AJIRA: a Lightweight Distributed Middleware for MapReduce and Stream Processing

Jacopo Urbani*, Alessandro Margara*, Ceriel Jacobs*, Spyros Voulgaris*, Henri Bal*
*Department of Computer Science
Vrije Universiteit Amsterdam, The Netherlands
{jacopo, a.margara, ceriel, spyros, bal}@cs.vu.nl

***Abstract*—Currently, MapReduce is the most popular programming model for large-scale data processing and this motivated the research community to improve its efficiency either with new extensions, algorithmic optimizations, or hardware.**

In this paper we address two main limitations of MapReduce: one relates to the model's limited expressiveness, which prevents the implementation of complex programs that require multiple steps or iterations. The other relates to the efficiency of its most popular implementations (e.g., Hadoop), which provide good resource utilization only for massive volumes of input, operating suboptimally for smaller or rapidly changing input.

To address these limitations, we present AJIRA, a new middleware designed for efficient and generic data processing. At a conceptual level, AJIRA replaces the traditional map/reduce primitives by generic operators that can be *dynamically* allocated, allowing the execution of more complex *batch* and *stream* processing jobs. At a more technical level, AJIRA adopts a distributed, multi-threaded architecture that strives at minimizing overhead for non-critical functionality.

These characteristics allow AJIRA to be used as a single programming model for both batch and stream processing. To this end, we evaluated its performance against Hadoop, Spark, Esper, and Storm, which are state of the art systems for both batch and stream processing. Our evaluation shows that AJIRA is competitive in a wide range of scenarios both in terms of processing time and scalability, making it an ideal choice where flexibility, extensibility, and the processing of both large and dynamic data with a single programming model are either desirable or even mandatory requirements.

I. Introduction

The current data deluge and dynamicity call for data processing models and infrastructures that offer (*i*) *flexibility* and *expressiveness*, (*ii*) *scalability*, and (*iii*) *efficiency*, to support complex and heterogeneous computations and low-latency, on-the-fly processing.

MapReduce [9], one of the most prominent models in this area, and Hadoop, its open-source implementation, offer a simple high-level interface to design and deploy programs over large clusters, automatically handling inter-node communication, fault tolerance, and synchronization. MapReduce and its derivatives achieve high performance by imposing a fixed and predefined execution schema, which limits their expressiveness and applicability. For example, implementing iterative or recursive algorithms using MapReduce is neither easy nor efficient [26].

The limitations of MapReduce stimulated significant research [19], which resulted in new extensions, algorithmic optimizations, and harnessing of new hardware (e.g., GPUs). To the best of our knowledge, none of the proposed works address *all* the requirements identified above. For example, the Map-Reduce-Merge extension [36] improves the efficiency and scalability of MapReduce by introducing a new "merge" operation, but does not address its efficiency as it relies on the same architecture.

In this paper, we contribute to this area by proposing AJIRA, a new middleware that focuses on satisfying the three requirements identified above. In particular, AJIRA provides good *scalability* while addressing two main limitations of the MapReduce model: *limited expressiveness* and *efficiency* [32].

With respect to expressiveness, AJIRA introduces a new generic processing model based on multiple actions that can be piped in concurrent and distributed chains of operations. A significant novel feature of our model is that each action can create new workflows at *runtime*, and use their output in the original workflow. This opens the door to the implementation of more complex programs than currently possible.

With respect to efficiency, AJIRA is designed to be significantly fast when dealing with small to medium data volumes, while remaining competitive on larger datasets. This was achieved by cutting down on unnecessary overhead, avoiding the use of a coordination master, and minimizing the extent of fault tolerance. Furthermore, contrary to Hadoop, AJIRA does not require a dedicated cluster. Instead, it is targeted to be used as a library, making deployment on Infrastructure-as-a-Service infrastructure much lighter. Efficiency for small to medium scale data is becoming increasingly relevant as MapReduce is being adopted for tasks beyond massive data analytics. Recent analysis of the Hadoop usage in the production clusters of large companies report that the median input size of MapReduce jobs is under 14GB [2], and that most of these jobs are very unlikely to fail [12].

This makes AJIRA a useful tool for a large variety of data processing tasks, and represents a first attempt of unifying expressiveness, efficiency, and scalability within a single programming model. We have already successfully used AJIRA not only for batch processing, but also for stream processing, whose requirements (e.g. low latency) often conflict with batch processing.

As a proof of concept, we implemented a number of batch and stream processing algorithms on top of AJIRA, and compared our system against four state of the art systems, i.e. Hadoop, Spark, Storm and Esper. Our evaluation shows that AJIRA is competitive both in terms of execution time and scalability in both scenarios. More in particular, AJIRA outperformed Hadoop with up to $370\times$ speedup in the best case, considering up to 32 machines and up to 19 TB of input data. At the same time, it proved to be between two and five times faster than Storm and Esper in processing streams of data. These results offer a first empirical demonstration of the value of the framework in combining the requirements for efficient, scalable, and flexible computation in a single solution.

The rest of the paper is organized as follows. Section II presents an overview of AJIRA, focusing on its programming model. Section III describes the architecture and implementation of AJIRA, while Section IV provides an evaluation of our middleware. Section V and Section VI conclude the paper, discussing related and future work.

II. THE AJIRA SYSTEM MODEL

AJIRA is a complex framework where multiple components interact with each other in a parallel and distributed environment, to perform computations in either *batch* or *stream* mode.

In AJIRA, data is represented by means of *tuples* composed of an arbitrary number of elements. AJIRA provides simple types of elements, such as integers, strings, and booleans, but the developer is free to define and use more complex data types if needed. The computation is performed by multiple *actions*, which are executed in sequences called *chains*. New chains can be created at *runtime*, and *merged* or *partitioned* in multiple concurrent flows. In the remaining of this section, we describe each of these concepts in more detail.

A. Actions

Actions are the basic computational units of AJIRA. An action is an operator that takes *one* tuple as input, and returns *zero*, *one*, or *more* tuples as output.

Each action inherits from an abstract class called `Action`, and defines its behavior by overriding the `process()` method, which receives a single tuple in input and produces zero, one, or more tuples as output. A number of actions for performing standard tasks (e.g., *Projection* and *GroupBy*) are provided by the framework, while the programmer typically has to implement custom actions for the problem in question.

The framework allows setting arbitrary parameters for each action instance individually. For example, an action that detects the top-k values of all its input tuples could be instantiated with different values of k.

Two important methods that actions can override are `startProcess()` and `stopProcess()`. The former is called exactly once for each instance of the class, before the framework starts feeding tuples to its `process()` method. This method can be used to initialize data structures required during processing. The latter, `stopProcess()`, is invoked to notify the action that it will not receive any more input tuples. An action can use this method to apply garbage collection on open resources (files, network connections, etc.), to compute statistics about the processing, or to clean its data structures before turning off. Notice that during the execution of the `stopProcess()` method, an action can still output new tuples. This can be useful to send some final tuples at the end of the flow with some aggregated information.

In the AJIRA model the programmer cannot assume that an action will be instantiated only once (although this *can* be enforced if required). Often, to improve performance, multiple instances of the same actions are created in parallel and are fed with *parts* of the entire input. Because of this, it is important that the code within an action is *self-contained*, so that `startProcess()` and `stopProcess()` can be invoked multiple times within the same program.

B. Chains

Typically a single action is not sufficient to implement the entire computation. Because of this, AJIRA allows actions to be piped into sequential *chains*, so that the output of one action becomes the input of the following one.

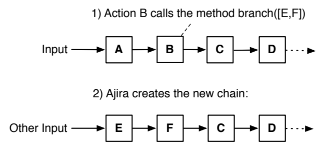
During every call of the `process()`, actions take exactly one tuple as input. If they produce one tuple as output, then the framework feeds to the next action in the chain. If an action, however, produces more than one tuple, these tuples are fed to the next action *one at a time*. That is, if an action produces k tuples, then the next action is sequentially invoked k times. This holds also for $k = 0$, that is, an action has the option to "cut off" the flow of a specific input by simply returning without any output tuples. This allows an action to act as a filter to the flow of computation.

C. Input Modules

The AJIRA chain executor takes the input for the first action in the chain using some *input modules*, which provide an abstract representation of a data source from which the program can retrieve tuples to process in the form of iterators.

Input modules constitute AJIRA's interface to arbitrary data sources. In that role they are in charge of accessing a data source, reading data in its native format, parsing it, and packing it into tuples that are subsequently handed to the first action of a chain. In addition, input modules have a second role. They decide on which physical node a newly created chain should be executed. This design decision is in line with AJIRA's explicit goal of constituting a *data-driven* computation framework. Placing computation *close* to data is crucial to efficiency, and input modules are the entities that possess the information about the data distribution. Besides data location, input modules can access additional metrics for selecting a node for execution, such as nodes' current load and resource availability.

AJIRA provides a number of default input modules (e.g., for accessing data from files), but the developer can easily define custom input modules to interact with arbitrary types of sources, such as databases, network connections, data streams (e.g., a Twitter feed) or any other proprietary source. Multiple input modules can be used within the same program, thus allowing the programmer to combine data from different data sources in a seamless way.


1) Action B calls the method branch([E,F])
Input
A
B
C
D
2) Ajira creates the new chain:
Other Input
E
F
C
D
(a) Branch

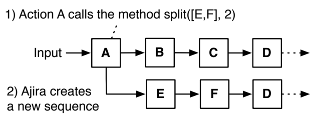
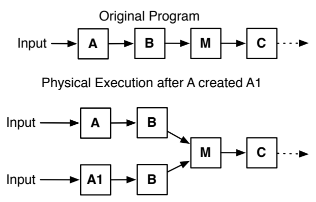
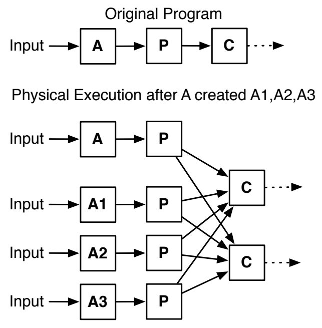

1) Action A calls the method split([E,F], 2)
Input
A
B
C
D
2) Ajira creates a new sequence
E
F
D
(b) Split

Fig. 1: Use of the branch and split primitives.



(a) Merge



(b) Partition

Fig. 2: Use of the merge and partition actions.

D. Flow of Computation

An AJIRA program starts by executing a single chain of actions, called the *root chain*. During the execution, any action is allowed to spawn additional chains, which will be executed concurrently on the same or a different physical node. In this way, the developer can implement complex execution plans where multiple concurrent flows of computation are created, split, or merged at runtime.

AJIRA offers two primitives for dynamic workflow management, *branch* and *split*, and two actions called *partition*, and *merge*, described below.

1) Branching: The *branch* primitive allows an action A to start a new chain. To this purpose, first A defines which actions should be executed in the new chain, and then it invokes a specific function to create a new branch. Notice that A defines only the action(s) for the new chain to *start* with, which are by default followed by A's successor actions in the parent chain.

To better explain how *branch* works, we report in Fig. 1a an example of branching. Suppose that the root chain consists of the sequence of actions `A`, `B`, `C`, and `D` (denoted `[A,B,C,D]` from now on), and action `B` performs a branch specifying some other input module and actions `E` and `F`. This generates a new chain, which runs in a *new thread*, either on the same or on a different physical machine, feeding the new input's tuples through the action sequence `[E,F,C,D]`. The actions `C` and `D` of the newly formed chain are separate instances, independent from the ones of the original chain. Apart from belonging to the same class, they do not share anything else.

Forcing a new chain to inherit the rest of the original chain (e.g., actions `C` and `D`) might seem counterintuitive. However, this is a very effective mechanism since it allows an action to parallelize its own computation by "cloning" its chain, without having to know its successor actions. Furthermore, *branching* primitive allows an effective implementation of iterations by simply having a "condition" action which continues to branch until the condition is no longer valid.

In our use cases the branch primitive turned out to be very useful to parallelize the computation. For example, consider action `B` being replaced by `ReadDirectory`, an action reading the content of a directory with n files. Using the *branch* primitive, this action can speed up the computation by first reading the list of files of the directory, and then creating n branches using another action `ReadFile` and passing a different filename to each branch. This way, `ReadDirectory` introduces parallelization in an elegant and transparent way, without knowing which actions come after it. With the branch primitive, the programming model encourages the user to split up the computation in independent blocks and allows each block to introduce parallelism while being agnostic to the rest of the computation, consequently maximizing its chances to be reused.

2) Splitting: The *split* primitive is an alternative way for an action to spawn a new chain. It is similar to branching, except for the following three points. First, rather than supplying the new chain with a new input module, it is fed with the output of the action performing the split. Second, the new chain is always executed on the same physical node as the original chain. And third, although the new chain does inherit the remaining actions of the original chain, the initiator of the split may require a number of inherited actions to be skipped.

For each tuple produced by an action that has performed a split, the action is free to choose whether it should be propagated along its original chain, along the new chain, or both of them. An action can also split multiple times, feeding its output to multiple parallel processing paths.

Fig. 1b depicts an example of a split operation. Action `A` invokes the split primitive with the sequence `[E,F]`. As in the case of the branch, such a call generates a new concurrent sequence that starts with `E` and `F`. However, in this case, the sequence does not receive its input from an input module, but from action `A`. The newly generated chain inherits all the following actions from the original chain but the first n, where n is an (optional) parameter defined in the *split* primitive. For example, in Fig. 1b, n equals to two: This means that after `[E,F]` the new sequence will contain `D` and all of its following actions in the original chain. If n is not specified, then no action is inherited. In our example, the new sequence would contain only `E` and `F`.

The *branch* and *split* primitives create multiple flows of computation that can be executed on the same or different machines. In the following, we discuss how such concurrent flows can be re-partitioned or merged in a single one.

3) Merging: When an action invokes the branch or split primitive, a new execution flow is created, which is completely independent from the existing ones. This is a precise design choice that enables a straightforward implementation where the same sequence of operations —i.e., the same chain— is executed in parallel on different inputs.

However, such behavior conflicts with all those cases where it is necessary that all the data is processed by a single flow, e.g., to compute an aggregation function. To deal with these cases, AJIRA offers a special *merge* action that ensures that multiple flows started in previous actions are merged back into a single flow. As an example, consider the execution reported in Fig. 2a. Here, the program starts executing a root chain with the generic sequence `[A,B,M,C]`, where `M` is the merge action. During the execution, action `A` creates a new branch, adding the action `A1`. At this point, AJIRA creates two instances of action `B`, one for processing the tuples produced by `A`, and one for processing the tuples produced by `A1`. The merge action forces the two execution flows to merge together into a single flow, which represents the input of the next (single instance of) action `C`. In this case, `C` cannot start unless all the other chains have finished. To ensure this, the node that should execute `C` keeps a trace of all the chains that should be merged and starts the final chain only after all the previous one have finished.

4) Partitioning: Finally, there are cases in which we need to reshuffle tuples from one or more flows to a new set of flows, based on some partitioning criterion. AJIRA provides this functionality through another special action, *partition*, which partitions the data flow of a chain into a given number of parallel output flows. This action accepts two important parameters: the number of flows (i.e., chains) to be created, and a user-defined hash function that maps tuples to flows. Note that the new chains will be uniformly distributed across all physical nodes AJIRA is deployed on. Such an operation is completely hidden from the user since the framework automatically handles all data transfers and possible synchronization issues in this process, irrespectively of whether chains run on the same or different physical nodes.

Fig. 2b demonstrates the use of the action *partition*. On top, we show a root chain as defined by the user. It includes three actions, `A`, `P`, and `C`, where `P` represents the special partitioning action. Now, let us assume that action `A` performed three branches for processing the input, that is, there are four chains executing in parallel. This situation is shown in the lower part of Fig. 2b. Note that these chains may be executing on the same or different physical nodes. If a user wants to reshuffle the tuples produced by `A`, `A1`, `A2`, and `A3` to *two* new chains, he only needs to write a hash function that decides which tuple is mapped to which chain (or use the default function that partitions through hashcode). As shown in Fig. 2b, this causes the instantiation of exactly *two* `C` actions, and a redistribution of the tuples among them.

III. AJIRA SYSTEM ARCHITECTURE

We will now discuss some details of the AJIRA implementation that are responsible for a large part of the performance described in Section IV.

A. Requirements

In Section I we identified three main requirements for AJIRA: expressivity, scalability, and efficiency. The system model described in the previous section fulfills the need for expressivity. During the development of AJIRA, we focused on the requirements for scalability and efficiency.

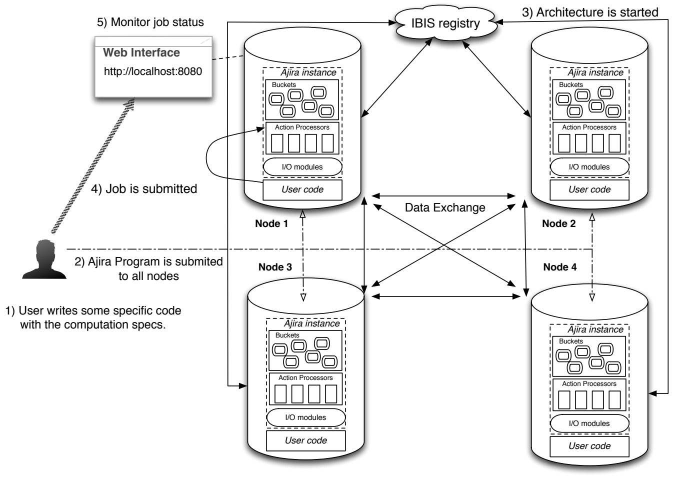


Fig. 3: Cluster view while executing an AJIRA program.

More in particular, we needed an architecture that could work in an Infrastructure-as-a-Service (IaaS) scenario, which consists of traditional computing clusters, where a number of machines are reserved for a limited amount of time and then returned to the cluster when the reservation expires. In this scenario, frameworks like Hadoop are not optimal since they require an elaborate deployment process where the data often needs first to be copied on the distributed filesystem.

Moreover, the high startup cost of each Hadoop job was unacceptable for most of our typical use cases, in which we had to perform very complex computations with hundreds of jobs having to be executed sequentially. We also noticed that a very large part of our computation did not require the advanced reliability features of Hadoop, and this was in line with the current usage of these frameworks in production environments. In fact, the MapReduce programs which run in production clusters of Microsoft and Yahoo! have a median input size of 14 GB while 90% of MapReduce programs at Facebook have an input below 100 GB [2], [38], [18]. Furthermore, more than 80% of the MapReduce jobs at Yahoo! terminate within less than 10 minutes and are very unlikely to fail [12]. For these types of jobs fault tolerance is often not a crucial feature since the cost of restarting is not high.

Finally, we often needed to perform continuous computations that produce new results as soon as new input becomes available. This is the typical purpose of event and stream processing systems, which analyze flows of data on-the-fly to detect patterns of situations of interest with low delay. This poses even more strict requirements in terms of efficiency.

Because of these reasons, we needed a framework with very little runtime overhead, which emphasizes performance rather than fault-tolerance and reliability. This motivates us to depart from the traditional Hadoop infrastructure to develop a radically new middleware.

B. Overview

We will now describe some salient features of the AJIRA architecture, which play a key role in shaping the performance and contribute to differentiate our approach from existing ones.

Fig. 3 presents the view of a cluster during the execution of an AJIRA program, indicating the main phases of the computation. First, the user writes a small program (we call it "user code"), which defines the AJIRA program that needs to be executed. The first operation of this program is to invoke a special AJIRA *init* function, which bootstraps the system. Within this function the nodes connect to a central registry, implemented using the Ibis IPL library [35], which implements a node discovery service. This registry is only used during system bootstrap: after that, nodes communicate directly with each other in a P2P fashion. When the bootstrap phase terminates, the user code submits its job to the local instance of AJIRA instantiated by the *init* function.

Next, the user selects a number of nodes to be used for the computation, and launches the user code on each node. When the init function has terminated, the user code submits the job to the local instance of AJIRA (this operation is performed only on one node). If requested by the user, AJIRA can start a lightweight HTTP server to provide a web interface to monitor the status of the program. When the program terminates, the architecture shuts down and returns the output to the user.

It should be emphasized that AJIRA does not require a cluster that is already up and running. Instead it starts the framework on-the-fly and shuts it down when the computation is finished. The AJIRA infrastructure is managed by the user during the execution of his program, without any communication with external programs (except for the Ibis registry). This makes the AJIRA system much more portable than cluster-based systems, since it can be started in different environments without additional configuration.

C. Implementation details

INTER-NODE COMMUNICATION. Being a distributed architecture, AJIRA does not only take care of performing the computation, but also handles node communication to exchange data and computation. For communication, AJIRA uses the Ibis message-passing library [35]. This library relies on the Ibis registry to manage the nodes in the network. As already mentioned, the registry is not involved in the computation, but is only used during the startup phase, when the *init* function is called on every node to start AJIRA.

The *init* function performs several other actions to bootstrap AJIRA: (*i*) it starts all the internal threads creating internal thread pools; (*ii*) it instantiates a number of point-to-point connections between the nodes; (*iii*) it initializes the input modules used by the program. Overall, these operations have a very little overhead and the startup of the architecture usually only takes a few hundred milliseconds.

THREADS MANAGEMENT. AJIRA does not create multiple processes during the computation. Each node runs all in a single (multi-threaded) application. This removes the need of (slow) interprocess communication, which is used in other frameworks like Hadoop. Moreover, AJIRA makes use of a thread pool in each node to reduce the cost of threads creation.

The chains are executed in a number of concurrent *worker* threads. Each worker takes one chain from a shared queue, which means that the chains are executed with a FIFO schedule. Then, it retrieves the input tuples from the input module, and starts feeding them to the actions in the chain. The execution of a single chain is a sequential process, carried out in a single thread.

BUCKETS. If the chain contains the actions *merge* or *partition*, the intermediate tuples may need to be transferred to other physical nodes. This operation is performed transparently by the framework using point-to-point connections and some special buffers called *buckets*. Buckets are an important data structure in AJIRA since their purpose is to temporarily store the tuples before being processed. During the transfer of data, buckets act as accumulator points since a node may produce tuples at a higher rate than they are being transferred.

In general buckets store their tuples in main memory, but can offload their content to disk when they become too large. Buckets can also sort the tuples if requested. In this case, the architecture performs external sorting using an additional pool of threads to speed up this operation.

Buckets are key components in making AJIRA suitable for both batch and stream processing. Indeed, in the case of batch processing, AJIRA collects data and transfers it only when enough tuples have been accumulated (potentially using compression, in case of network-bounded computations). Conversely, in case of stream processing, AJIRA tries to deliver tuples from node to node as soon as possible, focusing on minimizing the processing delay.

When a node has finished sending data to a bucket, it sends one last EOF message which contains the chain ID, and additional information about the children that the chain has generated. In some cases, one chain cannot start to read from a bucket until all the data has been sent. In this case, the bucket can use this information to determine when all chains writing data to it are finished.

It is worth to note that a bucket is an internal data structure which cannot be accessed directly by the user. The only possibility to interact with a bucket is through the actions *ReadFromBucket* and *WriteToBucket* which can be used to read and write the tuples in the chain to temporary buckets. This is a particularly useful feature if we need to perform some iterative computation on the same data, or if we want to temporarily store some data in the same node.

SERIALIZATION. Internally, AJIRA uses object serialization extensively, limiting the creation of new objects (which can cause significant performance problems) as much as possible. All the principal components of the ajira programming model, e.g., tuples, actions and chains, implement methods for efficient serialization to streams of bytes.

FAULT TOLERANCE. Given our requirements described above, we did not implement methods for fault tolerance and recovery that could slow down the performance. Instead, we implemented mechanisms to terminate gracefully in case of an exception. In AJIRA, if an exception is raised during the computation, the framework does not try to re-execute, but rather catches the exception and gracefully exits.

All the features described above have contributed to the development of a framework that aims at minimizing the overhead. Efficient communication, buckets, object reuse, data serialization, threads pools, and limited fault tolerance enable AJIRA to satisfy the requirements identified at the beginning of this section. In our experiments AJIRA starts up quickly, and closing a session is similarly fast. Section IV provides experimental evidence of the benefits of AJIRA in a number of typical tasks for both batch and stream processing.

IV. EVALUATION

The main goal of our evaluation is to measure the performance of AJIRA for both batch and stream processing. To this end, we chose a number of representative tasks and used them to compare our middleware with state of the art technologies from the two domains. First, we focus on batch processing tasks, and compare AJIRA with Hadoop and Spark [39]. Next, we analyze the efficiency of AJIRA for stream computations by comparing it against Esper [13] and Storm [33].

AJIRA is written in Java, and is available online[1]. Unless differently specified, the following experiments have been executed on the DAS-4 cluster[2] using up to 32 machines. Each machine is equipped with two quad-core Intel E5620 CPUs, 24 GB of main memory, and two hard disks of 1TB with RAID-0. The machines are connected with an 1 Gb Ethernet connection, and share a 20 TB disk using NFS.

A. Batch Processing

We consider three typical large data processing applications: *sorting*, *data compression*, and *data clustering*. We implemented these three applications on both AJIRA and Hadoop. In particular, we used Hadoop 1.0.3, configured with four mappers and four reducers per node and setting the task's maximum heap size to 1.5 GB.

We first consider *sorting* and *compression*. Sorting splits a data collection in several partitions and sorts each of them. We chose this task because Hadoop is highly optimized for it, since the intermediate grouping phase before the reduce is implemented with a local sort on each partition. Compression solves a real problem we addressed in the past: compressing a large collection of Web RDF data [31]. For this task, we re-implemented the MapReduce-based algorithm [34], which requires a sequence of three MapReduce jobs, in AJIRA. We chose this task because it is a real-world example of MapReduce that requires multiple jobs. For both applications, the input consists of RDF files generated using the well-known LUBM [15] benchmark. Each RDF file consists of multiple lines, each containing a list of URLs. We considered several inputs, ranging from 100 MB to more than 500 GB of compressed text (uncompressed, the biggest input is about 19 TB). For AJIRA, we stored the input on the 20 TB shared disk, while we copied it to the HDFS filesystem for Hadoop.

SORTING. In our first experiment, we launched the local sorting algorithm for different input sizes on eight nodes considering 32 partitions. Fig. 4a compares the performance of AJIRA and Hadoop: we observe that AJIRA outperforms Hadoop in all cases. Interestingly, the difference between AJIRA and Hadoop is much more significant for small inputs (less than 2 GB), where AJIRA is from 2.5 to 14.3 times faster. AJIRA remains 60% faster even when considering the largest dataset of 500 GB (19 TB uncompressed), which by far exceeds the median size of the inputs processed in production environments [2]. These results confirm the benefits of our design and implementation choices.

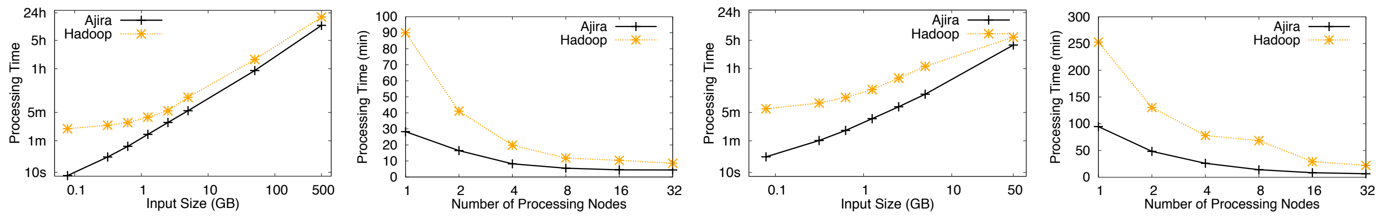
In our second experiment, we launched the sorting algorithm on a fixed input of about 6 GB compressed and varied the number of processing nodes from 1 to 32. Fig. 4b shows the execution times. From 1 to 8 nodes, the processing time decreases significantly for both AJIRA and Hadoop. When adding more nodes, the execution time starts to flatten, and the performance does not improve as much. In this experiment too, AJIRA outperforms Hadoop in all cases. Although the processing time of Hadoop drops faster when increasing the number of nodes, AJIRA is still twice as fast with 32 nodes.

COMPRESSION. We have repeated the previous two experiments on the same inputs, this time considering the compression algorithm. For this task, the Hadoop-based version requires three MapReduce jobs, while AJIRA can encode it in a single program. Therefore, we can evaluate whether the increased expressiveness of AJIRA is beneficial in terms of performance. Fig. 4c and Fig. 4d show the results. If we change the input size, the trend is similar to that observed in the previous application: AJIRA is up to 15 times faster when considering the smallest input (less then 100 MB) and remains more than 1.5 times faster when considering the largest input of 50 GB[3]. When we change the number of nodes, the advantage of AJIRA increases when moving from one to eight nodes (from 2.66 times to 4.85 times faster). Between 16 and 32 nodes the performance of Hadoop increases significantly, however AJIRA remains about 3.4 times faster in both cases.

K-MEANS. In the previous experiments, AJIRA significantly outperforms Hadoop. We investigated the reason behind this difference and concluded that the HDFS filesystem and the advanced reliability features of Hadoop might play an important role. Therefore, we decided to compare our performance against the Spark framework [39], which does not use a distributed filesystem. We chose Spark because: (*i*) it is a mature project recently being accepted in the Apache incubator initiative; (*ii*) it does not require a distributed filesystem; (*iii*) whenever possible it keeps data into main memory,

[1]http://jrbn.github.io/ajira
[2]http://www.cs.vu.nl/das4

[3]We did not run the 500 GB input for compression; it would simply have taken too long. Extrapolating the lines in Fig. 4c, however, may lead to the conclusion that Hadoop might be faster in this case.



(a) Sorting: Input size (b) Sorting: Number of nodes (c) Compression: Input size (d) Compression: Number of nodes

Fig. 4: Comparison of the performance of AJIRA against Hadoop.

	Small	**Medium**	**Large**
AJIRA	1.7 s	63.5 s	3505.4 s
Hadoop	633.0 s	1176.7 s	6162.6 s
Spark	6.0 s	213.8 s	4645.3 s

TABLE I: K Means: comparison with Hadoop and Spark

instead of offloading everything to disk; (*iv*) it is much more expressive than Hadoop since it allows iterative computation.

Initially we tried to implement a local sort using Spark. However, after some attempts and a request in the Spark user mailing list, we were unable to encode it in an efficient way that could be used for a fair comparison. Therefore, we decided to implement a different application which had an efficient implementation in Spark and compare the performance against AJIRA and Hadoop. We chose *K-Means clustering* [24], which is a popular data mining algorithm that clusters n vectors into k clusters. An interesting property of this algorithm is that it requires a loop: therefore, we could use it to test how efficient AJIRA is for iterative computations.

We generated three datasets: a *small* one, which consists of 10K vectors of four elements and ten cluster centers; a *medium* one, with 16M vectors and 100 cluster centers; and a *large* one, with 160M vectors and 10K centers. The small dataset takes about 244 KB of space, the medium 361 MB, while the large one takes about 3.6 GB. We ran this application on eight nodes, and set the default parallelism of Spark to 64, which provides the best results.

Table I presents the results. We notice that in the best case AJIRA is 370 times faster than Hadoop. However, if we compare with Spark, the difference is much smaller since AJIRA is a bit more than three times faster, and this difference is most likely due to some fault-tolerance mechanisms of Spark which introduce additional overhead. If we increase the input size and the complexity of the task, we notice that the performance of the three systems becomes more similar, but the advantage of AJIRA is still visible.

DISCUSSION. From the results presented above we can derive some conclusions. First of all, we notice that AJIRA is faster than Hadoop in all the scenarios we tested, especially on small inputs (up to 15 times faster in local sorting and compression and up to 370 times faster in k-means). Even though AJIRA still outperforms Hadoop with the largest input (which consists of 19 TB of uncompressed data for sort), the difference becomes smaller as the input size increases. We speculate that for even larger input sizes, Hadoop will eventually be faster than AJIRA. This trend is to be expected since the distributed filesystem becomes beneficial when the input is sufficiently large because reads are performed only on local disk (through HDFS). With AJIRA (and Spark) the input is being read by a single disk accessed through NFS, and this will become a performance bottleneck with larger inputs. Furthermore, as the size of the input or number of nodes increases, the chance of a failure increases. In this case, the fault-tolerance features of Hadoop might become crucial.

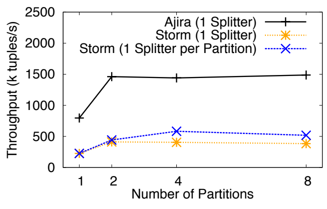
The comparison with Spark is interesting because both engines are designed for efficient computation even for small inputs. While Hadoop might indeed be better in scaling *up*, the other two approaches are much more efficient in scaling *down*, which is an equally important feature considering the current usage of MapReduce frameworks. In this context, the lightweight architecture allows AJIRA to also outperform Spark with a significant margin in our example scenario.

B. *Stream Processing*

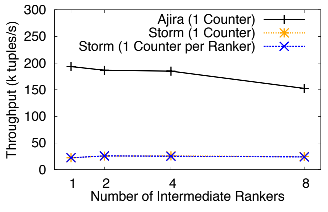
To evaluate the performance of AJIRA in stream processing, we compare it against two heterogeneous systems. The first is Storm [33], a distributed real-time platform designed to obtain high throughput on continuous computations. The second is Esper [13], a Complex Event Processing (CEP) engine that is known for its low latency processing. Both frameworks are mature and widely adopted: Storm is used as one of the main components in the computation infrastructure of Twitter, while Esper is used inside several commercial systems, e.g., Oracle CEP [23].

In this section, we consider three applications: *streaming wordcount*, *trending topics*, and a *CEP operator benchmarking*. The first two applications come from the official Storm example library. *Streaming wordcount* takes a stream of sentences, computes how many times each word has been received, and outputs the stream of updated counts. *Trending topics* takes a stream of words and computes the k most used words in the last n minutes. *CEP operator benchmarking* comes from the CEP domain, where high level queries are used to process input events: we implement the main building blocks used in such queries and test their performance.

STREAMING WORDCOUNT. In this application the processing is performed in two steps: first a number of splitters extract single words from sentences; then a number of counters store the count of each word in memory and update it as they receive new input. In this evaluation we introduce a third and final step to collect all the results so that we can measure

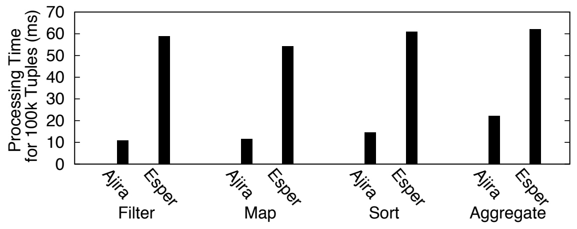


(a) Streaming Wordcount

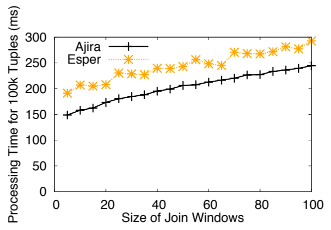


(b) Trending Topics

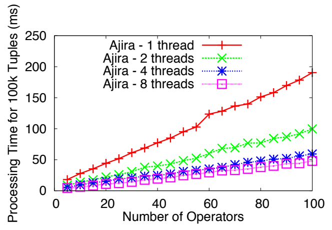
Fig. 5: Analysis of throughput and comparison with Storm.



(a) Single-stream operators



(b) Join operator



(c) Scalability: number of operators

Fig. 6: Analysis of processing time and comparison with Esper.

the overall throughput. The input consists of a set of random sentences which are generated by the system. Each sentence consists of ten words selected from a dictionary of 1000 words. To measure the maximum throughput of AJIRA and Storm, sentences are generated by a single process at maximum speed (without sleeping times). In all our experiments, we monitored the processing components to make sure that neither the generator of sentences nor the final measurement module introduces any performance bottleneck.

First we launched both AJIRA and Storm 0.8.2 on eight machines using one splitter and changing the number of word partitions (i.e., the number of counters). The results are presented in Fig. 5a, and show that AJIRA always outperforms Storm. Then, we tried to tune the parameters of Storm to verify whether it was a problem of cluster misconfiguration. This tought us that Storm has a higher throughput if the number of splitters is equal to the number of partitions. Because of this, we repeated the experiment with Storm configured in this way. However, even in this case AJIRA significantly outperforms Storm, being between 3.52X faster (with a single partition) and 2.87X faster (with 8 partitions).

TRENDING TOPICS. In the trending topics application, the computation is performed in three steps: first, a number of counters count the occurrences of each word in a given time window; then some intermediate rankers compute the ranking of words within partitions; finally a total ranker combines the rankings from each partition to provide a global order. To stress the systems under analysis, we recompute the ranking (of the top ten topics) every time a new word is received. Fig. 5b presents the results. Here too, AJIRA outperforms Storm in all cases with up to 8.6 times speedup. Interestingly, neither AJIRA nor Storm can increase their throughput when considering a higher number of partitions since the final total ranking operator is the main bottleneck.

CEP OPERATORS. Low latency processing is an important requirement for streaming applications. Consider for example a system for high frequency trading: reducing the delay for analyzing input may represent a significant advantage over competitors. Here, we study the processing latency of AJIRA when performing streaming computations using the main building blocks defined in every CEP engine. For these experiments we used Esper 4.9.0 and a single machine equipped with a Core i7-3615QM and 8 GB of RAM. In all our experiments, we first generated and stored 100K tuples in main memory, each of them consisting of four integer and four string attributes. Then, we sequentially submitted the tuples to the program using a special input layer, and measured the total time required to process all of them.

Fig. 6 shows the results we observed. In particular, it shows the processing time required for executing a single operator on the input tuples. We considered four operators:
Filter: selects the tuples based on the value of one attribute; in our settings, it discards half of the input tuples;
Map: transforms each input tuple by removing one of the attributes from its content;
Sort: performs an ordering of the tuples in a sliding window s of size 100. This means that it first collects 100 tuples and orders them according to the value of an attribute a; after receiving a new tuple t, it removes and outputs the first tuple in s (i.e., the one having the lowest value of a), and stores t;
Aggregate: computes and outputs the average value of an attribute over the last 100 received tuples.

From the figure, we notice that both AJIRA and Esper process the input with a very small delay, always below 70ms. However, AJIRA outperforms Esper in all the tests, with processing times from 2.5 to 5.5 times lower. AJIRA implemented all the operations in a single-threaded action; in contrast, Esper uses multiple threads and their synchronization introduces overhead. The higher performance of AJIRA is confirmed in Fig. 6b, where we measure the processing time for performing a windowed join over two streams. We repeated this experiment while increasing the size of the window. The impact on the runtime is less then linear for both AJIRA and Esper, but the advantage of AJIRA remains almost constant.

Finally, Fig. 6c measures the scalability of AJIRA when increasing the number of processing steps performed over the input (i.e., the number of actions that get executed). Unfortunately we could not replicate this experiment in Esper, since it adopts a high level language for the definition of the processing tasks that does not allow to control the number of operators each tuple moves through. In this experiment, we adopted only filter operators, changing their number from five to 100. We set all the filters in such a way that each operator receives some tuples in input. Since filtering can be applied independently on different tuples, we also investigated the benefits of parallelizing the computation, splitting the input over multiple processing threads. As Fig. 6c shows, the processing time of AJIRA increases linearly with the number of operators involved in the processing, meaning that no additional overhead is added. Moreover, the performance of the system scales linearly with the number of threads when moving from one to four; the benefits vanish with eight threads, since the machine does not have enough cores.

From these results we conclude that the overhead of AJIRA is very low, so that it not only performs batch processing with competitive performance to specialized systems, such as Hadoop, but also competes with and even outperforms systems explicitly designed for real-time processing.

V. Related Work

Batch and stream processing have been widely investigated in the past. To position our contribution, we discuss the current technologies and the related work proposed so far. Unfortunately, space constraints force us to focus only on the most relevant technologies, and redirect the reader to existing surveys, such as [19] for MapReduce and [7] for stream processing, for a complete overview.

Batch Processing. MapReduce [9] and Dryad [17] are among the first proposals to process very large amounts of data. While MapReduce organizes the computation in two phases, Dryad was more generic since it defined the computation as generic acyclic graphs (DAG). Both models did not support iterative or recursive computations, which are supported by AJIRA.

Many recent works addressed the limitations of MapReduce. Among them, some focus on adding iterations to Hadoop [5], [11]. AJIRA does not extend Hadoop but rather implements a completely new architecture. Other works focus on the efficiency of performing small MapReduce jobs [12], [30]. These works only target the functionalities of MapReduce, while AJIRA focuses on a more generic programming model.

Some works focus on offering programming models in which the execution graph can be specified at run-time, either partially, e.g. Pregel [22], or completely, like Ciel[26]. Pregel was specifically designed for graph computation, and this makes it difficult to compare with AJIRA, which was designed for more generic computations. Ciel is perhaps the closest to AJIRA in terms of expressivity, since it proposes a primitive called *spawn* that resembles the functionality of our *branch*. However, to the best of our knowledge Ceil does not support stream processing, does not allow the redirection of newly created flows in the existing ones, nor is it possible to imitate the functionality of our *split* primitive.

We found Spark [39], and its applications on streams [40], to be the closest framework to ours in terms of performance. Spark shares with AJIRA the goal of achieving high efficiency for generic computation but it is difficult to compare the two regarding expressiveness since Spark focuses on the definition of a high-level language while AJIRA offers actions and primitives that work at a lower level. Another distrbuted framework similar to ours is Naiad [25], which offers both high throughput for batch computation and low latency for stream processing. Conceptually, their approach differs significantly from ours since it proposes a computation representation with a directed cyclic graph called *timely dataflow*, while AJIRA uses multiple chains of operators.

Like Spark, other approaches also focus on the definition of higher-level languages, which are then translated into a programming model like MapReduce. The most popular are perhaps Pig Latin [28], which is a declarative language developed by Yahoo! to ease the processing of data with MapReduce, and DryadLINQ [37], which was designed to simplify the development with Dryad. Comparing AJIRA with such languages is problematic since they work at different levels: while Pig and DryadLINQ aim at hiding completely the complexity of the underlying programming paradigm, in AJIRA the goal is to expose the complexity with clear and easy-to-use APIs, hiding only the physical execution.

Stream Processing. Several specialized stream processing engines are capable of performing complex computations on streams of data. Despite their differences, these systems can be roughly organized into two main classes [7]: *Complex Event Processing (CEP) systems* [21], which aim at extracting higher level knowledge starting from (patterns of) low level observations, and *Data Stream Management Systems* (DSMSs) [3], [4] which perform continuous computations to process and transform the input data on-the-fly. Some modern commercial products (e.g., Esper [13], Oracle Event Processing [23]) embed both processing abstractions into a single solution.

Other systems, including Storm [33], HStreaming [16], MapReduce online [6], Nephele [20], and MARISSA [10], focus on providing MapReduce-like computation models to real-time processing. They have a lower expressivity than AJIRA since they do not support dynamic workflows. Furthermore, while these systems deliver high throughput, we provide empirical evidence of the higher efficiency of AJIRA against Storm, which is among the current state of the art.

The idea of organizing the computation into a workflow of basic operators has been widely exploited in stream processing, both at a conceptual level (e.g., see the Event Processing Network model [14]), and at a technical level, to allow parallel and distributed processing (e.g., work on operator placement [29] and deployment [8]). Several systems implement this model on clusters or large scale distributed scenarios [27], [1]. Nevertheless, these systems remain mainly non-distributed applications, which cannot scale to large amounts of data.

From this analysis, we see that both batch processing and streaming systems excel in some tasks but have limitations in others. AJIRA aims at providing a single programming model and architecture that is competitive in both domains.

VI. Conclusions

Modern organizations increasingly need processing models and infrastructures that combine flexibility, scalability, and efficiency. In this paper we introduced AJIRA, a new middleware that aims at satisfying these three requirements.

AJIRA proposes a new generic processing model where multiple actions can be piped in concurrent and distributed chains of operations. Each action can dynamically create new workflows at runtime, which might contribute with new data, and such expressivity enables the efficient implementation of complex programs. This processing model is supported by an efficient framework that minimizes the time to deploy new processing tasks, providing an ideal environment for small and large computations, without sacrificing scalability.

These characteristics enable AJIRA to support a variety of heterogeneous tasks such as data analytics, compression, clustering, and logic programming. Furthermore, the flexibility of AJIRA makes it suitable for stream processing and continuous computations over dynamic data, which introduce strict requirements in terms of efficiency and processing latency.

Our evaluation demonstrates the benefits of AJIRA in a wide range of scenarios when compared to state of the art research and commercial technologies for batch and stream processing, such as Hadoop, Spark, Storm, and Esper.

We believe that AJIRA distinguishes itself from the current approaches for its expressivity and efficiency, and could represent an ideal unifying technological layer for all those contexts that demand an efficient execution of complex processing tasks of both large scale and dynamic data.

ACKNOWLEDGMENTS. This project was partially funded by the COMMIT project, and by the NWO VENI project 639.021.335.

REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2), 2003.

[2] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-Up vs Scale-Out for Hadoop: Time to Rethink. In *Procs. ACM Symposium on Cloud Computing*, 2013.

[3] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: The Stanford Stream Data Manager. In *Procs. SIGMOD*, 2003.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Procs. PODS*, 2002.

[5] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient Iterative Data Processing on Large Clusters. *Proc. VLDB Endow.*, 3(1-2), 2010.

[6] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *Procs. NSDI*, 2010.

[7] G. Cugola and A. Margara. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surv.*, 44(3), 2012.

[8] G. Cugola and A. Margara. Deployment Strategies for Distributed Complex Event Processing. *Computing*, 95(2), 2013.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1), 2008.

[10] E. Dede, Z. Fadika, J. Hartog, M. Govindaraju, L. Ramakrishnan, D. Gunter, and R. Canon. Marissa: MapReduce Implementation for Streaming Science Applications. In *Procs. Int. Conf. on E-Science (e-Science)*, 2012.

[11] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.H. Bae, J. Qiu, and G. Fox. Twister: a Runtime for Iterative MapReduce. In *Procs. HPDC*, 2010.

[12] K. Elmeleegy. Piranha: Optimizing Short Jobs in Hadoop. *Procs. VLDB Endow.*, 6(11), 2013.

[13] 2013. http://esper.codehaus.org.

[14] O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publications Co., 2010.

[15] Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3, 2005.

[16] 2013. http://www.hstreaming.com.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Procs. EuroSys*, 2007.

[18] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Procs. SOSP*, 2009.

[19] K.H. Lee, Y.J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel Data Processing with MapReduce: a Survey. *SIGMOD Record*, 40(4), 2011.

[20] B. Lohrmann, D. Warneke, and O. Kao. Massively-Parallel Stream Processing Under QoS Constraints with Nephele. In *Procs. HPDC*, 2012.

[21] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[22] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a System for Large-Scale Graph Processing. In *Procs. SIGMOD*, 2010.

[23] S. McReynolds. Complex Event Processing in The Real World. 2007.

[24] T. M. Mitchell. *Machine Learning. 1997*, volume 45. 1997.

[25] D.G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, M. Abadi. Naiad: A Timely Dataflow System. In *Procs. SOSP*, 2013.

[26] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a Universal Execution Engine for Distributed Data-Flow Computing. In *Procs. NSDI*, 2011.

[27] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *Procs. ICDMW*, 2010.

[28] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a Not-So-Foreign Language for Data Processing. In *Procs. SIGMOD*, 2008.

[29] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *Procs. ICDE*, 2006.

[30] R. Power and J. Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *OSDI*, 2010.

[31] W3C Recommendation: RDF Primer, 2013.

[32] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and Parallel DBMSs: Friends or Foes? *Commun. ACM*, 53(1), 2010.

[33] 2013. http://storm-project.net.

[34] J. Urbani, J. Maassen, N. Drost, F. Seinstra, and H. Bal. Scalable RDF Data Compression with MapReduce. *Concurrency and Computation: Practice and Experience*, 2012.

[35] R. V. Van Nieuwpoort, J. Maassen, G. Wrzesińska, R. F. Hofman, C. J. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a Flexible and Efficient Java-Based Grid Programming Environment. *Concurrency and Computation: Practice and Experience*, 2005.

[36] H.C. Yang, A. Dasdan, R.L. Hsiao, and D. S. Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *Procs. SIGMOD*, 2007.

[37] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLinq: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI*, 2008.

[38] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: a Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Procs. European Conf. on Computer Systems*, 2010.

[39] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Procs. USENIX Conf. on Hot Topics in Cloud Computing*, 2010.

[40] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, I. Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Procs. SOSP*, 2013.