Too Big to Eat: Boosting Analytics Data Ingestion from Object Stores with Scoop

Yosef Moatti
IBM Research Haifa (Israel)
moatti@il.ibm.com

Eran Rom*
OpenStack Storlets Project
eran@itsonlyme.name

Raúl Gracia-Tinedo
Universitat Rovira i Virgili (Spain)
raul.gracia@urv.cat

Dalit Naor
Doron Chen
IBM Research Haifa (Israel)
dalit|cdoron@il.ibm.com

Josep Sampé
Marc Sánchez-Artigas
Pedro García-López
Universitat Rovira i Virgili (Spain)
josep.sampe|marc.sanchez|pedro.garcia@urv.cat

Filip Gluszak
Eric Deschodt
GridPocket (France)
filip.gluszak@gridpocket.com
eric.deschodt@gridpocket.com

Francesco Pace, Daniele Venzano, Pietro Michiardi
Eurecom (France)
francesco.pace|daniele.venzano|pietro.michiardi@eurecom.fr

***Abstract*—Extracting value from data stored in object stores, such as OpenStack Swift and Amazon S3, can be problematic in common scenarios where analytics frameworks and object stores run in physically disaggregated clusters. One of the main problems is that analytics frameworks must ingest large amounts of data from the object store prior to the actual computation; this incurs a significant resources and performance overhead. To overcome this problem, we present *Scoop*. Scoop enables analytics frameworks to benefit from the computational resources of object stores to optimize the execution of analytics jobs. Scoop achieves this by enabling the addition of ETL-type actions to the data upload path and by offloading querying functions to the object store through a rich and extensible active object storage layer. As a proof-of-concept, Scoop enables Apache Spark SQL selections and projections to be executed close to the data in OpenStack Swift for accelerating analytics workloads of a smart energy grid company (GridPocket). Our experiments in a 63-machine cluster with real IoT data and SQL queries from GridPocket show that Scoop exhibits query execution times up to 30x faster than the traditional "ingest-then-compute" approach.**

I. Introduction

These days, object stores, such as Amazon S3 [1], OpenStack Swift [8] or IBM Cleversafe [5], 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 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.

While object storage is simple and scalable, it does not allow to directly execute queries to extract value from data as with traditional databases or frameworks that provide co-located storage and compute (e.g., Hadoop, Dryad). Instead, to extract value from object datasets, common analytics platforms (e.g., Amazon Elastic MapReduce) run computations on compute clusters that are physically disaggregated from the object store [20]. 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 [27], [18].

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

*Much of the work described in this paper was done while Eran Rom was working at IBM Research Haifa.

[1] http://www.gridpocket.com

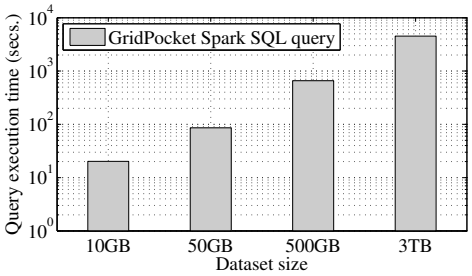


Fig. 1. Impact of the "ingest-then-compute" problem on GridPocket analytics.

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.

A. 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 functionalities of the system for handling new offloaded tasks.

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 functionality 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.

B. Contributions

To overcome the ingest-then-compute problem in disaggregated Big Data clusters, we present *Scoop*. Scoop 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. Scoop achieves this by offloading ETL-type and querying functions to the object store through a rich and extensible active object storage layer. Conversely to prior works [20], [35], the ultimate goal of Scoop is to extend the object store "on-the-fly" with new types of active storage functionalities. That is, administrators can install and deploy general-purpose code at the object store that can be then 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 [23], [19] into a disaggregated analytics ecosystem. Our Scoop 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 Scoop is publicly available (see Section V). Spark SQL interfaces with data source implementors, such as Spark CSV, with an API which permits to off-load selection and projection filtering tasks. Thus, while storlets allow to run arbitrary code on the object store, we concentrate on the filtering aspects of SQL queries.

To evaluate our system, we executed extensive experiments on a 63-machine cluster over real IoT data from GridPocket energy meters. Our results show that Scoop 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, Scoop enables GridPocket to benefit from the advantages of object storage, while making their analytics workloads much faster and more efficient.

In summary, the key contributions of our work are:

- Design of Scoop, a novel solution that exposes a parallel architecture to address the ingest-then-compute problem for data stored in object storage;
- Implementation of Scoop, 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 real life data from GridPocket workloads;
- The code of Scoop and the anonymized datasets are publicly available.

Roadmap: This paper is organized as follows. In Section II we discuss related work. Section III provides technical background for the rest of the paper. Section IV describes the design principles of Scoop, and Section V depicts our implementation to enable SQL predicate execution at the object store. In Section VI we present our validation framework and the results of our experiments. We discuss our future work and conclude in sections VII and VIII, respectively.

II. Related Work

The *ingest-then-compute* data life-cycle imposed by infrastructure disaggregation is a performance barrier for today's

data analytics frameworks [27]. 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 [14], [34], [31], or even replacing HDFS by optimized file systems to minimize the impact of disaggregation [36], [27]. 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 [26], [32], [38]. 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 [24], 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 [30], [33]. However, Diamond has not been targeted and applied on data analytics.

Regarding data analytics, Rhea [20] 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 [25] focuses on automatically optimizing MapReduce programs to reduce the data movements during the computation. More recently, Cybertron [35] combined data filtering with novel coding techniques to reduce IO overheads of analytics jobs.

A key difference between Scoop and these works lies in the storage layer. First, some of these works do not support actual data locality at the storage cluster as Scoop does; that is, Rhea [20] 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 Scoop goes far beyond this goal. Scoop 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 Scoop a more flexible active object storage system for analytics than prior works.

III. Technical Background

A. Spark Ecosystem

Apache Spark[2] 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.

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

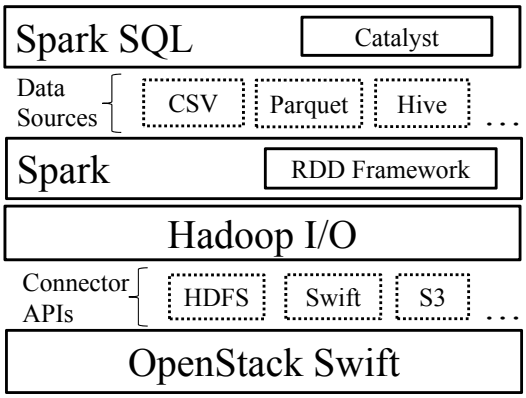


Fig. 2. Components in a Spark deployment to execute SQL queries on semi-structured data stored in an object store.

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) [37] 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 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 [3], [15]. 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.

B. 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 horizontal scaling of large deployments.

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[3] pipeline that enables developers to configure middlewares that intercept object requests with environment information.

IV. Scoop Design

The main goal of Scoop 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.

[3]https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface

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. Scoop achieves that by providing three abstractions: *pushdown task*, *analytics delegator*, and *pushdown filter*.

A. 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. In Scoop, 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 Scoop 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 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 provides 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. Scoop 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.

B. Scoop Architecture

Next, we present the design of Scoop on top of OpenStack Swift to overcome the ingest-then-compute problem in analytics platforms (see Fig. 3).

At the compute cluster, Scoop 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. 3, 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

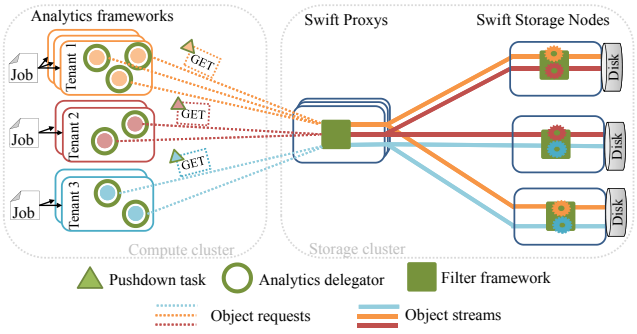


Fig. 3. Architecture of Scoop deployed on top of OpenStack Swift.

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, Scoop equips 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, Scoop 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, Scoop 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.

V. Implementation: Spark SQL Pushdown

In this section, we illustrate the implementation of Scoop and how we extended it to leverage SQL pushdown for Apache Spark on top of OpenStack Swift. As we show later on, our implementation enhances the daily operation of GridPocket, an energy grid company that executes typical IoT SQL workloads on semi-structured object-based datasets generated by smart energy meters. which often addresses IoT data sets too big to be cached in the Spark memory and which spawn recurring ingest bottlenecks.

A. Scoop Components

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

Specifically, we modified the Spark-CSV library [12], 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[4].

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[5], 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 [17]. 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 VII, 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: Scoop offers simple means for deploying and enforcing pushdown filters on a particular tenant or container via policies in OpenStack Swift [21]. Scoop 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 [7], [29], 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` [4]. 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 object node to one of the proxies instead of processing on the

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

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[6]. 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.

ETL often requires data transformations. Storlets permits this in the PUT data path. We use Storlet for data cleansing and for modifying the data format (e.g., split a column into multiple ones). These transformation simplify Spark workloads without requiring painful rewrites of huge data sets.

B. The Pushdown Process Flow

Next, we depict the workflow of Scoop 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. 4). 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. 4). 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 [12], 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 which can be accessed with the HDFS API.

Following the Hadoop RDD creation, a process called partition discovery takes place. This process involves partitioning

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

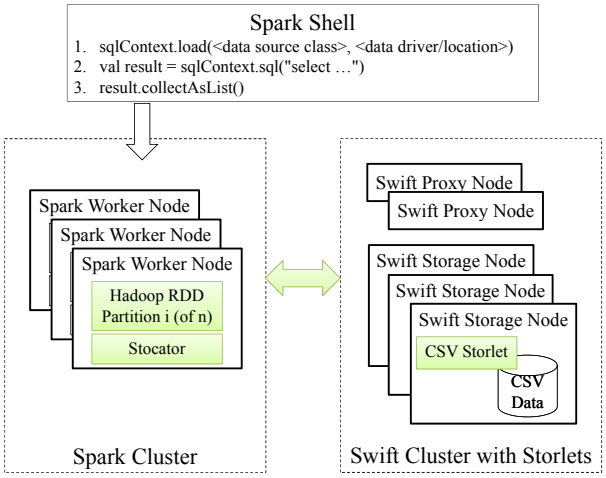


Fig. 4. Overview of Scoop's implementation to leverage Spark SQL pushdown for OpenStack Swift.

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 VII. 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. 4). At this point, Catalyst calculates the implied projection and selection filters and calls the appropriate 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. 4, 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 enables Scoop to greatly accelerate Spark SQL queries and mitigate the ingest-then-compute problem in disaggregated analytics clusters.

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 I. SET OF DATA INTENSIVE QUERIES TYPICALLY EXECUTED BY GRIDPOCKET DATA ANALYSTS.

VI. EXPERIMENTAL EVALUATION

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

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

Metrics: In our experiments, we make use of these metrics:

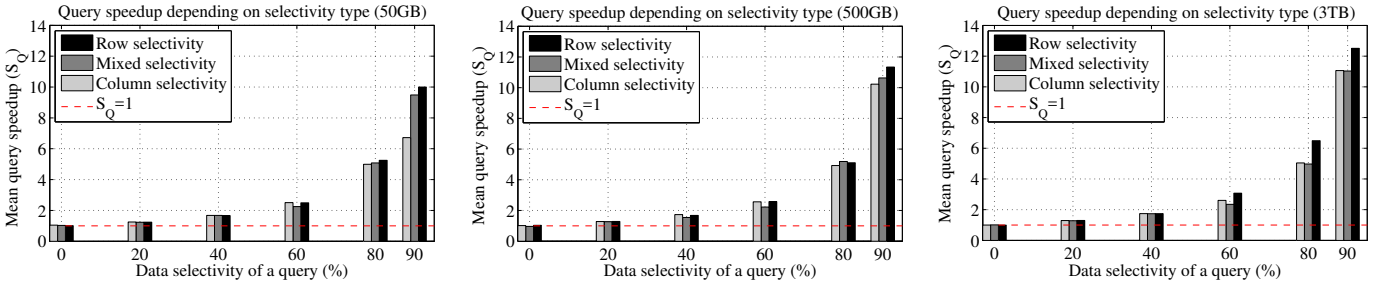
- *Query data selectivity*: This metric describes the percentage of data that would not be necessary for executing a given query and can be discarded. Normally, this metric refers to the data discard of the entire dataset (i.e., the number of bytes discarded). However, in some points of the evaluation, 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 Scoop as $S_Q = \frac{T_{no\_scoop}}{T_{scoop}}$. 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 Scoop, whereas a value $S_Q < 1$ means the opposite.
- *Resource usage (Network, CPU, Memory)*: We collect metrics that indicate the percentage of consumed machine resources when executing a query. We record these metrics in both compute and storage clusters.

The relationship between these metrics will let us understand the performance/cost trade-off of Scoop, 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. Also the data, upon being uploaded into the object store, was cleansed by an ETL storlet for which we do not evaluate the performance since our main objective is to measure and demonstrate query accelerations. 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[7].

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 I.

[7]https://github.com/gridpocket/project-iostack



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

Fig. 5. Analysis of query speedup (S_Q) for different types of query data selectivity and dataset sizes.

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 [6]. 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[8] [28]. All the nodes in the cluster run collectd[9] 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.

A. Scoop Performance Analysis

Next, we show a performance analysis of synthetic SQL queries of controlled data selectivity executed with/without Scoop. 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. 5 shows that Scoop 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. 5(b) we clearly see that

[8]http://zoe-analytics.eu
[9]https://collectd.org/

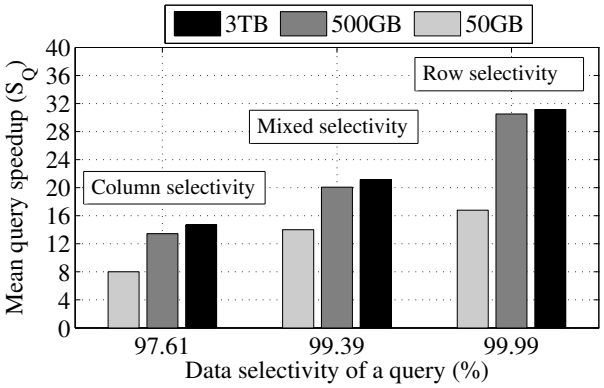
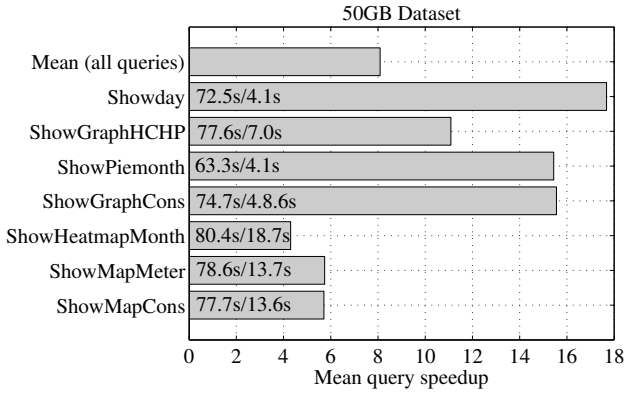


Fig. 6. Query speedup results (S_Q) for high data selectivity.

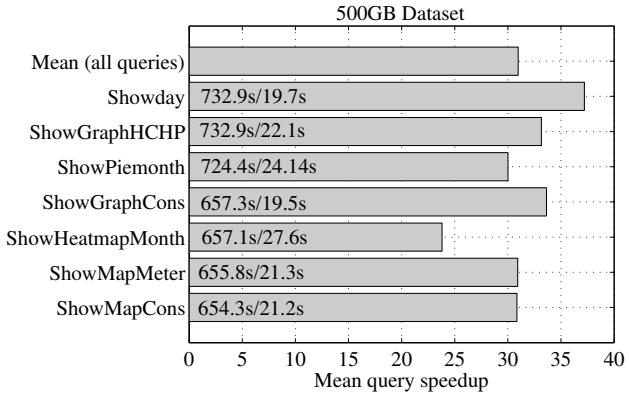
a query data selectivity of 80% provides a $S_Q \approx 5$, whereas discarding 90% of the dataset Scoop achieves $S_Q > 10$. In this sense, Fig. 6 shows in more detail query speedup results for very high data selectivity. Clearly, this scenario is favorable to Scoop; queries with high percentages of 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 I) exhibit query data selectivity values $> 90\%$ from the total dataset. This indicates that Scoop is a practical system to mitigate the ingest-then-compute problem in industrial environments.

Moreover, Scoop achieves significant query speedup even for *moderate percentages of data selectivity*. To wit, considering a mixed query data selectivity of 60%, we see that for the 50GB dataset Scoop 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.



(a) Small dataset (50GB).



(b) Medium dataset (500GB).

Fig. 7. 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.

We are also interested in observing the behavior of Scoop for low data selectivity. Appreciably, Fig. 5 depicts that Scoop 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 Scoop can efficiently handle workloads formed by queries of high and low data selectivity.

We verified that Scoop *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 Scoop, 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, Scoop behaves differently depending on the *dominant type of selectivity*. Fig. 5 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. 5 we analyze the impact of the dataset size. In our experiments, Scoop 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. 6 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.

B. Real use-case: GridPocket SQL queries

Next, we analyze the performance benefits of Scoop for typical data intensive SQL queries executed by GridPocket data scientists (see Table I). To this end, Fig. 7 shows the speedup that Scoop achieved in these queries for various dataset sizes (for clarity, bars in Fig. 7 also depict the "original"/"pushdown" absolute query times in seconds).

Appreciably, for a small dataset, Scoop 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. 7(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. 7(a) demonstrates that for a larger dataset Scoop achieves faster query execution times. Moreover, the speedup differences among queries are less important.

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. 7 is $4,814.7$ seconds. With Scoop, 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.

C. Pushdown vs Parquet

Next, we compare Scoop with other technologies that also mitigate the ingest-then-compute problem. Concretely, we perform a comparison with Apache Parquet [2]: a columnar storage format which can be used with Apache Spark and object stores. It provides two main benefits: i) Being columnar, it is possible to efficiently perform column projection; 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. 8 shows query speedup values for both Scoop 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

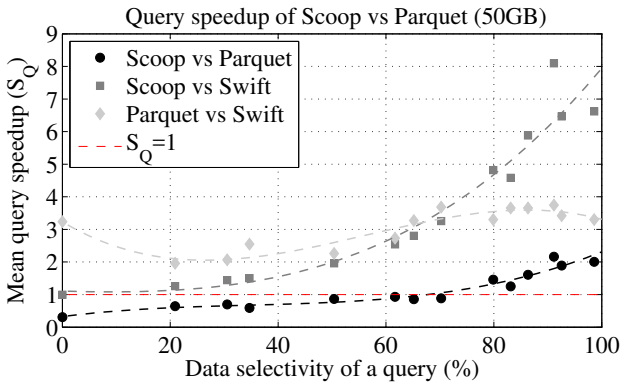


Fig. 8. Performance comparison of Scoop and Parquet for column selectivity.

compression is more beneficial (no data selectivity) or when the fraction of data selectivity is high.

Fig. 8 shows that Scoop query speedups exhibit a different behavior than with Parquet. For no data selectivity, Scoop 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 Scoop achieves superlinear S_Q values.

Our experiments show that Scoop exhibits higher performance than Parquet for data selectivity $\geq 60\%$ using the 50GB datasets. For instance, for 90% data selectivity Scoop queries are 2.16x faster than queries executed with Parquet. Note that Scoop 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 Scoop 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— Scoop provides higher query acceleration than Parquet. Moreover, 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.

D. Resource Usage

In what follows, we analyze how Scoop 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. 9 compares the resources consumed —from the compute cluster viewpoint— executing a GridPocket query (`ShowGraphHCHP`, 99% data selectivity) on the 3TB dataset with and without Scoop.

Fig. 9(a) shows that Scoop achieves CPU savings at the compute cluster; first, the average CPU consumption of Spark nodes to execute the given query is less than half for Scoop ($\approx 1.2\%$) compared to plain Spark/Swift ($\approx 3.1\%$). Second, and perhaps more importantly, if we consider the experiment execution time, Scoop 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. 9(b) illustrates the memory consumption at the compute cluster, which is a valuable resource shared across many jobs. As with CPU, Scoop also provides significant memory savings to Spark. At the peak, the average memory usage of Spark nodes is around 13.2% lower for Scoop 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. 9(b) shows that such amount of memory is kept in use for a period of time 12-15x longer than when using Scoop, which may prevent the concurrent allocation of new incoming jobs.

Interestingly, Fig. 9(c) points out that the network was the bottleneck for the ingestion of data. 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. 9(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. 9(c) reveals that Scoop 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. 9(c) shows that the load balancer exhibited an average transmission bandwidth of 189MBps to the compute cluster, and only during ≈ 120 seconds. Therefore, with Scoop both the datacenter network and Swift proxies have more resources to serve other jobs or services running in the system.

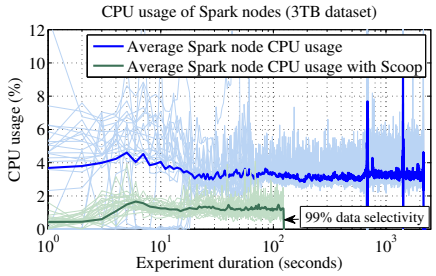
Naturally, all these benefits at the compute and network level come at the cost of using compute power at the object store. In terms of CPU, Fig. 10 shows that Scoop consumes on average a 23.5% of storage nodes CPU to execute projections/selections on the 3TB dataset, whereas this resource is almost totally idle in plain Swift (average CPU usage of 1.25% in storage nodes). Regarding memory, analyzing the execution of multiple SQL queries with Scoop, 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 Scoop 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.

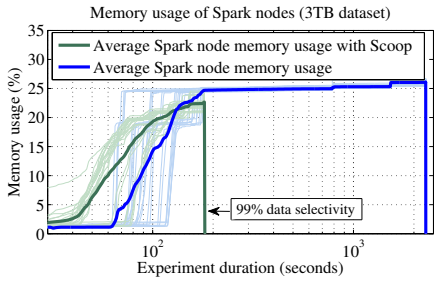
VII. Discussion

By implementing the Spark SQL pushdown use-case, we demonstrated that the concepts behind Scoop are powerful enough to accelerate and make more efficient analytics queries in disaggregated clusters. Our present and future steps focus on leveraging a more general, or even dynamic, form of cooperation between analytics frameworks and object stores:

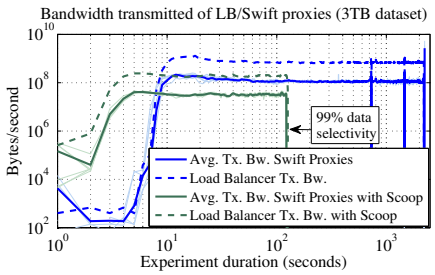
Beyond Spark-SQL pushdown. Boosting ingestion with Scoop leverages Spark-SQL data sources API to delegate projection and selection filtering tasks to the object store. SQL projections and selections, however, are only a particular case



(a) CPU usage of Spark nodes.



(b) Memory usage of Spark nodes.



(c) Transmitted bandwidth of load balancer and Swift proxies.

Fig. 9. Resource usage of Spark nodes in the compute cluster and the inter-cluster network with and without Scoop.

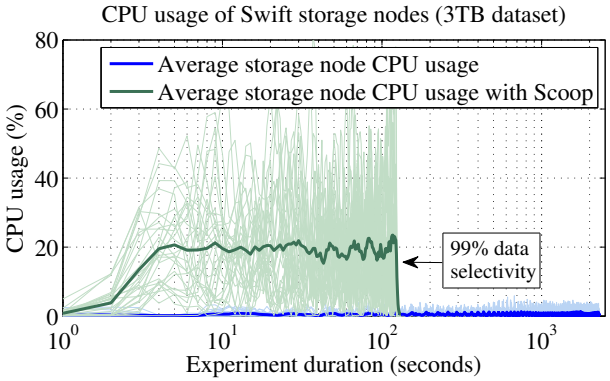


Fig. 10. CPU utilization of Swift storage nodes with and without Scoop.

of the computations that Scoop can carry out in the object store. At the storage side, Scoop provides a rich substrate to execute parallel streamline computations on data objects. From the analytics viewpoint, any computation which can be carried out in parallel and independently over disjoint parts of the input dataset could be pushed down by Scoop in form of a filter.

The capabilities of Scoop at the object store inspired us to generalize the delegation of computations described in this paper to solve many other problems beyond SQL pushdown. In fact, we already extended the Spark RDD [13] to allow the developer to write Spark jobs 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.

New object-aware data sources. Unlike other stores connected to Spark, that usually hold specific or limited data formats (e.g. [9], [10], [11]), 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 Scoop 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 [13]; 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 [13] 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 Scoop. With [13], the whole Hadoop layer can be bypassed by calling from the Spark-CSV layer directly.

Towards adaptive pushdown execution. With Scoop, 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.

To achieve this goal, we built Crystal[10], a control architecture that can dynamically orchestrate the execution of Storlets in OpenStack Swift [21], [22]. Similarly to [16], 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.

VIII. Conclusions

Scoop is a novel solution that mitigates the problems of executing analytics in disaggregated compute and storage clusters

[10]https://github.com/Crystal-SDS

by exploiting the computational capabilities of object stores. In particular, Scoop 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 Scoop is a practical solution for providing faster and more efficient analytics in disaggregated clusters.

ACKNOWLEDGMENTS

This work has been partly funded by the EU project H2020 "IOStack: Software-Defined Storage for Big Data" (644182). The hardware for the performance experimentation was granted by the OpenStack Innovation Center (OSIC). We would like to acknowledge Tal Ariel for helping collecting, analyzing and depicting the experimental data as well as Eran Raichstein for his helpful insights. Gil Vernik is the creator of Stocator; Gil wrote the first implementation of Storlet invocation from Spark and made invaluable contributions to the most recent implementation of the Storlets presented in this paper. We thank Gil for these contributions. We also thank Kevin Fouhety for his contributions on gathering and executing GridPocket workloads in the initial stages of this work.

REFERENCES

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

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

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

[4] Docker. https://docs.docker.com.

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

[6] Openstack innovation center. https://osic.org.

[7] Openstack storlets. https://github.com/openstack/storlets.

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

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

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

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

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

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

[14] 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.

[15] 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*, pages 1383–1394, 2015.

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

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

[18] 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.

[19] E. Friedman, P. Pawlowski, and J. Cieslewicz. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *VLDB Endowment*, 2(2):1402–1413, 2009.

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

[21] 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*, 20(3):10–18, 2016.

[22] R. Gracia-Tinedo, J. Sampé, E. Zamora, M. Sánchez-Artigas, P. García-López, Y. Moatti, and E. Rom. Crystal: Software-defined storage for multi-tenant object stores. In *USENIX FAST'17*, 2017.

[23] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *ACM SIGMOD'93*, volume 22, 1993.

[24] 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*, volume 4, pages 73–86, 2004.

[25] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for MapReduce programs. *VLDB'11*, 4(6):385–396, 2011.

[26] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ MapReduce for log processing. In *USENIX ATC'11*, page 115.

[27] M. Mihailescu, G. Soundararajan, and C. Amza. Mixapart: decoupled analytics for shared storage systems. In *USENIX FAST'13*, pages 133–146, 2013.

[28] F. Pace, D. Venzano, D. Carra, and P. Michiardi. Flexible scheduling of distributed analytic applications. In *IEEE/ACM CCGRID'17*, 2017.

[29] 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)*, pages 59–71, 2014.

[30] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia applications. In *VLDB'98*, pages 62–73, 1998.

[31] 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*, page 67, 2011.

[32] 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*, pages 119–132, 2013.

[33] R. Wickremesinghe, J. S. Chase, and J. S. Vitter. Distributed computing with load-managed active storage. In *IEEE HPDC'02*, pages 13–23, 2002.

[34] 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*, pages 524–534, 2014.

[35] 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*, volume 49, pages 895–908, 2014.

[36] 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*, PP(99):1–1, 2015.

[37] 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*, pages 2–2, 2012.

[38] 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*, pages 320–331, 2013.