous connectors have assumed file system semantics, in particular, achieving fault tolerance and allowing speculative execution by creating temporary files to avoid interference between worker threads executing the same task and then renaming these files. Rename is not a native object store operation; not only is it not atomic, but it is implemented using a costly copy operation and a delete. Instead the Stocator connector leverages the inherent atomicity of object creation, and by avoiding the rename paradigm it greatly decreases the number of operations on the object store as well as enabling a much simpler approach to dealing with the eventually consistent semantics typical of object stores. We have implemented Stocator and shared it in open source. Performance testing shows that it is as much as 18 times faster for write intensive workloads and performs as much as 30 times fewer operations on the object store than the legacy Hadoop connectors, reducing costs both for the client and the object storage service provider.
The Stocator connector has been in production use in IBM Cloud from 2016 for connecting the Apache Spark service to the IBM Cloud Object Storage.

2 Pluggable Spark SQL Filters

2.1 Introduction

Today data is being generated at an incredible pace, and that pace is continuously accelerating. Storing those bytes is not enough, clearly we need to be able to analyze that data in order to unlock its true value. The question is, how do we keep up with these increasingly large quantities of data ? We propose technology which aims to both improve the performance and lower the cost of analytics on truly big datasets.

Traditionally, systems such as map reduce [2] and its open source alternative Hadoop [3] were advocated for big data analytics. The Hadoop ecosystem originally started out where map reduce and HDFS would typically run on the same infrastructure, and a main design goal was to allow compute to run close to the storage [3, 4]. However, today it is a best practice to deploy and manage cloud compute and storage micro services independently, since each has differing requirements. For example low cost scalable storage requires storage rich hardware, whereas an analytics micro service requires compute and memory rich hardware. It should be possible to scale the storage, compute and memory independently. Apache Spark adopts this approach, and Spark compute clusters can run against any data source implementing the Hadoop File System API, including HDFS, Amazon S3, IBM COS and OpenStack Swift, where each of these storage systems is deployed and managed independently from Spark.

2.1.1 Some Background on Object Storage

Object storage is arguably the storage of choice for truly big datasets i.e. those in the petabyte range, because of its high capacity, scalability and low cost. Many traditional approaches to data storage such as classical relational databases do not scale to this level. Moreover, at this scale a low cost per byte stored is essential to ensure a feasible budget for the whole solution. For example, in the cloud, object storage will typically have at least an order of magnitude lower cost per byte per month than a NoSQL database [5, 6]. This is a very significant advantage of object storage, although object storage does present some challenges.

Objects are similar to files in filesystems, but there are important differences. Like files, objects have both data and metadata. Unlike files, object data is written once and never modified. You cannot append or update data to an object although you can overwrite it completely. There is no rename operation for objects, so this can only be done by rewriting the object with a new name and deleting the old object. Note that in object storage, metadata is a set of key value pairs associated with an object, and there are two kinds - system and user metadata. A user can choose what user metadata attributes to store with an object, and object metadata can be updated without rewriting the object. Objects are accessed via a RESTful API with commands to PUT, GET, POST and DELETE objects. Unlike filesystems, object storage has a flat namespace, although directory hierarchies can be simulated by using forward slashes '/' in object names. Analytics over object storage works best on equally sized objects where the objects are not too small i.e. roughly 10MB or larger.

2.1.2 Minimizing Bytes Shipped from Storage to Analytics Micro Services

Today's best practices to deploy and manage cloud compute and storage micro services independently leaves us with a problem: it means that potentially huge datasets need to be shipped from the storage micro service to the analytics micro service in order to analyze data. If this data needs to be sent across the WAN then this is even more critical. Therefore, it becomes of ultimate importance to minimize the amount of data sent, since this is the key factor affecting cost and performance in this context. For example, Amazon Athena, a cloud based SQL service which queries data in S3, bills users according to the amount of data scanned in S3 [7].

We now focus on Spark as our analytics engine, and consider SQL queries over structured amd semi-structured datasets. There are currently two main approaches to limit the number of bytes sent from object storage to Spark. The first is to use specialized column based formats such as Parquet[8] and ORC[9]. These formats provide column wise compression, which clearly reduces the number of bytes to be sent. They also support column pruning, so that only columns requested by a query need

to be sent to Spark. Finally they sometimes support specialized metadata which can be used to filter columns according to query predicates. This approach is format specific, and the techniques used do not apply to other formats such as csv and json, whereas we aim for a solution which can be applied to all formats. The second approach is to use Hive style partitioning [10] to name the objects using a certain convention. In this case, information about object contents is encoded into object names. For example, if we partition according to a date column then each object will contain data records with the same date, and the date is encoded in the object name. Note that this means that the date column can be omitted from the object contents. One can partition the data according to multiple columns and this results in multiple columns encoded into object names. Spark SQL understands this convention and can filter the set of objects retrieved when query predicates apply to partitioning columns. This can significantly reduce the number of objects sent to Spark and the number of bytes shipped. See the following object listing of an example GridPocket dataset partitioned by date:

```
GPMeterStream/dt=2017-08-21/4-I-000000-080000.csv
GPMeterStream/dt=2017-08-21/4-I-080000-120000.csv
GPMeterStream/dt=2017-08-21/4-I-120000-160000.csv
GPMeterStream/dt=2017-08-21/4-I-160000-200000.csv
GPMeterStream/dt=2017-08-21/4-I-200000-240000.csv
GPMeterStream/dt=2017-08-21/4-I-800000-840000.csv
GPMeterStream/dt=2017-08-21/4-I-840000-880000.csv
GPMeterStream/dt=2017-08-21/4-I-880000-920000.csv
GPMeterStream/dt=2017-08-21/4-I-920000-960000.csv
```

Hive style partitioning is an important technique which applies to all data formats, however it has limitations. Firstly, only one hierarchy is possible, which is similar to a database which can only have one primary key. Changing the partitioning scheme requires rewriting entire dataset, since the hierarchy cannot be changed without renaming all objects, and in object storage renames require rewrite from scratch. Moreover, this technique does not support range partitioning - one can only partition according to columns with discrete types e.g. date, city, age etc. It doesn't work well for timestamps and arbitrary floating point numbers e.g. velocity, temperature etc. Finally, a deep hierarchy may result in small and non uniform object sizes, and this can reduce performance.

2.1.3 The GridPocket Use Case

Minimizing bytes shipped from storage to analytics micro services is needed in order for GridPocket to efficiently analyze their customer data. GridPocket[11] is one of our IOStack partners, a smart grid company developing energy management applications for electricity water and gas utilities, with headquarters in France. GridPocket helps utility companies manage customer data consisting of meter readings and associated information. GridPocket's desired analysis of customer data includes analyzing and comparing a consumer's energy consumption with that of other consumers in their neighborhood. This could provide healthy peer pressure and useful insight leading to energy conservation and reducing a customer's energy bill. Therefore we applied our technology to help GridPocket efficiently compare their customers' energy consumption with the average consumption of their neighbors. Our target is to handle 1 million meters reporting every 15 minutes and records are approximately 100 bytes each. Allowing for 1 order of magnitude growth in each dimension gives 1PB in 3 months. See section 2.3 for more details on the GridPocket dataset and use case. The existing techniques we outlined in the previous section are not sufficient to handle this scenario since it involves geospatial queries, where latitude and longitude do not have discrete data types, so Hive style partitioning does not apply.

2.1.4 Our Contributions

IBM helped Gridpocket implement high scale anonymized energy comparison queries with an order of magnitude lower cost and higher performance than was previously possible. Spark SQL over Object Storage is suitable in this context because object storage is highly scalable and low cost storage

which provides RESTful APIs to store and retrieve objects and their metadata. Key performance indicators (KPIs) of query performance and cost in this scenario are the number of bytes shipped from Object Storage to Spark and the number of incurred REST requests. Pluggable Spark SQL Filters extend the existing Spark SQL partitioning mechanism with an ability to dynamically filter irrelevant objects during query execution. Our approach handles any data format supported by Spark SQL (Parquet, JSON, csv etc.), and unlike pushdown compatible formats such as Parquet which require touching each object to determine its relevance, it avoids accessing irrelevant objects altogether. We developed a mechanism for developing and deploying Filters, and implemented GridPocket's filter which screens objects according to their metadata, for example geo-spatial bounding boxes which describe the area covered by an object's data points. This leads to drastically lower KPIs since there is no need to ship the entire dataset from Object Storage to Spark if you are only comparing yourself with your neighborhood. Together with GridPocket, we developed analytics notebooks to demonstrate the technology. In addition we ran performance experiments on the GridPocket dataset, achieving up to 53x speedups.

2.2 Pluggable Spark SQL Filters: Design and Architecture

2.2.1 Metadata Indexing

Our aim is to go beyond Hive style partitioning, to further reduce the amount of data sent from object storage to Spark in order to answer a query. Our approach is to utilize object storage metadata to store information about dataset columns which can then be used for *data skipping* i.e. to show that an object is not relevant to a query and therefore does not need to be accessed or sent from object storage to Spark. In order to make this efficient, we index the metadata using Elastic Search, so that during query execution, we can quickly filter out irrelevant objects from the list of objects to be accessed by the query.

Note that this technique applies to all data formats, and avoids touching irrelevant objects altogether. We support various index types, and more (such as bloom filters which can be viewed a space efficient value list), can be added in future:

min/max this index type applies to ordered columns and stores the minimal and maximal values for an object for that column.

bounding box this index type applies to geospatial data and stores a list of geopoints which describe one or more bounding boxes within which the object's data points are contained

value list this index type applies to data types representing a list of values e.g. text can be viewed as a list of words. The index stores the set of values appearing in the object

For example, let's consider a query looking for heat wave data where the temperature exceeds 40 degrees celcius. We can create a min/max index over the temp column of the GridPocket dataset, and in this case an object with a max of 35 is cleary not relevant to the query. As another example, consider a query looking for the average energy consumption of neighbours living within a 5km radius of your house. In this case we can define a bounding box index on the lat and lng columns of the GridPocket dataset. An object whose bounding box metadata does not overlap with the query is not relevant to it.

Users can choose which columns to index and the index type per column. Additional columns can be indexed later on. The main requirement of an index is that there are no false negatives - if the index says an object is not relevant then this needs to be 100 percent correct. On the other hand false positives can be tolerated - if objects are unnecessarily shipped to Spark then this is correct albeit not optimal for performance.

Our technique does not require that the index be completely up to date. They key property is that the index is only used to prove that an object is **not** relevant. If new objects are added to an existing dataset, and queries are run before they are indexed, then they may be shipped to Spark unnecessarily. If objects are deleted from the dataset then they will not be considered for the query in the first place, even if there is metadata about the object in our index. Unlike a fully inverted index in a database, which stores an index record for each data record, our technique stores a metadata record per object, where each object can store many data records (typically thousands or more). Figure 1 shows example metadata corresponding to an object in the GridPocket dataset. Here we defined a bounding box index on the lat,lon fields, a value list index on the city field, and a min/max index on the temp field.

```
{
   "name": GPMeterStream/dt=2017-08-21/part-0008.csv",
   "metadata": {
     "location": [
        {
           "lat": 47.5,
           "lon": 4.2
        },
        ...,
        {
           "lat": 43.7,
           "lon": 3.4
        }
      ],
      "city": {
        "set": [
            "Killatell",
            ....,
            "Husssignemont"
            ]
        },
        "temp": {
           "min":  7.97,
           "max": 26.77
        }
    }
}
```

Figure 1: Example metadata for an object in the GridPocket dataset

2.2.2 Pluggable Spark SQL Filters Basic Concepts

Metadata Index The IoT data stream comprises records which continually flow in. We aggregate records into objects which we upload to the Object Store. For each object, we compute some metadata information (e.g., the minimum and maximum values of a given column within the given object). This metadata is then indexed (e.g., through ElasticSearch) to permit a quick retrieval of the list of objects with metadata corresponding to some SQL query.

Data partitioner handles the flow of IoT records. The Data Partitioner uses a "model" which permits to very quickly decide for each record to what logical partition it belongs. Thus, each partition handles a subset of the IoT flow and create objects along with their metadata. The model of the data partitioner may be updated from time to time to reflect data or even query distribution changes.

Pluggable Spark SQL Filters is a user developed piece of code that contains the logic to decide, given a) a specific object b) a SQL query, whether the object is relevant to the query. This answer may contain false positives (that is an object may be declared relevant while in fact it is not) but may not contain any false negative.
The filters typically will make use of metadata that was generated while the IoT data was ingested into the Object Store and then indexed.

Adding a filter is very simple: the developer has to implement the following interface:

```
trait ExecutionFileFilter {
  /**
    *
    * @param dataFilters query predicates for actual data columns (not partitions)
    * @param file a file that that would be read from spark internal file index
    * @return true if the file needs to be scanned during execution of this query
    */
  def isRequired(dataFilters: Seq[Filter], file: FileStatus) : Boolean
}
```

after that, the filter can be plugged-in using a spark configuration as in following example:

```
sqlContext.setConf("spark.sql.execution.fileFilter", "myCustomFilterClass")
```

Figure 2 summarizes interactions between these basic components:

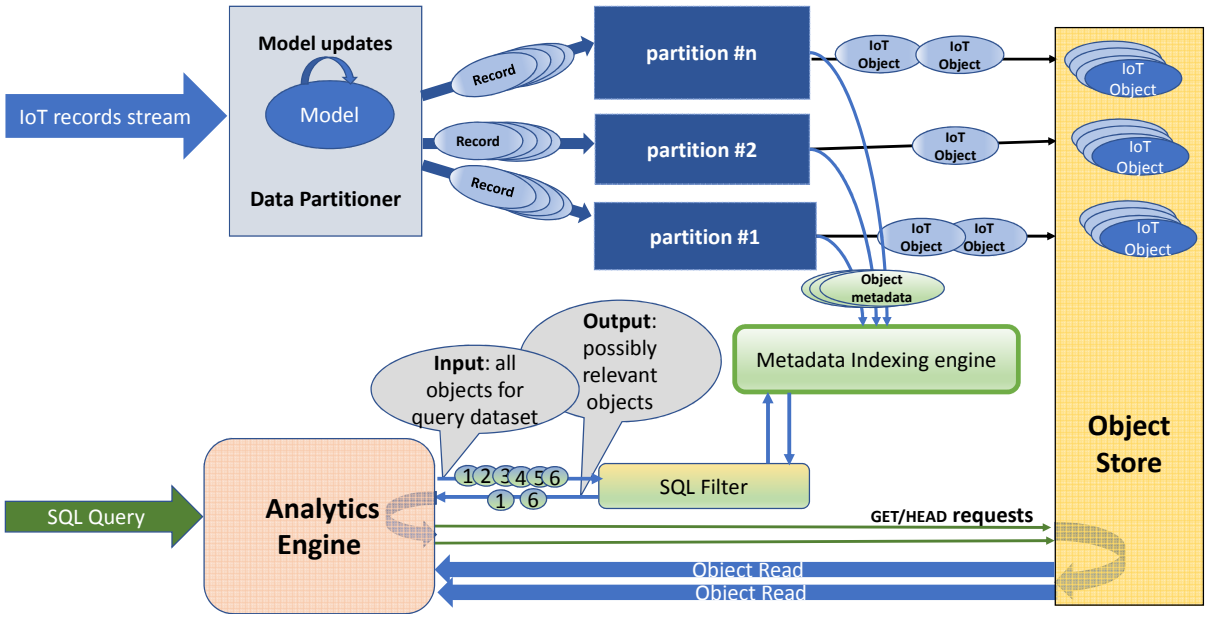

Figure 2: Basic architecture

2.2.3 Pluggable Spark SQL Filters Architecture

When running SQL queries against a data source, such as an object in csv format, for each query, Spark retrieves a file/object listing and maintains a data structure of files/objects which need to be retrieved, which we will call the *In Memory File Index*. Before reading the data from the file system/object store, a partition pruning step is carried out whereby files/objects using the Hive style naming convention which do not match the query predicates are filtered out. See the left hand side of figure 3 to see the typical Spark SQL query flow.

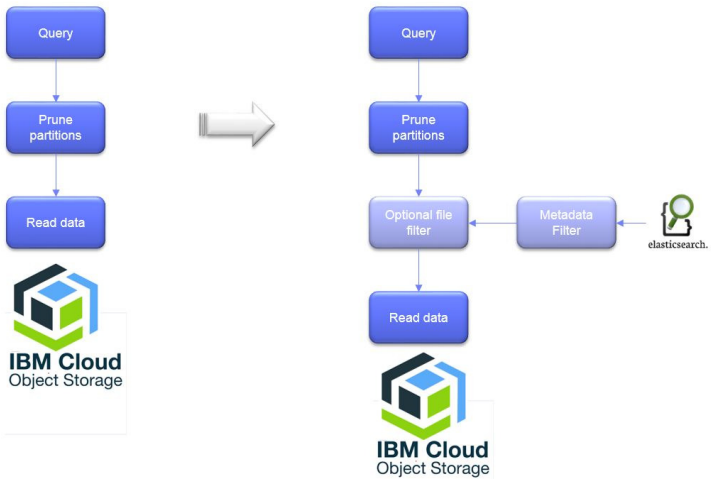

Figure 3: Spark SQL Query Execution Flow

Our technique complements Hive style partitioning by introducing an additional phase which queries Elastic Search to rule out irrelevant objects according to their metadata. This is depicted on the right hand side of figure 3.

We added this additional phase without changing core Spark by using the Catalyst experimental API. At the core of Spark SQL is the Catalyst optimizer, which is query optimization engine of Spark SQL. Catalyst was designed to be extensible and contains a general library for representing trees and applying rules to manipulate them. On top of this framework, there are libraries specific to relational query processing (e.g., expressions, logical query plans), and several sets of rules that handle different phases of query execution: analysis, logical optimization, physical planning, and code generation to compile parts of queries to Java bytecode. See figure 4 which shows the various Catalyst optimization phases.

As part of the ongoing design of Spark SQL, in order to make the Catalyst more extensible, an interface for adding external rules at the Logical Optimization phase was introduced and is currently labeled as experimental:

```
spark.sessionState.experimentalMethods.extraOptimizations
```

Using this interface we enhanced Catalyst with a new rule that we implemented to activate Pluggable File Filters during the query execution time. The rule will scan the logical plan tree and replaces the default *In Memory File Index* with our enhanced *Indexed Catalog* which extends the In Memory File Index. For example, if we execute the following code:

```
spark.sessionState.experimentalMethods.extraOptimizations = Seq(IndexedCatalogRule)
spark.read.csv("data.csv").createOrReplaceTempView("myTable")
spark.sql( "SELECT vid FROM myTable WHERE lat > 50.0")
```

the logical plan will initially be as shown on the left hand side of the diagram in figure 4. Using our Catalyst rule this is transformed to the logical plan on the right hand side of the diagram.

During query execution the Indexed Catalog will adds another filtering phase which queries Elastic Search as shown in figure 3, resulting in a finer grained selection of files to be retrieved when processing the given query.

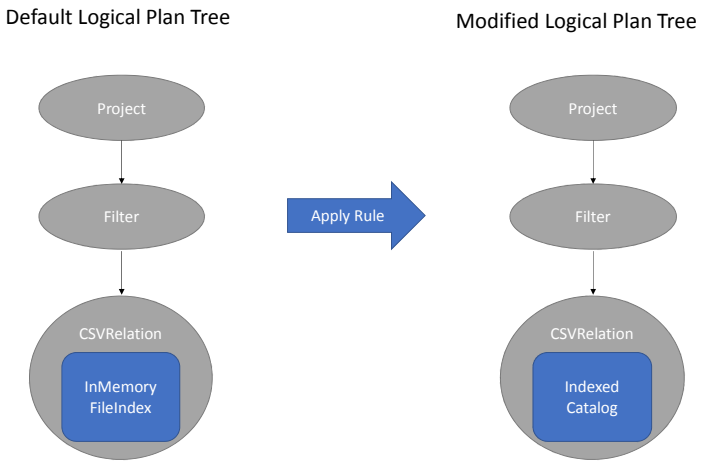

Figure 4: Catalyst Optimization Phases

Data sources Our work applies to all file based data sources implementing the Hadoop Filesystem API. This includes the Parquet, json and csv formats which can be used together with various filesystems, such as HDFS, and object stores, such as Amazon S3, IBM COS and the open source OpenStack Swift. See figure 5 which shows the Spark SQL stack.

2.2.4 Smart Data Partitioning

Metadata is a powerful tool, but if our data records are assigned to objects at random then metadata may not give useful information. How can we assign data records to partitions and objects and in a way that generates the most meaningful metadata? A scalable way to ingest IoT data to object storage is using Apache Kafka. When using Kafka, at ingest time we can assign records to Kafka partitions using a user defined partitioning function.

On the one hand, the size of IoT data records is typically up to a few hundred bytes. On the other hand, for read performance reasons, the objects that are uploaded to Object Store should typically be in the 50-128 MB range). Therefore objects aggregating IoT records can easily comprise 1 million records. In addition, objects can not be modified in Object Stores (that is to modify an object one has to completely overide it).
SQL query against the data set may specify a predicate against any of the columns defined by the schema of the data set. Each object which may contain one query relevant record is fully retrieved to process the query (typically one could not keep the metadata information which would permit to retrieve a byte range of the object).

For a given query against the data set, we designate by *accuracy* the percentage of the volume of the query relevant data over the volume of all the data that was read from the Data Store to satisfy the query.
The partitioning problem consists at separating the data set into objects and to compute and index the metadata of each of these objects so that:

1. The created objects' size is very close to a pre-specified size

2. The data points which are likely to belong to an as small as possible subset of objects, and can

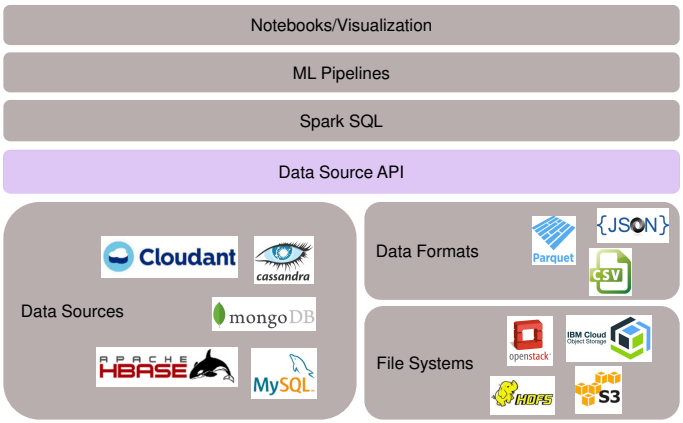

Figure 5: Spark SQL Stack

be described by suitable metadata to capture this. This goal is equivalent at maximizing the *accuracy* of the queries.

3. The partitioning algorithm is stateless (so that it can run efficiently in a distributed setting)

It should be noted that the flow of the IoT data is processed as a series of successive batches. Each batch keeps, analyzes and partitions the flow of the IoT data into object which are then uploaded to the Object Store. The longer the duration of these batches (noted T), the more effective the partitioning process the larger the quantity of data, the higher the oportunities to partition well the dataset. On the other hand, T is limited by two distinct factors: first the system requirements, for instance the quantity of memory needed to buffer the IoT data prior to uploading the resulting objects to the Object Store. Second, the user requirement that may limit the time during which fresh flowing IoT data can not be queried.

The number of the kafka partitions, and the indexing scheme are chosen so that the size of the IoT data flow that reaches each kafka partition during a period T is close to the targeted object size.
In the following we detail the design of two indexing methods that we developed and experimented.

2.2.5 Geospatial Data Partitioning

Grid Partitioning Our goal is to organize data in objects in a smart way so that the generated metadata associated to the objects will be as effective as possible. The natural phase during which we'll partition incoming data is the IoT records ingestion phase. As detailed in figure 7, we use a Kafka cluster which connects the data producers (from which the data records originate) to the data consumers (which aggregate records into objects and upload them to the Object Store). The Kafka architecture allows the user to define a partitioning function which will map each data record to the kafka partition that should handle it. Since we need to scale horizontally, the partitioner has to be stateless.

For geospatial data, we devised the grid partitioning. This partitioning basically divides the targeted map into a grid, with a precision that depends on the use case. Each data point belongs

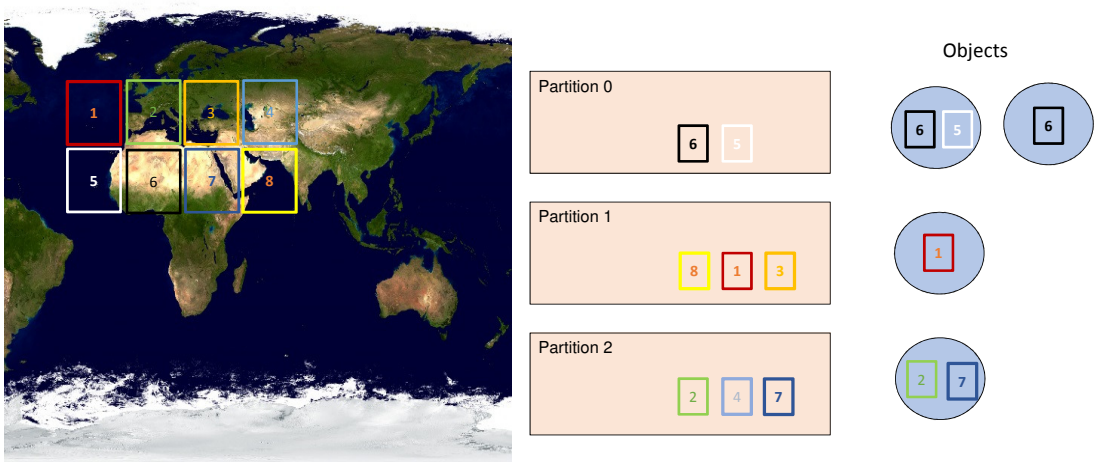

Figure 6: Grid Partitioner: cells to partitions

to a cell. Each cell is hashed to a partition. There are typically many more cells than partitions so that a partition will generally contain multiple cells. Each partition periodically generates objects. The metadata that will be created for any such object will describe a list of bounding boxes, one per participating cell. The left side of figure 6 depicts a grid of 8 cells, where each of the cells is mapped to one of the 3 possible partitions: cells 5 and 6 are mapped to partition 1, cells 1,3,8 to partition 2 and cells 2,4,7 to partition 2. The partitions will create objects (right side figure 6) where each created object will only comprise records coming from (part of) the cells mapped to the partition.

Figure 7 depicts:

- The IoT data flow from the IoT data producers down to the Object Store
- The flow of a SQL query, starting with the query and ending with the data that is subsequently read by the Analytics Engine

Each logical IoT data flow is associated with a kafka topic and will be pushed to a kafka cluster which forwards the IoT data pertaining to a given topic from possibly many producers to possibly many consumers.

The kafka producers write a given IoT data stream to its associated kafka topic. Each producer's partitioner routes the IoT records to kafka partitions. By default, the routing policy is round robin. For the geospatial indexing, the routing policy is as follows: first of all, we create a grid based on latitude and longude (with a given accuracy e.g., 1 decimal numbers), which covers the geographical area of the IoT dataset. Secondly, for a given IoT data record, we compute the latitude and longitude coordinates of the upper left corner grid cell to which the record pertains and then compute a

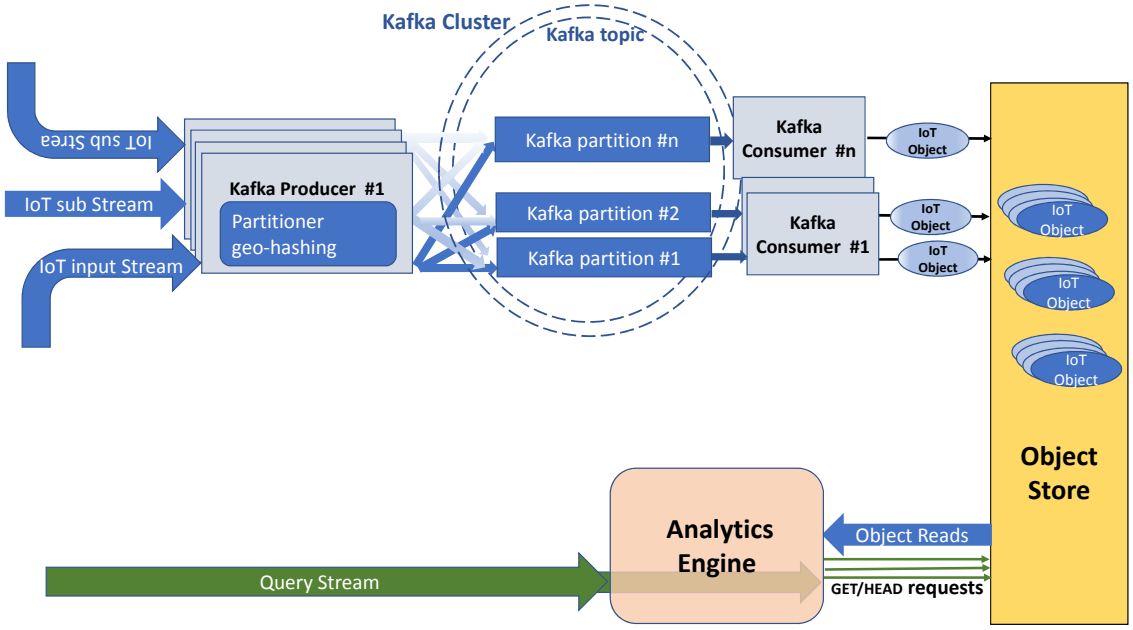

Figure 7: geo-spatial indexing architecture

hash number based on these two coordinates. We then choose the kafka partition as function of this computed hash number. The number of cells in the grid is generally much bigger than the number of kafka partition so that each kafka partition will receive the IoT data corresponding to many non geographically related cells. This permits to obtain quite even loads on the kafka partitions since each of them addresses a mix of sparce and dense cells.

The kafka consumers use an (open-source) kafka to Object Store connector to create and upload to Object Store IoT data objects.

2.2.6 Advanced Indexing Architecture

The *accuracy* obtained with geospatial indexing is limited by the following problems: first of all even when using queries which only have geographical related predicates, many non geographically related regions are mixed (as explained previously, since the grid specifies regions which are have more or less equal areas, we have to mix (non geographically related) regions to obtain an even load on the kafka partitions). Secondly, other columns than latitude and longitude are not indexed with this indexing scheme which obviously limits the *accuracy* for all queries which are not purely geographic.

We present in the following a novel indexing scheme which overcomes these drawbacks.

Before entering into design details, the following picture reproduces the architecture that was presented previously but with the following changes:

- A new “Data and Query Analysis” component is registered as a kafka listener but also listens to the query log. This component periodically computes a model that is then pushed to the

kafka producers and used by their partitioners

- The partitioner uses a model which permits to select the target partition for each IoT record in a smarter way than the previous hash based implementation

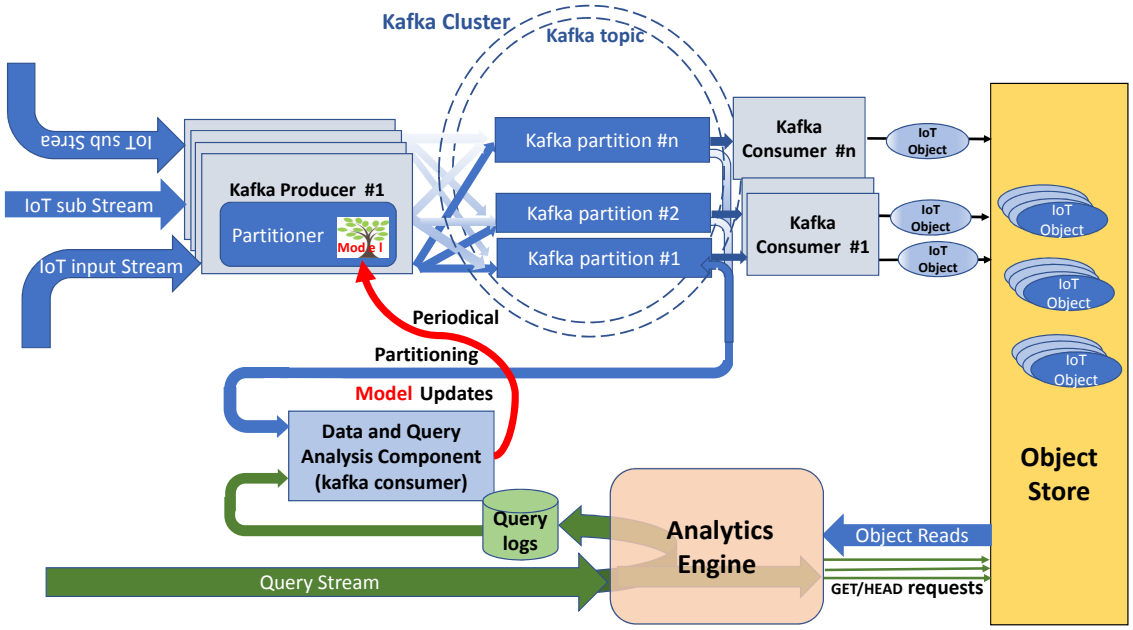

Figure 8: advanced indexing architecture

The Partitioning Model is periodically built with period T. It mainly consists of a bucket k-d-tree [12] which precisely defines a way of how to partition the space of the possible IoT records. This tree is build offline by analyzing D: the IoT data corresponding to the past T period. For simplicity, we assume that there exists an integer: h such that $D/2^h$ corresponds to an object size which is both acceptable and best in terms of performance.

The first targeted property (Objects having a uniform pre-specified size) is ensured by the (recursive) construction process of the k-d-tree over data points in D: at every stage we separate each of the temporary buckets (hyperrectangle) into 2 sub-buckets of equal data size. Thus, clearly after h recursive steps, we obtain 2^h final buckets of equal data size. The new model build at end of a T period will reflect possible data or query distribution changes and may replace the model currently in use by kafka partitioners if it presents any notable change. The third targeted property (statelessness) comes from the fact that a given k-d-tree (the partitioning model) can be distributed to the kafka partitioners which can each use this data structure independently to route the IoT data stream to the various kafka partitions. For a given data set, a k-d-tree represents some partitioning of the data set and may be built in many different manners.

Two main parameters may be varied when building a k-d-tree:

1. the choice of the dimension along which a hyperrectangle (or bucket) will be (further) partitioned

2. given such a partitioning dimension, the choice of the split value which will define the hyper plane that will separate the bucket rectangle into two sub buckets.

Canonical *k*-d-trees [12] are built by choosing the partitioning dimensions in a round-robin fashion and for each chosen dimension by using the median of the data population within a hyperrectangle as the split value. (the median value is chosen in order to generate equally sized objects).
Given a data set and given a bucket that is to be (further) partitioned, we aim at choosing as partitioning dimension, the dimension that will minimize the expected cost of future SQL requests. Given a finite query workload, and given a bucket, we could then evaluate the result of splitting on each dimension against this workload, by applying the full set of queries on the resulting partitioned data and measuring the resulting key performance indicators (KPIs). We can then choose the dimension which gives the best KPIs. This is the essence of our method.

In the following, we present the following example heuristics which permit to choose the partitioning dimension more efficiently:

1. The first way aims to approximate such an analysis by capturing the notion of “query workload shape”

2. The second way generalized the first way by capturing the notion of “query cut”

3. The third one further refines our analysis by defining a notion of “metadata compactness” which permits to decide between dimensions with similar query cuts

Query shape method This method fits well cases where the query patterns are very clearly defined. In this case we’ll choose the dimension which leads to hyper-rectangles buckets having most similar shapes to query shapes.
For example, let’s assume 4 dimensions: latitude, longitude, temperature and energy consumption. The Data and Queries Analysis component discovers that the typical geometry of the queries targets geospatial squares of length between 1 and 10 km with an additional constraint on the temperature column which is queried with a accuracy of 3 ° C. As a consequence, and as function of the data density, the query shape method will aim to separate the space into hyper-rectangles with a geometry that fits the discovered typical query geometry).

Query driven method This method extends the query shape method: we project the historical queries on the hyper-rectangle corresponding to the bucket to be partitioned and compute the *cutting value* for each dimension where the *cutting value* of a given dimension is defined as the number of historical queries which cut the hyper-rectangle along that dimension. We then approximate the dimension that yields the best KPI improvement by choosing the dimension with the highest *cutting value*.
This method can be refined by taking into account:

- the quality of a query cut, that is the proximity of the cut value with the median value relative to the size of the hyper-rectangle along the chosen dimension.

- the quality of the associated query. It may be approximated by taking into account the following factors:

  - to what degree is this cut relevant to the hyper-rectangle? In case this cut was "and-ed" with other predicates, what is the intersection between the hyper-rectangle defined by these predicates and the hyper-rectangle that we are dealing with?
  - the relevancy of the queried data: was the query issued against fresh data or only against old data (relative to the query time)? Is this query recent?

For example, let's assume a 3-dimensional space: x, y, and z and a given bucket. Let's further assume that within this bucket, if we would find out that the y dimension has the highest cutting value, then we'll choose it as the next separating one.

Metadata compactness method When the lack of query history prevents using one of the two first methods, or when the query driven method yields two or more dimensions with similar cutting values, we apply the metadata compactness method to decide between them.
This method consists at choosing as partitioning dimension the one which leads the resulting hyper-rectangles to have the most "compact" metadata representation, that is the one leading to hyper-rectangles (representing the metadata) with the smallest aggregated volume. This method fits well the cases were data is arranged in distinct "clumps", then much is to be gained by using a compact metadata representation where queries over areas having no data points will not match any objects. It is also well adapted to datasets with correlated columns.
Taking again the example of 3 dimensions: x, y and z, where the z value is strongly correlated with the value of x, but the pairs *(x,y)* and *(y,z)* have a zero correlation, if we'd found out that x and y dimensions have smallest but similar cutting values, then we'll choose the x dimension since it yields the smallest metadata.

2.3 Experimental Evaluation

We evaluated a prototype of Pluggable Spark SQL Filters via real experiments where both the fixed grid and the *k*-d-tree partitioners were used.

2.3.1 Data Generation

Gridpocket collaborated with IBM to develop an open source data generator, called metergen, [13] to closely simulate their customer workload. It should be noted that a lot of efforts were poured by the Gridpocket team to improve metergen during 2017 so that a) the spatial distribution of the meters is similar to real meters distribution b) the meters output data records with correlations between features similar to the correlations that were extracted from real data.
It generates a NoSQL dataset comprising of one large denormalized table with meter reading information according to the schema in table 2.3a. We use SQL queries to compare customer energy consumption with the average consumption of their neighbours.

The Fields for Gridpocket records			
Field Name	Unit	Data type	Usage
date	seconds	integer	timestamp of the meter reading
index	kilowatt hour	float	the meter electrical reading
sumHC	kilowatt hour	float	total energy consumed since midnight during off-hours
sumHP	kilowatt hour	float	total energy consumed since midnight during peak-hours
type	Enumerated	string	how the house is warmed: electricity or gaz
vid	number	integer	the meter id
size	square meter	integer	the apartment size
temp	celcius	float	temperature at meter location for specified date
city	not relevant	string	Name of the city of the meter location
region	number in [1,95]	integer	Number of the French region of the meter location
latitude	degrees	float	latitude of the meter location
longitude	degrees	float	longitude of the meter location

Table 2.3a: Gridpocket generated records

2.3.2 Experimental Setting

Our evaluation addressed the following objectives: i) demonstrate the value of our data skipping technique, specifically using the Gridpocket use-case and data ii) compare the obtained results for

the two data partitioners.
We did not had time to take into account the query history. This is left to future works.

Our environment comprises 2 Spark nodes running Spark v2.1.0. Each node has 32 cores and 128 GB of memory. We use Elastic Search version 5.3.0 and we use IBM COS service [14] as our Object Store.

Grid partitioner When using this partitioner, we divided the targetted map into a grid (with a given precision). Each data point belongs to a cell, and each cell is mapped thru the hash function to a Kafka partition (to which multiple cells will typically be mapped to).
The size of the cells in the grid is given by the "precision" of the grid as follows:

- Precision 1 => each square in the grid is a 11km x 11km square
- Precision 0 => coarser than precision 1, each square in the grid is a 110km x 110km square

Note that a Precision 2 does not make a lot of sense, since it would define cells of about 1 km^2 which would yield an average of about 2 meter per cell for our data set(see details in the following).

Dataset We used the Gridpocket data generator which yields meter density in accordance with its real customer data and which also yields correlations between data feature also in accordance with real data. Following are the main characteristics of the used generated data set:

- 1 million meters, where each meter reports every 15 minutes.
- The data records are of approximately 115 bytes long on the average
- We generated 1TB of Gridpocket data generator: one million meters for 3 months over the French metropolitan territory
- The resulting files are in csv format.
- We partitioned the data using the Grid hash Partitioner with precision 1 and 0.
- We partitioned the data using either 32, or 64 or 128 partitions.

Experiment details

- We randomly draw a meter from a meter list (which contains a "representative" meter for each non empty square of the grid).
- We calculate a square geobox which center is the drawn meter (the query dimensions are given in km and converted to lng/lat degrees)
- We submit a geobox query around the chosen meter which retrieves the data pertaining to the meters in that square and then does some computation.
- We then calculate the number of bytes that were retrieved for processing the query , both with and without pluggable filters .

2.3.3 Grid partitioner results

Figure 9 depicts the obtained results when using the fixed grid partitioner. It shows that we achieve our main goal: when using pluggable filters, we achieve between x17 and x2.5 reduction in the number of shipped bytes for a grid with precision 1. The degree of reduction depends on the number of partitions and the query bounding box size.
For larger query bounding boxes more data has to be shipped since more rows are needed to calculate the query result. We show that for precision 1, when decreasing the query bounding box size we succeed to similarly reduce the number of bytes shipped.
This graph also demonstrates that a larger number of partitions allows each partition to collect fewer grid cells overall and thus each partition has more "uniform" data. This explains why a larger number of partitions induces a larger reduction in bytes shipped.

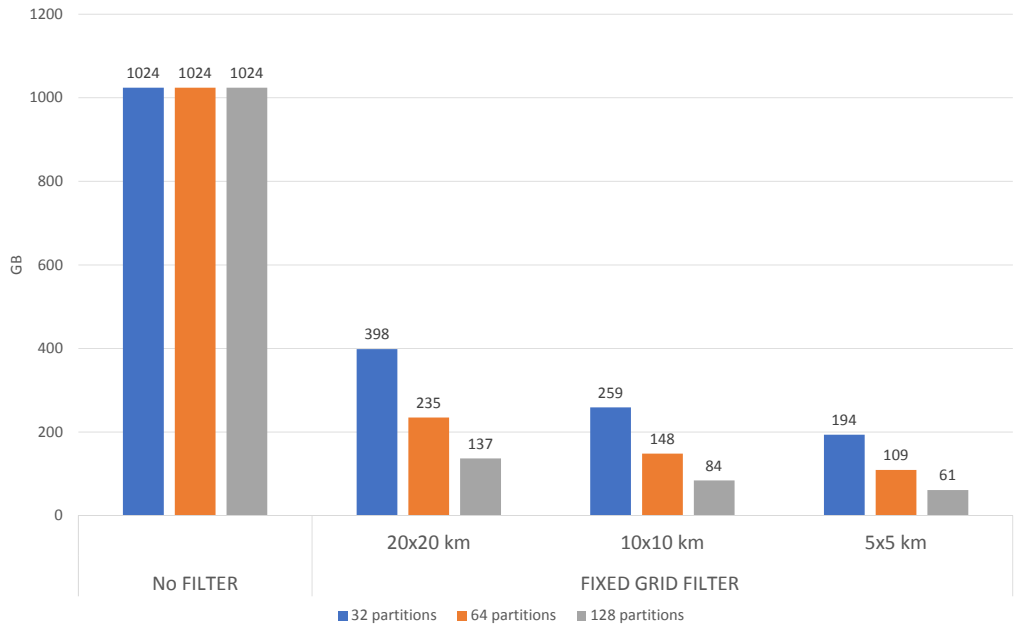

Figure 9: Shipped bytes with fixed grid - precision 1)

In figure 10, we used 128 partitions and compare between precision 1 (fine grained grid cells) and precision 0 (coarse grained grid cells).
Unlike with precision 1, with precision 0 we see a fairly constant reduction in bytes transferred across query bounding box sizes. This is because in all cases the size of the query bounding box is significantly smaller than the size of a grid cell. For the 20x20 query case, since a large area is retrieved, we obtain more benefit by using a coarse precision. This is because when using fine precision, we are disadvantaged by the fact that neighbouring grid cells are typically mapped to different partitions and therefore the data is spread across more objects. The conclusion here is that with grid partitioning, one needs to fit the precision used to the characteristics of the workload. If the workload changes dynamically this can be problematic. This was one of the incentives to devise and explore the *k*-d-tree partitioner.

Figure 11, depicts the final phase of a notebook created by the Gridpocket team. For a given meter and for a given period, it outputs the electrical consumptions of the meter versus the mean consumption of the neighbors. Finally (right side of the picture), it gives a global comparison for the specified period.

2.3.4 Advanced partitioner results

Figure 12 depicts the initial results obtained using the *k*-d-tree partitioner. For example, for a 5x5km^2 query using 128 partitions we observe a X53 improvement in terms of shipped bytes compared to when not using filters. All the partition sizes and bounding box query sizes we tested yield an X18 improvement or more. For all numbers of partitions and for all query bounding box sizes we see significant improvements over the grid partitioner (between 3 and 7 times improvement compared to fixed grid with precision 1). Note that the *k*-d-tree can handle multiple dimensions and here

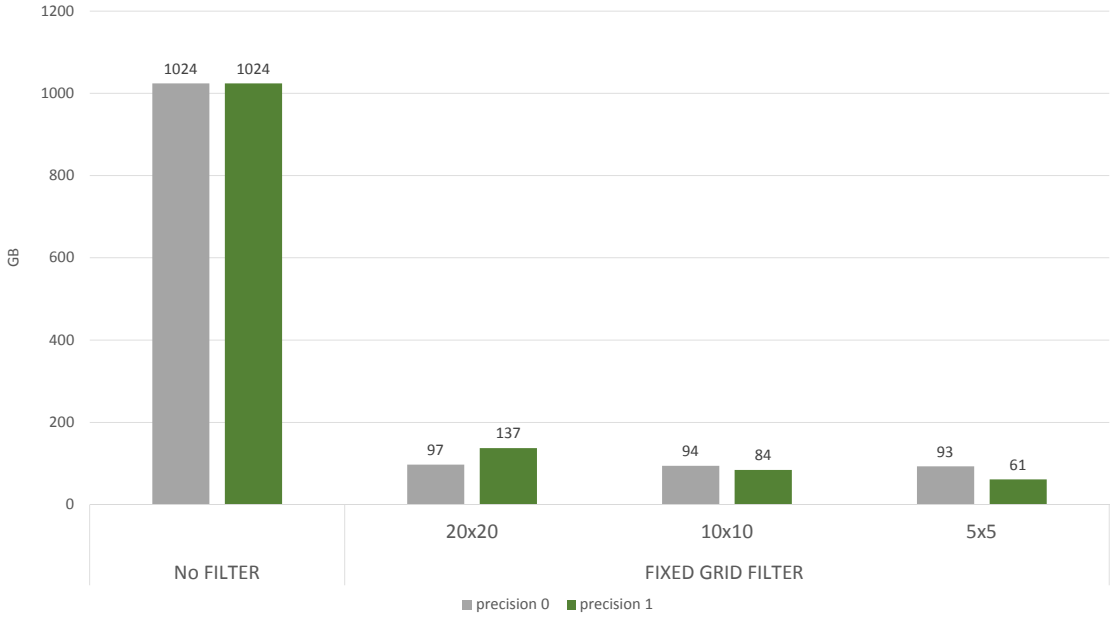

Figure 10: Shipped bytes with fixed grid - 128 partitions

we limit it to only 2 dimensions. We plan to continue our research in future by applying the *k*-d-tree partitioner to more than 2 dimensions.

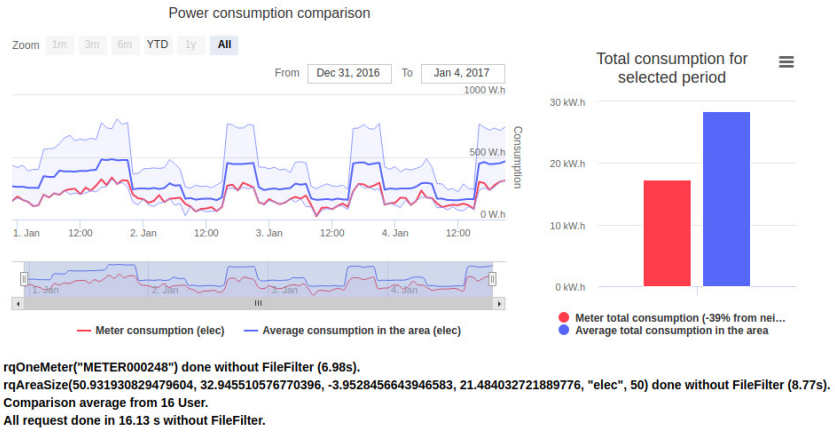

Figure 11: Power Consumption Comparison

2.4 Related Work

2.4.1 Pluggable Spark Filters

For Apache Spark SQL users, the most practical way to tackle data set ingestion is Hive style partitioning. Yet this method has some drawbacks: first of all, only a single hierarchy (similar to a database primary key) is possible, and no secondary hierarchy is possible. In addition the hierarchy cannot be changed without renaming all the files which is not feasible for common object storage systems since there is typically no rename operation. One other restriction is that partitioning can only be done against discrete types attributes such as gender or age. In addition, Hive style partitioning doesn't work well for types represented as arbitrary floating point numbers such as time-stamps or temperatures. Also a deep hierarchy will in general result into creating many small object which is detrimental to read performance from the object store.

It should be noted that using the Parquet format can sometimes reduce the amount of data shipped (See [15]). Indeed the minimum and maximum values for a column can be stored as metadata in the footer of a Parquet file which permits to find out that the file is not relevant to a query. However the footer of the parquet file still needs to be retrieved. Another feature of parquet permit to avoid this: a _common_metadata file can be maintained to filter out files. However, by default this feature is not enabled, and often cannot be used. As a matter of fact maintaining such a file is uneasy since it needs to be appended each time a new file is added to a folder.
Note that object storage systems do not support append. However our extension is useful for all formats and our implementation is independent of the file format chosen.

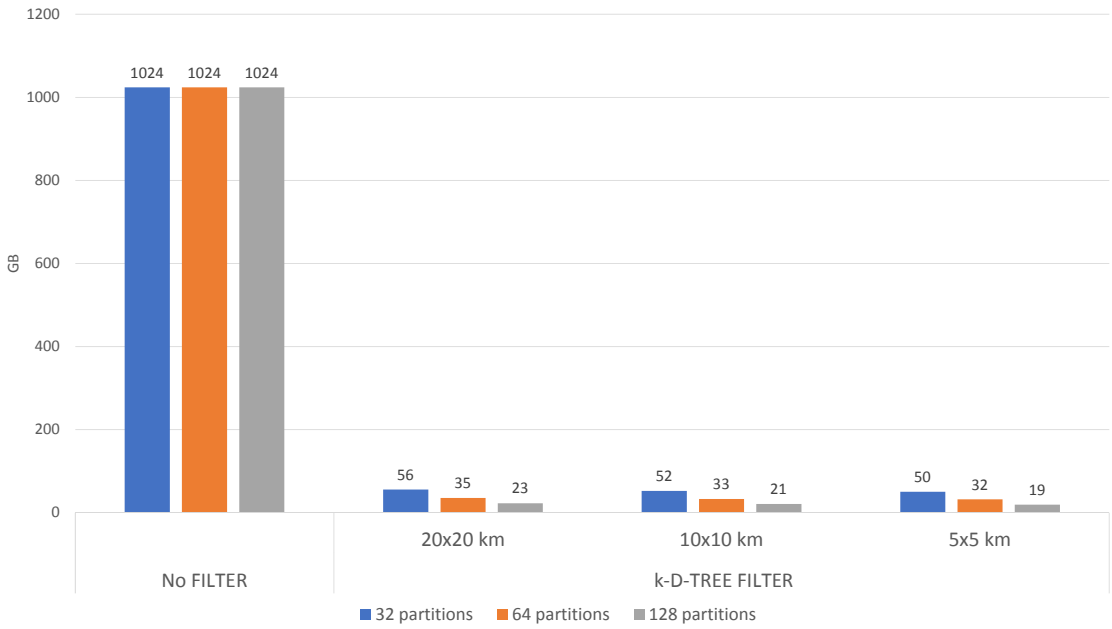

Figure 12: Shipped bytes with the advanced partitioner

2.4.2 Data Skipping

Data Skipping techniques have been used for optimizing traditional data bases [16, 17] and and detects ranges of column values that are not needed to satisfy a query and thus permits to avoid retrieving their associated pages from disk.

Regarding the use of data skipping techniques in Apache Spark, in October 2017, we concurrently presented our Pluggable Spark Filters work at the Spark Summit in Dublin [18], while the DataBricks team unveiled DataBricks Delta in the keynote [19] there and demonstrated Data Skipping indexes [20] which are part of DataBricks Delta. This shows that this is an extremely important optimization in practice. Our work handles geospatial data skipping and is extensible to additional indexes and data types, whereas the DataBricks implementation handles only min/max indexes. In addition, we explore smart data partitioning which is not addressed by the current DataBricks implementation.

2.4.3 Smart Data Partitioning

We independently developed the notion of *k*-d tree partitioning which uses dataset medians as cutting points for partitioning and uses query history in order to choose the partitioning dimensions. Subsequently a paper was published in September 2017 [21] which uses the same approach and similarly applies it to Apache Spark for the purpose of data skipping. The work in this paper goes beyond our work by providing an adaptive approach to repartition datasets on the fly according to a cost model. Our work focuses on object storage whereas this paper focuses on HDFS where appending data to the end of files is possible and so modifications would be required in order for it to work well with object storage. They focus on ordered data types and range partitioning only, whereas we also apply our work to geospatial and other data types and indexes including Bloom filters. A recent companion paper covers how their technique can be applied to join processing [22]. This is a cutting

edge research area which is also promising in terms of its applicability to analytics on real world big datasets.

2.4.4 Multi-dimensional indexing

Techniques based on Space-filling curves [23] such as Z-order curves (or Morton curves) [24] map a multi-dimensional space into a single indexing dimension represented by an encoding string (the metadata). These techniques can handle varying data density by issuing a geohash code of varying length. Possible usage of these techniques is to convert the query box into a one dimensional code range and to use it against the indexed data.
However, the main drawback of these techniques in the fact that the chosen space filling curve and the dataset points completely determines the partitioning. In addition the query history is not taken into account. Also, one cannot dynamically change the way partitioning is done, and Space-filling curves treat all dimensions in a symmetric way so no way to "prefer" one dimension over the other (e.g. to achieve metadata compactness and to fit the representation to the query distribution).

2.5 Conclusions and Further Work

During the two first years of the IOStack project, in WP4 we tackled the problem of reducing the amount of data transferred from the object storage micro service to the analytics micro service using active storage methods. During the third and last year of the IOStack project, we designed, implemented and proved the efficiency of a complementary and novel method to solve the same problem.

We demonstrated the efficiency of our method using our grid partitioner together with data skipping techniques implemented as Pluggable Filters. For a 1TB Gridpocket dataset, we observed a reduction in up to x17 in the number of bytes transferred (that is the number of bytes transferred was reduced from 1TB to less than 60 GB).

After analyzing how to further improve our data skipping technique, we devised a novel partitioning method which is based on the *k*-d-tree and also takes into account query history. We then presented initial experimental results for the *k*-d-tree partitioner. They show that usage of the *k*-d-tree based data partitioner, without taking into account query history, significantly improves the results obtained by our initial grid partitioner.

2.5.1 Further Work

All these research items that were driven under the IOStack project have opened wide and very interesting horizons. In particular the following research questions are of interest for further research:

1. To further validate the benefits of the *k*-d-tree partitioner experimentally
2. To explore the use of the *k*-d-tree partitioner with more than 2 dimensions
3. To perform a theoretical comparison analysis of the various partitioners
4. To extend our research to dataset columns for which min-max metadata is not suitable.
5. To research the improvement of the *k*-d-tree partitioner when we take into account query history

2.5.2 Gridpocket impact

Pluggable Spark SQL filters have demonstrated the very high potential of data skipping techniques applied to Apache Spark and Object Stores. These techniques provide a significant reduction in the amount of data read from object storage and transferred to Apache Spark. This was demonstrated for the GridPocket use case using various tests and benchmarks.

3 Stocator: High Performance Object Storage Connector for Spark

3.1 Introduction

Data is the natural resource of the 21st century. It is being produced at dizzying rates, e.g., for genomics by sequencers, for healthcare through a variety of imaging modalities, and for Internet of Things (IoT) by multitudes of sensors. This data increasingly resides in cloud object stores, such as AWS S3[5], Azure Blob storage[25], and IBM Cloud Object Storage[26], which are highly scalable distributed cloud storage systems that offer high capacity, cost effective storage. But it is not enough just to store data; we also need to derive value from it, in particular, through analytics engines such as Apache Hadoop[3] and Apache Spark[27]. However, these highly distributed analytics engines were originally designed 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 deliverable we present Stocator[28], 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; our work can be extended to work with the other parts of the Hadoop ecosystem.

Current connectors to object stores for Spark, e.g., S3a[29] and the Hadoop Swift Connector[30] are notorious for their poor performance[31] for write workloads and sometimes leaving behind temporary objects that do not get deleted. The poor performance of these connectors follows from their assumption of file system semantics, a natural assumption given that their model of operation is based on the way that Hadoop interacts with its original storage system, HDFS[32]. In particular, Spark and Hadoop achieve fault tolerance and enable speculative execution by creating temporary files and then renaming these files. This paradigm avoids interference between threads doing the same work and thus writing output with the same name. Notice, however, that rename is not a native object store operation; not only is it not atomic, but it must be implemented using a costly copy operation, followed by a delete.

Current connectors can also lead to failures and incorrect executions because the list operation on containers/buckets is eventually consistent. EMRFS[33] from Amazon and S3mper[34] from Netflix overcome eventual consistency by storing file metadata in DynamoDB[6], an additional strongly consistent storage system separate from the object store. A similar feature called S3Guard[35] that also requires an additional strongly consistent storage system 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.

Others have tried to improve the performance of object store connectors, e.g., the DirectOutputCommitter[36] for S3a introduced by Databricks, but have failed to preserve the fault tolerance and speculation properties of the temporary file/rename paradigm. There are also recommendations in the Hadoop open source community to abandon speculation and employ an optimization[37] that renames files to their final names when tasks complete (commit) instead of waiting for the completion of the entire job. However, incorrect executions, though rare, can still occur even with speculation turned off due to the eventually consistent list operations employed at task commit to determine which objects to rename.

In this deliverable we present a high performance object store connector for Apache Spark that takes full advantage of object store semantics, enables speculative execution and also deals correctly with eventual consistency. Our connector eliminates the rename paradigm by writing each output object to its final name. The name includes both the part number and the attempt number, so that multiple attempts to write the same part due to speculation or fault tolerance use different object names. Avoiding rename also removes the necessity to execute list operations to determine which objects to rename at task and job commit, so that a Spark job writes all of the parts constituting its output dataset correctly despite eventual consistency. This reduces the issue of eventual consistency to ensuring that a subsequent job correctly determines the constituent parts when it reads the output

of previous jobs. Accordingly we extend an already existing success indicator object written at the end of a Spark job to include a manifest to indicate the part names that actually compose the final output. A subsequent job reads the indicator object to determine which objects are part of the dataset. Overall, our approach increases performance by greatly decreasing the number of operations on the object store and ensures correctness despite eventual consistency by greatly decreasing complexity.

Our connector also takes advantage of HTTP Chunked Transfer Encoding to stream the data being written to the object store as it is produced, thereby avoiding the need to write objects to local storage prior to being written to the object store.

We have implemented our connector for the OpenStack Swift API[38] and shared it in open source[39]. 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 store service typically includes charges based on the number of operations executed, this reduction in the number of operations lowers the costs for clients in addition to reducing the load on client software. It also reduces costs and load for the object store 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 Spark that leverages object storage semantics, avoiding costly copy operations and providing 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 IBM Analytics for Apache Spark, an IBM Cloud service, and has enabled the SETI project to perform computationally intensive Spark workloads on multi-terabyte binary signal files[40].

The remainder of this section is structured as follows. In 3.2 we present background on object storage and Apache Spark as well as the motivation for our work. In 3.4 we describe how Stocator works. In 3.5 we present the methodology for our performance evaluation, including our experimental set up and a description of our workloads. In 3.6 we present a detailed evaluation of Stocator, comparing its performance with existing Hadoop object storage connectors, from the point of view of runtime, number of operations and resource utilization. 3.7 discusses related work and finally in 3.8 we conclude.

3.2 Background

We provide background material necessary for understanding the remainder of the section describing stocator. First, we describe object storage and then the background on Spark[27] and its implementation that have implications on the way that it uses object storage. Finally, we motivate the need for Stocator.

3.2.1 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. Object storage is typically accessed through RESTful HTTP, which is a good fit for cloud applications. This simple object semantics enables the implementation of highly scalable, distributed and durable object storage that can provide very large storage capacities at low cost. Object storage 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 AWS S3[5], Azure Blob storage[25], OpenStack Swift[41] and IBM Cloud Object Storage[26].

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. In particular, when listing the contents of a container a separator character, e.g., "/" or "*", between levels of naming can be specified as well as a prefix string, so that only the names for objects in the container starting with the prefix will be included. This is different than file systems where there is both hierarchy in the implementation as well as in naming, i.e., a directory is a special file that can contain other files and directories.

Common operations on object storage include:

1. PUT Object, which creates an object, with the name, data and metadata provided with the operation,
2. GET object, which returns the data and metadata of the object,
3. HEAD Object, which returns just the metadata of the object,
4. GET Container, which lists the objects in a container,
5. HEAD Container, which returns the metadata of a container, and
6. DELETE Object, which deletes an object.

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.

In order to enable a highly distributed implementation the consistency semantics for object storage often include some degree of eventual consistency[42]. 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 aspects of eventual consistency. For example, AWS[43] guarantees read after write consistency for its S3 object storage system, i.e., that a newly created object will be instantly visible. Note that this does not necessarily include read after update, i.e., that a new value for an existing object name will be instantly visible, or read after delete, i.e., that a delete will make an object instantly invisible. Another aspect of eventual consistency 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.

3.2.2 Spark

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

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. In that case the driver may start a new execution of the same task. The execution of a task may also be slow and in some cases the driver cannot tell whether the execution has failed or is just slow. Spark has an important feature to deal with these cases 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 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.

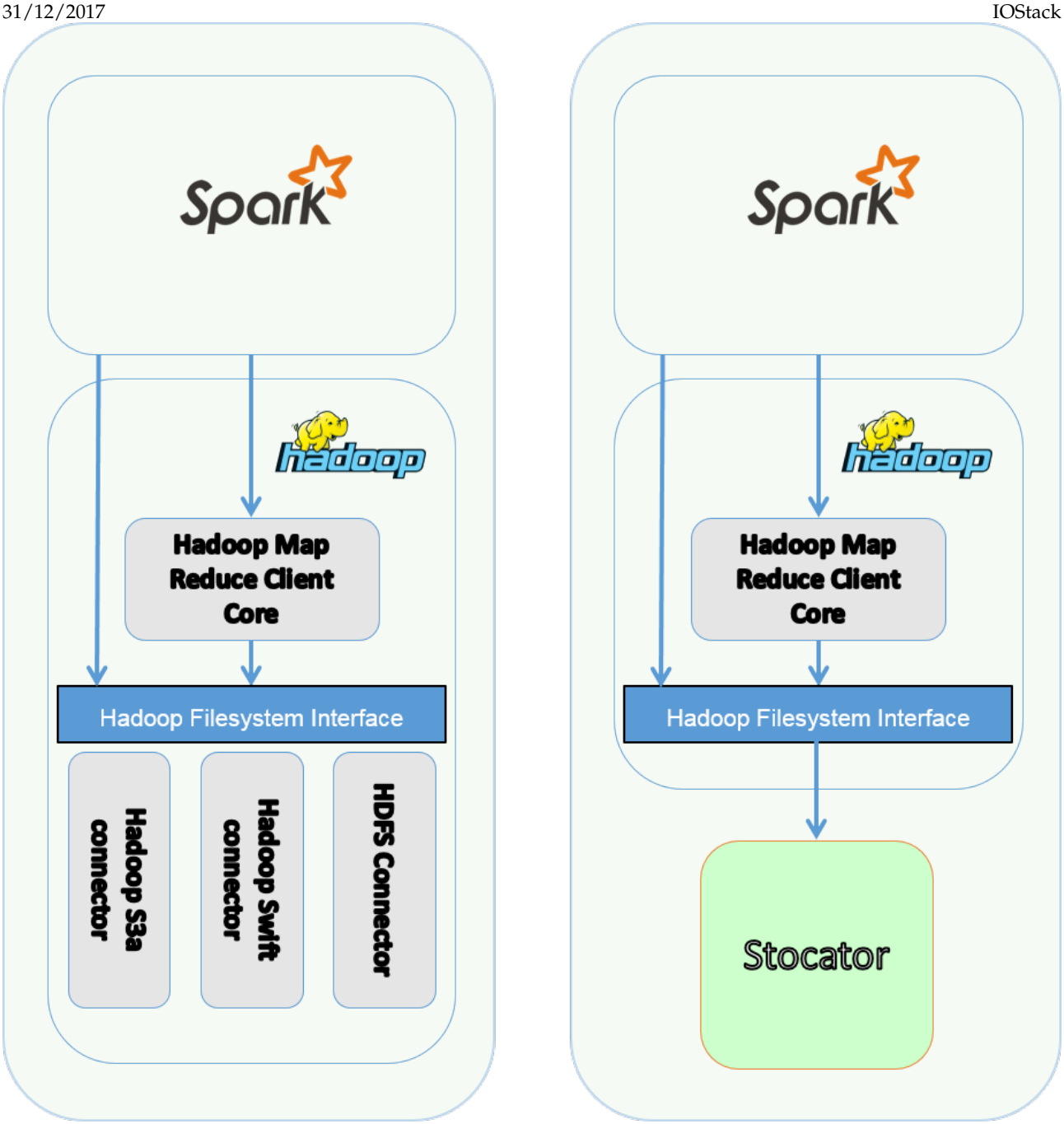

Figure 13: Hadoop Storage Connectors

Spark and its underlying storage Spark interacts with its storage system through Hadoop[3], primarily through a component called the Hadoop Map Reduce Client Core (HMRCC) as shown in the diagram on the left side in 13.

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 S3 object store API and a Swift connector for the OpenStack Swift object store 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,

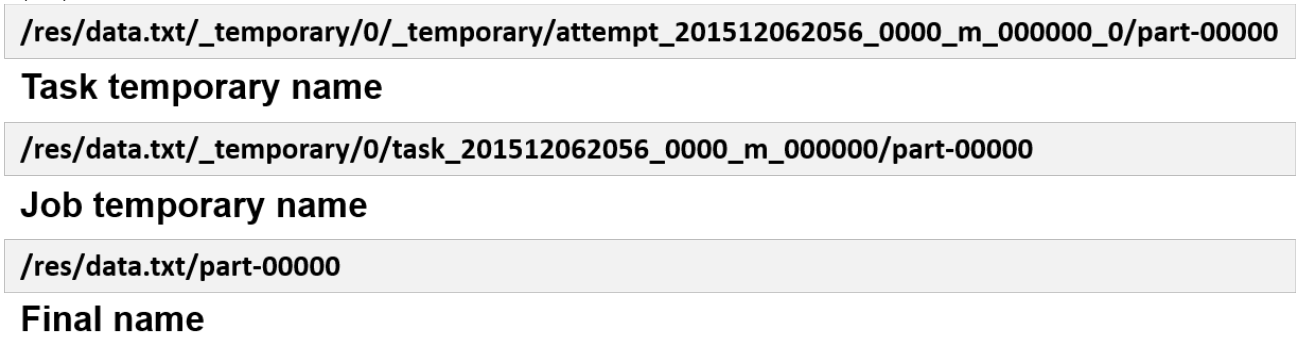

Figure 14: Sequence of names for part 0 of output from task temporary name to job temporary name to final name.

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. 14 shows the names for a task's output.

This two stage strategy of task commit and then job commit was chosen to avoid the case where incomplete results might be interpreted as complete results. However, 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. Accordingly, there is now a new version of the file output committer algorithm (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 directories 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[34, 33, 44].

3.3 Motivation

To motivate the need for Stocator we show 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 15. Spark and Hadoop were originally designed to work with a file system. Accordingly, 3.3a shows the series of file system operations that Spark carries out for the sample program.

1. The Spark driver and executor recursively create the directories for the task temporary, job temporary and final output (steps 1–2).

2. The task outputs the task temporary file (step 3).

3. At task commit the executor lists the task temporary directory, and renames the file it finds to its job temporary name (steps 4-5).

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

Figure 15: A Spark program that executes a single task that produces a single output object.

	Operation	File
1	Spark Driver: make directories recursively	hdfs://res/data.txt/_temporary/0
2	Spark Executor: make directories recursively	hdfs://res/data.txt/_temporary/0/_temporary/attempt_201702221313_0000_m_000001_1
3	Spark Executor: write task temporary object	hdfs://res/data.txt/_temporary/0/_temporary/attempt_201702221313_0000_m_000001_1/part-00001
4	Spark Executor: list directory	hdfs://res/data.txt/_temporary/0/_temporary/attempt_201702221313_0000_m_000001_1
5	Spark Executor: rename task temporary object to job temporary object	hdfs://res/data.txt/_temporary/0/task_201702221313_0000_m_000001/part-00001
6	Spark Driver: list job temporary directories recursively	hdfs://res/data.txt/_temporary/0/task_201702221313_0000_m_000001
7	Spark Driver: rename job temporary object to final name	hdfs://res/data.txt/part-00001
8	Spark Driver: write _SUCCESS object	hdfs://res/data.txt/_SUCCESS

Table 3.3a: The file system operations executed on behalf of a Spark program that executes a single task to produces a single output object.

4. At job commit the driver recursively lists the job temporary directories and renames the file it finds to its final names (steps 6-7).

5. 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. 3.3b shows the breakdown according to operation type.

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 15 Stocator executes just 8 REST operations: 3 PUT 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

Table 3.3b: Breakdown of REST operations by type for the Spark program that creates an output consisting of a single object.

4 HEAD object and 1 GET container.

3.4 Stocator algorithm

The right side of 13 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.

3.4.1 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 3.2 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. In particular, HMRCC asks to write a temporary file/object in a temporary directory of the form `<output-dataset-name>/_temporary/0/_temporary/attempt_<job-timestamp>_0000_m_000000_<attempt-number>/part-<part-number>`, where `<job-timestamp>` is the timestamp of the Spark job, `<attempt-number>` is the number of attempt, and `<part-number>` is the number of the part. Stocator notices this pattern and in place of the temporary object in the temporary directory, it writes an object with the name `<output-dataset-name>/part-<part-number>_attempt_<job-timestamp>_0000_m_000000_<attempt-number>`.

Finally, when all tasks have completed successfully, Spark writes a _SUCCESS object through HMRCC. 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; thus, 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.

3.4.2 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 one that has the most data. 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.

The second option is more complex to implement. Here at the time the _SUCCESS object is written, Stocator includes in it a list of all the successful execution attempts completed by the Spark 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 outlined above that was 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. The second option also does not need the fail-stop assumption. However, due to its simplicity we have implemented the first option in our Stocator prototype.

3.4.3 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 5 MB.

	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

Table 3.4a: Possible operations performed by the Spark application showed in 16

3.4.4 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 object multiple times, it should

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

Figure 16: A Spark program where three tasks each write an object part.

return the same result. Stocator uses a small cache to reduce these calls.

3.4.5 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.

16 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. 3.4a shows the 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.

3.5 Methodology

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

3.5.1 Experimental Platform

Infrastructure 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 24 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.

Workload	Input Size	Output Size
Read-Only	46.5 GB	0 MB
Read-Only 10x	465.6 GB	0 MB
Teragen	0 GB	46.5 GB
Copy	46.5 GB	46.5 GB
Wordcount	46.5 GB	1.28 MB
Terasort	46.6 GB	46.5 GB
TPC-DS	13.8 GB	0 MB

Table 3.5a: Workloads' details.

Our IBM Cloud Object Storage (COS) [45] 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 with round-robin so that connections opened by a Spark server with the object storage alternate between Accessers. Given that each of the three Spark servers has a 10 Gbps NIC, 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 and/or 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.

3.5.2 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 second group, with the suffix *Commit V2*, uses the version 2 of Hadoop FileOutputCommitter that reduces the number of copy operations towards the object storage (as described in Section 3.2). 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 3.4).

All experiments run on *Spark 2.0.1* with a patched [46] version of Hadoop 2.7.3 infrastructure. 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*.

3.5.3 Benchmark and Workloads

To study the performance of our solution we use several workloads (described in 3.5a), that are currently used in popular benchmark suites and cover different kinds of applications. The workloads

	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

Table 3.6a: Average run time

span from simple applications that target a single and specific feature of the connectors (micro benchmarks), to real complex applications composed by several jobs (macro benchmarks).

The micro benchmarks use three different 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 [47, 48] test suite, is the "Hello World" application for parallel computing. It is a read-intensive workload, that reads an input text file, computes the number of times each word occurs in the file and then writes a much smaller output file 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 [49, 50] implemented with DataBricks' Spark-Sql-Perf library [51]. It executes several complex queries on files stored in Parquet format [15]; 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 for the Read-only, Copy, Wordcount and Terasort benchmarks are divided into 128 MB objects. The outputs of the Copy, Teragen and Terasort benchmarks are also divided into 128 MB objects. We also run Spark with a partition size of 128 MB.

3.5.4 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.

3.6 Experimental Evaluation

We now present a comparative analysis between the different scenarios that we defined in 3.5.2. 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.

	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 3.6b: Workload speedups when using 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 3.6c: Ratio of REST calls compared to Stocator

3.6.1 Reduction in run time

For each workload we ran each scenario ten times. We report the average and standard deviation in 3.6a. The results shows that, when using a connector out of the box 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.

3.6b 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. 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.

3.6.2 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. Figures 17 and 18 show 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.

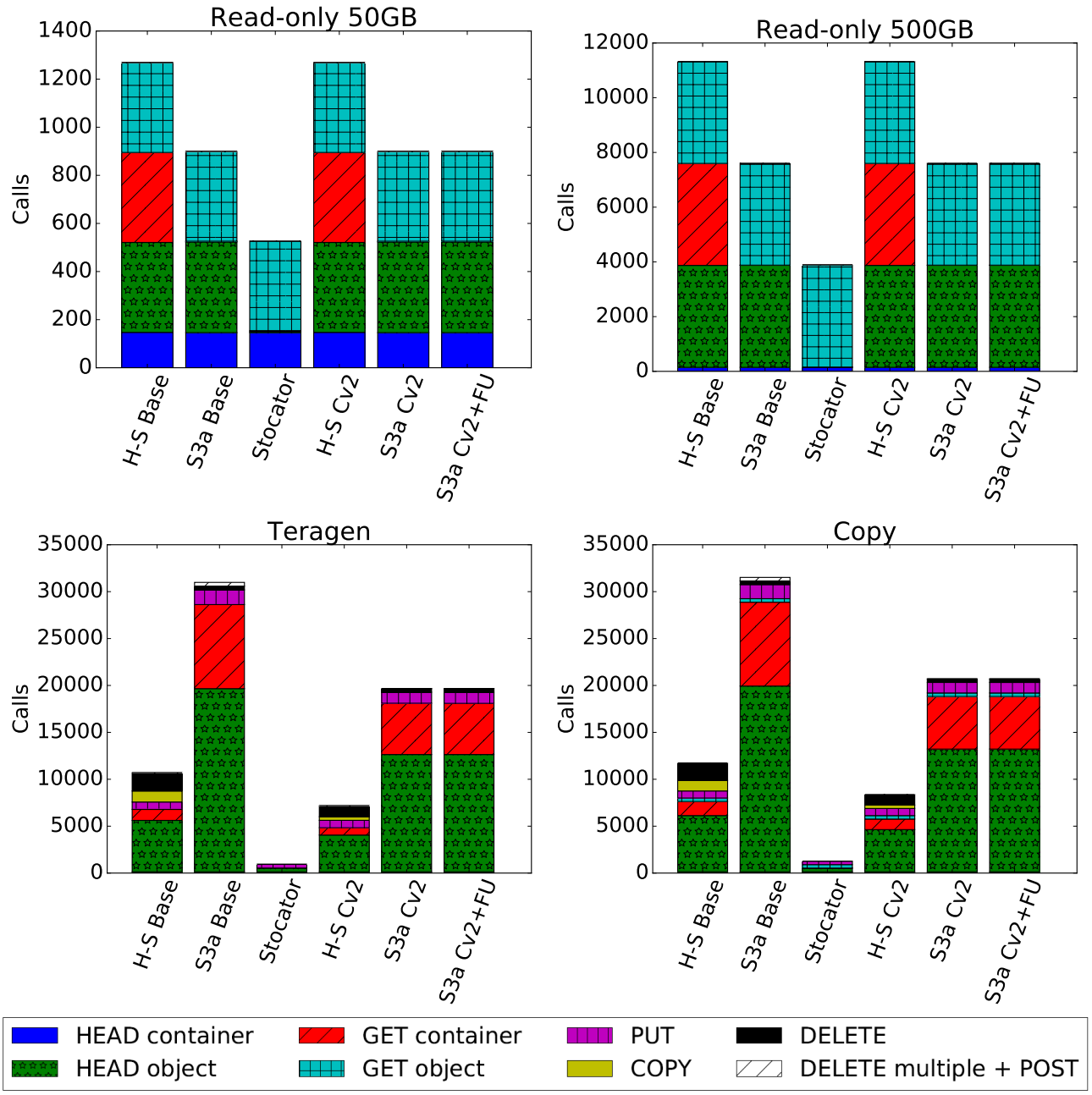

Figure 17: Micro-benchmarks REST calls comparison

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 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. 3.6c shows the number of REST calls that is possible to save by using Stocator. We observe that, for write-intensive workloads, Stocator

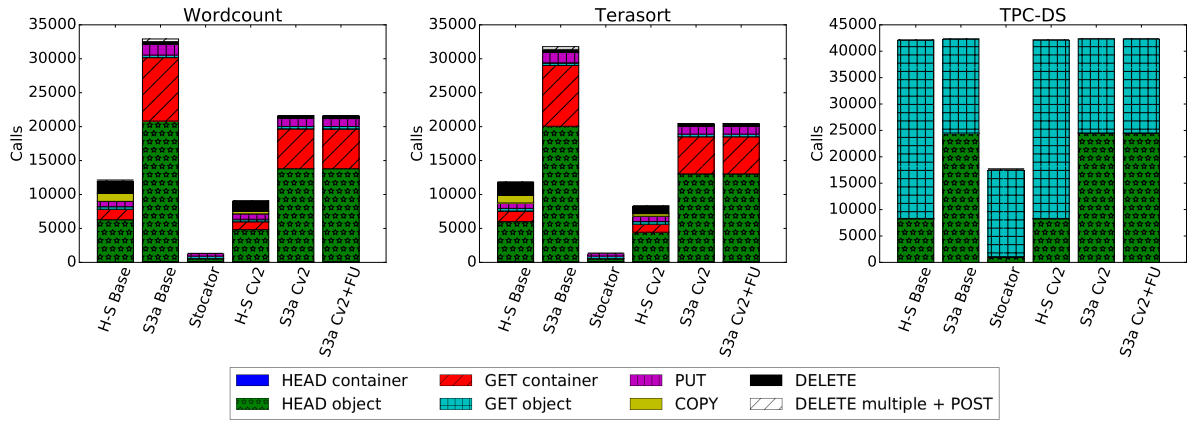

Figure 18: Macro-benchmarks REST calls comparison

	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	2.23
S3a Base	x1.63	x1.94	x27.82	x26.74	x26.84	x25.88	2.25
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	2.25
S3a Cv2 + FU	x1.63	x1.94	x17.55	x17.29	x17.34	x16.40	2.25

Table 3.6d: REST calls cost compared to Stocator for IBM, AWS, Google and Azure infrastructure

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 active.

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. 3.6d 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 [52], AWS [53], Google [54] and Azure [55]; 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 19 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 19 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.

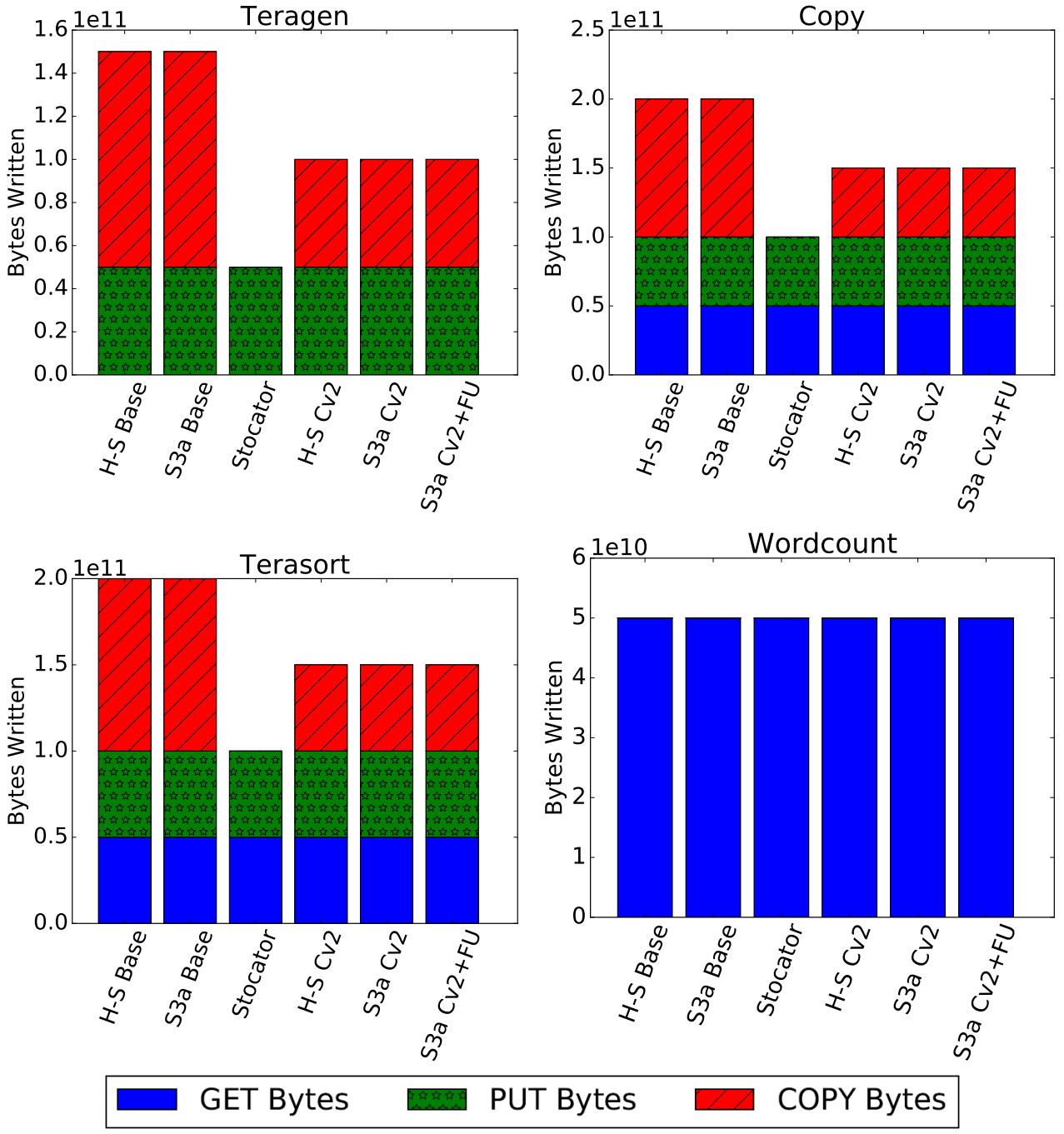

Figure 19: Object Storage bytes read/written comparison

3.7 Related Work

In the past several years there has been a variety of work both from academia and industry [56, 57, 58, 4, 59, 60, 61, 62, 63, 64, 65, 31, 42, 34, 66, 33, 67, 44, 35] that target the performance of analytics frameworks with different configurations of Compute and Storage layers. We can divide this work into two major categories that tackle performance analysis and eventual consistency.

Performance Analysis. Work from [56, 57, 58, 4, 59, 60, 61, 62, 63, 64] analyze the performance of analytics frameworks with different configuration of the Compute and Storage layer. All this work, albeit valid, base their conclusion on limited information, workloads and configurations that may not highlight some problems that exist when analytic applications connect to a specific Data or Storage layer solution. In particular Ousterhout et al. [56] 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, such work states that the runtime of analytics applications is generally CPU-bound rather than I/O intensive. A recent work [65] shows that this is not always true; moving from a 1Gbps to a 10Gbps network can have a huge impact on the application runtime. Another work [31] shows that is possible to further improve the run times by eliminating impedance mismatch between the layers, which can highly affect the run times of such applications; one in particular when using an Object Storage solution (e.g.; Openstack Swift [41, 68]) as the Storage layer. Concurrently there has also been some work from industry and open source to improve this impedance mismatch. Databricks introduced something called the DirectOutputCommitter [36] for 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 [37], 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.

Eventual Consistency. Vogels [42] addresses the relationship between high-availability, replication and eventual consistency. 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. In particular, when there is eventual consistency on the list operations over containers/buckets, current connectors from the Hadoop community for Swift API [30] and the S3 API [29], can also lead to failures and incorrect executions. EMRFS [33, 67] from Amazon and S3mper [34, 66] from Netflix overcome eventual consistency by storing file metadata in DynamoDB [6], an additional storage system separate from the object store that is strongly consistent. A similar feature called S3Guard [44, 35] that also requires an additional strongly consistent storage system 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.

3.8 Conclusions and Further Work

IBM started from the onset the IOStack project a research which has led to the development of Stocator: a high performance object storage connector for Apache which is presented in the second part of this report. 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.

Stocator has been made available to the open source community [39], it has already found its way to production starting from 2016 within IBM Analytics for Apache Spark, an IBM Cloud service. Also stocator has enabled the SETI project to perform computationally intensive Spark workloads on multi-terabyte binary signal files[40].

3.8.1 Further Work

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 (which should primarily require testing) and Hive.

3.8.2 Gridpocket impact

Stocator has been instrumental to demonstrate to the Gridpocket company, our IOStack use-case partner, the benefits of using Objects Storage and Spark, thanks to the huge speedup of the data upload to Object Store phase.

References

[1] "IOStack report D4.2." http://iostack.eu/deliverables/send/3-deliverables/24-d4-2.

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," Commun. ACM, vol. 51, pp. 107–113, Jan. 2008.

[3] "Apache Hadoop." `http://hadoop.apache.org/`.

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

[5] "Amazon S3." `https://aws.amazon.com/s3/`.

[6] "Amazon DynamoDB." `https://aws.amazon.com/dynamodb/`.

[7] "Athena." https://aws.amazon.com/athena/pricing.

[8] "Parquet Format." https://parquet.apache.org.

[9] "OCR Format." https://orc.apache.org.

[10] "Hive Style Partitioning." https://www.linkedin.com/pulse/hive-partitioning-bucketing-examples-gaurav-singh.

[11] "Gridpocket." http://www.gridpocket.com.

[12] "k-d-tree." https://en.wikipedia.org/wiki/K-d_tree.

[13] "Gridpocket Metergen Simulator." https://github.com/gridpocket/project-iostack/tree/master/meter_gen.

[14] "IBM COS Service." https://ibm-public-cos.github.io/crs-docs/.

[15] "Parquet Format." https://github.com/apache/spark/pull/15835.

[16] L. Sun, S. Krishnan, R. S. Xin, and M. J. Franklin, "A partitioning framework for aggressive data skipping," VLDB, 2014.

[17] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin, "Fine-grained partitioning for aggressive data skipping," SIGMOD, 2014.

[18] "Using Pluggable Apache Spark SQL Filters to help GridPocket users." https://spark-summit.org/eu-2017/speakers/paula-ta-shma/.

[19] "Databricks Delta." https://databricks.com/announcing-databricks-delta.

[20] "Data Skipping Index." https://docs.databricks.com/spark/latest/spark-sql/dataskipping-index.html.

[21] A. Shanbhag, A. Jindal, S. Madden, J. Quiane, and A. J. Elmore, "A robust partitioning scheme for ad-hoc query workloads," SoCC, 2017.

[22] Y. Lu, A. Shanbhag, A. Jindal, and S. Madden, "Adaptdb: Adaptive partitioning for distributed joins," VLDB, 2017.

[23] "Space Filling Curves." https://en.wikipedia.org/wiki/Spacefilling_curve.

[24] "Z-order curves." https://en.wikipedia.org/wiki/Z-order_curve.

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

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

[27] "Apache Spark." http://spark.apache.org/.

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

[29] "Amazon Web Services SDK for Java." https://aws.amazon.com/sdk-for-java/.

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

[31] F. Pace et al., "Experimental performance evaluation of cloud-based analytics-as-a-service," in 9th IEEE Intl Conf. on Cloud Computing, CLOUD, 2016.

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

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

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

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

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

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

[38] "Openstack swift api." https://developer.openstack.org/api-ref/object-storage/.

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

[40] 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.

[41] "OpenStack Swift." http://swift.openstack.org/.

[42] W. Vogels, "Eventually consistent," Communications of the ACM, vol. 52, pp. 40–44, Jan. 2009.

[43] "Amazon Web Services benefits." https://aws.amazon.com/application-hosting/benefits/.

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

[45] 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.

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

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

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

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

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

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

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

[53] "Amazon Web Services REST calls cost." https://aws.amazon.com/s3/pricing/.

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

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

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

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

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

[59] 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.

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

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

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

[63] 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.

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

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

[66] "Netflix S3mper." https://github.com/Netflix/s3mper.

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

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