2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)

λFlow: Automatic Pushdown of Dataflow Operators Close to the Data

Raúl Gracia-Tinedo, Marc Sánchez-Artigas, Pedro García-López
Universitat Rovira i Virgili
Tarragona, Spain
{raul.gracia,marc.sanchez,pedro.garcia}@urv.cat

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

Filip Gluszak
GridPocket
Sophia Antipolis, France
filip.gluszak@gridpocket.com

***Abstract*—Modern data analytics infrastructures are composed of physically disaggregated compute and storage clusters. Thus, dataflow analytics engines, such as Apache Spark or Flink, are left with no choice but to transfer datasets to the compute cluster prior to their actual processing. For large data volumes, this becomes problematic, since it involves massive data transfers that exhaust network bandwidth, that waste compute cluster memory, and that may become a performance barrier.**

To overcome this problem, we present *λFlow*: a framework for automatically pushing dataflow operators (e.g., `map`, `flatMap`, `filter`, etc.) down onto the storage layer. The novelty of λFlow is that it manages *the pushdown granularity at the operator level*, which makes it a unique problem. To wit, it requires addressing several challenges, such as how to encapsulate dataflow operators and execute them on the storage cluster, and how to keep track of dependencies such that operators can be pushed down safely onto the storage layer. Our evaluation reports significant reductions in resource usage for a large variety of IO-bound jobs. For instance, λFlow was able to reduce both network bandwidth and memory requirements by 90% in Spark. Our Flink experiments also prove the extensibility of λFlow to other engines.

***Index Terms*—data analytics pushdown; object storage; Spark**

I. Introduction

Needless to say, big data analytics frameworks have drawn much attention from both academia and industry in the last ten years. Google's MapReduce [1] was the first major platform developed to process large amounts of data in parallel. But soon enough, dataflow processing engines, such as Spark [2], Flink [3], and Tez [4], came into scene to overcome many of its shortcomings (e.g., simple API, disk-based communications). These analytics engines soon became cloud platforms, where compute and storage are physically disaggregated for a better management, elasticity and security properties [5]. Prominent examples include Amazon with its Elastic MapReduce (EMR) service [6], or the Netflix Big Data platform [7], among others.

A major drawback of compute-storage disaggregation is that dataflow engines require to transfer the whole dataset prior to its processing. While this penalty is low when the input dataset is small, data-intensive jobs can spend a significant fraction of the whole execution time just to transfer data from the storage cluster. Even worse, this transfer stage can exhaust network resources, bottleneck the system, and even have subtle implications on issues such as billing (e.g., Amazon Athena [8] charges customers for the data transferred from S3).

Luckily, many analytics applications only need to operate on a small subset of the original dataset, a transformed version of it, or a specific result. For this reason, the first operations found in many analytics jobs are devoted to prepare the working set. In the dataflow programming model, data scientists achieve this by applying dataflow operators (e.g., `filter`, `map`) on distributed datasets to filter out irrelevant data and transform the extracted data into a suitable format. Here lies an important opportunity for optimization. If this data preparation phase can be moved to the storage layer, resource usage can be greatly cut down. This not only saves network bandwidth, but it can also help to better provision valuable resources in the compute cluster such as memory [9]. By filtering out irrelevant data, the memory requirements in the computer cluster also decrease, and so does the cost of cloud services such as Amazon EMR, which depend on the instance type.

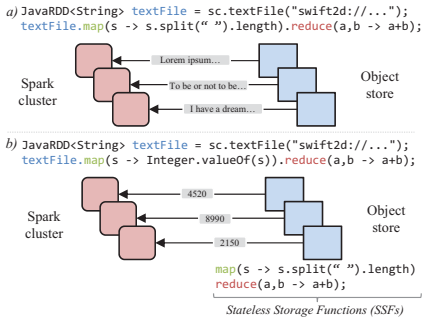


Fig. 1. Concept of dataflow operator pushdown in data-intensive analytics.

A. Scope and Challenges

That being said, the major question we address in this work is whether dataflow operators can be safely pushed down onto the storage layer to save resources. The answer is "yes", which opens a new avenue of investigations.

It must be noted that, although there exists some specialized pushdown mechanisms, e.g., for predicates in Spark SQL, or for Hadoop mappers [5], the main contribution of this work is that the granularity of the pushdown is realized at the operator level, which makes the problem much more challenging. For instance, it requires determining which operations within a job can be safely pushed down to the storage layer, or how to run these operations on the storage side.

To clarify this, we provide the following example. Fig. 1 (a) shows the code for a Spark's `countwords` application that outputs the total count of words from a collection of text

978-1-7281-0912-1/19/$31.00 ©2019 IEEE
DOI 10.1109/CCGRID.2019.00022

documents. This application is clearly data-intensive, as Spark must first ingest all the text files from the storage cluster.

We have observed that, due to their functional nature, many dataflow operators can be totally or partially re-implemented as *stateless functions operating on the data as a stream*. Fig. 1 (b) illustrates this idea. Instead of raw text data, the storage system transfers the "count of words from each file", which are finally added up in Spark. As can be noted, a key difference between both examples is that in Fig. 1 (b), the equivalent to Spark's `map` and `reduce` operators have been offloaded to the storage cluster. The result is not only a significant reduction of bandwidth, but also of the memory footprint, which is of high importance in Spark, and job run times.

B. Contributions

To put it in a nutshell, our goal is to enable the automatic pushdown of dataflow computations close to the storage. Our rationale is based on the fact that the dataflow execution model allows computation to happen in self-contained tasks, each one running on a (horizontal) partition of data. As the parallel tasks instances do not require interim communication between them, pushdown of dataflow operators becomes very amenable to the shared-nothing architecture of cloud storage systems. This endeavor raises several challenges:

Implementation of dataflow operators on the storage side. To run dataflow operators on the storage cluster, we leverage our previous work on Stateless Storage Functions (SSFs) [10], [11]. SSFs are data-parallel. That is, they can run in parallel on a separate data partition as dataflow engines do. Although they are stateless, i.e., they cannot share intermediate state, they are powerful enough to implement most of the IO-bound operators (e.g., `filter`, `map` and `reduce`). Also, SSFs can run either co-located on storage nodes [10], or in a disaggregated manner, so that compute power can scale out independently of the storage, as we showed in [11].

Automatic execution of the operators in the storage cluster. Our solution should be able to infer the "pushable" dataflow operators and use this information to translate parts of the job code into SSFs to be executed on the storage side. This process must be safe in the sense that the job output must be the same with and without applying our solution. Moreover, executing dataflow operators close to the data should be transparent to all the actors: the dataflow processing engine, the storage system and the data scientists that write applications.

We address all these challenges with λFlow: a framework for automatically pushing down dataflow operators close to the data. First, λFlow allows to plug-in job analyzers, which encapsulate the logic for deciding which dataflow operators are safe to be pushed down to the storage layer. Second, λFlow builds a dataflow execution layer for object stores. Our focus is on unstructured storage and object stores, which are a natural fit for big data. Third, λFlow automatically pushes dataflow operators down to the storage layer without human intervention. λFlow is non-intrusive to the operation of both the object store (OpenStack Swift) and the dataflow engines (Spark, Flink), thus retaining their fault tolerance properties.

Our results, obtained with a 12-machine cluster, show that λFlow significantly reduces resource usage in data-intensive analytics. We report data transfer reductions $\geq 90\%$, memory footprint reductions from 32.5% to 89.6%, and job speedups from 1.47x to 3.39x in the tested applications. We also present promising results for Apache Flink, which demonstrate the extensibility of the framework.

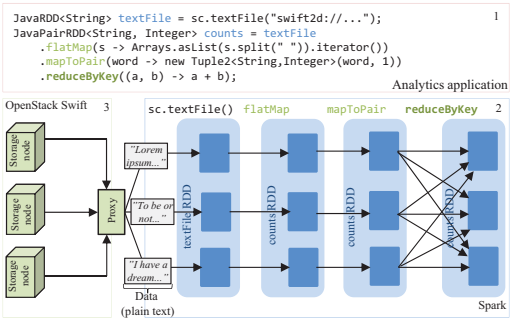


Fig. 2. Example of (1) an input `wordcount` application, (2) the DAG execution of Spark and (3) the data transfers from an object store.

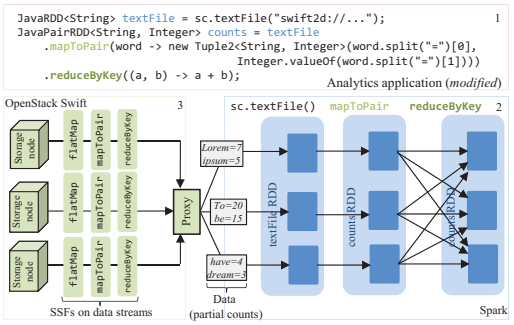


Fig. 3. Rationale behind dataflow operator pushdown: To push down some dataflow operators as SSFs and modify the input application accordingly.

II. Background

A. Dataflow Engines: Spark as an Example

Dataflow computing engines [2], [3], [4] let programmers manipulate distributed collections of objects across a cluster. In Spark, such collections, called *Resilient Distributed Datasets* (RDDs) [12] typically reside in memory to optimize iterative computations on large clusters. The Spark's programming API allows programmers to execute operations on RDDs through: *transformations*, which create a new dataset from an existing one (e.g., `map`, `filter`, `reduceByKey`, etc.); and *actions*, which return a value to the driver program as a result of a computation on the dataset (e.g., `reduce`, `first`, etc.).

Spark's data processing workflow is organized as a *Directed Acyclic Graph* (DAG), where each *vertex* corresponds to a (parallel) data processing *task* in which user-defined code is executed on a separate RDD partition. *Edges* represent a data flow between producer and consumer vertexes. Spark tasks are organized into sequential *stages* that belong to the same RDD.

B. Object Storage with OpenStack Swift

Due to the relevance of unstructured data, object stores are becoming a pervasive substrate for big data storage [13], [14], [15]. A popular example is OpenStack Swift, which is a highly scalable object storage service (see Fig. 2). It offers a simple HTTP RESTful API to store (`Put`), retrieve (`Get`), and

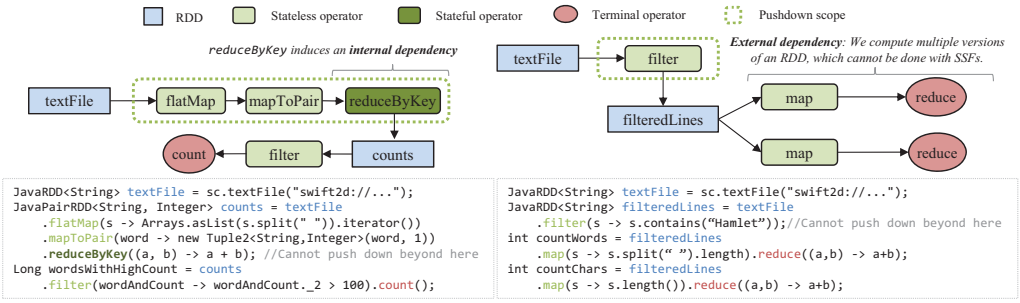


Fig. 4. Examples of internal (left) and external (right) dependencies in analytics applications to be handled by an automatic pushdown mechanism.

delete (`Delete`) objects. Data objects are organized into data containers. Internally, Swift exhibits a two-tier architecture that consists of *proxies and storage nodes*. The former are in charge of authentication and access control of storage requests. Upon reception of a valid request, the proxy server routes it to the corresponding storage nodes for storage.

C. Data-intensive Analytics in the Cloud: A Wordcount Job

To better understand the operation of dataflow engines in disaggregated cloud infrastructures, Fig. 2 depicts the code and the DAG execution stages of a `wordcount` Spark application reading data from an object store (OpenStack Swift). In this case, the application first declares an RDD to hold in memory the text lines from the remote files kept in Swift (`textFile`). Then, a second RDD (`counts`) maintains the counts of each word in the data set by applying a set of transformations on `textFile`. First, we split each line into words (`flatMap`), and then we `map` each word into `(word,1)`-pairs that are further summed up via `reduceByKey`.

When Sparks runs this job, it first detects that the first RDD points to an external dataset that should be transferred from the storage. Thus, each Spark task executes an object transfer that are served in parallel by storage nodes in Swift. As it is readily evident, this process consumes an important amount of inter-cluster network, as well as compute resources related to the execution of transfers by Spark tasks. As visible in the execution DAG of Fig. 2, tasks receiving data objects execute `flatMap` and `mapToPair` in a parallel, stateless fashion without dependencies on other data partitions. Finally, the partial results or word counts for each Spark task are stored in memory and aggregated via a `reduceByKey` operator.

III. Dataflow Operator Pushdown

Here we elaborate on the main design aspects behind λFlow.

A. Stateless Storage Functions as Dataflow Operators

As the pushdown granularity is at the operator level, storage-equivalent functions must preserve the same semantics. This means, for instance, that the storage equivalents must be able to work on independent data partitions (i.e., data parallelism), and be sequentially composable, so that the storage operators can be chained together as in the original dataflow engine.

To meet these requirements, we leveraged our prior work on object storage [10], [11] to implement the storage equivalents to dataflow operators. We called this programming abstraction Stateless Storage Functions (SSFs). To provide the same effect as their analogous dataflow operators, SSFs were designed to be data-parallel and sequentially composable, while ensuring correctness of results. If we revisit the `wordcount` example, this means that the Spark's `reduceByKey` operator should have to receive exactly the same input, irrespective of whether `(word, count)` pairs were computed by Spark (Fig. 2) or by SSFs (Fig. 3).

The execution model for SSFs is analogous to the execution of tasks within a dataflow stage. That is, SSFs are pipelined and directly run on data streams from storage requests without generating persistent intermediate results. As visible in Fig. 3, the execution context of SSFs are regular object requests. This enables SSFs to capitalize on the parallelism available across storage nodes. The stateless nature of SSFs enables us to hide their underlying distributed execution from data scientists.

Further, it is worth to note here that our dataflow execution model goes beyond "classical" active storage [16], [17]. The reason is that as SSFs operate on data streams, instead of at the disk-driver level, we have full flexibility of where to deploy the SSF execution engine, i.e., it can be co-located with the storage nodes [10], or deployed as a separate compute service [11].

B. Automatically Pushing Down Dataflow Operators

As our goal is to decrease the expensive initial data-transfer between the storage and compute clusters, it is readily evident that our pushdown mechanism can only act before the dataset is loaded into the compute cluster. Once the data touches the compute cluster (e.g., a Spark cluster), it is wiser to apply the dataflow operators over the in-memory, partitioned collections for better performance. Our key observation, however, is that for many practical applications, the dataset size tends to reduce notably after the application of the first dataflow operators. In these cases, pushing down the first operators saves resources (bandwidth and memory) and decreases processing burden at the compute cluster. Indeed, CPU utilization is typically low in the storage cluster. Consequently, a pushdown mechanism like ours should bring a global improvement to the datacenter.

We notice here that although there exist some optimization algorithms and planner techniques [18], [19] (e.g., merging multiple `filter` operators) that manipulate RDD operators, none of them addresses the three following specific problems:

Internal dependencies: We distinguish two types of pushable operators: *stateless* and *stateful*. While stateless operators (e.g., `map`, `flatMap`, `filter`, etc.) can be safely pushed down onto the storage layer with no side effects, stateful operators (e.g., `reduceByKey`, `distinct`, etc.) can only be partially pushed down (i.e., *internal dependency*). This would require

storage functions to sharing intermediate state, thus becoming stateful. This is visible in Fig. 4 (left). In this example, Spark still needs to execute a final`reduceByKey` to aggregate the partial "word counts" outputted by the `reduceByKey` storage functions. Consequently, the execution of any SSF beyond the first stateful operator on an RDD (or derived one) would lead to invalid computation results.

External dependencies: Once the transformed data has been transferred to the compute cluster, our pushdown model does not allow any subsequent dataflow operator to execute on the storage side. This limitation has some implications when a user wants to operate on multiple versions of the same dataset.

To better understand this, let us give an example. In Fig. 4 (right), we first instantiate a `filteredLines` RDD that only keeps the text lines containing the word "Hamlet". After this instruction, we derive two different results from the `filteredLines` RDD: the counts of words and the counts of characters of all the lines that contain the word "Hamlet". This is possible for Spark, as `filteredLines` is persistent. Unfortunately, pushing down any or both two `map` operations in `countWords` and `countChars` would lead to incorrect results. To avoid this, divergent versions of a dataset should be detected and only push down those operators that are *common to all RDD versions*. That is, in Fig. 4 (right), it is only safe to push down the `filter` operator on `filteredLines`. We call this mismatch as an *external dependency*.

Code modification: Once dataflow dependencies are resolved, our automatic pushdown mechanism is ready to instruct the storage layer to execute a set of SSFs. But executing SSFs at the storage side requires to adapt the original application code accordingly. In general, the functional style of dataflow APIs enable us to define simple code modification rules in this regard [20]. On the one hand, stateless operators that have been pushed down can be simply removed from the application code. On the other hand, a pushed stateful operator must still be executed in the application code. Such rules are shown in Fig. 3. That is, both `flatMap` and `mapToPair` are removed from the modified application, whereas `reduceByKey` is still present. Automatic pushdown mechanisms may even adopt more elaborated rules. To wit, a `count` terminal operator on an RDD could be replaced by a `reduce` operator on the Spark side plus a `count` operator on the storage layer.

By default, an application could expect raw strings to be read from the storage. However, depending on the executed SSFs, the application's input could become the *string representation of a computed object*. This is shown in Fig. 3 where the modified `wordcount` receives `(word, count)` pairs from the storage as strings. Therefore, an automatic pushdown mechanism should contain the logic to add an extra `map` operator to convert the string representation of objects from the last SSF output (e.g., tuples, lists) into actual objects.

IV. λFlow Architecture

λFlow is a framework for pushing down dataflow operators close to the data, and in particular, onto object stores, to reduce unnecessary network traffic and memory requirements. As a framework, λFlow abstracts users away from many (low-level) issues. To start, it performs code analysis (dependencies) and translates the "pushable" parts into storage functions on behalf of the user. Second, it makes this process automatic: A user just submits a job, and the framework is in charge of running the dataflow operators close to data. Third, the user does not need to manage the cloud analytics infrastructure. Simply put, λFlow remains oblivious to the operation of the object store and the dataflow engine.

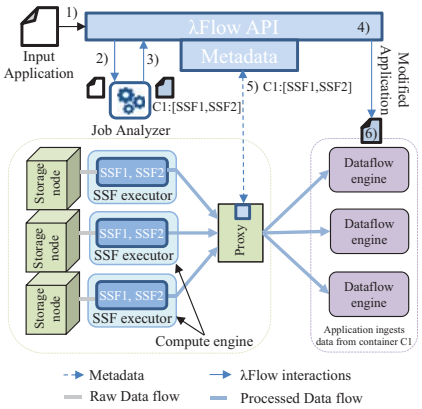


Fig. 5. Architecture and lifecycle of λFlow.

From a user viewpoint, the main λFlow components are:

Job analyzer: A job analyzer receives as input an application's code and performs two main tasks: i) it identifies the dataflow operators to be pushed down, and ii) it rewrites the job's code accordingly. Developing a new job analyzer is, in general, easy to do, as functional dataflow APIs are amenable to automatic parallelization [20].

SSF executor: As input, SSF executors receive a pipeline of SSFs and execute it on data objects in a streaming fashion. SSF executors exploit the joint benefits of lambda processing offered in many programming languages and the potential of computing close to the data [5], [9].

Fig. 5 depicts the architecture and lifecycle of λFlow. The λFlow API exposes the available operations to administrators for managing the extension points of the framework. For instance, this API allows administrators to install a new job analyzer for Flink applications, or to deploy an SSF executor for Python functions at the object store. The state information of λFlow is stored in the metadata store.

Upon submission of an analytics application (Fig. 5, step 1), the API triggers the appropriate *job analyzer* as defined by the administrator. Job analyzers are pluggable and must fulfill a simple contract (Fig. 5, steps 2, 3): i) a job analyzer should expect as input the `original code` of the analytics application, and ii) should return a tuple `<modified code, <container, [SSFs]>>`. Note that λFlow allows to push down SSFs at the level of data container granularity for applications computing on multiple datasets. Thus, job analyzers are pluggable pieces of logic for developing techniques like static analysis on input code of analytics applications [5].

Following its lifecycle, λFlow stores identified SSFs at the metadata store and submits the modified application as an executable binary to the compute cluster (Fig. 5, steps 5, 6).

λFlow comprises a compute engine within the object store that permits to deploy SSF executors in storage nodes, and therefore, to exploit in parallel their spare compute resources. The compute engine intercepts data streams from requests and injects them into SSF executors along with <order=type/body> pairs representing SSFs. Thus, upon a new storage request, a middleware placed at the proxy node interacts with the metadata layer to retrieve the SSFs to be executed. The middleware attaches SSFs to the storage request headers as metadata attributes, which are available as input to SSF executors when the request proceeds. We can co-locate proxies and metadata servers to minimize communication.

Developers can deploy SSF executors in λFlow to execute SSFs close to the storage. They receive as input i) the input/output data streams of object requests, and ii) a set of <order=type/body> tuples describing the signature and body code of the SSFs to execute and their execution order. The developer's responsibility is to correctly execute on a data stream its associated SSFs. Note that λFlow allows several SSF executors to be deployed at the storage side; an administrator defines which ones applies for a given container.

V. Implementation

We implemented λFlow[1] as an extension of Crystal [10], [21], a software-defined storage framework for object stores. We use Redis 3.0.5 to store metadata. As a compute engine at the storage side (Fig. 5), we used the OpenStack Storlets framework [22] that provides OpenStack Swift with the ability to run computations on data streams at storage nodes in a secure and isolated manner, making use of Docker containers [23]. Invoking a storlet on a data object is done in an isolated manner so that the only data accessible by the computation is the object's data and its user metadata. We exploited the raw stream processing capabilities of storlets and its supported runtimes (Java, Python) to develop SSF executors. We also use an interception middleware (WSGI) i) to contact the metadata layer (e.g., once per request to get the SSFs to execute, pick the correct SSF executor), ii) and to inject data streams into the storlets framework, so Swift remains agnostic to the execution of SSF computations.

The development of the λFlow took 6.6K lines of code (LoC): 400 LoC for the API, 5K LoC for the job analyzer, 1K LoC SSF executor storlet, and 200 LoC in other parts.

A. Job Analyzer for Java Analytics Applications

Our job analyzer for analytics applications (Spark Java) supports a subsetset of dataflow operators that we considered as potentially beneficial in terms of bandwidth reduction. This includes filtering (`filter`, `distinct`, `sample`) and simple transformations (`map`). We also considered other operators that do not save up bandwidth by themselves (`flatMap`, `mapToPair`), but that could feed other operators that do reduce transfers (`reduceByKey`). Note that our pushdown mechanism also applies to some final operators or actions on in-memory data sets (e.g., `count`, `first`, `min/max`, and `reduce`). On the other hand, we discarded operators like sorting or grouping (e.g., `sortBy`, `groupBy`) as they would not provide benefits from a resource usage viewpoint, as well as other ones that are not applicable to our model (e.g., `persist`, `cache`, `join`).

Our job analyzer builds a graph data structure (`FlowControlGraph`) per in-memory dataset —such as a RDD— that points to a data container or that is derived from a RDD that does. The root node of such graph contains the RDD name, type and if the RDD points to a data container (e.g., `textFile(container)`) or to another RDD. The rest of nodes are operators executed on it. To ease dependency detection, the `FlowControlGraph` of an RRD has pointers to the root nodes of all the derived RDDs at the operator node from which it was derived. We discard declared RDDs that are unrelated with data containers.

Our job analyzer uses the `FlowControlGraph` to solve dependencies and change code according to the criteria in Section III. The selected dataflow operators are then translated into SSFs. To this end, our job analyzer contains an extensible set of translation rules depending on the dataflow engine API.

Spark: Our job analyzer exploits the great synergy between the Spark RDD API and the Java 8 Stream API, which is the platform used to execute SSFs at the storage side (see Section V-B). Thus, for many RDD operations that can migrated at the storage side, rules are simple as there is a direct equivalent in the Java Stream API. For API calls without a direct Java Stream equivalent, rules encapsulate the translation logic (e.g., `reduceByKey` can be achieved using the Collector API).

Flink: To demonstrate the extensibility of λFlow, we use a Flink's job analyzer that extends Spark's job analyzer. This version implements translation rules for `map`, `filter`, and `reduce` operations in DataSet objects.

Thus, during the translation phase, such rules are applied on each node of the `FlowControlGraph` so that dataflow operators become Java Stream equivalents for the SSF executor. Internally, the job analyzer also updates `FlowControlGraph` nodes representing dataflow operators by adding the signature, type and body of the Java Stream equivalents [29]; this information is needed to compile SSFs at the storage side.

B. SSF Executor Exploiting Java 8 Lambdas

We implemented an SSF executor as a Storlet that allows the execution of Java 8 lambdas passed by parameter on object data streams. As input, it gets from the metadata layer a set of <order, type/body> pairs describing the SSFs to be applied on a data stream and their execution order, as defined by the job analyzer. To this end, the SSF executor compiles on runtime the input code for the SSF input pairs. Upon successful compilation, the SSF executor encodes the byte-level stream into a Java 8 Stream and applies the SSFs on each record. If the compilation raises an unexpected error, the data object is retrieved to the user without applying SSFs.

Our SSF executor contains two optimizations to minimize overheads. First, the encoding overhead related to transforming byte-level data streams into text is performed lazily and triggered when SSFs are to execute. Second, we implemented a cache of SSFs, so a function must be compiled only the first time it reaches the SSF executor. Subsequent executions of a given SSF reuse the compiled function object stored in the cache. Overall, we confirmed that the compilation time overhead of simple Java 8 lambdas falls between 3ms-9ms

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

Application	Dataset	Description	λFlow optimization opportunities
Countwords (`countwords`)	Wikipedia Backups (57.1GB) [24]	Count the number of words of a text-based dataset.	Compute the number of words per object instead of transferring text.
Wordcount (`wordcount`)	Wikipedia Backups (57.1GB) [24]	Get the number of occurrences per word in a text-based dataset.	Compute (`word,count`) pairs for each data object instead of transferring text.
Exploratory query (`query`)	GridPocket IoT Data (138.3GB) [25]	Get the top 10 meters from Paris that consumed most energy in 2012.	Filter the rows that do not belong to 2012/Paris, build (`meter,energy`) tuples and select max values.
Windowed statistics (`win_stats`)	GridPocket IoT Data (138.3GB) [25]	For records within a given period (e.g., 30 mins) in time-series data, get (avg, min, max) triples.	Compute tuples with maximum meter energy measurement per time-slot at the storage side.
Joint log failure correlation (`log_corr`)	NASA Apache logs (95.6GB) [26]	Compute per-hour time-series correlation for bad HTTP requests ($40X/50X$ codes) across two log containers.	Filter non-error lines from logs and compute partial time-series with errors per time-slot.
DNA sequence similarity (`genomics`)	1000 Genomes Project FastQ DNA (177.5GB) [27]	Compute cosine similarities across DNA sequences containing the pattern `GATTACA`.	Select only sequences containing `GATTACA` and transform them into numeric vectors.
K-means (`kmeans`)	UbuntuOne Trace (97.7GB) [28]	Classify UbuntuOne users according to their types of storage operations (k=5, iterations=20).	Discard useless data and perform partial counts of user operations, which are the values to cluster.

TABLE I
ANALYTICS APPLICATIONS USED IN OUR EVALUATION.

in most cases. We consider this overhead to be affordable, especially considering the benefits reported in Section VI.

C. Applicability and Current Limitations

Currently, our SSF executor is limited to the runtime compilation of object types available in the JDK 8 (e.g., primitives, string, collections, etc.) or any other library linked during the lambda executor packaging. However, we do not dynamically handle cases in which lambdas operate on user-defined objects; this would require to push down and compile classes, in addition to SSFs. Our job analyzer does not yet support code expansion. The body of the lambda to be pushed down should contain the actual logic, instead of a function reference.

VI. VALIDATION

A. Experimental Setting

Objectives: Our evaluation addressed the following objectives: i) Demonstrate the automatic dataflow operator pushdown capabilities of λFlow for various analytics applications; ii) Understand the potential benefits of λFlow in terms of data transfer and reduction of memory usage for data-intensive analytics; iii) Examine the impact of λFlow on the execution times of applications, including under multi-tenancy; iv) Verify that λFlow preserves the fault tolerance mechanisms of both dataflow engines and object stores; v) Show that λFlow can optimize multiple dataflow engines.

Analytics: Table I summarizes the analytics applications used in our evaluation. The applications in Table I are real-world use-cases for which λFlow transparently inferred the operators to be executed at the storage side, as well as the modifications required in the input code.

Table I also depicts the public datasets used in our evaluation. To explore the impact of object sizes, we test data objects from 20MB (GridPocket) up to 196MB (Apache logs), and also heterogeneous object sizes within a dataset (DNA sequences). In some cases (e.g., Apache logs), we replicated the original dataset to obtain more significant data volumes.

Platform: In our experimental setting, the object store was a deployment of OpenStack Swift Ocata version with OpenStack Storlets [22] as the computing platform for our SSF executor. We ran analytics applications on Spark 2.1.1. We also made use of the Spark listener system to obtain fine-grained monitoring information of an application lifecycle [30]. We also experimented with Flink 1.3.2. Spark and Flink interact with Swift via the Stocator object storage connector [31].

Application	SSFs (Java 8) executed close to the data	Transfer reduction	Data ingested	Memory reduction
`countwords`	map → reduce	99.978%	12MB	89.8%
`wordcount`	flatMap → filter → map → collect	95.096%	2.8GB	36.9%
`query`	filter → map → filter → map → collect	99.997%	4.2MB	74.4%
`win_stats`	filter → map → collect	89.877%	14GB	32.5%
`log_corr`	map → filter → map → map → map → collect (x2)	99.940%	57MB (x2)	85.9%
`genomics`	filter → filter → map	97.697%	4.1GB	48.2%
`kmeans`	filter → map → filter → map → collect	99.916%	82MB	54.9%

TABLE II
RESOURCE SAVINGS PER APPLICATION ACHIEVED BY λFLOW.

We ran our experiments in a 12-machine cluster formed by 8 Dell PowerEdge 320 nodes (Intel Xeon E5-2403 processors); 1 of them acts as Swift proxy node (28GB RAM, 1TB HDD, 4-core) and the rest are Swift storage nodes (16GB RAM, 2x1TB HDD, 4-core). We used 3 Dell PowerEdge 420 (32GB RAM, 1TB HDD, 12-core) nodes as compute nodes to execute analytics. One large node ran the OpenStack and λFlow services (i.e., API, metadata store). Nodes in the cluster were connected via 1 GbE switched links.

B. Results

Data transfer reduction. We inspect the benefits of λFlow in terms of data transfer reduction in disaggregated infrastructures. Table II depicts the pipeline of SSFs per application that our job analyzer (Section V-A) automatically pushed down to the storage and the obtained data transfer reduction.

Table II demonstrates that dataflow operator pushdown is a practical approach to reduce bandwidth costs for data-intensive analytics. That is, all the applications in our evaluation show a data transfer reduction $\geq 90\%$, considering either one or multiple data containers. Therefore, λFlow can save significant bandwidth resources that can be delivered to other tenants.

Interestingly, the applications in Table II are heterogeneous in terms of data transfer reduction. On the one hand, applications like `countwords` or `query` require a very small amount of data from the whole dataset (order of MBs). On the other hand, other applications like `wordcount` require from every request an output from the SSF pipeline of non-negligible size (e.g., <`word,count`> pairs), which leads to a larger data ingestion. Fig. 6 illustrates such an heterogeneity with time-series plots of bandwidth consumption for several applications. All in all, we observe that λFlow greatly reduces

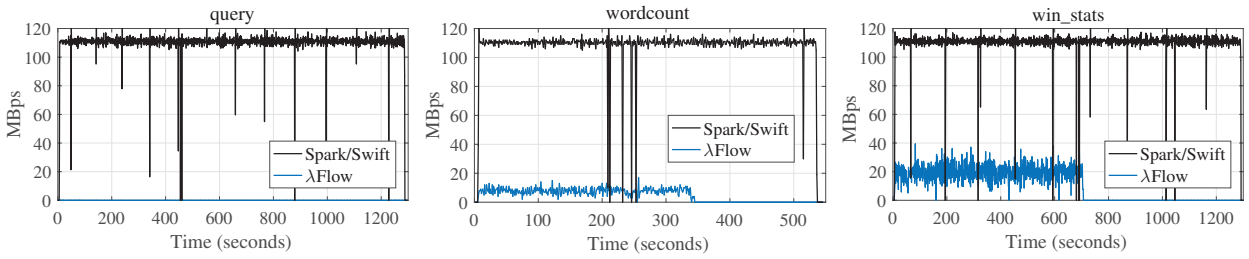


Fig. 6. Time-series view of bandwidth consumption during data transfer stages of vanilla Spark/Swift and λFlow for several applications.

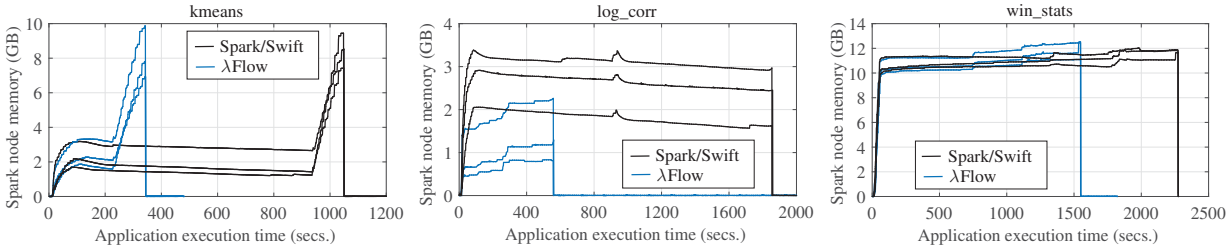


Fig. 7. Time-series per-node memory usage for three applications w/wo λFlow.

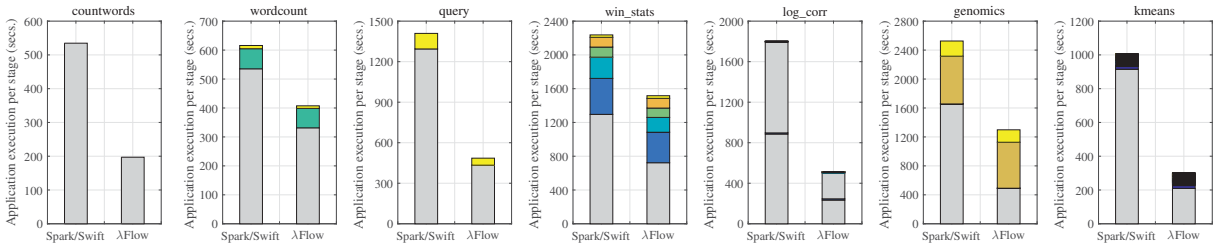


Fig. 8. Execution times per stage for all applications considered w/wo λFlow. Grey areas in stacked bars represent data transfer stages.

bandwidth costs compared to vanilla Spark/Swift.

Table II shows that the λFlow was able to configure complex SSF pipelines at the storage side. From the SSF pipelines, we realize that the data reduction for each application is achieved in a different way. That is, most applications filter useless rows (`filter`), others select a particular part of a record (`map`), and other applications directly compute partial results (e.g., sums, counts, maximum finding) for each object (`reduce`). This goes beyond ETL tasks and opens the door to investigate the feasibility of richer types of SSFs in λFlow.

Memory footprint. We evaluate how λFlow helps to reduce the memory usage of data-intensive applications. Table II includes the memory usage reduction that λFlow achieved for all the tested applications (using `free` linux utility). Specifically, the memory usage of an application A on a node n was calculated as $M^n_A = \sum_{t=0}^{t^{end}} m(t) - m(0)$, where $m(t)$ provides the memory consumption of a node n at a certain point in time t until the end of the experiment (t^{end}). Thus, we calculated the total memory usage reduction of A with λFlow as $\sum_{n \in \mathcal{N}} M^n_{A_{\lambda Flow}} / \sum_{n \in \mathcal{N}} M^n_{A_{vanilla}}$.

Our results show that λFlow exhibits remarkable memory usage reduction for all the applications, ranging from 32.5% (`win_stats`) up to 89.8% (`countwords`). This proves that pushing dataflow operators down to the storage cluster does not only provide important bandwidth savings, but also greatly reduces memory usage for data-intensive analytics. In fact, Table II suggests that there is a correlation between the reduction levels of data transfers and memory usage.

Fig. 7 shows the memory usage behavior of three applications in a time-series fashion with and without λFlow. We identified that the memory reduction achieved by λFlow for these applications exhibits two different profiles. On the one hand, all the applications consume less memory given that λFlow reduces their execution times. Many applications allocate a similar amount of memory during their execution with and without λFlow, but for shorter periods of time. On the other hand, we noted that applications exhibiting high data transfer reduction (e.g., `query`, `log_corr`) also had reduced the memory allocation for Spark RDDs, irrespective of the application's execution time. This can be confirmed by comparing the maximum memory consumption values of Spark nodes with and without λFlow. In conclusion, λFlow can allow Spark to significantly reduce memory consumption for data-intensive analytics.

Application completion times. Next, we examine how the dataflow operators pushed down to the object store, as well as the changes in the original application code, impact application execution times. Fig. 8 show the execution times per-stage for all the applications in Table I.

At first glance, Fig. 8 shows that λFlow clearly improves execution times in all the tested applications. Specifically, we measure the speedup S for an application A as $S = T^A_{vanilla}/T^A_{\lambda Flow}$, where T^A stands for an A's execution time. Given this metric, λFlow achieves speedups that range from

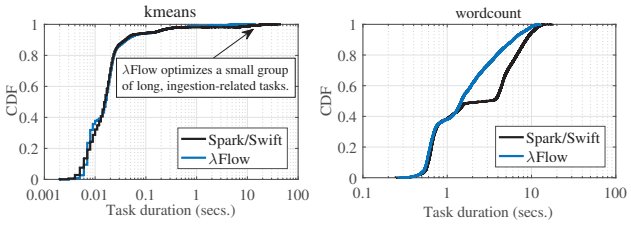
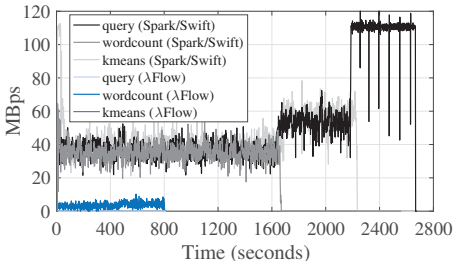
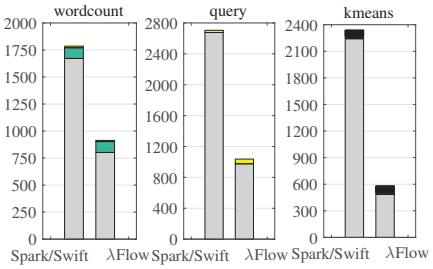


Fig. 9. Distribution of application task execution times w/wo λFlow.



(a) Bandwidth consumption under multi-tenancy.

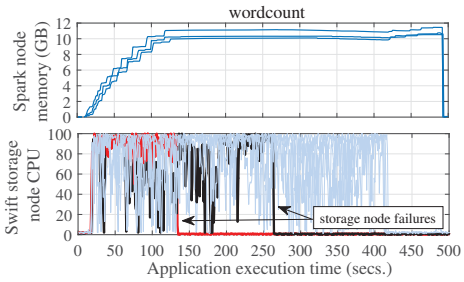


(b) Execution times per-stage of parallel applications.

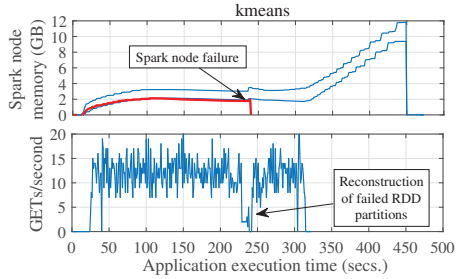
Fig. 10. Execution times per stage and time-series bandwidth for 3 applications in parallel w/wo λFlow.

$S = 1.47$x (`win_stats`) to $S = 3.39$x (`log_corr`). In addition to the speedup gain in the `win_stats` application, GridPocket also benefits from an speedup of $S = 2.90$x for the `query` application. We note that there is a strong correlation between the percentage of data transfer reduction (Table II) and the speedup achieved by λFlow. This is because the data transfer reduction achieved by executing SSFs at the object store removes the network bottleneck in our cluster. In future work, we will model the implications of the infrastructure and the computational complexity of SSF pipelines on speedup gains with λFlow.

λFlow speedups come from boosting application stages related to data transfers (depicted as grey stacked bars in Fig. 8). To this end, the job analyzer extracts from the code operators that are safe to push down to the storage side on RDDs related to data containers. Thus, the code of the input application is modified by: i) removing the operators to be executed on the storage side, and ii) adding an initial transformation to adapt the application to the new data format computed by SSFs (if necessary). Appreciably, these code changes only impact on the first stages of applications, which are those that execute data transfers. That is, the modified code version of an application shows the same sequence of execution stages than the original one, thus preserving their behavior beyond the stages related to data transfers.



(a) Storage node failures executing `wordcount`.



(b) Compute node failures executing `kmeans`.

Fig. 11. Execution of applications with λFlow under server failures.

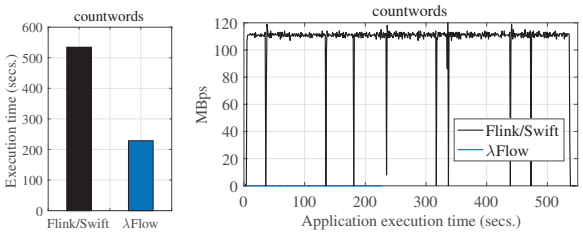
Fig. 9 describes the execution time of Spark tasks. Conversely to what happens with stages, task execution times exhibit greater differences among applications. This mainly depends on the fraction of tasks related to data transfers, for which applications exhibit different profiles. In applications like `kmeans`, most of the tasks are short and related to iterative in-memory computations (algorithm convergence). Clearly, λFlow only improves the execution of a small group of long tasks (distribution tail in Fig. 9) related to data transfers. However, in applications such as `wordcount`, there is a higher fraction of data-transfer tasks. Such tasks exhibit shorter execution times, whereas tasks related to in-memory computations (shuffle, reduce of `(word,count)` pairs) are similar to the original application.

Multi-tenant workloads. Next, we explore the benefits of λFlow in shared analytics infrastructures. Fig. 10 shows the execution times and bandwidth usage of 3 applications in parallel (one application per compute node).

Fig. 10(a) shows that sharing resources in a multi-tenant scenario leads to significantly higher execution times. To wit, `query`, `wordcount` and `kmeans` execution times are 1.94x, 2.92x and 2.83x larger compared to when they ran alone in the cluster. In part, this slowdown was due to the sharing of compute resources. However, the most important bottleneck was network bandwidth in our environment. This is why most of an application's execution time was related to data transfer.

Fig. 10(a) also demonstrates that λFlow works well in multi-tenant scenarios. Compared to vanilla Spark/Swift, λFlow achieves execution time speedups of $S = 2.61$x, $S = 1.95$x, and $S = 3.98$x in `query`, `wordcount` and `kmeans` applications, respectively (Fig. 10(b)). In our cluster, these speedups are similar or even better than the ones obtained when running applications alone.

Fault tolerance. We inspect the execution of λFlow in the



(a) Execution time. (b) Data transfer reduction w/wo λFlow.

Fig. 12. Pushdown experiment with Apache Flink (`countwords` application).

presence of failures, which are common at large scale. As λFlow is non-intrusive to the operation of Swift and Spark, we wanted to show that both systems retain their fault tolerance.

Fig. 11(a) shows the execution of `wordcount` under two storage node failures (out of seven storage nodes). The upper plot shows the memory usage of Spark nodes, whereas the lower plot describes the CPU consumption of storage nodes executing SSFs. Appreciably, failed storage nodes are easy to identify since they stop consuming CPU power before the experiment terminates. In particular, the presence of these two consecutive failures lengthened the execution time by 15.7% compared with the non-failure scenario. The reason is that failed storage nodes reduced the overall compute power available for running SSFs. Anyway, the application could end successfully, as we used a 3-way replication scheme to tolerate up to two failures of the same data container.

Fig. 11(b) shows induced failures in Spark executing a `kmeans` application. As illustrated in the memory usage plot, we disconnected one Spark node from the cluster (out of three) when the data transfers were completed. Upon the failure event, Spark again triggered the data-transfer phase to reconstruct the missing RDD partitions stored at the failed node. Once the RDD was reconstructed, the remaining Spark nodes executed the clustering algorithm successfully.

In conclusion, we confirm that data scientists can exploit λFlow at large-scale, given that both Spark and Swift retain their respective fault tolerance mechanisms.

Flink's Proof-of-concept. λFlow has been designed to optimize multiple dataflow analytics engines. To demonstrate this, we applied our Flink's job analyzer (see Section V-A) to the `countwords` application (Fig. 12).

Fig. 12(b) shows that λFlow can save bandwidth resources for the Flink `countwords` application similarly to what we saw for Spark (99.978% data transfer reduction, as the same dataflow operators are pushed down). Moreover, λFlow achieves a speedup of 2.34x. Therefore, our Flink experiments confirmed that λFlow can optimize multiple engines embracing the dataflow programming model.

VII. Related Work

A key goal of λFlow is to reduce resource consumption of data-intensive analytics. A main aspect to achieve this goal relates to the "data ingestion problem", which has gained traction in the last years [32]. From the analytics viewpoint, some works optimize the workflow of MapReduce programs to minimize data movements [33]. In fact, λFlow acts as an optimization mechanism on input applications to push down dataflow operators that reduce data transfers. Other works augmented Hadoop/Spark with indexes on structured storage (HBase) for pruning useless data partitions [34]. Interestingly, indexing mechanisms in object stores could also help to filter out entire data objects [35], [36]. This technique is complementary with our work, as λFlow computes on data streams of the data objects that are required by the index.

From a storage perspective, other works have focused on interfacing Hadoop with enterprise file systems to bridge the gap between legacy data stores and compute clusters [37], [38], or even replacing HDFS by optimized file systems to minimize the impact of compute/storage disaggregation [39]. Conversely, λFlow exploits the concept of SSF as a building block to optimize data intensive analytics. Although computing close to the data is a well-known technique (e.g., active storage, co-processors) [16], [17], λFlow is the first work to mimic the dataflow execution model close to the data.

Executing SSFs close to the data may be seen as a form of predicate/user-defined function pushdown in the database literature [40], [41]. Regarding unstructured data, our previous efforts [9] aim at translating an equivalent concept to SQL pushdown in object storage. In fact, Spark SQL can be extended to pushdown predicates to new data sources with compute power [19]. However, λFlow goes beyond pushdown techniques: i) it pushes down general dataflow operators defined on in-memory datasets (not SQL-specific); ii) as the application code analysis occurs off-line, λFlow can be extended to multiple dataflow engines (e.g., Spark, Flink); and iii) the storage system is agnostic to the operation of λFlow, as it only interacts with the compute engine running SSFs [22].

Others works focus on how to coordinate the distributed execution of analytics computations. For instance, Gaia [42] is a geo-distributed machine learning system that automatically scatters computations within a datacenter and across datacenters to minimize data transfers in wide-area analytics. Rabkin et al. [43] propose a close-to-the-data abstraction called DataCube. DataCubes enable executing filter and transformations to optimize bandwidth in wide-area stream analytics. But λFlow does not aim at replacing existing compute/storage systems. Instead, it augments existing systems with automatic operator pushdown to improve efficiency.

Gkantsidis et al. [5] present Rhea, a system that filters data on the storage side to minimize data ingestion in Hadoop. Rhea provides a static analysis tool to automatically infer the filters to apply at the storage cluster. Despite sharing the same spirit, our work presents key differences compared to Rhea. First, Rhea focuses on optimizing SQL filters/selects in Hadoop, whereas λFlow targets dataflow programming APIs, thus coping with diverse analytics applications and engines. λFlow exploits parallel execution of SSFs on storage nodes, while Rhea relies on a proxy to perform filtering tasks that may become a bottleneck and a single point of failure.

In summary, λFlow is the first framework that automatically pushes down dataflow operators directly inside the storage layer. We believe that the emergence of the dataflow programming model [44] supports the ideas behind λFlow, motivating its use for optimizing multiple dataflow engines.

VIII. Conclusion and Future Work

In this work, we have introduced λFlow: a framework for automatically pushing down dataflow operators close to the data. λFlow leverages the notion of stateless storage function to run dataflow operators (Spark) on an disaggregated storage cluster (OpenStack Swift), thus reducing their consumption of bandwidth and memory resources, while boosting application performance. In our implementation, both Swift and Spark remain fault-tolerant and agnostic to the pushdown process.

Our experiments have demonstrated that λFlow can achieve significant reductions in resource consumption for a large variety of data-intensive jobs (data transfer reductions $> 90\%$ and memory usage reductions from 32.5% to 89%). λFlow is also able deliver performance improvements (i.e., job speedups from 1.47x to 3.39x), specially in multi-tenant scenarios where bandwidth is scarce (i.e., job speedups up to 3.98x), as shown by our evaluation. Our Flink experiments have also proven the extensibility of λFlow.

λFlow also opens new research directions: i) new theoretical models to predict the gain of pushing down dataflow operators to the storage layer; and ii) novel elastic deployment models for SSF executors in the cloud [11].

Acknowledgment

This work has been partly funded by the EU project H2020 "CloudButton: Serverless Data Analytics Platform" (825184) and Spanish research project "Software Defined Edge Clouds" (TIN2016-77836-C2-1-R).

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Comm. of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *USENIX HotCloud'10*, vol. 10, no. 10-10, p. 95, 2010.
[3] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.
[4] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache tez: A unifying framework for modeling and building data processing applications," in *ACM SIGMOD'15*, 2015, pp. 1357–1369.
[5] C. Gkantsidis, D. Vytiniotis, O. Hodson, D. Narayanan, F. Dinu, and A. I. Rowstron, "Rhea: Automatic filtering for unstructured cloud storage." in *USENIX NSDI'13*, vol. 13, 2013, pp. 2–5.
[6] "Amazon elastic mapreduce," https://aws.amazon.com/en/emr.
[7] "Big data at netflix: Faster and easier," https://conferences.oreilly.com/strata/big-data-conference-ny-2015/public/schedule/detail/42498.
[8] "Aws athena pricing," https://aws.amazon.com/en/athena/pricing.
[9] Y. Moatti, E. Rom, R. Gracia-Tinedo, D. Naor, D. Chen, J. Sampe, M. Sanchez-Artigas, P. Garcıa-Lopez, F. Gluszak, and E. Deschdt, "Too big to eat: Boosting analytics data ingestion from object stores with scoop," in *IEEE ICDE'17*, 2017, pp. 309–320.
[10] R. Gracia-Tinedo, J. Sampé, E. Zamora-Gómez, 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, pp. 243–256.
[11] J. Sampé, M. Sánchez-Artigas, P. García-López, and G. París, "Data-driven serverless functions for object storage," in *ACM/IFIP/USENIX Middleware'17*, 2017.
[12] 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*, 2012, pp. 2–2.
[13] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran, "Object storage: The future building block for storage systems," in *IEEE MSST'05*, 2005, pp. 119–123.
[14] "Amazon s3," https://aws.amazon.com/en/s3.
[15] L. Rupprecht, R. Zhang, B. Owen, P. Pietzuch, and D. Hildebrand, "Swiftanalytics: Optimizing object storage for big data analytics," in *IEEE IC2E'17*, 2017, pp. 245–251.
[16] E. Riedel, G. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia applications," in *VLDB'98*, 1998, pp. 62–73.
[17] J. Piernas, J. Nieplocha, and E. J. Felix, "Evaluation of active storage strategies for the lustre parallel file system," in *ACM/IEEE Supercomputing'07*, 2007, p. 28.
[18] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *VLDB'09*, vol. 2, no. 2, pp. 1626–1629, 2009.
[19] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *ACM SIGMOD'15*, 2015, pp. 1383–1394.
[20] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. R. Pietzuch, "Making state explicit for imperative big data processing." in *USENIX ATC'14*, 2014, pp. 49–60.
[21] "Crystal GitHub," https://github.com/Crystal-SDS.
[22] "OpenStack Storlets," https://github.com/openstack/storlets.
[23] "Docker," https://www.docker.com.
[24] "Wikimedia downloads," https://dumps.wikimedia.org/.
[25] "Gridpocket iot data," https://github.com/gridpocket/project-iostack.
[26] M. F. Arlitt and C. L. Williamson, "Web server workload characterization: The search for invariants," *ACM SIGMETRICS Performance Evaluation Review*, vol. 24, no. 1, pp. 126–137, 1996.
[27] "1000 genomes project," http://www.internationalgenome.org/.
[28] R. Gracia-Tinedo, Y. Tian, J. Sampé, H. Harkous, J. Lenton, P. García-López, M. Sánchez-Artigas, and M. Vukolic, "Dissecting ubuntuone: Autopsy of a global-scale personal cloud back-end," in *ACM IMC'15*, 2015, pp. 155–168.
[29] "Javaparser github," https://github.com/javaparser.
[30] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks." in *USENIX NSDI'15*, vol. 15, 2015, pp. 293–307.
[31] G. Vernik, M. Factor, E. K. Kolodner, E. Ofer, P. Michiardi, and F. Pace, "Stocator: a high performance object store connector for spark," in *ACM SYSTOR'17*, 2017, p. 27.
[32] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum, "In-situ mapreduce for log processing," in *USENIX ATC'11*, 2011, p. 115.
[33] E. Jahani, M. J. Cafarella, and C. Ré, "Automatic optimization for MapReduce programs," *VLDB'11*, vol. 4, no. 6, pp. 385–396, 2011.
[34] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: making a yellow elephant run like a cheetah (without it even noticing)," *VLDB'10*, vol. 3, no. 1-2, pp. 515–529, 2010.
[35] P. Ta-Shma, "Using pluggable spark sql filters to help gridpocket users keep up with the jones' (and save the planet)," https://spark-summit.org/eu-2017/events/using-pluggable-apache-spark-sql-filters-to-help-gridpocket-users-keep-up-with-the-jones-and-save-the-planet.
[36] A. Shanbhag, A. Jindal, S. Madden, J. Quiane, and A. J. Elmore, "A robust partitioning scheme for ad-hoc query workloads," in *ACM SoCC'17*, 2017.
[37] 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.
[38] 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*, 2011, p. 67.
[39] M. Mihailescu, G. Soundararajan, and C. Amza, "Mixapart: decoupled analytics for shared storage systems." in *USENIX FAST'13*, 2013, pp. 133–146.
[40] J. M. Hellerstein and M. Stonebraker, "Predicate migration: Optimizing queries with expensive predicates," in *ACM SIGMOD'93*, vol. 22, no. 2, 1993.
[41] E. Friedman, P. Pawlowski, and J. Cieslewicz, "SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions," *VLDB'09*, vol. 2, no. 2, pp. 1402–1413, 2009.
[42] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-distributed machine learning approaching lan speeds." in *USENIX NSDI'17*, 2017, pp. 629–647.
[43] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman, "Aggregation and degradation in jetstream: Streaming analytics in the wide area." in *USENIX NSDI'14*, vol. 14, 2014, pp. 275–288.
[44] "Apache beam," https://beam.apache.org.