CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2016; **28**:2487–2502
Published online 28 January 2016 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3779

SPECIAL ISSUE PAPER

Watershed-ng: an extensible distributed stream processing framework

Rodrigo Rocha*,†, Bruno Hott, Vinícius Dias, Renato Ferreira, Wagner Meira and Dorgival Guedes

Department of Computer Science, Universidade Federal de Minas Gerais, Belo Horizonte, MG, Brazil

SUMMARY

Most high-performance data processing (a.k.a. big data) systems allow users to express their computation using abstractions (like MapReduce), which simplify the extraction of parallelism from applications. Most frameworks, however, do not allow users to specify how communication must take place: That element is deeply embedded into the run-time system abstractions, making changes hard to implement. In this work, we describe Wathershed-ng, our re-engineering of the Watershed system, a framework based on the filter–stream paradigm and originally focused on continuous stream processing. Like other big-data environments, Watershed provided object-oriented abstractions to express computation (filters), but the implementation of streams was a run-time system element. By isolating stream functionality into appropriate classes, combination of communication patterns and reuse of common message handling functions (like compression and blocking) become possible. The new architecture even allows the design of new communication patterns, for example, allowing users to choose MPI, TCP, or shared memory implementations of communication channels as their problem demands. Applications designed for the new interface showed reductions in code size on the order of 50% and above in some cases. The performance results also showed significant improvements, because some implementation bottlenecks were removed in the re-engineering process.
Copyright © 2016 John Wiley & Sons, Ltd.

Received 2 March 2015; Revised 10 November 2015; Accepted 24 December 2015

KEY WORDS: distributed systems; watershed; big data; frameworks

1. INTRODUCTION

The explosion of data available to researchers today has led to the growth of the area of high-performance data processing (a.k.a. big data). To obtain the best results in this area, tools have to be available to experts in the data domain. However, they seldom are also experienced in parallel programming, creating the need to develop frameworks that can explore the parallelism intrinsic to applications with a simple API. Google's MapReduce [1] (and its open-source implementation, Hadoop [2]) is certainly one of the the most well-known examples of such frameworks, as well as Spark [3], a newer contender in this arena.

Most frameworks, however, do not allow users to specify how communication between computer nodes must take place: That element is deeply embedded into the run-time system (RTS), making changes hard to implement.

Watershed [4] is another framework developed with similar objectives. It is a distributed stream processing system for massive data streams, derived from Anthill [5], a previous framework

*Correspondence to: Rodrigo Rocha, Department of Computer Science, Universidade Federal de Minas Gerais Belo Horizonte, MG, Brazil.
†E-mail: rcor@dcc.ufmg.br

Copyright © 2016 John Wiley & Sons, Ltd.

developed for batch processing. Both were inspired in the *data flow* model. Stream processing systems comprise a collection of modules that compute in parallel and that communicate via *data streams* [6]. That is, filters obtain data from channels, transform them, and send them to output channels. In its original form, as presented by Ramos *et al*. [4], Watershed presented streams as black boxes, elements that could not be altered by the user. Not only that, but its orientation towards continuous stream processing made it hard to use when developing more traditional, batch/file-oriented applications, something its predecessors allowed.

To change that, we performed the re-engineering of the Watershed framework, to make the stream abstraction extensible by the programmer. By isolating stream functionality into appropriate classes, combination of communication patterns and reuse of common message handling functions (like compression and blocking) became possible.

Our goal with this redesign was that users could extend the framework with other communication elements, better fitted to their algorithms. For example, users should be able to add transformations to a data stream in an elegant fashion, without requiring neither new processing filters to be written nor changes to be made to their application code. One common case is that of message aggregation: In many applications, to reduce overhead due to the transmission of many small messages, it is often good for performance to block a large number of data items into a single message to be sent over the network. In the original version of Watershed, each developer had to add their version of such blocking code (and its unblocking counterpart) to their application code. In the re-engineered version, Watershed-ng (*Next Generation*[‡]), it should be possible for programmers to reuse a common block/unblock element that would be added directly to the stream, decoupled from the application code.

The new architecture should also make it easy to integrate Watershed-ng with the Hadoop ecosystem, because a new stream class could be easily created to integrate with the HDFS, the Hadoop File System [8], both for reading and writing data in parallel. This integration would also make the re-engineering process easier, because much of the application deployment code was simplified by using Hadoop's YARN [9] and Zookeeper [10] modules.

In its new form, the architecture will also allow the design of new communication patterns, for example, allowing users to choose MPI, TCP, or shared memory implementations of communication channels as their problem demand. That would help also in making Watershed-ng simpler to use in the development of batch/file-oriented applications, as streams could be extended to better serve that form of processing.

Our contributions are the discussion of the new abstractions added to the framework, which simplified the task of the programmers, and of the description of our experience integrating our framework to the Hadoop ecosystem. Our results show that applications designed using the new API showed reductions in code size on the order of 50% and above in some cases. Not only that, but the re-engineering process allowed us to remove some implementation bottlenecks, resulting in significant performance improvements compared with the original version and bringing Watershed-ng performance to levels similar to those of Hadoop and Spark. Although our goal was not to improve performance compared with those systems, our results show that Watershed-ng performs at the same level as Hadoop for simple applications and even outperforms Spark for the k-nearest neighbors (kNNs) application in our tests.

The remainder of this work describes our design decisions during this re-engineering process. Section 2 describes the new framework and discusses the impact of the new abstractions available for programmers. The architecture of the re-engineered RTS and its integration to the Hadoop ecosystem are detailed in Section 4, and performance results are shown in Section 5. Finally, Section 6 describes related work, and Section 7 provides some conclusions and discusses ways in which the work can be extended.

[‡]The name refers to the once-famous TV series *Star Trek: The Next Generation*, commonly known among its fans as STTNG. The first use of *ng* in this way in Computer Science can probably be attributed to IPng, the Next Generation IP, first name of what came to be IPv6 [7].

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

2. WATERSHED-NG

In the original Watershed, an application was defined as a set of user-defined filters, connected by pre-defined streams. Data flowed through streams and get processed in filters, potentially producing new data elements. Parallelism was implicitly declared, because all filters can execute concurrently, as long as they have data to process, and each filter might have multiple instances executing in parallel. Data from a stream were distributed among instances according to that stream's policy.

In Watershed-ng, those principles remain, but one of the goals of the re-engineering process was to turn streams into first-class abstractions, which could also be customized by the user. That customization could be achieved by programming explicit actions or by combining different existing elements.

The interface implemented by all objects in Watershed-ng offers an initialization and termination methods.

- The *start* method is the prolog processing of the filter's instance execution. It is intended to perform a setup operation at the beginning of the instance execution.
- The *finish* method is the epilog processing of the filter's instance. It is intended to perform a cleanup operation at the end of the instance execution.

Those two methods are complemented by methods specific to each element in the framework, discussed next.

2.1. Filters

In Watershed, and Watershed-ng, a processing module is composed by a filter, a set of input ports, and a set of output ports, as illustrated in Figure 1. We can attach a stream channel to each input port and attach one or more stream channels to each output port. Streams are unidirectional: When a module is specified to consume the data produced by another module, a stream channel dynamically binds them via their input and output ports, respectively.

Each filter may have multiple executing instances, in a number that is currently defined by the user. Filter instances receive data from input ports in an event-based fashion. After a filter has processed the data, it may send the transformed data through its output ports, triggering new delivery events at the filters connected to the other end of that specific stream.

A filter is the producer for another filter when the former produces an output stream of data that is an input stream of the latter. Similarly, a filter is the consumer from another filter when the former consumes an input stream of data that is produced as an output by the latter.

The methods that must be implemented by each filter are as follows:

- The *process* method is an event handler that is triggered by one of the input stream channels, whenever there are new data to be processed by that particular instance of the filter. The filter must explicitly produce its output data through the output stream channels. Considering the concurrent aspect, mutable shared variables must be manipulated in a thread-safe manner.
- The *onInputHalt* is the event method that is concurrently triggered when one of the input streams has halted, propagating the end-of-stream signal.

2.2. Streams

Processing filters are connected via data stream channels. A stream basically consists of a sender, a deliverer, a stack of encoders, and a stack of decoders, as depicted in Figure 2. We can notice from

Figure 1. Abstraction of a processing module.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

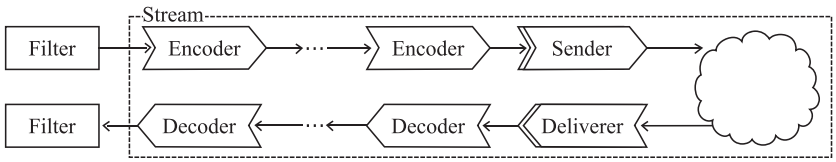

Figure 2. The decomposition of a single stream into the actual objects connecting two filters.

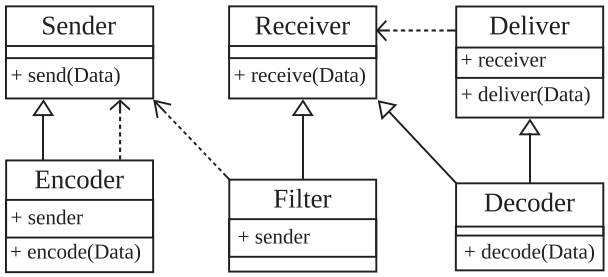

Figure 3. Simplified class diagram of the module components.

the figure that there is no stream object by itself: A stream is actually a combination of different elements, which together implement the communication channel.

Senders and deliverers implement the actual data communication, possibly crossing different execution domains. They also agree on the transmission medium and the communication protocol. Examples of implementation choices might be communication through files in a distributed file system; a TCP connection; MPI messages; a shared memory buffer, if the origin and destination are different processes in the same machine; or even an direct method activation, if the objects exist within a single execution domain.

The sender must define a *send* method, which is called when there are data to be sent through the stream. The communication protocol is implemented inside that object. On the other hand, the deliverer is responsible for extracting data from the specific transmission medium (no method from the framework is activated at this level; the object must implement its own receiving mechanism). When new data are received, the deliverer must activate an event in the next object of the transmission chain (a decoder or a filter) to have that data transferred and processed.

Encoders and decoders operate on the data flow between filters and senders/deliverers. They can be used as pairwise data transformations during the data transmission through a stream, for example, for cryptography, data compression, and data aggregation and disaggregation. In this case, the stack of encoders and the stack of decoders must agree in the sequence of execution, in such a way that the decoder of the last encoder will be the first to be executed, and vice-versa. That is enforced by the configuration language, which requires that paired transformations be added to a stream in a single operation.

We can also use either an encoder or a decoder as a single, one-way transformation, for example, a selector that only forwards data elements that match a given pattern, to encode data stream records as JavaScript Object Notation (JSON) encoded strings or to change a temperature stream from Fahrenheit to Celsius.

Figure 3 illustrates the relations between the different objects that may be involved in the data flow through a stream. It shows that many of the elements share common interfaces, given their roles in the communication: An encoder has the same interface as a sender, because it may or may not be in the way between a filter and the later; similarly, filters and decoders share the receiver's interface.

The main difference between the *encode* method and the *process* method of the filter API is that the encoder belongs to one of the output stream channels of a filter; therefore, there is only one output destination, which is the next sender in the output sender stack. Nevertheless, the class hierarchy is organized in such a way that filters can, in principle, also be used as encoders/decoders.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

That is useful, for example, to implement the functionality of MapReduce *combiners*, as shown in the examples, later on.

Each of the objects that can be combined in a stream, besides providing a method to process the flowing data, must also process events triggered by the closing of a stream, or the end of the data flow. The *onProducerHalt* is the event triggered when all the producers of the deliverer's stream channel have finished their execution.

The Watershed-ng default library of streams contains basic implementations that are common in most of the data processing applications [11], such as over the network broadcast, round-robin, and labeled stream; local file line and writer; and HDFS file line and writer. The library also contains some encoders and decoders, such as data compressor, data decompressor, itemizers, and a regular expression-based filter.

2.3. Termination mechanism

In its original design, Watershed considered all streams as continuous and had no explicit termination semantics. Anthill, on the other hand, had a clear termination algorithm, which was based solely on the state of communication channels [5]. In the process of re-engineering Watershed, we decided to use a more explicit termination criteria, based on Pregel's halting mechanism [12]: The termination of a module is dependent on every module instance *voting to halt*. A filter has an event called *onInputHalt*, which is triggered when an input stream ends, and another event called *onAllInputsHalt*, which is automatically triggered when all the input streams terminate. Both filters and stream deliverers have a function called *halt*. When the stream deliverer calls the *halt* function, it triggers the *onInputHalt* event of the filter/receiver after it. When the filter/receiver calls the *halt* function, it sends a signal to the Watershed master specifying that that particular instance of the module has voted to halt. By the time all instances of a given module have halted, the Watershed master removes the module from the execution environment.

The mechanism for propagating the termination signal of a filter through a stream channel is the responsibility of the communication protocol used to implement that particular stream, where the deliverer associated with it must trigger the *onInputHalt* event. Consumers can then decide to halt when all of its data producers finish, based on the propagation of the end-of-stream signal.

Based on the halting mechanism, we can define stream channels as either finite or continuous. Finite streams have well-defined and pre-determined duration, such as a stream that reads data from a file. Continuous streams have undetermined duration, depending on an outside *actor* to finish its execution or on the end of the data stream, such as a stream that reads data from a web feed, a stream that writes data into a file or a stream that reads data from the network.

With this halting mechanism, it is possible to implement batch applications on top of Watershed, as illustrated in Section 5. In a sense, the implementation of a Watershed batch application becomes equivalent to an Anthill application [5], which was one of our original goals for this work.

2.4. Loading a module

When the user wants to load a new module, M_1, he or she sends the module descriptor to the Watershed master. The master schedules the slave nodes that will execute each instance of the module. Afterwards, the master sends the instance descriptors to the scheduled slave nodes. Finally, it verifies the bindings among the modules. The consumer is the one responsible for describing the stream channel components it needs, as represented by Figure 2, that will create the binding with the producer. Therefore, different consumers are able to use different stream policies to read from the same filter output port.

For each filter M_0 that is a producer for M_1, the Watershed master sends a stack of encoders and a sender for each instance of M_0, which will be properly attached to the output port, as specified by M_1. In the same fashion, for each filter M_2 that is a consumer of a data stream produced by M_1, the master provides a stack of encoders and a sender for each instance of M_1, which will be properly attached to its output port, as specified by the input stream of M_2.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

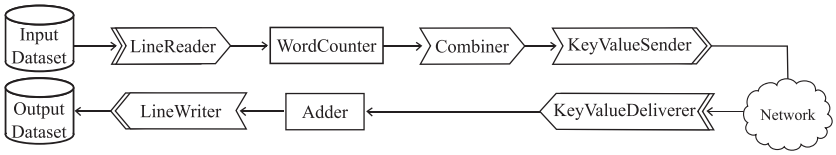

Figure 4. Word frequency filter–stream application design.

```
package sample.wordcount;

import java.util.AbstractMap.SimpleEntry;
import hws.core.Filter;

public class WordCounter extends Filter{
   public void process(String src, Object obj){
      String data = (String)obj;
      String []words = data.split("\\W+");
      for(String word : words){
         Object pair = new SimpleEntry<String,Integer>(word, 1);
         outputChannel().send(pair);
      }
   }
}
```

Figure 5. Watershed source code for the WordCounter filter of the word frequency application.

3. WATERSHED PROGRAMMING EXAMPLE

To illustrate how the new Watershed can be used to implement a simple application, we show how to compute word frequency in a large text dataset.[§]

Word frequency is a batch processing application for massive datasets that is very suitable for the MapReduce abstraction [1]. Figure 4 presents an implementation in Watershed for the word frequency application that is conceptually equivalent to that described for the MapReduce abstraction. The application consists of two filters: the WordCounter, which is equivalent to a Mapper, and the Adder, which is equivalent to a Reducer. The communication between the WordCounter and the Adder filters uses a labeled stream, which implements the shuffle pattern of communication [11]: Key–value pairs are delivered based on the result of a mapping function (usually hash) to the key.

The LineReader implements an input-only stream that reads the dataset line by line from a file and feeds them to instances of the WordCounter filter (Watershed currently allows files in a local file system and HDFS). The LineWriter implements the symmetrical operation, writing the output of the Adder filter instances to the file system.

Adder and WordCounter are the filters responsible for most of the computation required for the application. WordCounter receives lines from a file input stream and breaks them in words. For each word in a line received, it then produces a pair containing $< word, 1 >$. The Adder filter receives pairs in the form $< word, frequency >$ and adds all the counts for each given word to finally produce globally unique pairs $< word, frequency >$ with the global frequency of each word. WordCounter and Adder are shown in Figures 5 and 6, respectively.

KeyValueSender and KeyValueDeliverer belong to the default library of network streams already available in Watershed. Together, they implement the labeled stream communication pattern (a map operation). That guarantees that all pairs produced for a given $word$ will all be mapped to the same instance of the filter in its output, independent from where they were produced or how many instances of each filter exist.

The combiner element in this example was so named because it has the same role as a combiner in MapReduce [2]: It locally aggregates the counting frequency for each word produced by the WordCounter filter, forwarding only locally unique pairs $< word, frequency >$. In Watershed, we can think of a combiner as one of many possible transformations applied to the data stream, so it

[§]That is intentionally the same canonical example used by Hadoop authors, so we can compare systems later.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

```
package sample.wordcount;

import java.util.Map;
import java.util.Map.Entry;
import java.util.concurrent.ConcurrentHashMap;

import hws.core.Filter;

public class Adder extends Filter{
   private Map<String, Integer> counts;

   public void start(){
      super.start();
      counts = new ConcurrentHashMap<String, Integer>();
   }

   public void finish(){
      for(Entry<String,Integer> entry: counts.entrySet())
         outputChannel().send(entry.toString());
      super.finish();
   }

   public void process(String src, Object obj){
     Entry<String, Integer> data = (Entry<String, Integer>)obj;
     String word = data.getKey();
     Integer val = data.getValue();
     if(counts.containsKey(word))
        counts.put(word, val+counts.get(word));
     else counts.put(word, val);
   }
}
```

Figure 6. Watershed source code for the Adder filter of the word frequency application. The combiner encoder, when used, shares the same code.

```
<?xml version="1.0"?>
<filter name="adder" file="sample.jar" class="sample.wordcount.Adder" instances="2">
    <input>
        <channel name="net">
            <sender class="hws.channel.net.KeyValueSender" />
            <deliver class="hws.channel.net.KeyValueDeliver" />
            <encoders>
                <encoder class="sample.wordcount.Combiner" file="sample.jar" />
            </encoders>
        </channel>
    </input>
    <output>
        <channel name="writer">
            <sender class="hws.channel.lfs.LineWriter" >
                <attr name="path" value="/watershed/wordcount" />
            </sender>
        </channel>
    </output>
</filter>
```

Figure 7. Configuration file for the Adder filter of the word frequency application using the Watershed framework.

is actually the implementation of an encoder in the stream abstraction. The code of the combiner is basically the same of the Adder filter (Figure 6) and was removed for brevity.

In Figure 5, we see that the WordCounter has only one method of interest in this case: `process`. It is activated by Watershed when a new data element arrives to be processed. The parameters then identify the input stream (`src`) and the data object received (`obj`). We can also observe how a filter can handle multiple output streams: They are available through a method `outputChannel()`. Sending a message over a stream is then just a matter of activating a method `send` on the desired stream.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Filter Adder (Figure 6) is somewhat more complex. In order to output a single pair for each $word$, it must keep a local dictionary to store partial counts. To do that, it defines `start` and `finish` methods to build such data structure and to output the final counts to the output files. As previously mentioned, the combiner has a very similar structure in this case.

Each filter must be separately configured to identify its connections and define the number of instances and other information. Figure 7 shows the configuration file for the Adder filter when the local file system is used. First, the `filter` itself is identified in terms of the jar file that contains it and the class name. Then, the input and output ports are listed. For those ports, the `name` attributes define the names of the streams that become visible to the filter code (like in the `outputChannels()` method.) For the adder, the input stream is a network-level labeled stream, given the classes chosen for the sender and the deliverer (`.KeyValue`). The input stream also has an encoder associated, the combiner, whose class and jar file are also identified. The output port is a stream directed to a file in the local file system of the compute node where it is executed (`lfs.LineWriter`, with the path of the file also provided).

4. WATERSHED-NG ARCHITECTURE

In its original implementation, Watershed was a monolithic, stand-alone system. All elements of execution, such as resource allocation, scheduling, and fault tolerance and I/O interfaces were controlled internally. That resulted in a very complex RTS and limited the range of techniques available to those that could be implemented by the local team.

In its original version, Watershed used static scheduling: The user assigned filter instances to specific machines in the application configuration file, which had to be adapted each time the number of machines available changed, for example. Access control and job execution were implemented using MPI commands, and MPI was also used as the communication substrate. Furthermore, there was no allocation control, and different users running applications in the cluster at the same time could easily encumber each other's progress by blindly choosing the same node to execute some filter instances.

Recently, many frameworks, like Spark [3] and Storm [13], have started leveraging functionalities from the Apache Hadoop ecosystem [14] to simplify their implementation. Besides the MapReduce programming model, the current Hadoop version (2.x) has been organized in a modular fashion, making isolated functionalities available to be repackaged and integrated with other solutions.

To avoid the problems found with the original version of Watershed, in the new implementation, we decided to use Yet Another Resource Negotiator (YARN) [9] as the scheduler/resource allocator. As Hadoop's resource scheduler and negotiator, YARN offers the service of a basic platform for the management of distributed applications in a higher level of abstraction. By doing so, it even allows different frameworks to coexist in a single installation.

Besides using YARN as the process scheduler and the resource manager, Watershed-ng uses Zookeeper [10] to implement the coordination service for the multiple processes in each application and HDFS [8], both as the distributed file system for applications and as the means of distribution for configuration and executable files. Besides those preexisting services, the Watershed-ng framework consists of three main components: (i) the *JobClient*, responsible for submitting the application to be executed; (ii) the *JobMaster*, which controls the execution of the processes that compose the application; and (iii) the *InstanceDriver(s)*, which execute and monitor each filter instance. The complete architecture is illustrated in Figure 8.

4.1. Starting an application

In this section, we describe the sequence of steps performed when starting the execution of an application in Watershed-ng, including the interaction with modules of the Hadoop ecosystem. That sequence is shown in Figure 9; in the discussion that follows, numbers in parenthesis refer to the interactions in that figure.

When a user wants to start a Watershed-ng application, she or he must start a JobClient with the configuration files that describe which filters must be run and how they are connected (an example

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

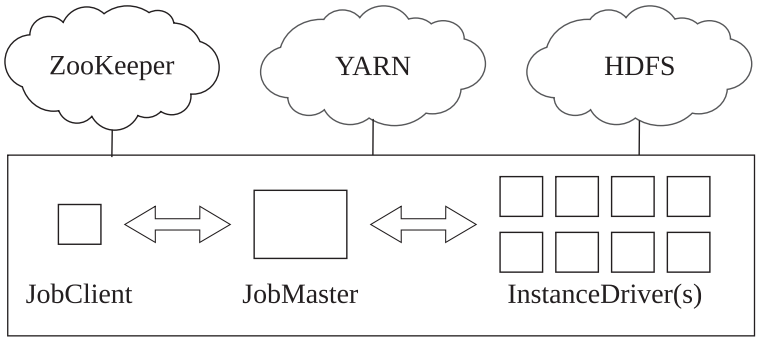

Figure 8. A high-level overview of the Watershed-ng architecture implemented in the Hadoop ecosystem. YARN, Yet Another Resource Negotiator; HDFS, Hadoop File System.

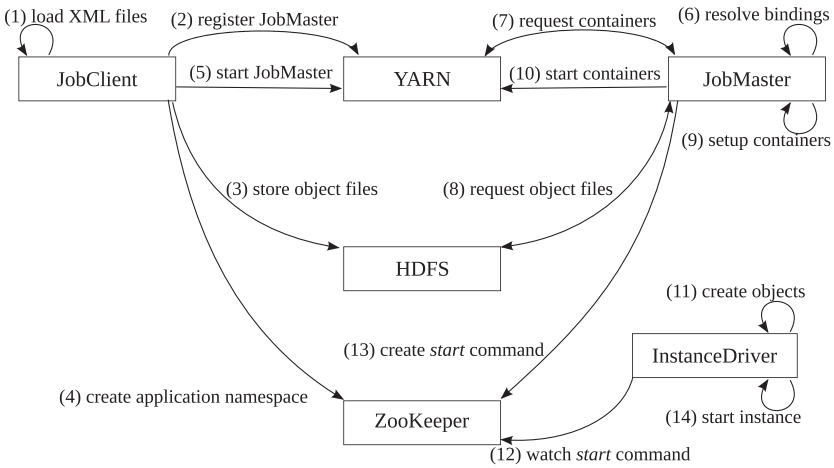

Figure 9. Relations between the multiple modules of Watershed-ng as an application is configured and distributed and begins execution. The multiple steps illustrated are detailed in this section.

of such file is shown in Figure 7). JobClient is the user interface to a Watershed-ng application job. It parses the configuration files (1), identifying the filters and streams defined by the user and their interconnections. After that, (2) it contacts YARN to register the code that will execute the JobMaster element, responsible for scheduling Watershed-ng programs.

The JobClient also has some responsibilities concerning internal management of the job, for example, creating the initial ZooKeeper sub-tree structure for the namespace of that particular application job (4), which will be necessary for the distributed coordination of the job, and creating an HDFS sub-tree for the namespace of that particular application job (3), which is mainly used for sharing specific libraries and for logging.

Once those configurations are in place, the JobClient then contacts YARN to issue a request to start a JobMaster process (5). After that, it waits for the Manager response.

Yet Another Resource Negotiator allocates a resource container and starts the JobMaster, passing to it the objects identified by the client (5). The JobMaster basically receives the module descriptors, parses them (6), requests containers from YARN to run each module instance (7), and sets up the environment with the shared libraries of that particular application job (8). While requesting the containers, depending on the scheduling strategy of the Application Master, it offers a set of the containers' requirements to the YARN Resource Manager, for example, specifying main memory capacity and number of CPU virtual cores and also computing node addresses for the containers.

For each container, the appropriate configurations are made (6), and a separate InstanceDriver is executed (10), receiving as parameters the description of the actual filter instance that must be started.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

The InstanceDriver receives a module-instance descriptor, instantiates the objects that compose the module instance (11), registers its state with ZooKeeper (12), and waits for a signal from the JobMaster to start the application (13). When it receives the start signal, it finally manages the execution of the instance (14). The InstanceDriver also has some responsibilities concerning coordination management, such as creating a signal of the instance termination using ZooKeeper. It is also responsible for generating the *onProducersHalt* event, which is also the result of a ZooKeeper signal.

The JobMaster observes the progress of that application's drivers. When it sees that all filters are ready to proceed, it triggers a signal to all drivers in ZooKeeper. When all filter instances terminate, it returns the control to the JobClient, which then cleans up the application environment.

4.2. Streams implementation

The first Watershed version had hardwired stream implementations that used MPI and offered the basic policies inherited from Anthill: broadcast, round-robin, and labeled (mapped). There was no predefined API to interface to a file system: Programmers were responsible for opening, reading, and writing files directly, what was not always simple, considering the distributed nature of the computation.[¶] The new version, Watershed-ng, allows users to define their own stream modes and how they are implemented. To show the flexibility and to provide a minimum set of functionalities, we implemented the original Anthill policies using TCP sockets, a pair or streams to read/write files in the local file system of a compute node, and also a pair of HDFS streams to read and write directly from HDFS.

The local read/write streams are easily implemented with in-memory queues that interface to simple file reading/writing threads. The name of the files to use is provided by the configuration files and verified by the InstanceDriver when it builds the stream. The configuration file in Figure 7 shows the Adder filter configured to write to a local directory.

For HDFS, we have created a writer and a reader stream for the cases when the output of a filter must be stored to a file or when a filter must read from a file, respectively. Right now, we only support HDFS line-oriented files. The *LineReader* stream partitions the input dataset evenly among the instances of the filter that requested to read, considering both the size in bytes of each partition and that lines are unique atomic entities. That is, no line is divided between two partitions or sent to more than one consumer. The *LineWriter* stream is a simple output interface with HDFS, where each data produced is stored as a line in an HDFS file.

5. EXPERIMENTAL EVALUATION

In this section, we evaluate Watershed-ng in terms of performance and size of code necessary to implement a given algorithm. For the applications described next, we compared their performance with other available implementations of the same algorithms and compared the code developed in each case based on lines of code (LOCs) to have a notion of code complexity in each case. The systems used for comparison were Anthill, the original Watershed, Hadoop, and Spark.

5.1. Applications

In this analysis, we considered two batch Watershed-ng applications: word frequency and kNNs.

5.1.1. Word frequency. The word frequency problem and the new Watershed-ng implementation were discussed in Section 3. It is particularly suitable to be processed using the MapReduce model, so it is a good application to compare Watershed-ng with Hadoop. For the performance comparisons, we considered only versions using a combiner-like element to reduce the number of messages in transit, because that is known to be the more efficient implementation. In Anthill, that functionality

[¶]That was actually one of the reasons why it allowed users to define instance placement exactly in terms of machines: to guarantee that a filter that needed to read a file stored in a certain node's local file system would be executed in the right compute node.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

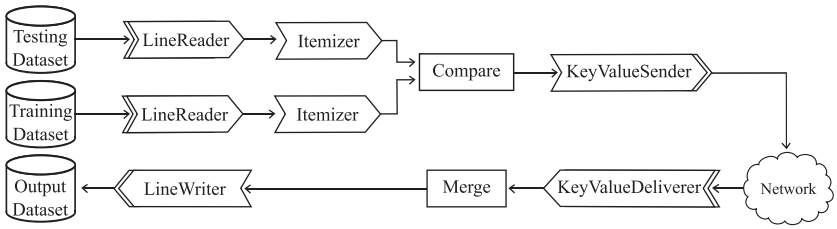

Figure 10. k-Nearest neighbors filter–stream application design.

had to be implemented directly inside a filter; for Watershed-ng and Hadoop, a combiner was provided; for Spark, the operators in the language implicitly implement the effect of a combiner (that is the default).

5.1.2. k-Nearest neighbors classifier. As a second experimental batch application, we have implemented a common data mining classification algorithm, the kNNs classifier. The kNNs classifier predicts the class of a given input as the majority class among its kNNs computed from the training dataset [15].

Figure 10 illustrates the design of the kNNs algorithm using the Watershed-ng programming mode. We use the same LineReader stream channels already available in the framework. The input stream passes through an itemizer, which is implemented by a decoder, so that each input datum is forwarded as a key–value pair, with the input data as the value and an incremental identifier as the key. The identifier is important for the merge phase, when the partial results of the top kNNs are combined for each *key*, comprising the classification based on the global kNNs.

Each instance of the compare filter receives the full testing dataset and only a disjoint portion of the training dataset. Afterwards, each instance produces its top kNNs relative to its portion of the training dataset. In the compare–merge communication, the network-based labeled stream was used to guarantee candidates were grouped properly. The merge filter computes the global top kNNs and outputs the testing dataset with the inferred classification.

The Spark implementation of kNNs uses the concept of resilient distributed datasets (RDDs) intrinsic to its programming model. The training dataset is read into an RDD, which partitions it into several chunks, distributed among the computing nodes. Afterwards, each test input point is replicated to each partition with training data points. Within each partition, kNNs are distributedly computed for each test input point. Finally, the various neighbors are merged and processed, computing the overall kNNs for each input point, producing the final classification based on the majority class.

For the following analysis, we did not consider the Hadoop implementation of kNNs we had. Because kNNs' execution pattern does not fit the MapReduce programming model well, the programming model constrained the development, and it resulted in a slower, complex (and long) code when compared with all others.

5.2. Code complexity

We used the metric of LOCs as an indicator of the complexity programming with each of the frameworks. Although the LOC metric has limitations, our point is that for sample sizes as the ones considered here, when every sample was produced by programmers experienced in the platform, a significant difference in the number of LOCs to implement the same algorithm can serve as a first indicator of the relative power of expression of the frameworks in question.

In this analysis, we considered only the code versions with the combiner feature, because they showed better performance results. Besides the code developed for Watershed-ng, we considered the code for implementations previously developed for Anthill and for the original Watershed. We only considered the Hadoop implementation for the word frequency application, because it fits well with the MapReduce programming model. For Spark, we considered algorithm implementations in

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Table I. Lines of code in the implementation of each application in the various frameworks.

	Application	
Framework	Word count	kNNs
Anthill	210	332
Original Watershed	127	400
Watershed-ng	65	145
Hadoop	64	NA
Spark*	18	83

*Spark uses a different programming language paradigm, because it is implemented in a functional programming language.
kNNs, k-nearest neighbors.

Scala, a functional language, which produced the lowest numbers (Spark can be used with Scala, Python, and Java).

Table I summarizes the results. The code written for the Watershed-ng was smaller than the code for its antecessors (Anthill and the original Watershed). The reductions achieved with the new framework come from the fact that file access is now embedded in the stream definition, and the new stream structure makes it easy to add transformations to the data (such as the combiner) using encoders. Because the word count application fits well in the Hadoop programming model, the programs are of similar size in that case. Spark requires considerably less LOCs when compared with the other frameworks. That is mostly due to the use of the Scala versions, because Scala, a functional language, tends to be highly expressive, while the other frameworks are implemented in imperative programming languages. However, imperative languages are still the most dominant programming language paradigm in overall usage. In particular, the Java version of the kNNs Spark code was much larger than the other versions considered.

5.3. Performance evaluation

For performance evaluation, experiments were conducted in a cluster with six computer nodes, where one of them was configured as master and the others behaved as slave nodes. Each node had an Intel Xeon processor® with four CPU cores, a clock of 2.5 GHz, and 8 GB of RAM. We performed a total of five executions for each experiment, and the results shown are the average of those results.

Our intent with these tests is to show that the new architecture does not hinder performance. Because our major goal was to make the programming task simpler for the application developer, we do not expect Watershed-ng to outperform an up-to-date parallel processing system, like Hadoop or Spark, but to have results close to it. We do not show results for the original Watershed. As previously mentioned, that implementation was geared towards continuous, persistent stream processing and used a storage-backed implementation of streams, which made its performance unacceptable for batch applications — that was one of the main reasons for the re-engineering process, after all.

5.3.1. Word frequency. The input dataset was a 1-GB portion of the English Wikipedia articles dump[‖] adding up to a total of about 61.5 million lines of text. Figure 11 shows the results as we varied the size of the sample that was fed to the application.

Anthill had the best performance for small datasets, but it did not scale well for larger sizes. The initial gains were due to the fact that it uses local files and a very simple start-up procedure, so it has lower overhead to star than the other systems, all of which use HDFS and YARN. Although word counting fits well to the MapReduce programming model, Watershed-ng outperforms it, slightly. Spark has clearly the best performance results overall. That is mostly due to it having a much lighter

[‖]Obtained from http://dumps.wikipedia.org/enwiki on January 12, 2015.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

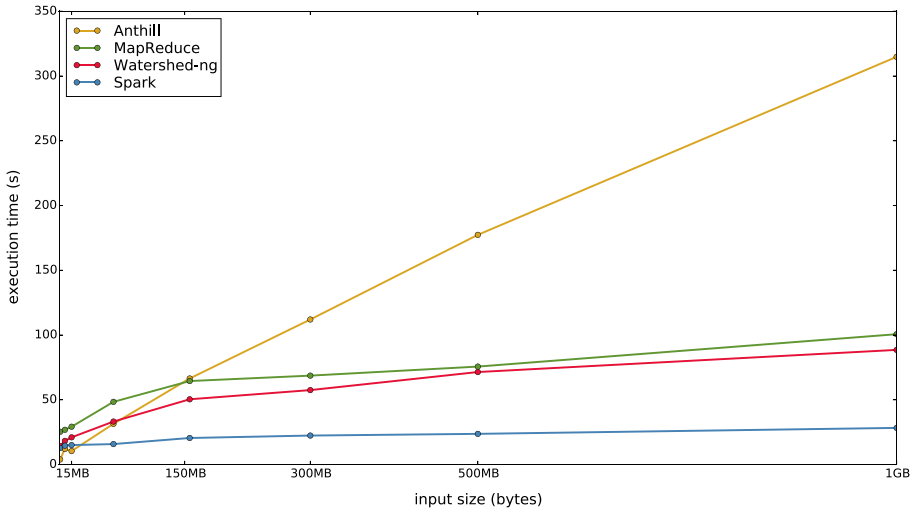

Figure 11. Word frequency with local aggregation (combiner): execution time for different input sizes, for Anthill, Hadoop (MapReduce), Watershed-ng, and Spark.

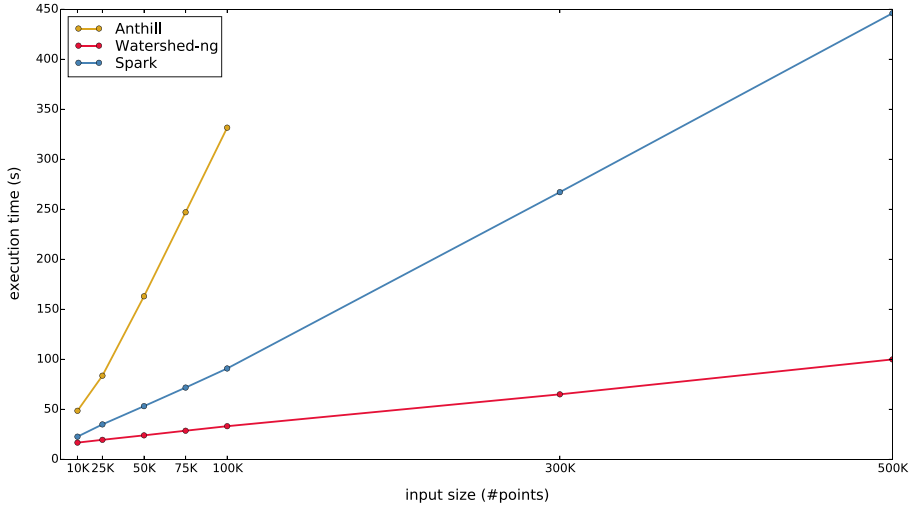

Figure 12. k-Nearest neighbors classifier: execution time for different input sizes, for Anthill, Spark, and Watershed-ng using the local file system or Hadoop File System (Anthill can only use the local file system).

RTS and because the functional operators used in the implementation allow for better optimizations to be applied.

These results show that Watershed-ng has a reasonable performance, even in problem with a very easy parallelization model. It slightly outperforms Hadoop, while being simpler to use. In this problem, it is not able to match Spark's performance, however.

5.3.2. k-Nearest neighbors classifier. For kNNs, we used a synthetic dataset that generated points in a two-dimensional space using a set of centroids around which points were distributed according to a normal distribution. The experiments were run with a value of k equal to 50, a total of 20,000 training samples, and different amounts of samples to be classified. Figure 12 shows the performance results in this case.

In this case, Anthill times raise much faster, and sets with more than 75,000 samples took too long to complete. We studied the performance data in detail and found that it was because

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

the kNNs implementation used small messages (one for each two-dimensional data point). That incurred in a lot of overhead in that system's communication substrate, which was not designed for small messages.

In this more complex application, Watershed-ng scales better than Spark and is about five times faster for 500,000 points. That seems to be due to the fact that Spark operates in a more synchronized way, and the kNNs implementation creates wide dependencies in the Spark's RDD lineage. In Watershed-ng, the pipeline computation is performed in a more asynchronous manner, with the filters compare and merge being computed in parallel.

6. RELATED WORKS

Anthill [5], the predecessor to Watershed, is a distributed processing system that implements the filter–stream abstraction. This model uses a data flow approach to decompose the applications into a set of processing units referred to as filters. Data are transferred through the filters using streams, which allow fixed-size untyped data buffers to be transferred between filters. Anthill has an RTS that is able to execute stream processing applications with static topology. It was suitable for several parallel data mining applications [5, 16], exploiting maximum parallelism of the applications by using task parallelism, data parallelism, and asynchrony.

Watershed [4], on the other hand, was developed with stream processing in mind, similar to systems like MillWheel [17], S4 [18], and Storm [13]. In it, like in those systems, applications may be never ending, processing data that arrive continuously through streams. The focus in Watershed was on allowing filters to be attached to any data source in the system dynamically, using type templates to describe the kind of data they needed. Streams should exist independently of filters and should be stored for later access to past data. The dependence on data templates and the need to persist all data imposed heavy performance penalties to applications that did not require such functionalities. By returning streams to Anthill semantics, Watershed-ng is able to guarantee high performance for batch-oriented applications; On the other hand, the new organization of the object model allows the developer to redefine the database-oriented semantics from the original Watershed by creating a new stream class that offers that functionality, and that can be used only when it is necessary.

The modular architecture from Watershed-ng was also heavily inspired by the new version of Apache Hadoop [14]. In Hadoop 2, the programming model (MapReduce) was more clearly isolated in a specific module, and other functionalities, like resource management, the distributed file system, and a stable storage module for fault tolerance, were all implemented separately. Besides being organized around more orthogonal modules, Watershed-ng even makes direct use of some of Hadoop's modules in its implementation, integrating the two systems to a point that they can be used side by side in a same infrastructure.

Besides Hadoop, another framework that has gained popularity recently is the Spark [3] cluster computing framework, which lets users seamlessly intermix streaming, batch, and interactive queries. It is built around the concept of RDDs, abstractions that implicitly define how communication takes place. Watershed streams have similar goals as RDDs to hide the details of distributed communication from the programmer. However, by exhibiting streams as first-class objects, it also allows the experienced programmer to define specific communication patterns.

In that sense, the emphasis on making the communication element (streams) more flexible that drove this re-engineering effort can be related to Coflow [11], a networking abstraction to express the communication requirements of prevalent data parallel programming paradigms. Coflow makes it easier for the applications to convey their communication semantics to the network, which in turn enables the network to better optimize common communication patterns. Similarly to the concept of communication pattern offered by Coflow, in the Watershed model, the users are able to specify the communication pattern precisely, and they are also allowed to develop their own communication channels. We are considering to make further use of this fact in the future to integrate resource and network scheduling based on the idea of coflows, by extracting more information about traffic patterns from the streams themselves.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

This paper extends a previous workshop paper, which discussed the re-engineering of Watershed and described the new stream abstractions [19]. This current work extends that by including complete code examples of the use of the new functionalities (part of Section 3), describing details of the integration of Watershed with the Hadoop ecosystem (Section 4), and by providing more detailed performance results, including some data on code complexity (Section 5). All of those elements are being presented for the first time here.

7. CONCLUSION

Multiple frameworks have been proposed to ease the task of programming big-data applications. Most of them, however, allow developers to express only their processing needs — Communication is treated as a black box inside the framework, to which programmers have no access. In this work, we have described the process of re-engineering Watershed, our stream processing system, to make streams first-class objects in the framework. By doing that, programmers can now implement communication channels that better fit their needs, as well as to reuse code to add functionalities to an existing communication channel.

The re-engineering process allowed us to streamline the implementation, and a modular approach reduced code development by reusing modules already available in the Apache Hadoop ecosystem, like YARN and Zookeeper. The new stream abstractions made it possible to easily extend the original Watershed to include a simple interface to HDFS, simplifying the programmer's task of obtaining access to the large volumes of data to be processed.

Our results show a significant reduction in code complexity, based on LOCs, as well as an improvement in performance compared with the two systems that preceded it. When compared with Hadoop in a case where the application was well matched to that system's programming model, our system's performance was comparable. When compared with Spark in a more complex application, Watershed's asynchronous execution yielded better performance.

As future work, we intend to continue the development of new streams, and new encoders, such as a message aggregator to reduce the penalty of small messages in the network. We believe that by implementing more optimized communication protocols, we can improve Watershed performance even further. We are also considering to integrate into our framework a solution to explore network management and optimization, like Orchestra and Coflow, using the semantic patterns of network streams in Watershed.

ACKNOWLEDGEMENTS

This work was partially sponsored by Fapemig, Capes, CNPq, and the National Institute of Science and Technology of the Web, InWeb (MCT/CNPq 573871/2008-6).

REFERENCES

1. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM* 2008; **51**(1):107–113.
2. White T. *Hadoop: The Definitive Guide: The Definitive Guide*. "O'Reilly Media, Inc.": Sebastopol, CA, 2009.
3. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: cluster computing with working sets. *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, Berkeley, CA, USA, 2010; 10–10.
4. de Souza Ramos TLA, Oliveira RS, de Carvalho A, Ferreira RAC, Meira W. Watershed: a high performance distributed stream processing system. *2011 23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, IEEE, Viktória, ES, Brazil, 2011; 191–198.
5. Ferreira RA, Meira W, Guedes D, Drummond LMA, Coutinho B, Teodoro G, Tavares T, Araujo R, Ferreira GT. Anthill: a scalable run-time environment for data mining applications. *17th International Symposium on 2011 23rd International Symposium on Computer Architecture and High Performance Computing, 2005. SBAC-PAD 2005*, IEEE, Rio de Janeiro, RJ, Brazil, 2005; 159–166.
6. Stephens R. A survey of stream processing. *Acta Informatica* 1997; **34**(7):491–541.
7. Bradner S, Mankin A. The recommendation for the IP next generation protocol. RFC 1752 (Proposed Standard), January 1995. http://www.ietf.org/rfc/rfc1752.txt,.
8. Shvachko K, Kuang H, Radia S, Chansler R. The Hadoop distributed file system. *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, IEEE, Washington, DC, USA, 2010; 1–10.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

9. Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, *et al.* Apache Hadoop YARN: Yet Another Resource Negotiator. *Proceedings of the 4th Annual Symposium on Cloud Computing*, ACM, New York, NY, USA, 2013; 5:1–5:16.
10. Hunt P, Konar M, Junqueira FP, Reed B. Zookeeper: wait-free coordination for internet-scale systems. *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, Vol. 8, Berkeley, CA, USA, 2010; 11–11.
11. Chowdhury M, Stoica Ion. Coflow: a networking abstraction for cluster applications. *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, ACM, New York, NY, USA, 2012; 31–36.
12. Malewicz G, Austern MH, Bik AJC, Dehnert JC, Horn I, Leiser N, Czajkowski G. Pregel: a system for large-scale graph processing. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ACM, New York, NY, USA, 2010; 135–146.
13. Toshniwal A, Taneja S, Shukla A, Ramasamy K, Patel JM, Kulkarni S, Jackson J, Gade K, Fu M, Donham J, *et al.* Storm@ twitter. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ACM, New York, NY, USA, 2014; 147–156.
14. Murthy A, Vavilapalli VK, Eadline D, Markham J, Niemiec J. *Apache Hadoop YARN: Moving Beyond Mapreduce and Batch Processing with Apache Hadoop 2*. Pearson Education: New Jersey, 2013.
15. Zaki MJ, Wagner Meira Jr. *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press: New York, 2014.
16. Veloso A, Meira W Jr, Ferreira R, Neto DG, Parthasarathy S. Asynchronous and anticipatory filter-stream based parallel algorithm for frequent itemset mining. *Knowledge Discovery in Databases: PKDD 2004*, Springer, Berlin, 2004; 422–433.
17. Akidau T, Balikov A, Bekiroğlu K, Chernyak S, Haberman J, Lax R, McVeety S, Mills D, Nordstrom P, Whittle Sam. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment* 2013; **6**(11):1033–1044.
18. Neumeyer L, Robbins B, Nair A, Kesari A. S4: distributed stream computing platform. *2010 IEEE International Conference on Data Mining Workshops (ICDMW)*, IEEE, Washington, DC, USA, 2010; 170–177.
19. Rocha RC, Ferreira R, Jr. WM, Guedes D. Watershed reengineering: making streams programmable. *Proceedings of the 26th IEEE International Symposium on Computer Architecture And High Performance Computing Workshops, SBAC-PAD Workshop 2014*: SBC, Paris, 2014; 120–125.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe