u

HORIZON 2020 FRAMEWORK PROGRAMME

IOStack

(H2020-ICT-2014-7-1)

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

D4.1 SDS Services for Analytics Design and Specification

Due date of deliverable: 31-12-2015
Actual submission date: 31-12-2015

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

Summary of the document

Document Type	Deliverable
Dissemination level	Public
State	v1.0
Number of pages	19
WP/Task related to this document	WP4/T4.1
WP/Task responsible	IBM
Author(s)	Dalit Naor, Eran Rom, Yosef Moatti, Shelly Garion
Partner(s) Contributing	IBM
Document ID	IOSTACK_D4_1_Public.pdf
Abstract	This document reviews the architecture and current implementation of the SDS services for analytics in IOStack. In particular, it describes how storlets, which implement "Compute empowered object stores", enable the acceleration of Spark Big Data analytics on data stored in object stores through push down of SparkSQL.
Keywords	storlet, analytics, spark, csv, computation push-down, swift

Table of Contents

—

1 Executive summary

Software Defined object stores are the fundamental storage for cloud data centric applications and services. Object stores play a major role in the IOStack vision of providing rich and dynamic policy driven storage functions that are easily managed, and in particular to support true native analytics on data that is performed **inside** the actual data nodes.

Designed for internet-scale both in terms of capacity and in terms of usage patterns, object stores such as OpenStack Swift [6] offer basic storage functionality. The IOStack architecture however requires a richer functionality, and specifically it needs to provide suitable computing vehicle for analytics on the data store. "Compute empowered object stores" are designed to provide means to execute the richer functionality required by the IOStack architecture.

"Compute empowered object stores" enable the collocation of compute and storage in an isolated and controlled manner so that compute functions can be executed securely near the data. With that, it is possible to implement the IOStack concept of "policy driven filters" for this fundamental type of storage system. In this project we demonstrate how the collocation of compute with data facilitates the important use case of Big Data analytics as a service in an optimized manner; furthermore, it also supports data privacy. This report summarizes the progress achieved so far in IOStack in the design and implementation of "Compute empowered object stores". In summary:

- We completed the design of "Compute empowered object stores" and how they are integrated with the overall IOStack architecture to enable policy-driven filters.
- An open source implementation of "Compute empowered object stores" via the storlets engine (based on Docker Containers) has been released.
- We completed and released an open source integration between the Apache Spark analytics engine and compute-empowered object stores.
- We completed the design and release an initial implementation of an important filter for the analytics use-case, called the "CSV storlet" and integrated it with Apache Spark for SQL push-down; performance evaluation is currently underway.
- Dissemination summary: The open source storlets engine and the connection between Spark and object stores were presented and disseminated in numerous forums this year [1, 2, 3, 4, 5].

In section 2, we provide an overview of "compute empowered object stores", and describe the current state of its implementation. In section 3, we describe the storlet engine architecture and implementation. Section 4 describes the integration between an analytics engine (Spark) and object stores, and how this basic integration is accelerated by pushing down computations from Spark to the object store using storlets. In section 5, we describe the implementation of the CSV storlet which employs filtering and data reduction on the source. In section 6, we describe in details how SQL pushdown is implemented on the Spark side. In section 7, we describe our future plans focusing on Gridpocket's Smart Metering usecase extending the work described in this document. We conclude with a summary in section 8.

2 Compute Empowered Object Store

Compute empowered object stores are object stores that allow the user to upload a computer program into the store and then invoke it over data that resides in the store. The object store we use in the IOStack project is Openstack Swift. Swift is a widely deployed object store that holds hundreds of petabytes in various deployments across the globe. In the IOStack project, Swift is empowered by the storlet engine to allow users to upload and execute computations near the data. The storlet engine was originally invented and written by IBM; the following work, partially done in the context of IOStack, is now an Openstack project [7] (since June 2015).

The computational objects that run inside the object store system are called storlets. Conceptually, they can be thought of the object store equivalent of database store procedures. The main idea behind storlets performing the computation near the storage is saving on the network bandwidth that is required to bring the data to the computation. Moreover, in some cases it also increases security since a storlet can transform the data and prepare it for the consumer needs, without exporting the raw data out of the data store.

Running a computation inside a storage system, involves two major aspects: one is resource isolation and the other is data isolation. Resource isolation ensures that the computation does not consume too many resources, so that the storage system stability and on-going operations are not compromised. Data isolation ensures that the computation can access only the data it is supposed to access. Achieving resource and data isolation is done by sandboxing the computation. In the storlets open source project the sandboxing technology used is Docker [8] a very popular management framework of Linux containers [9].

Our storlets implementation supports two scenarios referred to as the PUT and GET scenarios. In the PUT scenario the storlet is invoked during object upload, where instead of keeping the data (and user metadata) as they are being uploaded, the storage system keeps the result of the storlet invocation over the uploaded data and metadata. This scenario is useful for e.g. metadata extraction: consider a case where the uploaded data is a jpg object, the storlet can extract the jpg information (resolution, geographic coordinations, etc.), and keep it as Swift metadata. We mention that analytics engines work on semi-structured or structured text data. The ability to extract metadata from binary data during their upload essentially enables analytics engines to later make queries on what was originally non queryable binary data in a simple and straightforward way.

In the GET scenario the storlet is invoked during object retrieval, where instead of getting the object's data (and metadata) as kept in the object store, the user gets back the result of the storlet invocation on the object's data (and metadata). This scenario is useful for e.g. analytics pre-filtering: consider an analytics program that is done over logs, where we are only interested in 'ERROR' lines. A storlet that runs near the data can filter out all the other lines resulting in a reduced bandwidth usage between the store and the analytics engine. The GET scenario enables both the analytics as a service use case described in section 4 as well as the the data privacy use case described in section 7

The storlet engine is an IBM technology which has been developed over several years. It was brought as an asset to IOStack and has been developed further in the context of IOStack. The work on storlets that was done in the context of IOStack consists of:

- Tailoring it to the analytics use case (as outlined in the section 4)
- Maturing the asset and opening it to the open source community

This document is focused on the storlet engine in the context of the analytics as a service use case. To make the document self contained, we bring necessary background on the engine. We mention that a comprehensive and detailed documentation on the API, design and implementation of the storlet Engine can be found in [7]

3 Storlet Engine Background

For completeness we give below background in the storlet engine. some of this material was described in the Cosmos deliverable D4.1.2 (pertains to Cosmos WP4)

3.1 Introduction

Storlets are computational objects that run inside the object store system. Conceptually, they can be thought of the object store equivalent of database store procedures. The basic idea behind storlets is performing the computation near the data thus saving on the network bandwidth required to bring the data to the computation. The storlet functionality in IOStack is developed in the context of the Openstack Swift object store.
The high level architecture section below describes how we integrate the storlet functionality into Swift.
Running a computation inside a storage system, involves two major aspects: one is resource isolation and the other is data isolation. Resource isolation has to do with making sure the computation does not consume too many resources, so that the storage system stability and on-going operation are not compromised. Data isolation has to do with making sure that the computation can access only the data it is supposed to access. Achieving resource and data isolation is done by sandboxing the computation. The sandboxing technology section below describes in more details the way in which the storlets computation is isolated.
Storlets provide a mechanism for data analysis where computation is enabled to run close to the storage. In addition, APIs are defined in the appendix which allow developers to write application specific analysis using storlets.

3.2 High Level Architecture

3.2.1 Swift Architecture Essentials

At a high level Openstack Swift has two layers: A proxy layer in the front end, and a storage layer at the back end. Users interact with the proxy servers that route requests to the backend storage layer. A typical flow of a request that operates on some object is as follows: the request hits the proxy server that (1) authorizes the request and (2) looks up in which storage server the requested data is kept. Then the proxy server forwards the request to the designated storage server (also known as object server). See figure 1 – Request flow in Swift.

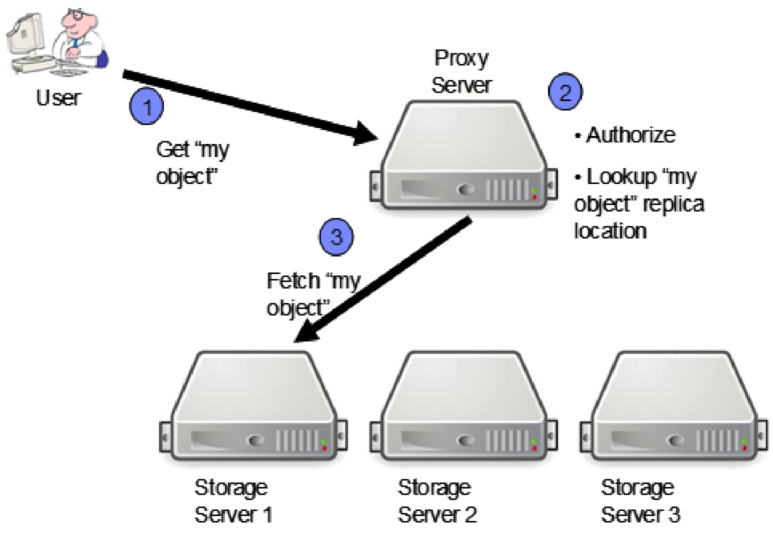


Figure 1: Request Flow in Swift

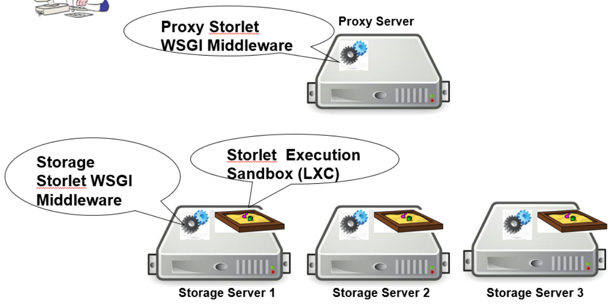

Proxy Storlet WSGI Middleware
Proxy Server
Storage Storlet WSGI Middleware
Storlet Execution Sandbox (LXC)
Storage Server 1
Storage Server 2
Storage Server 3

Figure 2: The storlets' high level components: WSGI middleware in the proxy and storage servers, and a sandbox in each of the storage servers. Each storlet is executed in a daemon that runs inside the sandbox

Swift is implemented using WSGI [14] technology that allows to plug-in functionality into the request processing. Each request hitting a WSGI based server goes through a pipeline of such plug-ins, called middleware. For example, among the plug-ins that consist the pipeline at the proxy server are an authorization middleware, quota related middleware and a 'router' middleware that forwards the request to the appropriate server according to the location of the request target resource.

3.2.2 Storlets' High Level Architecture Components

The storlet functionality adds the following components to Swift. See figure 2.

Storlet Execution Sandbox A sandbox where the storlet code is running in an isolated manner protecting the system. For the sandboxing technology we chose to use Docker [8] which is a leading tools in the domain of Linux containers [9] management. Docker offers as salient features its ability to easily build and deploy applications that have to be executed inside Linux containers.
In contrast to traditional virtualization, Linux containers provide an operating system level virtualization rather then hardware virtualization which makes them lightweight. Linux containers are mainly based on two Linux kernel features:

- Control Groups. Control groups allow controlling the resource consumption at the level of a process. For example they can be used for limiting a process to use only a subset of the machine's core, set a limit on the IO bandwidth a process can use with a certain device, and completely block the access to certain hardware devices.
- Namespaces. Namespaces allow wrapping a global system resource so that it appears to a process as if it has its own instance of the resource. For example, a mount namespace provides a process with what looks like a root file system while effectively it sees only a portion of the host's root file system. Another important type of namespaces is the user namespace. User namespaces allow a process within the namespace to have the root user id (0) and have root privileges, while outside of the namespace it has no special privileges. Thus, a process running as root in some user namespace could send signals to other processes running in the same namespace, while it will not be able to send any signals to processes running outside of the namespace.

The storlet middleware The storlet middleware is a standard Swift middleware that intercepts all user requests that involve storlets invocation. The middleware makes sure that the storlet is running inside the docker container and routes the request's data to the storlet running inside the container.

3.2.3 Storlets' Invocation Flow in the Get Scenario

A storlet is invoked by adding a designated header to the Swift GET request. When such a request hits the proxy server, the proxy storlet middleware validates that the issuing user has access to the

required storlet. As descried in the above flow, the proxy server then routes the request to a storage server where the requested object resides.
The storage service middleware that runs on the server where the object resides opens the object's file and passes its file descriptor to a daemon that executes the storlet. Together with the object's file descriptor, the storage server storlet middleware passes a pipe file descriptor through which the storlet can send back the computation results. The computation results are then sent back to the user.
The following figure describes the interaction between the storage server storlet middleware and the daemon running the storlet code.

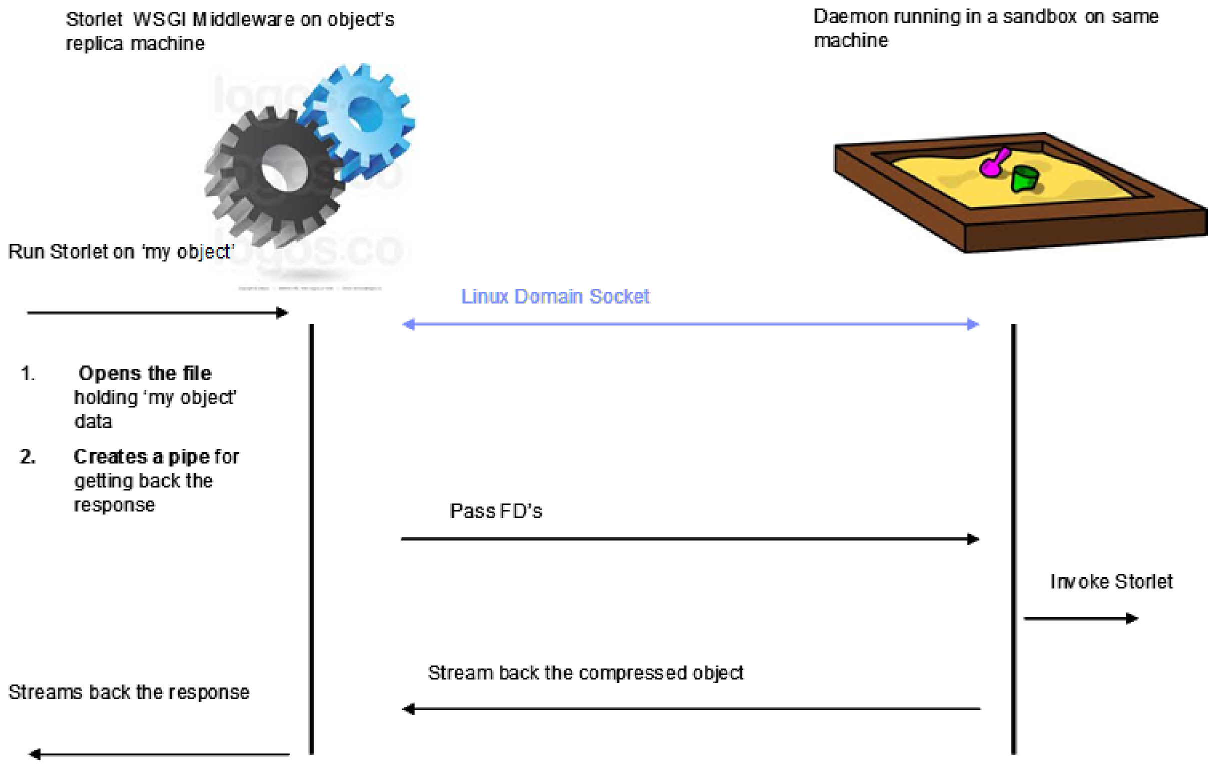

Storlet WSGI Middleware on object's replica machine
Daemon running in a sandbox on same machine
Run Storlet on 'my object'
Linux Domain Socket
1. Opens the file holding 'my object' data
2. Creates a pipe for getting back the response
Pass FD's
Invoke Storlet
Stream back the compressed object
Streams back the response

Figure 3: The interaction between the storlet middleware on the storage server and the sandbox running on the same machine. The middleware got a request for running a storlet on an object named 'my object'. The middleware is communicating with the sandbox via a Linux domain socket to pass the designated file descriptors

3.2.4 Storlets' Invocation Flow in the Put Scenario

A storlet is invoked by adding a designated header to the Swift PUT request. When such a request hits the proxy server, the proxy storlet middleware validates that the issuing user has access to the required storlet. The storlet middleware in the proxy server then passes the request data to daemon that executes the storlet in the proxy server. As with the GET scenario together with the request data, the proxy server storlet middleware passes a pipe file descriptor through which the storlet can send back the computation results. The computation results are then continue with the Swift PUT flow, that creates several copies of the data.
We mention that in both GET and PUT the daemon is sandboxed in a way that it cannot access any I/O devices of the storage or proxy servers. The only communication channels it has with the outside world are the file descriptors it is given from the middleware.

3.3 The Storlet Engine Multitenancy and Run Time Model

The storlet engine multi-tenancy model is based on Swift's accounts. In Swift an account is a set of containers and is typically associated with a certain customer and acts a a unit of billing. In Swift there is a notion of an account admin user that has full access to all containers the account, but no access to other accounts.
Following that model, the storlet engine requires that each account will have a designated Swift container to holds storlet objects. Moreover, each account has a separate docker container image. This container image is used to run a Docker container dedicated to that account. All storlets that are executed over data belonging to a certain account run inside the account's dedicated Docker container. Since the data belonging to an account can reside on each of the Swift storage nodes there is an instance of the account Docker container on each of the Swift nodes.
Figure 4 describes the run time multi-tenancy model within a Swift node. On the left hand side of the figure is the storlet middleware that was described in the previous sections. On the right hand side are the 'per account' Docker containers. Each container runs a daemon factory and a set of storlet daemons:

3.3.1 The Storlet Daemons

A storlet daemon is a process dedicated to execute code of a certain storlet. This means that if a user of some account uploaded a storlet for doing compression and a storlet for doing encryption, then Docker containers belonging to this account would potentially run a compression daemon and an encryption daemon. We mention that this reflects a performance oriented design choice where instead of spawning a process per user request to invoke a certain storlet (or even a docker container instance per request) there is a 'per storlet daemon'. Thus, all the invocation requests for doing compression over data that belong to account X and that reside in node Y would end up being served by the compression storlet daemon that runs within the Docker container dedicated to account X that runs on node Y. The storlet daemons are written as generic processes that can load a storlet Java class at runtime. Thus, when a storlet daemon is spawned it gets as input which storlet code to load. Once the Storlet daemon loads the storlet code it listens on a dedicated channel (the 'per storlet daemon Sbus') for storlet invocation requests. Through this channel the storlet middleware passes storlet invocation requests over objects together with the object's file descriptors.

3.3.2 The Factory Daemon

The factory daemon spawns a 'storlet daemon' per request from the storlet middleware. The daemon factory gets executed as soon as the Docker container is initialized and it listens on the 'Factory Sbus' through which the storlet middleware passes requests to start or stop 'per daemon storlets'.

3.4 From Writing to Deploying and Running a Storlet

Writing a storlet involves implementing a single API called 'Invoke'. This API call gets the following inputs:

- An input stream representing the object data. On PUT this is the object content uploaded by the user. On GET this is the object content as stored in the system.
- An output stream where the storlet writes the computation output.
- The object's metadata. This is the object's uploaded metadata (on PUT) or the kept metadata (On GET)
- Execution parameters. When invoking a storlet the user can specify invocation parameters.
- Logger. The storlet code can emit logs that end up getting to the Swift node syslog where the storlet runs.

Once the Storlet is written it should be packaged as a jar file and uploaded to a designated container within the user account. While the storlet is being uploaded as any other Swift data object it eventually needs to end up running inside a Docker container on potentially any of the Swift nodes. The steps below describe how his happens:

- When uploaded, the storlet is kept as a data object in a container called 'storlet' (that container name is configurable).
- When the storlet middleware on a certain node receives a storlet invocation request:
  - It checks if there is a storlet daemon that runs this storlet. This check is done against the factory daemon in that Docker container.
  - If no daemon currently runs the storlet, it checks whether there is a local copy of that storlet. If no local copy exists it initiates a Swift internal GET to download it from the 'storlet' container. Once downloaded the storlet is copied into the Docker container.
  - Once there is a copy of the storlet inside the container, the storlet middleware initiates a request to the factory daemon to spawn a storlet daemon that will load the storlet code.

3.5 Storlet Engine Deployment

In this section we briefly describe several deployment related tools that come with the storlet engine.

3.5.1 cluster deployment scripts

These are a set of scripts that can take a configuration file that describe an existing Swift cluster and deploy all the storlet engine components on top of it.

3.5.2 S2AIO

s2aio or 'swift storlets all in one' is a script that installs from scratch both swift and the storlet engine on a single - typically virtual - machine. The script is used by the openstack CI system to install the engine for tests upon every commit.

3.5.3 Bring your own image

Different storlets can potentially be dependent on non trivial software stacks. One can think of storlets that require non-trivial image processing libraries or C matlab code. As described in section 3.4 the storlet size cannot be too large as it needs to get copied into a certain node synchronously during a user request processing. Thus, we have added the ability for a tenant admin to be able to bring a Docker image to be deployed on all nodes. To enable that we have written the following scripts:

- Create tenant. This script generates a default Docker image that includes the relevant storlets code (e.g. the daemon factory) and saves it in a designated docker images Swift container.
- Deploy tenant image. A Script that takes a Docker image from the designated images Swift container and deploys it across all the cluster nodes. We mention that there might be a running Docker container that needs to be stopped as it is running using the previous image. the script takes care to stop any existing running container in a graceful manner.

Using the above two scripts, a tenant admin can download the default image from her account, augment it with whatever software and upload the resulted image back to the Swift images container. The Swift operator can then invoke the deploy tenant image script to deploy the updated image across the cluster. Once deployed any existing running Docker container will exit gracefully. The next storlet invocation request would cause a new container instance to start while using the updated image.

3.6 Storlet Conclusions

The storlet engine brings a holistic solution for running computations near the data. As such it can serve as a cornerstone for the implementation of IOStack when applied to object stores.

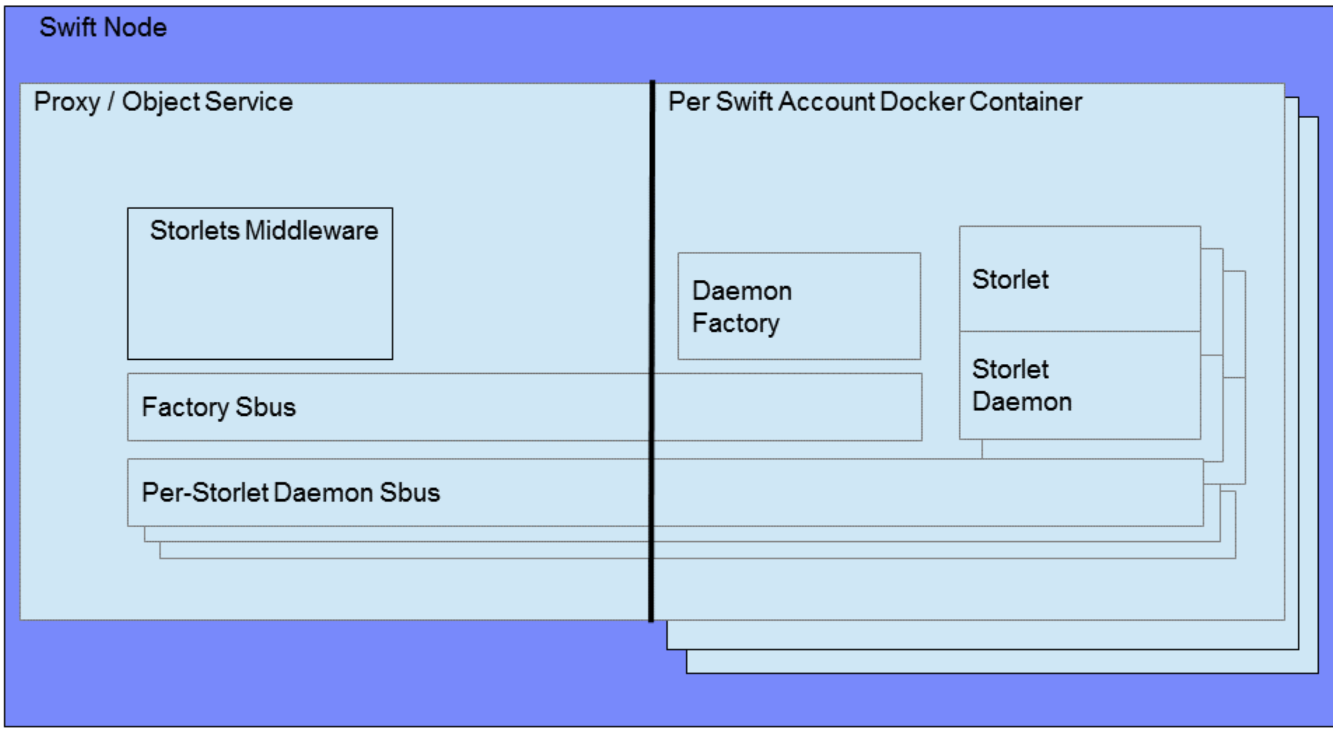


Figure 4: The Storlet Engine Multitenancy and Run Time Model

4 Pursuing Analytics Acceleration with the Storlet Engine

In order to analyze the continuously growing amounts of data, one needs to use cluster computing eco-systems, such as Apache Hadoop or the most recent open-source analytics engine Apache Spark. Apache Spark can analyze both batched and streaming data, using advanced machine learning and graph processing algorithms as well as answer SQL queries. An important feature of spark is its ability to analyze data from multiple data sources. For example, Apache Spark can analyze data stored in object stores, HDFS, and various data bases as depicted in 5

Much of the semi-structured data today is stored in object stores in the cloud using formats such as CSV. However, in order to perform analytics on this data with engines such as Apache Spark, the data needs to be structured, as it is queried using e.g. the Spark SQL module [10].
A major contribution of this WP is to show that storlets can accelerate this type of Spark Analytics on data that resides in Swift object stores. Analytics acceleration using the storlet engine is done through Spark SQL. **Spark SQL** is a Spark extension that allows to run SQL queries over data in Spark, thus utilizing Spark's ability to distribute query processing within the Spark cluster. Our acceleration work allows to push some of the SQL query processing over CSV files to the object store using the storlet engine. By doing so the amount of data being loaded into Spark is smaller, thus saving both on bandwidth and on memory and CPU consumption within the Spark cluster.

In the next section we describe an end-to-end scenario of Spark SQL push down to Swift using the storlet engine. We then proceed to describe the design and implementation of this mechanism, highlighting the challenges involved in implementing the push down.

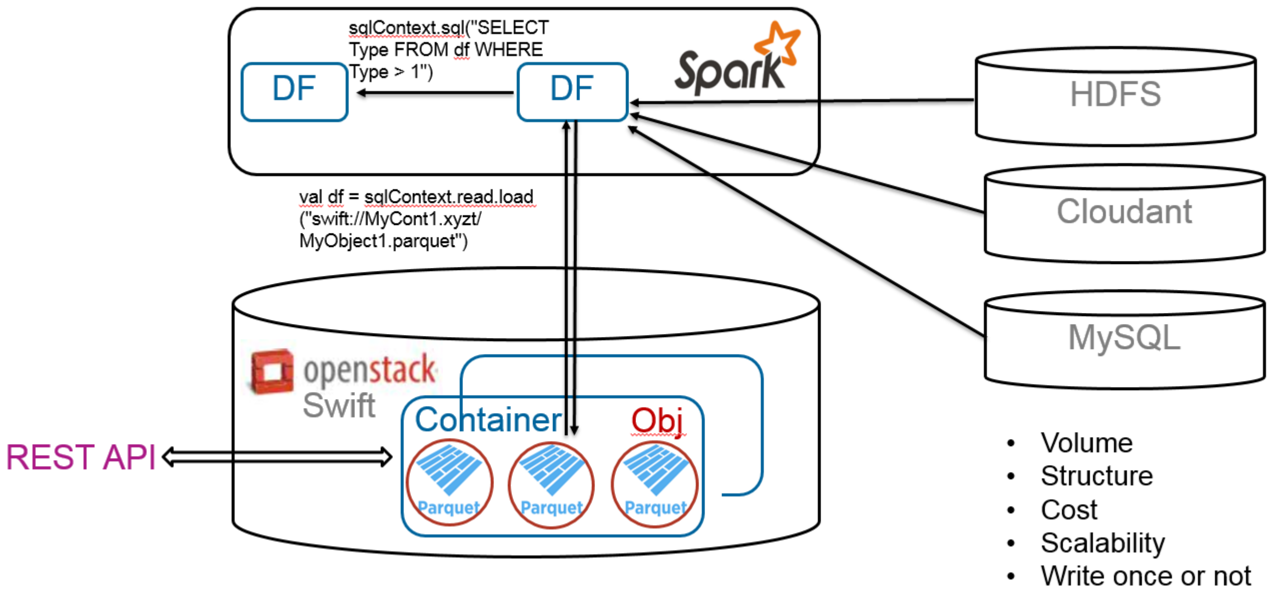


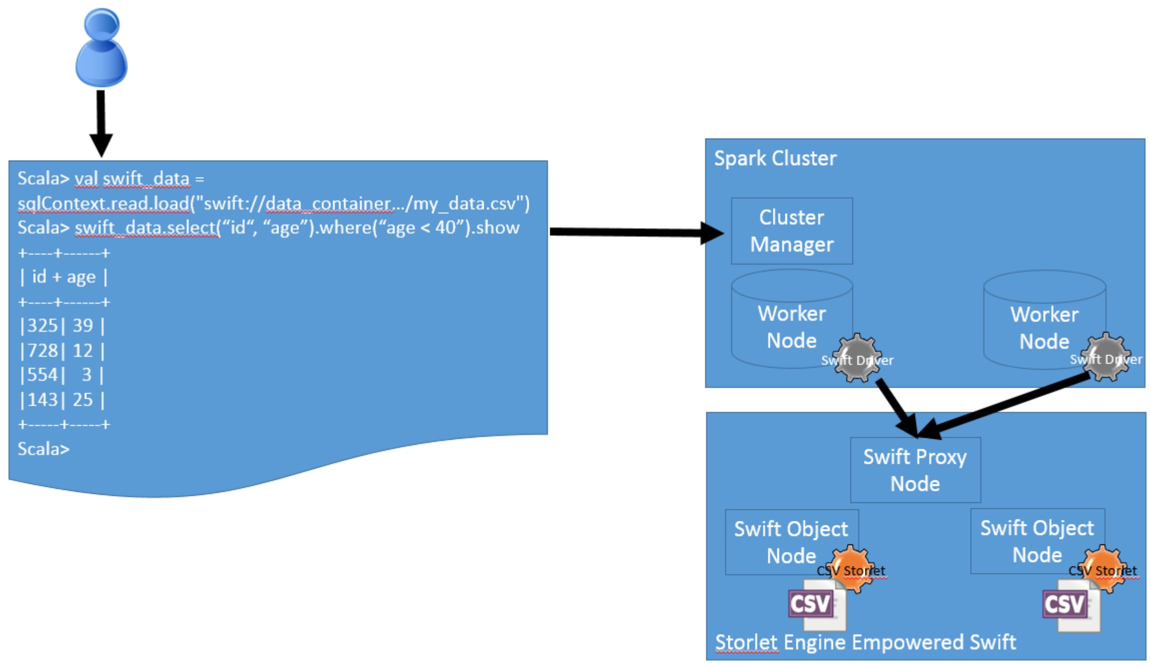
Figure 5: Swift and Spark SQL Access API

4.1 Spark SQL Push Down Flow

Consider an example of an SQL query over a large data set that is stored in a CSV object in the object store called my_Data.csv. The SQL query selects a subset of rows from this data set. The Spark SQL push down to the object stored in Swift involves several stages – See figure 6.

- The flow typically begins with a user interacting with Spark using a Spark shell[1]. The Spark shell is integrated with the Spark SQL component.
- The user defines a Spark data frame, whose associated data is a csv file residing in Swift. This definition triggers an internal Spark process called 'partition discovery'. During that process Spark uses an underlying Swift specific driver to partition the data into a set of Swift objects and ranges within those objects[2]. Thus, a partition is of the form (object, range). Each partition is going to be processed by a different Spark task, and those tasks are distributed amongst the Spark worker nodes. We stress that at this point no actual data has been read from Swift.
- Once the data frame is defined, the user can perform an SQL query over it. We note that this query has both column selection (e.g. id and age in this example) and row filtering (e.g. rows with age < 40).
- The submitted SQL query triggers the various tasks (one per partition) to start executing across the Spark worker nodes. Each such task starts by reading the partition data from Swift.
- The actual reading is done using the Swift driver that does an HTTP GET request from Swift to get the data. **With our implementation of push down via storlets, the driver issues a GET request that also triggers the CSV storlet over the Swift CSV object**.

[1]Depicted flow applies also for Spark batch invocations
[2]In our example the data frame consists of a single object, but in general the user can specify a collection of objects



```
Scala> val swift_data =
sqlContext.read.load("swift://data_container.../my_data.csv")
Scala> swift_data.select("id", "age").where("age < 40").show
+----+------+
| id + age |
+----+------+
|325| 39 |
|728| 12 |
|554|  3 |
|143| 25 |
+-----+-----+
Scala>
```

Figure 6: Request Flow in Swift

- The storlet engine executes the CSV storlet on the Swift object nodes, **where the actual data is stored and sends the selected subset of data to Spark**. Recall that Swift splits large objects over multiple data nodes, in addition Swift replicates each data object several times (typically 3) for object of any size. Hence different partition task executions, each having a different range within the object, can get to a different replica of the object in parallel. As a result, the pushed down work is distributed amongst the Swift object nodes.

4.2 Spark SQL Push down Challenges and Implementation

Our SQL push down solution consists of two parts:

1. Changes required to the Spark analytics
2. A CSV storlet (detailed in section 5)

We divide the discussion on challenges and implementation accordingly.

Enhancing Spark to support pushdown via storlets As we mentioned in the end-to-end scenario, whenever Spark is given a data source it performs a process called "partition discovery", during which Spark partitions the data into a set of Swift objects and ranges within those objects. Each partition is going to be processed by a different Spark task. Those tasks are distributed amongst the Spark worker nodes.
We stress that the partition set is created based on the data size being specified and is **independent** of the user SQL query. That is, Spark first builds the partition set, and when a query comes in, its processing is distributed according to that fixed partition set.
Spark reads data lazily, delaying the actual read actions until the data is actually needed. Therefore during partition discovery no actual data is being read from Swift: that is the data is read **implicitly** when the user submits a query.

Challenges. The partition discovery flow brings the following challenges:

1. The first challenge has to do with the partition discovery being done before we know anything about the query. Given a query, some objects may not be relevant at all, while other objects

may require a lot of processing. This challenge of doing SQL aware "partition discovery" is a major structural change in Spark, and it is now a subject of discussions in the Spark community, which IBM plans to be part of in order to improve our push down mechanism.

2. The second challenge has to do with the fact that the data is read from Swift implicitly. Using the storlet engine to do the push down, we need it to hook into the read operation in order to specify that a storlet should be invoked during the read operation, in addition the relevant parameters that reflect the user's query should also be passed as part of the read. Our work addresses that challenge by modifying Spark to propagate the query parameters to the Spark tasks.

3. A third challenge imposed by Spark, is that Spark uses a chain of inherited legacy Hadoop drivers to read data. That chain ends with the Swift specific driver that performs the actual read from Swift. In order to utilize the storlet engine we needed to pass all along the chain the information that would allow the Swift driver to:

   (a) Invoke the CSV storlet as part of the GET.

   (b) Pass the query parameters (“age”, “id”, “where age<40” in our example) to the storlet.

5 Filtering data on the source: The CSV Storlet Implementation

The CSV storlet is a storlet written to process filtering and data reduction over CSV files. It is invoked via a GET request initiated by the Spark chain of drivers responsible for reading partition data.
The filtering is done both on columns and on rows according to SQL query and parameters passed from Spark. Spark passes two parameters to the CSV storlet:
The first parameter describes the indexes of the requested columns. If this parameter is omitted or points to an empty list of columns then no column selection will be done, meaning that the returned rows will contain all the columns.
The second parameter describes a set of conditions to be applied over the selected columns values. Those two parameters are essentially the parameters that are propagated to all the Spark tasks allowing each of them to invoke a storlet execution.

5.1 Implementation details

For the implementation of the CSV storlet we have chosen two open source packages:

1. The univocity open source [11] CSV parser version 1.5.6. This package allows to do CSV parsing as well as column projection. That same package is used by the spark-csv project [12] to do the exact same CSV parsing when done on the Spark side.

2. Various parquet [13] packages used for row filtering. Spark uses the same packages to perform that very task when row filter is not pushed down.

During the implementation of the CSV storlet we ran into the challenges described below:

Granularity of record processing For CSV processing, we have used the univocity package. Two naïve extreme ways of using the package are to:

1. Process the target file one record at a time – this may be very slow; or

2. Load the whole file into memory allowing to process it in one step which is not feasible for large files.

We chose a middle ground where records are handled in a batched way: a given number of records are read at once and processed. This allows a tradeoff between memory usage the performance penalty of reading one record at a time. To facilitate this, the CSV storlet has the rowsPerBatch parameter (whose defaults is 100), which specifies the number of rows being read from the CSV file at a time. The performance implications of this parameter are described is described in section .

CSV file breakdown CSV files usually have a header in their first line that describes the schema. This scheme is needed as a parameter for the CSV storlet. Since Spark tasks read ranges of a file, the storlet cannot rely on having the header information as it can be invoked over a range that does not include that header.
Since the scheme header is too long to be passed as an HTTP string (as needed for a storlet invocation), the implementation passes column indices instead of names.

6 A Concrete example: the CSV Storlet

In this section we bring several code snippets of interest from the invocation stack of the CSVStorlet.

6.1 Storlet invocation mechanism in the swift haddop driver

We fist show few critical APIs and their modifications for the pushdown mechanism within the Spark/Hadoop frameworks:

- When a Spark SQL query is performed, the Spark framework creates a number of tasks where each task accesses a specific object and range of bytes.
Since that data in question resides in Swift, the swift hadoop driver is invoked. This essentially causes the constructor of the *SwiftNativeInputStream* class to be called.
The following code snippet shows the modifications done to this constructor which basically boil down to adding the two pushdown parameters to the URI of the request.

```
public SwiftNativeInputStream(SwiftNativeFileSystemStore storeNative,
    FileSystem.Statistics statistics, Path path, long bufferSize,
    String filterPredicate, String filterColumns)
        throws IOException {
  this.nativeStore = storeNative;
  this.statistics = statistics;
  this.path = path;
  if (bufferSize <= 0) {
    throw new IllegalArgumentException("Invalid buffer size");
  }
  this.bufferSize = bufferSize;
  //initial buffer fill
  StringBuilder dataLocationURI = new StringBuilder();
  boolean hasFilter = false;
  if (!filterPredicate.equals("")){
      hasFilter = true;
      dataLocationURI.append(path.toString()).append("?whereClause=")
      .append(filterPredicate);
  }
  if (!filterColumns.equals("")){
      hasFilter = true;
      dataLocationURI.append(";selectedFields=").append(filterColumns);
  }
  LOG.debug("Data location URI: " + dataLocationURI.toString());
  if (hasFilter)
      this.path = new Path(dataLocationURI.toString());
  this.httpStream = storeNative.getObject(this.path).getInputStream();
}
```

The last line of constructor of the *SwiftNativeInputStream* class invokes the *getObject* API of 'storeNative' which is an instance of *SwiftNativeFileSystemStore*. This *getObject* call would eventually lead to the invocation of the "doGet" API of the SwiftRestClient class.
The parameters of the pushdown, as previously seen, have been appended to the the URI parameter.

Upon detecting at least one such parameter, the *doGet* API will add the header which will cause the storlet invocation in the Swift middleware.

```
/**
 * Exec a GET request and return the input stream of the response
 *
 * @param uri URI to GET
 * @param requestHeaders request headers
 * @return the input stream. This must be closed to avoid log errors
 * @throws IOException
 */
private HttpBodyContent doGet(final URI uri, final Header... requestHeaders)
        throws IOException {
  return perform("", uri, new GetMethodProcessor<HttpBodyContent>() {
    @Override
    public HttpBodyContent extractResult(GetMethod method) throws IOException {
      return
        new HttpBodyContent(
          new HttpInputStreamWithRelease(uri, method),
          method.getResponseContentLength()
        );
    }

    @Override
    protected void setup(GetMethod method) throws
                SwiftInternalStateException {
      // The following two lines cause the necessary header to be added
      // if the CSVStorlet is to be invoked
      if (uri.toString().contains("whereClause")
          || uri.toString().contains("selectedFields"))
          method.addRequestHeader("X-Run-Storlet", "CSVStorlet-1.0.jar");

      setHeaders(method, requestHeaders);
      if (isXNewestHeader) {
          setXNewestHeader(method);
      }
    }
  });
}
```

6.2 Storlet invoke API implementation

Within the Swift middleware, the storlet invocation header is detected, along with the name of the storlet to be invoked as well as the invocation parameters. The middleware then causes the invocation of the storlet within its docker container.
The following code snippet gives an example of how a Storlet is written. Basically, it has to implement the single API (invoke) of the IStorlet interface The inputStreams parameter gives access to the stream of bytes resulting from the read of the targeted object while the outputStreams parameter will be used by the storlet to stream out the potentially modified stream of bytes. The "parameters" parameter is a Map in which all the needed parameters to the Storlet have been added.

A brief description of the code is as follows:
The storlet entry point is the *invoke* API (which is the sole API of the *IStorlet* interface).
We first build a a *ReaderEnv* object out of the passed *parameters* Map. The constructor of this object

checks the validity of the parameters passed within *parameters* and implements an internal API which gives access to the important parameters of the storlet invocation.
We then prepare the *BufferedReader/Writer* objects which permit to cope with potentially big input (and output) objects.
We then build a *FilterPredicate* instance out of the specific filter predicate that pertains to this storlet invocation. Where FilterPredicate is an interface pertaining to a Parquet [13] package which represents an expression tree describing the criteria which discriminates the records of the CSV file to be kept.
We can then build the *CSVBatchedPredicateProcessor* instance where the *CSVBatchedPredicateProcessor* extends the univocity class *BatchedColumnProcessor* so as to handle CSV files.
Finally the *CSVBatchedPredicateProcessor* instance is used to invoke the parsing of the targeted CSV file.

```
@Override
public void invoke(
           final ArrayList<StorletInputStream>  inputStreams,
           final ArrayList<StorletOutputStream> outputStreams,
           final Map<String, String> parameters,
           final StorletLogger log )
      throws StorletException {

   ReaderEnv env = new ReaderEnv(parameters, log);
   CsvWriter writer = null;

   // creates a CSV parser
   csvSettings = ReaderBuilder.getSettings(env, log);
   StorletInputStream sis = inputStreams.get(0);
   StorletObjectOutputStream storletObjectOutputStream;
   storletObjectOutputStream =
      (StorletObjectOutputStream)outputStreams.get(0);

   if (env.isStorletInvocation())
           storletObjectOutputStream.setMetadata(sis.getMetadata());
   InputStream is = sis.getStream();

   StorletOutputStream sos = outputStreams.get(0);
   OutputStream os = ((StorletObjectOutputStream) sos).getStream();
   BufferedReader bufferedReader = null;
   BufferedWriter bufferedWriter = null;
   PredicateBuilder predicateBuilder = new PredicateBuilder();
   String predicateString = env.getTheWhereClause();
   FilterPredicate filterPredicate =
      predicateBuilder.getFilterPredicate(predicateString);

   try {
      String encodingString = env.getParam(
         ReaderEnv.FILE_ENCRYPTION,
         ReaderEnv.DEFAULT_FILE_ENCRYPTION);

      bufferedReader = getReader(is, log, encodingString);
      bufferedWriter = getWriter(os, log, encodingString);
```

```
        int rowPerBatch = (int)env.getLongParam(
            ReaderEnv.ROWS_PER_BATCH,
            ReaderEnv.DEFAULT_ROWS_PER_BATCH);
        CsvWriterSettings writerSettings = new CsvWriterSettings();
        writer = new CsvWriter(bufferedWriter, writerSettings);

        CSVBatchedPredicateProcessor batchedColumnProcessor =
            new CSVBatchedPredicateProcessor(
                rowPerBatch,
                filterPredicate,
                writer,
                booleanCodeOptions,
                intCodeOptions);

        csvSettings.setRowProcessor(batchedColumnProcessor);
        String[] selectedFields = env.getSelectedFields();

        String[] schema = env.getSchema();

        if (schema != null && schema.length>0)
                csvSettings.setHeaders(schema);

        if (selectedFields != null && selectedFields.length>0)
                csvSettings.selectFields(selectedFields);

        Utils.doubleLogPrint(log, "Starting␣to␣parse...␣");
        new CsvParser(csvSettings).parse(bufferedReader);
    } catch (StorletException e) {
        throw e;
    } catch (RuntimeException e) {
        throw new StorletException(message);
    } finally {
        try {
                if (writer != null)
                        writer.close();

                if (bufferedWriter != null)
                        bufferedWriter.flush();

                is.close();
                os.close();
        } catch (IOException e) {
                log.emitLog("got␣IOException:␣" + e.getMessage());
        }
    }
    log.emitLog("CSVStorlet␣invocation␣completed");
}
```

7 Future Plans

The IOStack consortium includes the Gridpocket company. Gridpocket brings to IOStack the **Smart electricity metering** use case. In this use case data is collected from smart meters and then aggregated into objects stored in Swift. In order to extract and optimize usage patterns and other analytics on this data, Spark SQL can be used.

The following characteristics of the data involved in this scenario makes storlets most appealing.

Data Size Assuming 100 bytes per reading, one reading every 10 minutes and one million end users, the use case generates approximately 5TB a year. This estimation is expected to grow as more meters and users become available. Loading this data set into the Spark compute cluster is not feasible, therefore a careful selection of the data per analytics experiment is needed. Specifically, the data reduction/selection architecture described hereby can be handy. In some cases, we expect that the data relevant for a particular analysis can be **as small as 1% of the whole data set** and as such may fit into a given Spark cluster.

Privacy Privacy is a major concern when it comes to energy data as these readings may reveal sensitive information such as regular times where the residents are absent and other private behavioral patterns. To support data privacy, relevant columns with private information can be filtered, data can be aggregated, averaged, sampled and normalized **at the data source** prior to being uploaded to the analytics engine.

In Y2 we plan to explore how this use case can benefit from the Spark/Swift and Storlets architecture in the context of the overall IOStack architecture. In particular we believe that the use case can benefit from the the Spark SQL pushdown using the CSV storlet.

7.1 Consent Management architecture

In the following we detail the architecture of the Consent management and explain how it integrates with the IOStack architecture.

Our solution provides logging and auditing access to sensitive data, managing and enforcing privacy and consent policies, as well as providing the ability to anonymize sensitive data. With such a solution in place, trust becomes a differentiator while auditing and compliance overhead is decreased for both the data processor and controller.

Our solution comprises of consent-management enforced analysis on data stored in Object Storage in the Cloud by Apache Spark. Data may be processed and analyzed only when the data subject has given his consent, is necessary for a contract, needed for compliance with a legal obligation, or is necessary for a task carried out for public interest, and the given consent must contain a (legitimate) purpose. Any data access attempt for which there is no consent for the specified purpose will either be blocked, or the data will be anonymized or obfuscated. The consent management will support advanced policies, for example: instead of outputting entire data records, the application will provide an anonymized sample, or a computation on the data such as average or median values, or an aggregation of the data such as summation.

The enforcement of the consent management policies may be done in several ways. Either in the Object Storage using privacy enabled push-down capabilities, such as Storlets, or in the Spark analytics engine, using for example the “Secure Spark” capabilities. Although IBM aims at investigating both options in order to obtain proven isolation and enforcement of the consent, in the context of IOStack we will pursue the enforcement of privacy through filters in the Swift object store implemented. We aim to provide tools and libraries for “privacy and compliance by design", while provide compliance solutions for existing applications without requiring changes to the application. The approach is to make privacy and compliance part of the IT infrastructure, and to ensure close coupling of all data with relevant consent and policies.

7.1.1 Privacy use-case scenarios

The objectives of the privacy use-case scenarios are:

- Demonstrate “privacy-by-design” architecture using IOStack
- Demonstrate data privacy protection of smart metering data

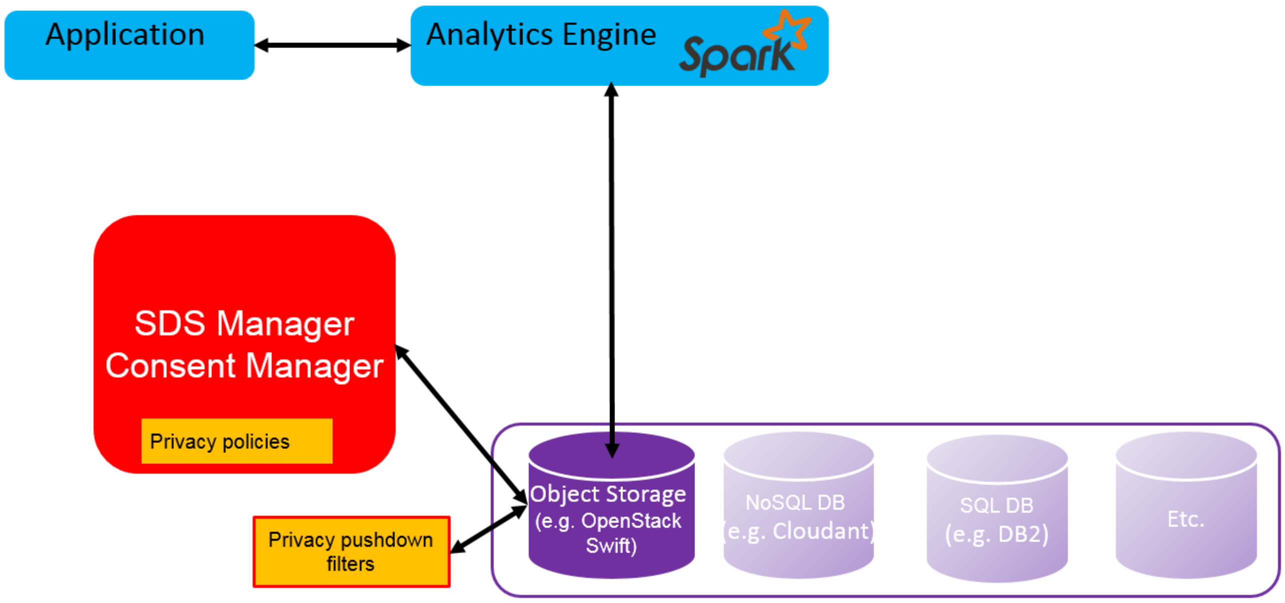


Figure 7: Consent Management architecture in IOStack Architecture

- Demonstrate the “Consent Management” concept

The data description for the use case is as follows:

- Time series of smart meter reading: meter_ID, time/date [ISO], cumulative power consumption [kWh].
- Additional data: smart meter owner data, e.g.:
  name, address of meter, phone, e-mail, postal address, individual consumer electrical consumption summaries

There are two privacy use-case scenarios that are detailed below. Both use-case scenarios assume that the data will be stored in Swift Object Store – energy consumption as the object, and additional data (name, address, etc) as metadata, and the queries will come from Spark.

1. Customer energy savings recommendation:
   The customer gives his consent to the company to provide recommendations regarding how to decrease energy consumption. The following data would be mandatory for such a service:
   - meter_ID
   - Consumer name
   - postal address
   - Aggregate information about the specific consumer's energy consumption (by month, day of week, by day, hour, 4 hours (or any interval), time of day – morning, afternoon, evening, night)

   The following data would be optional:
   - Address of meter (home of the consumer) – perhaps to compare to usage of neighbors?
   - Email

- Phone number
- Energy usage for the specific consumer by specific date and time

2. Prediction of energy needs:
   The customer gives his consent to the company to use his data for the purpose of future prediction of energy consumption by the company. For this type of service, typical privacy settings would likely be:

   - meter_ID is not necessary
   - Consumer name, phone and email are not necessary
   - Phone number
   - Address is needed, but only at an aggregate level, such as perhaps the zipcode, county or state

Our plan is to work with Gridpocket towards pursuing the elaboration of the privacy use case that will both involve the consent management architecture and the storlet mechanism.
Work is under way to define this in details.

8 Summary

We have shown how to perform analytics acceleration using Spark Analytics over CSV data stored in Swift object stores. This required modifications to the Spark partition discovery mechanism, as well as extending the storlet engine and implementing the CSV storlet. These contributions together perform selective data filtering at the data source which may substantially reduce the amount of data needed to transfer into the Spark engine.
Our future goals for Y2 are:

- To quantify the data reduction for some key generic datasets and sample queries, and measure the potential performance acceleration.
- Pursue and develop the use case with Gridpocket

References

[1] OpenStack (Israel, June 2015)
`http://www.slideshare.net/openstackil/ibm-swift`

[2] OpenStack (Tokyo, November 2015)
`https://www.openstack.org/summit/tokyo-2015/videos/presentation/storlets-making-swift-more-software-defined-than-ever`

[3] OpenStack (Vancouver, May 2015)
`https://openstacksummitmay2015vancouver.sched.org/eran_rom.1sq7eu5t?iframe=no#.VkHrBL9OdSA`

[4] IBM Research blog spot
`http://ibmresearchnews.blogspot.com.es/2015/04/storlets-from-research-prototype-to.html`

[5] IBM InterConnect 2015
`https://www.youtube.com/watch?v=4iez3otZH4o`

[6] OpenStack Swift
`http://docs.openstack.org/developer/swift`

[7] IBM Storlets
`https://github.com/openstack/storlets`

[8] Docker technology
`https://github.com/docker/docker`

[9] Linux Containers
`https://linuxcontainers.org/`

[10] Apache Spark SQL
`http://spark.apache.org/sql/`

[11] Univocity
`http://www.univocity.com/`

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

[13] Apache Parquet
`https://parquet.apache.org/`

[14] WSGI
`http://wsgi.readthedocs.org/en/latest/`