HORIZON 2020 FRAMEWORK PROGRAMME

# IOStack

(H2020-644182)

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

# D4.2 SDS Services for Analytics Prototype and Initial Evaluation

Due date of deliverable: 31-12-16
Actual submission date: 31-12-16

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

# Summary of the document

| | |
|---|---|
| **Document Type** | Deliverable |
| **Dissemination level** | Public |
| **State** | v1.2 |
| **Number of pages** | 25 |
| **WP/Task related to this document** | WP4 / T4.2 |
| **WP/Task responsible** | IBM |
| **Leader** | Yosef Moatti (IBM) |
| **Technical Manager** | Dalit Naor (IBM) |
| **Quality Manager** | Raúl Gracia-Tinedo (URV) |
| **Author(s)** | Yosef Moatti (IBM), Raúl Gracia-Tinedo (URV) |
| **Partner(s) Contributing** | IBM, URV |
| **Document ID** | IOStack_D4.2_Public.pdf |
| **Abstract** | This deliverable reviews the implementation of the SDS services for analytics in IOStack as of end of 2016. In a first part, this document reviews the design and implementation of the Apache Spark SQL pushdown technology, which enables the acceleration of Spark Big Data analytics on data stored in object stores through push down of Spark SQL filters. In a second part it reviews and discuss the experimental evaluation that we performed on a large cluster granted by the OpenStack Innovation Center. |
| **Keywords** | storlet, analytics, spark, csv, SQL, computation push-down, swift, OSIC |

## History of changes

| Version | Date | Author | Summary of changes |
|---|---|---|---|
| 1.0 | 25-11-2016 | Yosef Moatti | First version |
| 1.1 | 14-12-2016 | Yosef Moatti | Revision after first round of reviews |
| 1.2 | 17-12-2016 | Yosef Moatti | Revision after second round of reviews |

## Table of Contents

## 1 Executive summary

From the onset of the IOStack project, the WP4 team's assessment was that the Apache Spark project would soon become central in the big data space. This was soon confirmed in 2015, as Apache Spark overcame the Apache Hadoop project, see for instance [1, 2]. A second assessment was that an Apache SQL pushdown to Swift prototype would very well demonstrate data reduction techniques which are a main goal of WP4. Discussions with the Gridpocket company strengthened our thinking that such a technology could be an important enabler for Apache Spark adoption for big data analytics .

The following paragraph gives a more detailed rationale for this technology: extracting value from data stored in object stores, such as OpenStack Swift, can be problematic in common scenarios where analytics frameworks and object stores run in physically disaggregated clusters, as is mostly common when Apache Spark is used. One of the main problems is that analytics frameworks must ingest large amounts of data from the object store prior to the actual computation; and this incurs a significant resources and performance overhead. To overcome this problem, we aimed at developing a technology which would enable analytics frameworks to tap unused computational resources at object stores to both off load filter computation from Spark analytics jobs and also alleviate the networking bottleneck that is too often observed during the data ingest phase. This goal would be achieved by offloading ETL-type and SQL querying functions to the object store which would be instrumented with the Storlet [3] technology acting as a rich and extensible active object storage layer.

The first year of the IOStack project, was devoted to lay the basis for reaching this goal: first by stabilizing and open sourcing the Storlet technology, which would become instrumental for running within Swift the pushed down SQL filter tasks; and secondly by outputting the initial version of the Stocator connector which became in 2016 the first industry class connector of Apache Spark to Object Stores.

During the second year, our efforts were devoted at consolidating the Stocator connector, and setting up a prototype of the Spark SQL pushdown prototype. We then validated this prototype against the Gridpocket use case and finally analyzed its performance and scale-up capabilities through extensive experiments conducted in a 63-machine production class cluster with generated IoT data and SQL queries from GridPocket. These experiments shown that our technology exhibits query execution times up to 30x faster than the traditional "ingest-then-compute" approach.

The described WP4 achievements result from the cooperation between the following IOStack consortium partners:

- IBM, which designed, implemented and tested both the Spark and Object Store parts of the prototype.
- Arctur, which provided the IOStack testbed
- Eurecom, which provided the Zoe tool which permitted the installation of the various Spark clusters that we used both in validation and performance testing phases
- Gridpocket, which provided the use case, including scenario, data, queries as well as the notebook code which drove part of the experiments
- URV, which provided both the Crystal software (for easy installation and usage of Storlets) as well as their expertise at interpreting and describing the experimental data that was collected in the experimental phase

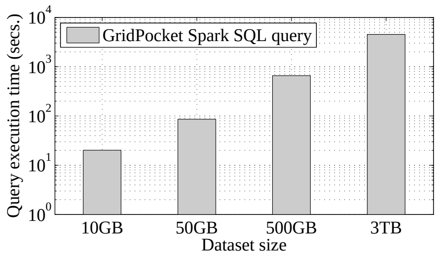


Figure 1: Impact of the "ingest-then-compute" problem on GridPocket analytics.

## 2 Introduction

These days, object stores, such as Amazon S3 [4], OpenStack Swift [5] or IBM Cleversafe [6], accumulate enormous amounts of non-structured or semi-structured data. A key reason for the popularity of object storage is its scalability, availability and cost effectiveness properties. But perhaps more importantly, its simplicity of use via HTTP RESTful APIs brings unique opportunities to easily automate the storage of data from any remote source.

A simple storage interface is preferable in numerous scenarios and use cases, from Internet-of-Things (IoT) to server logs amounting to a few terabytes. Due to their volumes and velocity, servers and sensors autonomously store data "as is" in object stores, without further processing and structuring. This is the case for companies such as GridPocket[1]: a smart energy grid company which is part of the IOStack consortium and which provided the real life use case that motivated this research. In the company's daily operation, hundreds of thousands of smart meters automatically collect and store energy consumption measurements from users across several European cities. Thanks to the scalability properties of object storage, an increasing number of GridPocket meters can continuously store energy measurements, while the system can scale out to satisfy the storage demands of the company.

However, simplicity and flexibility on the storage side come at a cost. This is particularly visible when extracting value from data. While this is less of a concern for dedicated clusters that provide both storage and computation, and frameworks such as Hadoop or Dryad that co-locate computation with data, it is an important issue for elastic clouds and analytics-as-a-service platforms such as Amazon Elastic Map-Reduce[2] (EMR). In these platforms, object stores are *physically disaggregated* from the compute clusters running the analytics [7]. Although disaggregating storage from computation has advantages (e.g., administration, security), the unintended consequence of such separation is that analytics frameworks must first ingest large amounts of data to perform the computation.

In practice, executing analytics in disaggregated compute and storage clusters presents some problems: i) Inter-cluster network bandwidth may be saturated due to parallel data ingestions from multiple analytics jobs; ii) The data ingestion phase consumes extra resources (e.g., CPU, RAM) from a compute cluster shared across multiple tenants; and iii) Analytics jobs suffer from overhead related to ETL (Extraction-Transformation-Loading) tasks used to prepare raw data objects. In the literature, this phenomenon is known as the *ingest-then-compute* or *store-first-query-later* problem [8, 9].

Unfortunately, the ingest-then-compute problem is also present in the company's daily operation. GridPocket data scientists execute Spark SQL workloads with heavy data ingestion against energy measurement data stored in an object store. As shown in Fig. 1, executing a given query on increasingly larger datasets involves a linear growth in query completion times. Hence, while GridPocket datasets are continuously growing, the capacity to execute analytics on such data is insufficient due to the overhead of data ingestion. Furthermore, as analytics frameworks evolve towards better performance (e.g., Spark SQL v2.0 is x2-x10 faster than v1.6), the inefficiencies derived from the ingest-then-compute problem will become a dominant bottleneck for many companies.

[1]`http://www.gridpocket.com`
[2]https://aws.amazon.com/en/emr

## 2.1 Scope and Challenges

To address the Big Data ingestion problem in object storage, an architecture that integrates ETL and querying functions in the object store with Big Data analytics models is needed. Among other things, such an architecture should overcome two main challenges:

**Task offloading**. To drive data ingestion with the object store, we need a communication channel that enables the analytics framework to offload processing operations to the object store. Specifically, upon job execution, the analytics framework should be able to define the task to be executed at the object store close to the data, to improve the performance or efficiency of the job at hand.

**Rich active storage layer**. The smartness of the storage layer should not be single purpose. Conversely, the challenge is to enable an object store to execute general-purpose code close to the data. Such code should be easily deployed to extend the functionality of the system for handling new offloaded tasks. We could imagine, for instance, enforcing some degree of data privacy by ensuring that the data (e.g., IoT electrical readings) can only be retrieved with a minimal granularity (which would be a function of the identity and purpose of the reader).

Rethinking the problem of GridPocket, solving these challenges would enable us to add SQL functionalities into the object store so it can cooperate with the analytics framework by becoming a *queriable data source*. For instance, we could extend the object store with the functionality necessary to execute SQL projections and selections directly where data lives. This would reduce data ingestion and the need for processing power at the compute cluster. As a result, GridPocket data analytics would become more scalable and efficient.

## 2.2 Contributions

To overcome the ingest-then-compute problem in disaggregated Big Data clusters, we describe in this document the *pushdown technology* that we developed and which exposes a parallel architecture that enables analytics frameworks to leverage the computational resources of object stores to accelerate the execution of jobs and make them more efficient. The described technology achieves this by offloading ETL-type and querying functions to the object store through a rich and extensible active object storage layer. Contrary to prior works [7, 10], the ultimate goal of the IOStack technology is to extend the object store "on-the-fly" with new types of general-purpose code executed close to the data, thus meeting the requirements of new offloaded tasks.

As a proof-of-concept, we translated a concept akin to "predicate pushdown" in the traditional database literature [11, 12] into a disaggregated analytics ecosystem. Our implementation enables efficient execution of SQL queries on raw Comma-Separated Value (CSV) data stored in OpenStack Swift, to accelerate GridPocket analytics. At the analytics side, we extended the CSV data source in Apache Spark, which can now offload SQL projections and selections on parallel object requests against Swift. At the object store, we contributed to the OpenStack Storlets: a framework to intercept and execute sandboxed code on object requests in OpenStack Swift. Among other things, we extended Storlets with the capability of efficiently executing computations close to the data. Moreover, we created a new Storlet code to perform the SQL projection and selection filtering on CSV objects. The source code of the various components of the IOStack pushdown technology are publicly available (see Section 6).

To evaluate our system, we executed extensive experiments on a 63-machines cluster over real IoT data from GridPocket energy meters. Our results show that our pushdown technology can accelerate the end-to-end SQL processing time on the semi-structured data by *up to* 30 *times* depending on the dataset size and amount of filtered data. Consequently, the IOStack developed technology enables GridPocket to benefit from the advantages of object storage, while making their analytics workloads much faster and more efficient.

In summary, as of end of 2016, the key contributions of the technologies developed in WP4 are:

- Design of a novel solution that exposes a parallel architecture to address the ingest-then-compute problem for data stored in object storage;

- Implementation of the technology which allows Spark SQL to transparently offload the execution of selections and projections to OpenStack Swift, and hence, where the data lives;
- Validation of our system in a production cluster and generated data from GridPocket workloads;
- The release to open source of the various code components
- The public availability of the anonymized datasets

*Roadmap*: This document is organized as follows: in section 3 we discuss related work; section 4 provides technical background for the rest of this document; section 5 describes the design principles of the IOStack pushdown technology; section 6 depicts our implementation to enable SQL predicate execution at the object store and in Section 7, we present our validation framework and the results of our experiments. We discuss our future work and conclude in sections 8 and 9, respectively.

## 3 Related Work

The *ingest-then-compute* data life-cycle imposed by infrastructure disaggregation is a performance barrier for today's data analytics frameworks [8]. This problem has attracted interest from the research community in various ways. On the one hand, recent works have focused on interfacing Hadoop with enterprise file systems to bridge the gap between legacy data stores and compute clusters [13, 14, 15], or even replacing HDFS by optimized file systems to minimize the impact of disaggregation [16, 8]. On the other hand, in-situ analytics aim at improving Big Data acquisition (i.e., data collection, transmission, and pre-processing). Essentially, in-situ data processing benefits from the compute capacity of data producers to execute computations and filters during data acquisition [17, 18, 19]. This approach reduces the amount of data that should be transferred and eventually siloed, as well as the overhead of exporting raw data from the storage cluster to perform analytics.

Perhaps, the vein of research closest to this work refers to the application of active storage techniques to mitigate the impact of compute/storage disaggregation. To wit, Huston et al. presented Diamond [20], an active storage architecture that provides early discard of useless data in interactive search. Conceptually, both Diamond and our system exploit the potential benefits of data filtering at the data store via active storage techniques [21, 22]. However, Diamond has not been targeted and applied on data analytics.

Regarding data analytics, Rhea [7] is a system for transparently filtering unstructured data in MapReduce via SQL projections and selections. Rhea relies on a filter compilation engine to transparently add SQL-like filters to MapReduce jobs which are executed on a filtering proxy at the storage side. A similar effort called Minimal [23] focuses on automatically optimizing MapReduce programs to reduce the data movements during the computation. More recently, Cybertron [10] combined data filtering with novel coding techniques to reduce IO overheads of analytics jobs.

A key difference between our developed technology and these works lies in the storage layer. First, some of these works do not support actual data locality at the storage cluster as done with our technology; that is, Rhea [7] resorts to a proxy entity that executes data filtering, but all the data should be read from the storage servers to the proxy. Second, these systems support only a limited number of data filters, as they consider a particular use case (e.g., SQL predicate filtering). The storage layer of our technology goes far beyond this goal. Our technology extends the object store with a sandboxed platform that can execute custom pushdown filters on object requests exploiting data locality. Analytics applications can communicate with the object store to dynamically execute these filters (e.g., SQL filter, complex calculations, data compression), which are explicitly managed via simple policies. This makes our technology a more general and flexible active storage system for analytics than prior works.

## 4 Technical Background

### 4.1 Spark Ecosystem

Apache Spark[3] is a general-purpose cluster computing framework that was developed at UC Berkeley. It was designed for iterative workloads and provides both APIs in Scala, Java, R and Python, and libraries for stream and graph processing, machine learning and SQL.

Spark offers a simple programming API that lets programmers manipulate distributed collections of Java or Python objects across a cluster through operations like `map`, `filter`, and `reduce`. Such collections called Resilient Distributed Datasets (RDDs) [24] reside in memory to optimize computations on large clusters. For instance, the Scala code below counts lines including the word "Spark" in a text file:

```
textFile = spark.textFile("hdfs://...")
lines=textFile.filter(line=>line.contains("Spark"))
println(lines.count())
```

This code creates an RDD of strings called `textFile` by reading an HDFS file, then applies `filter` to obtain a derived RDD, `lines`. It then performs a `count` on this data. RDDs are immutable and

[3]Apache Spark Project `http://spark.apache.org`

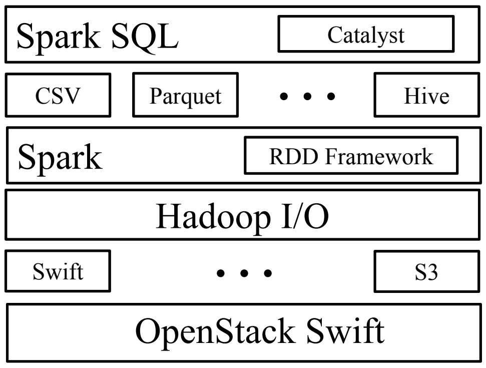


Figure 2: Components in a Spark deployment to execute SQL queries on semi-structured data stored in an object store.

lazily evaluated. They are also resilient because they maintain lineage information for reconstructing lost partitions. The critical improvement of Spark as compared with Hadoop lays in its ability to cache intermediate results in memory as RDDs. However, caching is mostly beneficial for iterative algorithms (e.g., for machine learning). Ad-hoc querying and data exploration, instead, require access to data that is harder to cache, because in such cases, data access patterns are less prone to benefit from the limited amount of RAM available for caching. Hence, caching does not alleviate the "ingest-then-compute" problem.

As visible in Fig. 2, there are libraries on top of the engine that facilitate different types of analytics workloads. Our work focuses on Spark SQL: a library for structured data analysis. Spark SQL is essentially a generic engine for distributed structured data manipulation. The operations on data are done using SQL queries and a programmatic API (i.e., Data Frames API). Out-of-the-box Spark SQL can read various data formats, such as data coming from Parquet, JSON and Hive tables.

For other data sources or formats, Spark SQL offers the "Data Sources API". Implementing these APIs enables us to import new formats into Spark SQL. In a nutshell, this API is used by Spark-SQL to translate some *foreign* data format into a common representation of structured data that Spark SQL knows how to work with. We focus on the Spark-CSV library which is an implementation of the Data Sources API for importing CSV formatted data into Spark SQL.

Spark SQL uses a query optimizer called Catalyst [25], [26]. Given a SQL query, the optimizer *extracts* the projection and selection filters implied by the query. These extracted filters are then used by Spark SQL with the customized flavors of the data source API. As we describe later on, we leverage Catalyst filter extraction together with the Data Sources APIs to offload the filtering work from Spark to the object store.

At the lowest level, Fig. 2 shows that Spark interoperates with Hadoop, in that it can manage data from any storage system supported by Hadoop, including HDFS or S3. For interoperation with Swift, the connector supports Hadoop's input/output APIs, although many of their operations are not native to Swift, such as moving, copying or renaming directories.

## 4.2 OpenStack Swift

OpenStack Swift is a highly scalable object storage system that can store a large amount of data through a RESTful HTTP API similar to that of Amazon S3. It provides a simple API to store (`PUT`), retrieve (`GET`), and delete (`DELETE`) objects. The access path to an object consists of exactly three elements: /account/container/object. The object is the exact data input by the user, while accounts and containers provide a way of grouping objects. Nesting of accounts and containers is not supported.

To achieve high scalability, Swift exploits the synergy between a flat object ID space and consistent hashing via a hash-based data structure called *ring*. The ring guarantees access load balancing across nodes within the cluster; this results in higher performance and storage capacity as more nodes are added to the cluster. Moreover, Swift can be run on commodity servers, which facilitates the

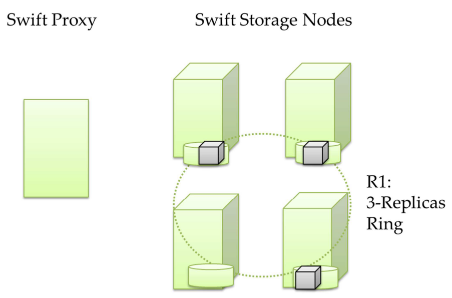


Figure 3: Swift Architecture

horizontal scaling of large deployments.

As shown in Fig. 3, internally, Swift exhibits a two-tier architecture that consists of proxy and object servers. The former are in charge of authentication, authorization and access control enforcement of storage requests. Upon reception of a valid request, a proxy server routes it to the corresponding object servers for storage. Object servers are also responsible for handling the replication of objects across available disks to reach the defined data availability threshold, and for managing objects. Both proxies and storage nodes include a WSGI[4] pipeline that enables developers to configure middlewares that intercept object requests with environment information.

## 5 Design

The main goal of developed technology is to enable analytics frameworks to benefit from the computational resources of an object store for optimizing job execution in disaggregated clusters. Indeed, the cooperation between analytics frameworks and object stores can be exploited in many ways. For instance, the object store may execute projections/selections defined in a SQL query to avoid transferring unnecessary data to the compute cluster. Alternatively, it can perform aggregations on individual object requests to facilitate the construction of graphs from a large dataset.

To solve this challenge, at the analytics framework, we provide existing analytics tasks with a means of delegating or "pushing down" specific computations to the object store. The object store, in turn, needs to have a rich and extensible compute layer that makes it capable of executing various types of calculations and ETL tasks based on incoming requests. Our technology achieves that by providing three abstractions: *pushdown task*, *analytics delegator*, and *pushdown filter*.

### 5.1 Concepts

**Pushdown task**: A pushdown task is the *work being delegated* to the object store. In practice, a pushdown task is represented as a piece of metadata attached to an object request. It embodies the trade-off between the consumption of compute resources at the storage cluster and the acceleration of analytics jobs. With our technology, a pushdown task is interpreted broadly; for instance, it may consist of predicates to filter from an SQL query or a partial computation to be executed on object request (e.g., aggregations, statistics). Naturally, both the analytics framework and the object store require an *end-to-end orchestration* to cooperate on a given pushdown task.

**Analytics delegator**: The analytics delegator is integrated with the analytics engine and enables our technology to *push down tasks to the object store*. The main purpose of the analytics delegator is to appropriately tag parallel object requests with the correct metadata to execute pushdown computations at the object store. Thus, upon the submission of a job, the analytics delegator works within the distributed task execution flow by attaching to each data partition the pushdown task that will be executed at the object store.

**Pushdown filter**: A pushdown filter (not to be confused with a Pushdown task) is a piece of programming logic that a system administrator can inject into the object store to perform custom computations. In our system, the behavior of pushdown filters is defined by two main properties: i) A pushdown filter is *triggered upon an incoming object request* with the appropriate metadata that pro-

[4]`https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface`

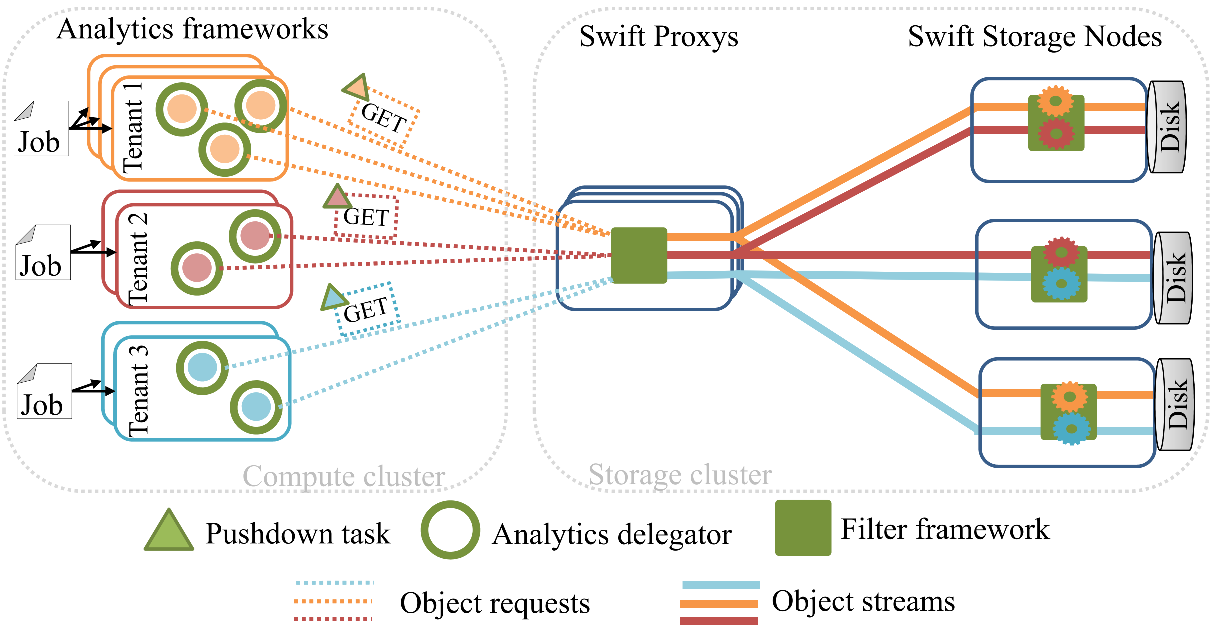


Figure 4: Architecture of our pushdown technology deployed on top of OpenStack Swift.

vides instructions to do so; ii) The execution of a pushdown filter occurs within the context of *a single inbound/outbound data stream* of an object request. This means pushdown filters are not designed to communicate among them at runtime to perform distributed coordinated computations. Our pushdown technology offers a powerful compute layer general enough to run a variety of calculations on object, from ETL and data discard tasks, to more complex numerical and statistical processes. A key feature of pushdown filters is that the instrumented object store is oblivious to their execution, and needs no modification to its implementation code to support them.

### 5.2 Architecture

Next, we present the design of our pushdown technology on top of OpenStack Swift to overcome the ingest-then-compute problem in analytics platforms (see Fig. 4).

At the compute cluster, our pushdown technology is able to delegate computations to the object store to accelerate submitted analytics jobs of different tenants via the *delegator component*. As shown in Fig. 4, an analytics job is broken into tasks that are distributed among the cluster nodes. Thanks to the delegator, analytics tasks that form the job can now add the appropriate *pushdown task* at each object request generated. This is achieved by piggybacking specific metadata fields in the HTTP `GET` request executed against the object store. Note that each tenant sharing the compute cluster may be executing very disparate analytics jobs. Therefore, each interceptor should inject the appropriate pushdown task to each job to trigger the correct computations in the object store.

At the storage cluster, we equip the object store with a general-purpose and powerful computation layer to *execute the pushdown filters* defined in the metadata of each object request. The execution of pushdown filters is performed on a request's data stream. This means multiple jobs can execute parallel pushdown filters on `GET` requests of the same object; all of them will receive their "own filtered version" of the object, whereas the stored object will remain unaltered. In particular, our pushdown technology is able to execute several pushdown filters on a single request (i.e., pipelining), as well as to decide the execution stage of a pushdown filter (i.e., proxy/storage node). Moreover, our pushdown technology can be extended "on-the-fly" with new pushdown filters. A third party integrating a new pushdown filter only needs to contribute the logic; the deployment and execution of the filter is managed by the system.

## 6 Implementation: Spark SQL Pushdown

In this section, we illustrate the implementation of our pushdown technology and how we extended it to leverage SQL pushdown for Apache Spark on top of OpenStack Swift. Our implementation is targeted at enhancing the daily operation of GridPocket, an energy grid company that executes heavy SQL workloads on object-based datasets generated by smart energy meters.

### 6.1 Architectural Components

**Delegation of Spark SQL predicates**: To delegate SQL queries we used the Spark Data Sources API.

Specifically, we modified the Spark-CSV library [27], which allows to import CSV data into Spark. The technical background is that the Data Sources API has several flavors. The simplest flavor is called `Scan`. `Scan` takes no parameters, and is expected to return all the originally "foreign formatted" data in the common representation used by Spark SQL. A more complex flavor is the `PrunedScan` API which takes a *selection filter as a parameter*, and returns the selected columns in the common representation. The `PrunedScan` API can be seen as a generic Spark-SQL mechanism for enabling the Data Source library not only parsing the formatted data, but also to filtering it. Further, the `PrunedFilteredScan` API flavor takes both a *projection and selection filters*, thus enabling passing both filter types to the Data Source library. In our implementation, we augmented the Spark CSV library with the `PrunedFilteredScan` Data Source API. Concretely, most of our work focused on enabling this library to push down projections/selections to OpenStack Swift, so data filtering is done at the storage cluster instead of at the compute cluster[5].

To read CSV data, we use Spark in cooperation with Hadoop, which is responsible for reading the data from the physical storage while taking care of logical records that may be split between partitions. To this end, Hadoop can work with a set of drivers that manage data from various sources. In this work, we extended Stocator[6], a high-speed connector to object stores. Stocator optimizes many aspects of the data access to object stores, as compared with the standard Hadoop driver, and optimizes the performance of managing large datasets in OpenStack Swift (e.g., metadata, file renaming). This is demonstrated, for example, in experimental measurements reported in [28]. We modified Stocator so that it could inject pushdown tasks in object requests issued to Swift; that is, HTTP requests issued by Spark tasks to ingest data objects are tagged with the appropriate metadata (e.g., projections/selections) to execute both projections and the selections at the object store. We also extended the Hadoop RDD so that the projection and selection filters propagate through the RDD's partitions all the way down to Stocator.

In Section 8, we discuss how our work is evolving towards a framework which generalizes the current SQL pushdown capabilities and bypasses the Hadoop layer.

**Pushdown filter framework**: Our technology offers simple means for deploying and enforcing pushdown filters on a particular tenant or container via policies in OpenStack Swift [29]. It intercepts storage requests and executes the pushdown filters specified on the request's metadata at storage nodes. To this end, we contributed to the OpenStack Storlets framework [3, 30], which allows running computations, called *storlets*, in the object store. Storlets provides a powerful extension mechanism to OpenStack Swift —without changing its code— to run computations close to the data in a secure and isolated manner making use of `Docker` [31]. With Storlets a developer can write code, package and deploy it as a regular object, and then explicitly invoke it on data objects as if the code was part of the Swift's WSGI pipeline. Request interception can occur not only at the proxy but also at the object servers thanks to the Storlet's WSGI middleware integrated in Swift, which "wraps" storage requests and responses.

In this work, we extended the Storlets framework with two important capabilities: Pipelining and staging execution control (i.e., proxy/storage node) of pushdown filters. In addition, the Storlet WSGI middleware in Swift was extended to support running Storlets at storage nodes for byte ranges; this was fundamental to match the natural operation of Spark tasks, which work on specific byte ranges of objects. This for two reasons: first, to avoid transferring the full object from the ob-

[5]Available at: `https://github.com/iostackproject/Scoop-csv-sql-pushdown`
[6]Available at: `https://github.com/SparkTC/stocator`

ject node to one of the proxies instead of processing on the targeted byte range directly at the object node, and second, to benefit from the higher concurrency provided by the Swift object nodes pool as compared with Swift proxy nodes pool.

**CSVStorlet**: Writing pushdown filters to accelerate analytics jobs is developer-friendly. In the code snippet below, we observe that a system developer only needs to create a class that implements an interface (`IStorlet`), providing the actual data transformations on the object request streams (`iStream`, `oStream`) inside the `invoke` method. This model makes it possible to implement a wide variety of storage-side calculations to reduce inter-cluster data movement and improve performance of analytics frameworks.

```
public class StorletName implements IStorlet{

    @Override
    public void invoke (
        ArrayList<StorletInputStream> iStream,
        ArrayList<StorletOutputStream> oStream,
        Map<String,String> parameters, StorletLoggerlogger)
            throws StorletException{
            //Develop pushdown filter logic here
    }
}
```

As a proof-of-concept, we contribute a storlet that can perform projection and selection filters over CSV data[7]. The `CSVStorlet` is a Java code that adheres to the Storlets API; it gets as input a stream of the locally stored CSV formatted data along with the projection and selection filters as extracted by Catalyst, and outputs the filtered data.

## 6.2 The Pushdown Process Flow

Next, we depict the workflow involved for pushing down SQL predicates from Spark to OpenStack Swift. To this end, we follow a simple example where the user interacts with Spark using the Spark-Shell interpreter (Fig. 5). The commands entered are initially processed by the Spark client that generates a staged execution plan where each stage consists of multiple tasks to be executed in parallel on Spark worker nodes.

The flow begins with the user specifying i) a data source class implementation that matches the data format, as well as the ii) dataset location (step 1 in Fig. 5). Part of the data location is the name of the storage driver to be used to read the data. In our example, the driver is Stocator and the location is a path in Openstack Swift, which may represent a container with multiple data objects. The class is the `Spark-CSV` class [27], as we are executing Spark SQL queries on CSV data. At this point, Spark initiates the CSV class that in turn creates a Hadoop RDD, which essentially represents data that resides in a data store in its original format and that can be accessed with the HDFS API.

Following the Hadoop RDD creation, a process called partition discovery takes place. This process involves partitioning the data set and associating each partition with a task. Each of these tasks, when scheduled for execution will be dynamically associated to one of the Spark worker nodes. For Hadoop RDDs, the underlying storage driver checks the total size of the data specified by the user and divides the total size by the HDFS chunk size. In traditional HDFS implementations, the chunk size is the size of the atomic placement unit in the file system. That is, each file is made of chunks that are spread over the HDFS cluster. As such, this number has system-wide implications. However, this is not the case for object stores. We elaborate more on that in Section 8. We also note that the partition discovery process takes place before a query has been specified.

The user defines a SQL query over the data, and collects the results as a list (steps 2,3 in Fig. 5). At this point, Catalyst calculates the implied projection and selection filters and calls the appropriate

[7] Available at: `https://github.com/openstack/storlets/tree/master/StorletSamples/java/CsvStorlet/src/org/openstack/storlet/csv`

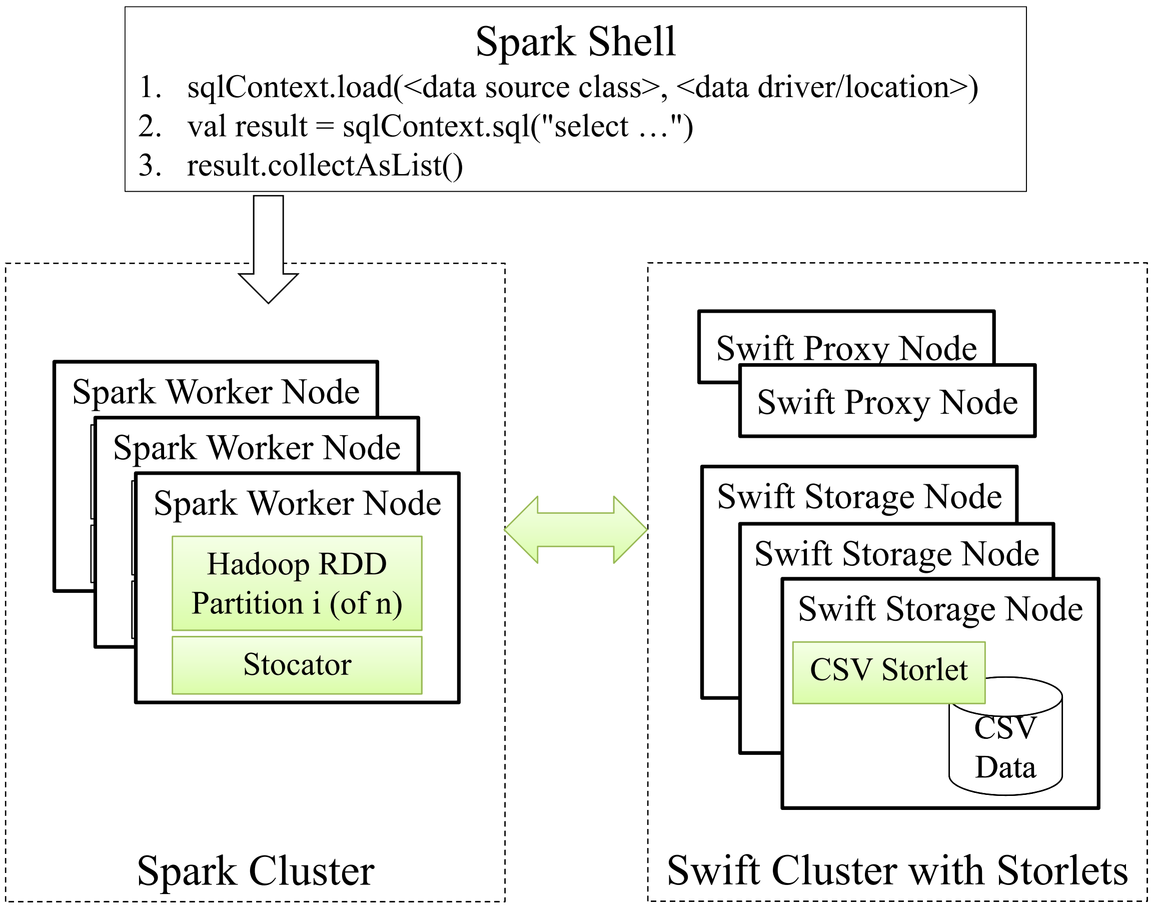


Figure 5: Overview of our implementation to leverage Spark SQL pushdown for OpenStack Swift.

data source API of the CSV class. The original Spark CSV class only supported projection/selection filtering execution *within* the compute cluster after ingesting the entire dataset. In Fig. 5, our CSV component extends the former CSV class to push down both SQL projection and selection from user queries to Swift.

Within the data source API that is called, each partition of the Hadoop RDD invokes Stocator to send a `GET` request to Swift to retrieve the chunk of the data for which it is responsible. To allow pushdown this `GET` request was changed to invoke the `CSVStorlet` so as to get back the filtered data. Therefore, for each partition Stocator sends a `GET` request with the `CSVstorlet` invocation. The storlet reads the data directly from disk, applies the appropriate pushdown filters defined for that tenant and sends back the filtered data. The resulting filtered data gets back to the worker node from which the request originated. The filtered data from all partitions are then further processed in each worker which run the part of the SQL query that was not pushed down. The local worker output is then aggregated back within the Spark client which completes the query processing.

Next, we show how this workflow greatly accelerates Spark SQL queries and mitigates the ingest-then-compute problem in disaggregated analytics clusters.

## 7 Experimental Evaluation

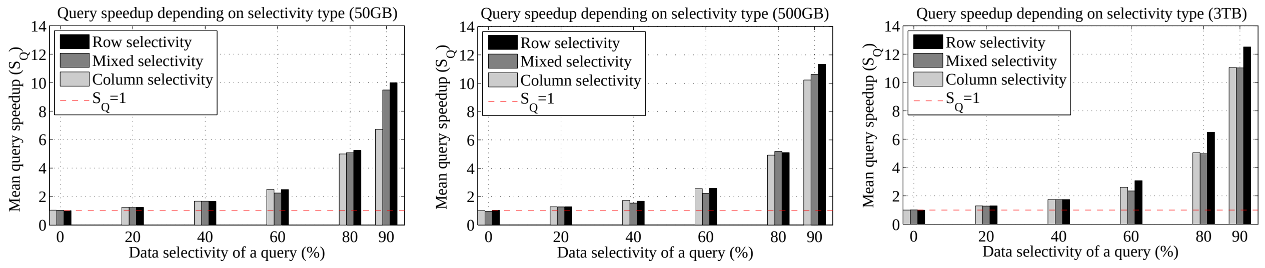
We evaluated a prototype of our technology for Spark and OpenStack Swift in terms of performance and overhead.

**Objectives**: Our evaluation demonstrates the contributions of this work by showing that it : i) provides an important acceleration of Spark SQL queries; ii) enables a real use-case to speed up its analytics workloads; iii) has an attractive resource consumption trade-off.

**Metrics**: In our experiments, we resort to two main metrics:

| *Query name* | *Query description* | *SQL query syntax* | *Column selec.* | *Row selec.* | *Data selec.* |
|---|---|---|---|---|---|
| ShowMapCons | Compute the per meter aggregated consumption, allowing to display results either in a heatmap or a per state aggregated consumption. | `SELECT vid, sum(index) as max, first_value(lat) as lat, first_value(long) as long, first_value(state) as state FROM largeMeter WHERE date LIKE ’2015-01%’ GROUP BY SUBSTRING(date, 0, 7), vid ORDER BY SUBSTRING(date, 0, 7), vid` | 92.00% | 99.62% | 99.97% |
| ShowMapMeter | In order to display a cluster map, obtain each meter with its info (city, Id,...) | `SELECT vid, sum(index) as max, first_value(city) as city, first_value(lat) as lat, first_value(long) as long, first_value(state) as state FROM largeMeter WHERE date LIKE ’2015-01%’ GROUP BY SUBSTRING(date, 0, 7), vid ORDER BY SUBSTRING(date, 0, 7), vid` | 92.00% | 99.54% | 99.97% |
| ShowMapHeatmonth | Get daily data for a given month for a (slider) parametric per day display. | `SELECT SUBSTRING(date, 0, 10) as sDate, sum(index) as max, first_value(lat) as lat, first_value(long) as long FROM largeMeter WHERE date LIKE ’2015-01%’ GROUP BY SUBSTRING(date, 0, 10), vid ORDER BY SUBSTRING(date, 0, 10), vid` | 92.00% | 99.54% | 99.96% |
| Showgraphcons | Obtain the consumption of meters in Rotterdam for the Jan. 2015. | `SELECT SUBSTRING(date, 0, 10) as sDate, sum(index) as max, vid FROM largeMeter WHERE city LIKE ’Rotterdam’ AND date LIKE ’2015-01-%’ GROUP BY SUBSTRING(date, 0, 10), vid ORDER BY SUBSTRING(date, 0, 10), vid` | 99.99% | 99.55% | 99.99% |
| ShowPiemonth | Obtain consumption for a specific subset of state consumption. | `SELECT SUBSTRING(date, 0, 10) as sDate, state as vid, sum(index) as max FROM largeMeter WHERE state LIKE ’U%’ AND date LIKE ’2015-01-%’ GROUP BY SUBSTRING(date, 0, 10), state ORDER BY SUBSTRING(date, 0, 10), state` | 99.99% | 99.99% | 99.99% |
| ShowGraphHCHP | Obtain data for drawing peak versus shallow hour consumption. | `SELECT SUBSTRING(date, 0, 10) as sDate, vid, min(sumHC) as minHC, max(sumHC) as maxHC, min(sumHP) as minHP, max(sumHP) as maxHP FROM largeMeter WHERE state LIKE ’FRA’ AND date LIKE ’2015-01-%’ GROUP BY SUBSTRING(date, 0, 10), vid ORDER BY SUBSTRING(date, 0, 10), vid` | 99.99% | 99.94% | 99.99% |
| Showday | Get the data for displaying the consumption of any specified hour of a given month. | `SELECT SUBSTRING(date, 0, 13) as sDate, sum(index) as max, vid FROM largeMeter WHERE city LIKE ’Rotterdam’ AND date LIKE ’2015-01-%’ GROUP BY SUBSTRING(date, 0, 13), vid ORDER BY SUBSTRING(date, 0, 13), vid` | 99.99% | 99.99% | 99.99% |

Table 6.2a: Set of data intensive queries typically executed by GridPocket data analysts.



(a) Small dataset (50GB). (b) Medium dataset (500GB). (c) Large dataset (3TB).

Figure 6: Analysis of query speedup ($S_Q$) for different types of query data selectivity and dataset sizes.

- *Query data selectivity*: This metric describes the percentage of data that would not be necessary for executing a given query and can be discarded. Normally, this metric refers to the data discard of the entire dataset (i.e., the number of bytes discarded). However, in some points of the evaluation, we also use the term selectivity to refer to the percentage of discarded data corresponding to filtered columns or rows by a query (column/row selectivity).
- *Query speedup ($S_Q$)*: This metric describes the relation between the execution time of a query with and without our pushdown technology as $S_Q = \frac{T_{no\_pushdown}}{T_{pushdown}}$. We measure execution times from a client perspective; it includes the time of ingesting data from Swift and the Spark SQL processing time. A value $S_Q > 1$ reflects a gain in performance by our pushdown technology, whereas a value $S_Q < 1$ means the opposite.

The relationship between these two metrics will let us understand the performance/cost trade-off of our pushdown technology as well as the situations in which it outperforms other technologies.

**Datasets**: The datasets used in this evaluation are anonymized versions of CSV files containing energy consumption values captured by 10K GridPocket smart energy meters. Anonymized datasets have exactly the same structural characteristics as the original ones, which means that from the viewpoint of our performance measurements, anonymization has no effect. In our experiments we make use of three dataset sizes: *Small*: 438 million rows (50GB); *Medium*: 3,900 million rows (500GB);

*Large*: 21,099 million rows (3TB). All these datasets have identical structure, with 10 columns, and every row represents a reading taken every 10 minutes. We also created a tool to generate synthetic data that mimics the structural properties of GridPocket's datasets[8].

**Queries**: First, we employ a set of real life SQL queries typically used by data scientists in the GridPocket platform to analyze the feasibility of our pushdown implementation. The queries that have been selected are data intensive and their data selectivity percentages are shown in Table 6.2a.

Apart from real SQL queries, we performed additional experiments to understand the behavior of our pushdown system. To this end, we executed synthetic queries on GridPocket datasets with controlled fractions of data selectivity. In particular, we executed specific experiments to analyze the impact of *row*, *column* and *mixed* data selectivity.

Moreover, for the sake of statistical significance, the results shown for each query are based on at least 15 executions.

**Platform**: We executed our experiments in the OpenStack Innovation Center (OSIC) testbed [32]. The platform consists of 63 servers (HP DL380 Gen9) equipped with 2X 12-core Intel E5-2680 v3 @2.50GHz, 256GB RAM, 12X 600GB 15K SAS - RAID10 and Intel X710 Dual Port 10 GbE NICs. The organization of the servers is as follows: i) An identity manager machine running Keystyone (Mitaka version); ii) 1 HA-Proxy load balancers backed up with VRRP; iii) 6 Swift proxy/metadata servers (Mitaka version); iv) 29 object servers (Mitaka version); v) 25 Spark v1.6 workers, a Spark (standalone) master node, and a Spark client driving the experiments. To facilitate large deployments, the software of Spark workers was running in Docker containers managed via Zoe[9]. All the nodes in the cluster run collectd[10] v5.4 in background to get resource usage metrics (CPU, memory, network).

The 3-replica object-ring was defined over 10 disks in each of the 29 nodes (290 disks altogether). The container and account rings were defined over 10 disks in each of the 6 proxies (60 disks altogether). All nodes were configured with a master-master bond over 2X10Gbps ports. The bonds were used to setup a data network to serve all replication as well as workload traffic.

## 7.1 Performance Analysis

Next, we show a performance analysis of synthetic SQL queries of controlled data selectivity executed with/without pushdown. Concretely, we focus on: i) the impact of the *query data selectivity* on performance, ii) the role of *selectivity type* (row/column/mixed) and, iii) the *dataset size*.

First, Fig. 6 shows that pushdown exhibits higher speedup values as the query data selectivity increases; more interestingly, such increase in performance is *superlinear with data selectivity*. To illustrate this, in Fig. 6(b) we clearly see that a query data selectivity of 80% provides a $S_Q \approx 5$, whereas discarding 90% of the dataset pushdown achieves $S_Q > 10$. In this sense, Fig. 7 shows in more detail query speedup results for very high data selectivity. Clearly, this scenario is favorable to
pushdown queries with high percentages of data selectivity may benefit from execution times up to 31 times shorter than the traditional "ingest-then-compute" approach. The reason for this behavior lies deeply on the type of bottleneck at hand: For low data selectivity, the network load balancer is the bottleneck, so an increment of data selectivity provides a near-linear performance improvement. However, we found that from $\approx 60\%$ of data selectivity onwards, the bottleneck progressively shifts from the network to the computational power of storage nodes. That is, for high data selectivity, the amount of data transferred to the compute cluster does not saturate the network, but utilizes significant compute resources from Swift storage nodes.

Overall, we observed that real life queries executed by GridPocket data scientists (Table 6.2a) exhibit query data selectivity values $> 90\%$ from the total dataset. This indicates that our pushdown technology can translate into a practical system to mitigate the ingest-then-compute problem in industrial environments.

Moreover, our pushdown technology archives significant query speedup even for *moderate percentages of data selectivity*. To wit, considering a mixed query data selectivity of 60%, we see that for

[8]`https://github.com/gridpocket/project-iostack`
[9]`http://zoe-analytics.eu`
[10]`https://collectd.org/`

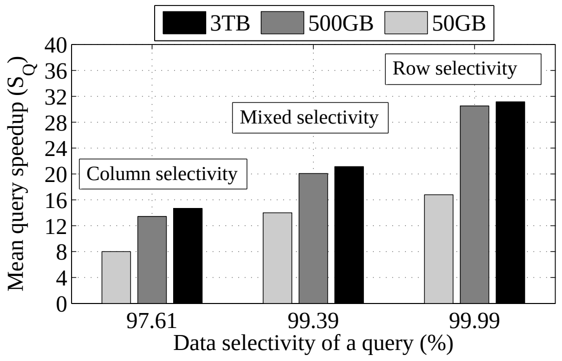


Figure 7: Query speedup results ($S_Q$) for high data selectivity.

the 50GB dataset pushdown exhibits a $S_Q = 2.25$, whereas for the 3TB dataset its performance is $S_Q = 2.35$. In other words, in this setting queries experienced an absolute improvement in execution time of 40.92 and 2631.56 seconds for the 50GB and 3TB datasets, respectively.

We are also interested in observing the behavior of our pushdown technology for low data selectivity. Appreciably, Fig. 6 depicts that our pushdown technology presents performance values $S_Q \approx 1$ for data selectivity values close to 0. That is, for no data selectivity —i.e., ingest the entire dataset— we registered a worst-case mean speedup penalty of 3.4% compared to plain Spark/Swift. In terms of performance, this suggests that our pushdown technology can efficiently handle workloads formed by queries of high and low data selectivity.

We verified that pushdown *does not exhibit higher performance variability* than vanilla Spark/Swift. For instance, the standard deviation values of query execution times for the 50GB (mixed selectivity) dataset range between $0.43 - 2.04$ and $0.05 - 1.33$ for plain Spark/Swift and when pushdown is used, respectively. We noticed a similar behavior for other types of selectivity and dataset sizes. This means that our approach for discarding data at the object store does not introduce additional performance variability.

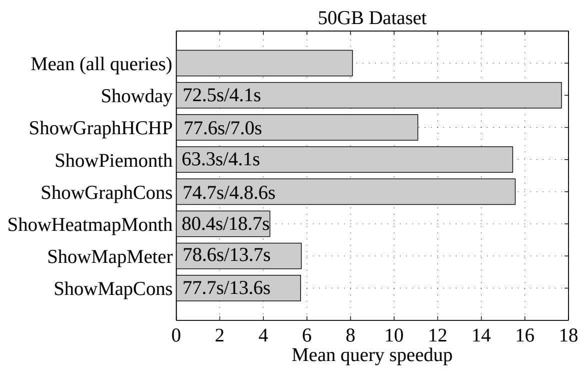
Interestingly, for high data selectivity percentages, pushdown behaves differently depending on the *dominant type of selectivity*. Fig. 6 shows that, in general, row selectivity exhibits higher performance compared to column/mixed selectivity. A reason for this behavior may reside on our `CSV Storlet` implementation; discarding an entire row by evaluating a condition may be more efficient than selecting and concatenating multiple columns in the output stream.

Finally, in Fig. 6 we analyze the impact of the dataset size. In our experiments, pushdown exhibits higher query speedup values *for larger datasets*. For instance, for 90% column selectivity, $S_Q = 6.72$ and $S_Q = 12.51$ for the 50GB and 3TB datasets, respectively. Fig. 7 illustrates this phenomenon more clearly, for high data selectivity rates. The reason is related to the testbed infrastructure at hand; queries executed against the 50GB dataset did not fully utilize the network and storage resources, unlike the case of larger datasets. This is supported by the fact that the performance increase between 500GB ($S_Q = 10.23$) and 3TB datasets is smaller.

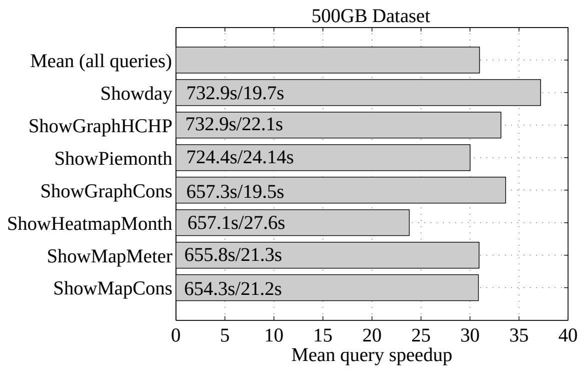
## 7.2 Real use-case: GridPocket SQL queries

In a battery of experiments (Fig. 8), we demonstrate the performance benefits of our pushdown technology for typical data intensive SQL queries executed by GridPocket data scientists (see Table 6.2a).

Fig. 8 shows the speedup of GridPocket queries for various datasets. Appreciably, for a small dataset, our pushdown technology achieves query speedups ranging from $S_Q = 4.1$ to $S_Q = 18.7$. Such differences in speedup for a given query are due to its percentage of data selectivity. That is, in Fig. 8(a) the fastest query (`ShowDay`) exhibits a data selectivity of 99.99%, whereas for the slowest ones the data selectivity is 92.05%. Further, in line with our previous observations, Fig. 8(a) demonstrates that for a larger dataset pushdown achieves faster query execution times. Moreover, the speedup differences among queries are less important.



(a) Small dataset (50GB).



(b) Medium dataset (500GB).

Figure 8: Analysis of query speedup ($S_Q$) for real GridPocket queries over various dataset sizes. Horizontal bars have an $x$ / $y$ annotation where $x$ represents the mean query execution times when using the traditional "ingest-then-compute" approach and $y$ the mean query execution times when using the "pushdown" approach.

As one can infer, for a company like GridPocket these improvements are significant. That is, in the case that each query requires to import a different 500GB dataset to the compute cluster, the total execution time of the set of queries in Fig. 8 is $4,814.7$ seconds. With developed pushdown technology data scientists in GridPocket could execute the same set of queries only in 155.48 seconds. This results represents a key step towards efficient analytics in a commercial setting.

## 7.3 Pushdown vs Parquet

Next, we compare our pushdown technology with other technologies that also mitigate the ingest-then-compute problem. Concretely, we perform a comparison with Apache Parquet [33]. Parquet is a format to store large datasets that provides two main benefits: i) It stores the data in a column-wise manner, so that it is possible to efficiently discard entire dataset columns; ii) Parquet stores highly optimized compressed data, which reduces the volume of network transfers. Note that Spark is in charge of carrying out the tasks of (de)compressing data and discarding columns in Parquet format.

Fig. 9 shows query speedup values for both pushdown and Parquet. Compared with ingesting data from plain Swift, Parquet offers significant speedups for very low query data selectivity. The explanation is simple: Importing compressed data from Swift makes the ingestion phase shorter, which is the dominant cost in the queries executed. Besides, for data selectivity of 0, Spark does not need to execute computations to discard columns in Parquet format, which may also represent an additional cost. Observably, the computation costs associated with Parquet seems to offer a better trade-off either when data compression is more beneficial (no data selectivity) or when the fraction of data selectivity is high.

Fig. 9 shows that pushdown query speedups exhibit a different behavior than with Parquet. For no data selectivity, our pushdown technology provides no performance benefit to the system, as no data can be discarded. Nevertheless, as we observed before, this experiment confirms that for higher fractions of data selectivity the pushdown technology achieves superlinear $S_Q$ values.

Our experiments show that pushdown exhibits higher performance than Parquet for data selectivity $\geq$ 60% using the 50GB datasets. For instance, for 90% data selectivity queries run with pushdown are 2.16x faster than queries executed with Parquet. Note that the pushdown technology achieves even better performance for mixed or row data selectivity, which cannot be shown as they are not supported by Parquet. Our experiments also indicate that the data selectivity threshold in which the pushdown technology outperforms Parquet is smaller for larger datasets.

We conclude that for SQL queries with high data selectivity —as the ones executed in GridPocket use cases— the pushdown technology provides higher query acceleration than Parquet. Moreover,

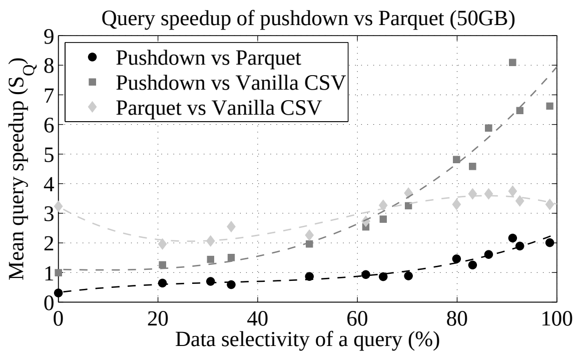
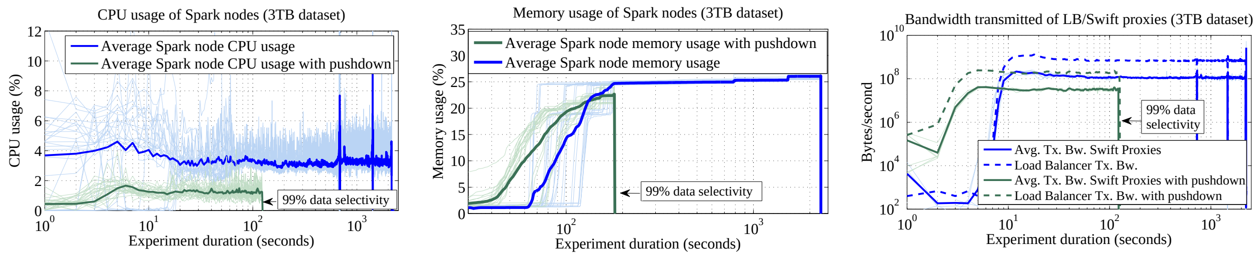


Figure 9: Performance comparison of Pushdown and Parquet for column selectivity.



(a) CPU usage of Spark nodes. (b) Memory usage of Spark nodes. (c) Transmitted bandwidth of load balancer and Swift proxies.

Figure 10: Resource usage of Spark nodes in the compute cluster and the inter-cluster network with and without pushdown .

as our compute layer in Swift can accommodate general-purpose computations, we will explore intelligent combinations of data filtering and compression for low data selectivity queries.

## 7.4 Resource Usage

In what follows, we analyze how the pushdown technology trades-off spare compute power at the object store to minimize resource usage in a shared compute cluster and an over-subscribed inter-cluster network. In particular, Fig. 10 compares the resources consumed —from the compute cluster viewpoint— executing a GridPocket query (`ShowGraphHCHP`, 99% data selectivity) on the 3TB dataset with and without pushdown.

Fig. 10(a) shows that pushdown achieves CPU savings at the compute cluster; first, the average CPU consumption of Spark nodes to execute the given query is less than half when pushdown is used ($\approx$ 1.2%) compared to plain Spark/Swift ($\approx$ 3.1%). Second, and perhaps more importantly, if we consider the experiment execution time, the pushdown technology reduces the number of CPU cycles for 97.8% to compute that query. These benefits are directly related to the fact of filtering data at the storage side, which shortens the experiment and avoids Spark to execute data filtering prior to the actual computations.

Fig. 10(b) illustrates the memory consumption at the compute cluster, which is a valuable resource shared across many jobs. As with CPU, the pushdown technology also provides significant memory savings to Spark. At the peak, the average memory usage of Spark nodes is around 13.2% lower for the pushdown technology than using vanilla Swift. The main reason for which memory savings are not higher is because Spark discards useless data prior to the computation of the SQL query. In addition, Fig. 10(b) shows that such amount of memory is kept in use for a period of time 12-15x longer than when using pushdown, which may prevent the concurrent allocation of new incoming jobs.

Interestingly, Fig. 10(c) points out that the network was the bottleneck for the ingestion of data.

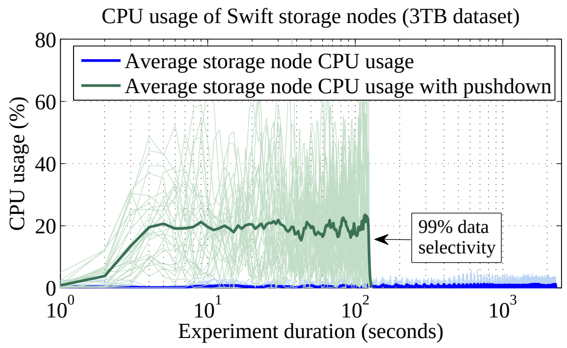


Figure 11: CPU utilization of Swift storage nodes with and without pushdown .

To inform this argument, the machine acting as load balancer was using a 10Gbps link to transfer data between storage and compute clusters. Observably, for plain Spark/Swift the 10Gbps link of the load balancer machine was close to saturation during the data ingestion phase of the query; to wit, Fig. 10(c) shows that the transmitted bandwidth to the compute cluster was close to 10Gbps. It is also visible that Swift proxy nodes were responsible for saturating the load balancer serving parallel requests of Spark tasks.

In contrast, Fig. 10(c) reveals that the pushdown technology heavily offloads the inter-cluster network. Both the load balancer and the Swift proxies only serve a small fraction of the total data and for a much shorter period of time. That is, Fig. 10(c) shows that the load balancer exhibited an average transmission bandwidth of 189MBps to the compute cluster, and only during $\approx$ 120 seconds. Therefore, when using the pushdown technology both the datacenter network and Swift proxies have more resources to serve other jobs or services running in the system.

Naturally, all these benefits at the compute and network level come at the cost of using compute power at the object store. In terms of CPU, Fig. 11 shows that the pushdown technology consumes on average a 23.5% of storage nodes CPU to execute projections/selections on the 3TB dataset, whereas this resource is almost totally idle in plain Swift (average CPU usage of 1.25% in storage nodes). Regarding memory, analyzing the execution of multiple SQL queries with pushdown we observed that Swift storage nodes exhibited a near constant memory usage between $4\% - 6\%$. Both CPU and memory overheads are related to the Docker container used to run Storlets plus the code execution.

We conclude that our pushdown technology exhibits an attractive resource usage trade-off; it incurs affordable CPU/memory overhead on Swift storage nodes in exchange of high query performance acceleration and significant reduction of resource usage at the compute cluster and the inter-cluster network.

## 8 Discussion and future steps

By implementing the Spark SQL pushdown use-case, we demonstrated that the concepts behind our pushdown technology are powerful enough to accelerate and make more efficient analytics queries in disaggregated clusters. Our present and future research will follow at least one of the following directions:

- *Generalize cooperation between analytics frameworks and object stores*
- *Towards adaptive pushdown execution*
- *Implement a complementary technology of data reduction*

as detailed in the following.

**Generalize cooperation between analytics frameworks and object stores**. Boosting ingestion with SQL pushdown techology leverages Spark-SQL data sources API to delegate projection and selection filtering tasks to the object store. SQL projections and selections, are very important since direct SQL queries are not only by themselves, a very important use case, but also, higher level libraries such the mllib Spark library implicitly issue SQL queries. However, SQL queries are only a particular case of the computations that our pushdown technology can carry out in the object store. And since at the storage side, our pushdown technology provides a rich substrate to execute parallel streamline computations on data objects, it is natural to work at generalizing the pushdown concept to other big data use cases.

From the analytics viewpoint, any computation which follows the map-reduce like pattern:

- In a first phase (map), a set of sub-tasks can be run independently, each on its specific subset of input dataset (these tasks are the push-down candidates)
- In a second phase (reduce), all the output of the first phase are merged in some way

could be pushed down in a way similar to that of the SQL pushdown.

The capabilities of our pushdown technology at the object store inspired us to generalize the delegation of computations described in this document to solve many other problems beyond SQL pushdown. In fact, we already extended the Spark RDD [34] to allow the developer writing Spark tasks that explicitly invoke computations at the object store via simple primitives. Thus, our new RDD: i) Provides programmatic means to explicitly execute Storlets in OpenStack Swift from the code of a Spark task; ii) holds the Storlet invocations output as its distributed dataset; and iii) embeds the knowledge of partitioning the input dataset to parallel tasks. Our current work is to research how this general form of task offloading may optimize other Spark analytics jobs.

Another way to generalize our present pushdown prototype is to handle data kept in non textual formats: Unlike other stores connected to Spark, that usually hold specific or limited data formats (e.g. [35], [36], [37]), object stores are not limited in the types and data formats they can store. Hence, one can imagine different types of Spark jobs ingesting information from non-textual data thanks to developed pushdown filters; examples include bringing EXIF metadata from JPEGs or text from PDF documents. Besides, to delegate tasks to the object store on textual data, in this work we have modified Spark's Hadoop RDD. However, in addition to the complexity of Hadoop APIs, this approach partitions the dataset according to the underlying HDFS chunk size. While natural for HDFS, the chunk size is not adapted to object stores. In object stores it seems more adequate to partition according to, for instance, the number of replicas and the compute parallelism available in the nodes.
Our current work addresses these drawbacks [34]; we provide a Spark-CSV alternative that makes use of a new RDD implementation, which is well aware of the `CSVStorlet` output. More generally, the idea behind [34] is to pair a Storlet that does a certain function, e.g. extract textual metadata from a binary object, to an appropriate RDD that is Storlet-aware. This approach makes it possible to extend the pushdown concept to additional non-textual data formats, broadening the scope of the applications of our pushdown technology With [34], the whole Hadoop layer can be bypassed by calling from the Spark-CSV layer directly.

**Towards adaptive pushdown execution**. With our technology, an administrator can deploy pushdown filters on the object store and enforce them on a particular tenant's requests. However, this decision is static, meaning that the fact of enforcing a pushdown filter is done without taking into account the workload conditions. It is not hard to imagine that, under peak workloads and CPU/-parallelism constraints at the object store, an administrator may decide that only "gold" tenants enjoy the pushdown service, whereas "bronze" tenants will ingest data in the traditional way. We can also imagine that the effectiveness of the filter could be modeled —e.g., in the SQL pushdown filter by approximating the data selectivity— and contribute to the decision of whether the pushdown filter should be applied or not. Clearly, the system should dynamically take these decisions based on real-time monitoring information and transparently to the administrator.

This goal can be reached thanks to the IOStack developed Crystal software[11], which implements a control architecture that can dynamically orchestrate the execution of Storlets in OpenStack Swift [29]. Similarly to [38], our future work encompasses the design of execution cost models for pushdown filters into control processes. Such control processes will take as input workload or resource metrics from the storage cluster to decide on runtime whether to execute a pushdown filter or not for a specific tenant.

**Implement a complementary technology of data reduction for Spark SQL**. The SQL pushdown data reduction technology permits to reduce data from a set of objects by applying the SQL filter at the data source against each of the objects, thus potentially reducing data for each of the objects pertaining to the data set. Another very different and complementary way to pursue data reduction for SQL is to detect objects as non-relevant and filter them out (not even reading them). The Hive partitioning technology[39] goes along this philosophy, however it has its own limitations both in terms of flexibility (e,g,. data has to be partitioned ahead of time in a manner that will fit the implicit partitioning of future request, also this method does not fit for columns containing data pertaining to a non discrete value set) and effectiveness (e.g., objects may still have be read although not relevant). The basic idea is to improve this technology by enriching the meta-data of each object with lightweight statistics which describe its various columns. This meta-data will permit to filter out the object (without having to access it altogether) in many cases where it will not be relevant to a future request. As for the SQL pushdown technology, the benefits of this data reduction method extend beyond reducing file/object access to reducing network load, improving task scheduling and overall performance. Such a technology nicely complements the SQL pushdown technology to handle the situations when data sets are composed of many small to medium size objects for which the meta-data can be enriched ahead of the query time.

## 9 Conclusions

In this document we describe the current state of the technology developed by the IOStack consortium within the framework of WP4. We describe a novel solution that mitigates the problems of executing analytics in disaggregated compute and storage clusters by exploiting the computational capabilities of object stores. This technology addresses this challenge by enabling analytics frameworks to delegate ETL-type and querying functions to the object store, which is in turn equipped with a rich and flexible active storage layer. We instantiated this concept by enabling Apache Spark SQL to offload projections and selections to OpenStack Swift, in order to execute them close to the data. Our experiments in a production cluster with real-world datasets and SQL workloads demonstrate that our developed technology is a practical solution for providing faster and more efficient analytics in disaggregated clusters.

[11]`http://crystal-sds.org`

## 10 Code Related Appendix

This appendix describes the various software parts that were produced or modified in WP4:

1. Stocator (WP4 produced)
2. Joss library (feature added)
3. The Storlet middleware (Open sourced + feature added)
4. Spark-csv modifications (WP4 produced)
5. Spark modifications (prototype code) (WP4 produced)
6. Spark modifications (new code) (WP4 produced)
7. Hadoop modifications (WP4 produced)
8. The CSV Storlet (WP4 produced)

### 10.1 Stocator

| | |
|---|---|
| Web page | https://spark-packages.org/package/SparkTC/stocator |
| Source Code | https://github.com/SparkTC/stocator |
| Documentation | https://github.com/SparkTC/stocator/blob/master/README.md |
| Continuous Integration | https://travis-ci.org/SparkTC/stocator/ |

Stocator, an Spark to Object stores connector, has been created for WP4 in 2015 and has become in 2016 the first industrial class connector between Spark and Swift (as of 2016, Stocator became the default connector between Spark and Object stores for the BlueMix "Spark on demand" service. As explained in the following, it is more general and can been extended to other Object Stores.

Apache Spark can access multiple data sources that include object stores like Amazon S3, OpenStack Swift, IBM SoftLayer, and more. To access an object store, Apache Spark uses Hadoop modules that contain drivers to the various object stores.

Apache Spark needs only small set of the object store functionalities. Specifically, Apache Spark requires the following operations: listing the containers, listing the objects of a given container creation, object read, and getting data partitions. Hadoop drivers, however, must be compliant with the Hadoop eco system. This means they support many more operations, such as shell operations on directories, including move, copy, rename, etc. which are not native object store operations. Moreover, Hadoop Map Reduce Client is designed to work with file systems and not with object stores. The temporary files and folders it uses for every write operation are renamed, copied, and deleted. All this leads to dozens of useless requests targeted at the object store. It's clear that Hadoop is designed to work with file systems and not object stores.

Stocator, although implementing the Hadoop is implicitly designed for the object stores, it has very a different architecture from the existing Hadoop driver. It does not depend on the Hadoop modules and interacts directly with object stores.

Stocator is a generic connector, that may contain various implementations for object stores. It was initially provided with complete Swift driver, based on the JOSS package, however it can be very easily extended to more object store implementations.

Installation and user manual can be found on line as URL mentioned in previous table.

### 10.2 Joss library modifications

| | |
|---|---|
| Web page | http://joss.javaswift.org/ |
| Source Code | https://github.com/javaswift/joss |
| Documentation | https://github.com/javaswift/joss/blob/master/README.md |

Joss is a java library code which provides a Java client to Swift. Stocator uses Joss to access Swift.

We added to Joss the ability to add client defined headers to REST requests to Swift. This was critical since it enabled to add the necessery requests which permit to run the CSVStorlet for the Spark SQL pushdown use case. This work has been a contribution of a related EU project (ForgetIT) as detailed in the following patch https://github.com/javaswift/joss/commit/7ab086490085ef2ac924ef4dd60c858767

The added patch supports adding general headers to Swift object upload and download operations. Example: suppose you use Swift Storlets and you want to run a storlet on a download operation. You can use addHeader to add one header specifying the storlet you wish to run, and other headers to specify the storlet parameters:

```
Header header1 = new GeneralHeader(
        "X-Run-Storlet",
        "SomeStorletJarFile.jar");
Header header2 = new GeneralHeader(
        "X-Storlet-Parameter-1",
        "SomeParameterName:SomeParameterValue");

DownloadInstructions downloadInstructions =
        new DownloadInstructions().addHeader(header1).addHeader(header2);
}
```

### 10.3 The Storlet Framework

| | |
|---|---|
| Web page | https://wiki.openstack.org/wiki/Storlets |
| Source Code | https://github.com/openstack/storlets |
| Documentation | http://storlets.readthedocs.io/ |
| Tests | Unit tests and functional tests |
| Continuous Integration | https://github.com/openstack-infra/ project-config/blob/master/jenkins/jobs/projects.yaml |

In WP4, we operated the following changes and additions to the Storlet framework:

- Open source the project (previously an IBM software)
- Upgrade to an Openstack informal project
- Upgrade to an official Openstack project (as of the coming micata OpenStack release)
- Multiple fixes
- Addition of a critical feature for Big data analytics: enable storlets to run at Object nodes when byte ranges are specified

Up to June 2016, the storlet middleware gave ability to run storlets when byte ranges were specified, however the storlet was to be run at one of the proxy servers. It soon became clear that this was a very big performance hit, and this for the following reasons:

- Spark tasks are typically assigned byte ranges and issues REST object GET requests towards object stores that specify byte ranges

- When a byte range GET request has to be processed by a Swift proxy node, the totality of the object has to be sent from one of the object nodes to the proxy node, therefore numerous concurrent such request easily can disrupt the internal network of the object store.
- Typically a Spark SQL query against a large data set will spawn many concurrent Storlet requests, therefore if the storlets are run on the not so numerous proxy nodes, their CPU resources will easily be bogged down, whereas when this same number of storlets are spread over the object nodes, we can see a much milder impact in term of CPU load at the swift nodes.

### 10.4 Spark-csv modifications

| | |
|---|---|
| Source Code | https://github.com/iostackproject/Scoop-csv-sql-pushdown |
| Documentation | https://github.com/iostackproject/Scoop-csv-sql-pushdown /blob/master/README.md/ |

In the first paragraph of sub-section 6.1, we gave a detailed description of the changes that we brought to this package. The Documentation is pointed to in previous table

### 10.5 Spark modifications (working prototype code)

| | |
|---|---|
| Source Code | https://github.com/iostackproject/Scoop-csv-sql-pushdown |
| Documentation | https://github.com/iostackproject/Scoop-csv-sql-pushdown /blob/master/README.md/ |

This code displays three scala file that were modified to enable the pushdown functionality:

- HadoopRDD.scala
- RDD.scala
- MapPartitionsRDD.scala

All of these classes are in the core/src/main/scala/org/apache/spark/rdd directory.

### 10.6 Spark modifications (new code)

| | |
|---|---|
| Source Code | https://github.com/eranr/spark-storlets |
| Documentation | https://github.com/iostackproject/ Scoop-csv-sql-pushdown/blob/master/README.md |

This code is similar to the prototype code except that it defines the CsvStorletRdd class as extending RDD class and not HadoopRDD. This has the following consequences:

- The code had not to mingle with the Hadoop code
- The code is mostly independent of the Spark code
- It connects to Swift through a direct usage of Joss
- Last but not least, it is no longer dependent of the "HDFS chunk size" which indeed is of no relevance for Object Stores but which for legacy reasons commands the number of partitions generated for a given data set.

### 10.7 Hadoop modifications (working prototype code)

| | |
|---|---|
| Source Code | https://github.com/iostackproject/Scoop-csv-sql-pushdown |
| Documentation | https://github.com/iostackproject/Scoop-csv-sql-pushdown /blob/master/README.md/ |

This code displays the two java files that were modified to enable the pushdown functionality:

- FileInputFormat.java
- LineRecordReader.java

These two classes are in the hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-core/src/main/java/org/apache/hadoop/mapred directory. and permit to patch the hadoop 2.7.1 code.

When testing an earlier version of the pushdown code, we discovered that the results of the SQL queries were not always exact, this was due the fact that the regular hadoop code was filling the gap between logical records (what should be done) and the fact that in general the byte range that defines a Spark partition cuts the last logical record, by doing an additional read to complete the generally last record which was broken. This code could not work when the SQL filter logics was pushed down to the data source side since the hadoop/Spark code had no clue of how to fix broken records. The main difference with the regular code is that when pushdown is used, we can not associate a received record to a given byte range of the object to which it pertains. In order to solve this tricky problem, we pushed the logics which permit to "repair" broken records to the object store side, basically by enabling the storlet to read more than the string byte range corresponding to a given partition. The FileInputFormat file comprises the fix which permitted to obtain with the pushdown code exact SQL results.

The whole project (that is comprising Spark CSV, Spark and Hadoop) can be built thanks to the BuildPushdownSpark.sh which has also been added to the same github repository.

### 10.8 CSVStorlet

| | |
|---|---|
| Web page | https://github.com/openstack/storlets/tree/master/StorletSamples/java/CsvStorlet |
| Source Code | https://github.com/openstack/storlets/tree/master/StorletSamples/java/CsvStorlet |
| Documentation | http://storlets.readthedocs.io/ |

This storlet is invoked automatically from the Spark side. Obviously the main job of the storlet is to implement the SQL column projection and the row selection. In addition the storlet solves a tricky problem which is comes from the fact that a Spark partition is defined as a physical byte range in general have a partial CSV record both at its beginning and its end. This problem, which also exists without pushdown is harder when pushdown is used where filtering is done at the Storlet side. We solved this problem by augmenting the byte range to be read by the storlet with a sufficiently long tail. This permits to the storlet to fix the problem as follows:

- in a first phase, the storlet will discard the prefix of the input byte stream till a record separator is read.
- in a second phase, the storlet will handle all the following logical records from the byte stream, while the total number of bytes is less than the Spark partition length (passed as a parameter)
- in a final phase, which starts when the number of read characters reaches the Spark partition length, the storlet will continue to read from the byte stream till an end of record character is encountered, process this last logical record and then stop to process the byte stream

The storlet is invoked by the Spark side mainly with following parameters:

- the string which defines the requested columns
- the string which defines the filter of the where clause to be run by the storlet
- the Spark partition length

## References

[1] "Apache spark adoption by the numbers." https://www.datanami.com/2016/06/08/apache-spark-adoption-numbers.

[2] "Apache spark survey 2016." https://databricks.com/blog/2016/09/27/spark-survey-2016-released.html.

[3] "Openstack storlets." https://github.com/openstack/storlets.

[4] "Amazon S3." https://aws.amazon.com/en/s3.

[5] "Openstack swift." http://docs.openstack.org/ developer/swift.

[6] "IBM Cleversafe." https://www.ibm.com/cloud-computing/products/storage/object-storage.

[7] C. Gkantsidis, D. Vytiniotis, O. Hodson, D. Narayanan, F. Dinu, and A. Rowstron, "Rhea: automatic filtering for unstructured cloud storage," in USENIX NSDI'13, pp. 343–355, 2013.

[8] M. Mihailescu, G. Soundararajan, and C. Amza, "Mixapart: decoupled analytics for shared storage systems.," in USENIX FAST'13, pp. 133–146, 2013.

[9] M. J. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre, "Continuous analytics: Rethinking query processing in a network-effect world.," in CIDR, 2009.

[10] T. Xiao, Z. Guo, H. Zhou, J. Zhang, X. Zhao, C. Ye, X. Wang, W. Lin, W. Chen, and L. Zhou, "Cybertron: Pushing the limit on i/o reduction in data-parallel programs," in ACM OOPSLA'14, vol. 49, pp. 895–908, 2014.

[11] J. M. Hellerstein and M. Stonebraker, "Predicate migration: Optimizing queries with expensive predicates," in ACM SIGMOD'93, vol. 22, 1993.

[12] E. Friedman, P. Pawlowski, and J. Cieslewicz, "SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions," VLDB Endowment, vol. 2, no. 2, pp. 1402–1413, 2009.

[13] R. Ananthanarayanan, K. Gupta, P. Pandey, H. Pucha, P. Sarkar, M. Shah, and R. Tewari, "Cloud analytics: Do we really need to reinvent the storage stack," in USENIX HotCloud'09, 2009.

[14] E. H. Wilson, M. T. Kandemir, and G. Gibson, "Will they blend?: Exploring big data computation atop traditional hpc nas storage," in IEEE ICDCS'14, pp. 524–534, 2014.

[15] W. Tantisiriroj, S. W. Son, S. Patil, S. J. Lang, G. Gibson, and R. B. Ross, "On the duality of data-intensive file system design: reconciling HDFS and PVFS," in ACM SC'11, p. 67, 2011.

[16] C. Xu, R. Goldstone, Z. Liu, H. Chen, B. Neitzel, and W. Yu, "Exploiting analytics shipping with virtualized MapReduce on HPC backend storage servers," IEEE Transactions on Parallel and Distributed Systems, vol. PP, no. 99, pp. 1–1, 2015.

[17] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum, "In-situ MapReduce for log processing," in USENIX ATC'11, p. 115.

[18] D. Tiwari, S. Boboila, S. S. Vazhkudai, Y. Kim, X. Ma, P. Desnoyers, and Y. Solihin, "Active flash: towards energy-efficient, in-situ data analytics on extreme-scale machines.," in USENIX FAST'13, pp. 119–132, 2013.

[19] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T.-A. Nguyen, J. Cao, H. Abbasi, S. Klasky, et al., "Flexio: I/o middleware for location-flexible scientific data analytics," in IEEE IPDPS'13, pp. 320–331, 2013.

[20] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki, "Diamond: A storage architecture for early discard in interactive search.," in USENIX FAST'04, vol. 4, pp. 73–86, 2004.

[21] E. Riedel, G. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia applications," in VLDB'98, pp. 62–73, 1998.

[22] R. Wickremesinghe, J. S. Chase, and J. S. Vitter, "Distributed computing with load-managed active storage," in IEEE HPDC'02, pp. 13–23, 2002.

[23] E. Jahani, M. J. Cafarella, and C. Ré, "Automatic optimization for MapReduce programs," VLDB'11, vol. 4, no. 6, pp. 385–396, 2011.

[24] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in USENIX NSDI'12, pp. 2–2, 2012.

[25] "Deep dive into spark SQL's catalyst optimizer." https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html.

[26] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark sql: Relational data processing in spark," in ACM SIGMOD'15, pp. 1383–1394, 2015.

[27] "Spark-CSV." https://github.com/databricks/spark-csv.

[28] D. V. D. C. P. M. Francesco Pace, Marco Milanesio, "Experimental performance evaluation of cloud-based analytics-as-a-service," in IEEE CLOUD'16, p. In press, 2016.

[29] R. Gracia-Tinedo, P. García-López, M. Sánchez-Artigas, J. Sampé, Y. Moatti, E. Rom, D. Naor, R. Nou, T. Cortés, and W. Oppermann, "Iostack: Software-defined object storage," IEEE Internet Computing, vol. 20, no. 3, pp. 10–18, 2016.

[30] S. Rabinovici-Cohen, E. A. Henis, J. Marberg, and K. Nagin, "Storlet engine for executing biomedical processes within the storage system," in Business Process Management (BPM'14), pp. 59–71, 2014.

[31] "Docker." https://docs.docker.com.

[32] "Openstack innovation center." https://osic.org.

[33] "Apache Parquet." https://parquet.apache.org.

[34] "Spark-storlets." https://github.com/eranr/spark-storlets.

[35] "Phoenix-spark." https://github.com/apache/phoenix/tree/master/phoenix-spark.

[36] "Spark-cassandra-connector." https://github.com/datastax/spark-cassandra-connector.

[37] "Spark-cloudant." https://github.com/cloudant-labs/spark-cloudant.

[38] C. Chen, Y. Chen, and P. C. Roth, "Dosas: Mitigating the resource contention in active storage systems," in IEEE Cluster'12, pp. 164–172, 2012.

[39] "Improving query performance using partitioning in apache hive." `http://blog.cloudera.com/blog/2014/08/improving-query-performance-using-partitioning-in-apache-hivel.`