Project Number 318763

D3.7 – Hadoop with Real-Time Enhancements

Version 1.0
1 December 2014
Final

Public Distribution

University of York

Project Partners: **aicas**, **HMI**, **petaFuel**, **SOFTEAM**, **Scuola Superiore Sant'Anna**, **The Open Group**, **University of Stuttgart**, **University of York**, **Brno University of Technology**

Project Partner Contact Information

aicas Fridtjof Siebert Haid-und-Neue Strasse 18 76131 Karlsruhe Germany Tel: +49 721 66396823 E-mail: siebert@aicas.com	**HMI** Markus Schneider Im Breitspiel 11 C 69126 Heidelberg Germany Tel: +49 6221 7260 0 E-mail: schneider@hmi-tec.com
petaFuel Ludwig Adam Muenchnerstrasse 4 85354 Freising Germany Tel: +49 8161 40 60 202 E-mail: ludwig.adam@petafuel.de	**SOFTEAM** Andrey Sadovykh Avenue Victor Hugo 21 75016 Paris France Tel: +33 1 3012 1857 E-mail: andrey.sadovykh@softeam.fr
Scuola Superiore Sant'Anna Mauro Marinoni via Moruzzi 1 56124 Pisa Italy Tel: +39 050 882039 E-mail: m.marinoni@sssup.it	**The Open Group** Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of Stuttgart Bastian Koller Nobelstrasse 19 70569 Stuttgart Germany Tel: +49 711 68565891 E-mail: koller@hlrs.de	**University of York** Neil Audsley Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325571 E-mail: neil.audsley@cs.york.ac.uk
Brno University of Technology Pavel Smrz Bozetechova 2 61266 Brno Czech Republic Tel: +420 54114 1282 E-mail: smrz@fit.vutbr.cz	

Contents

List of Figures

Document Control

Version	Status	Date
0.9	Complete First Draft	28 November 2014
1.0	QA for EC Delivery	1 December 2014

Executive Summary

This document constitutes deliverable *D3.7 – Hadoop with Real-Time Enhancements* of work package 3 of the JUNIPER project.

The purpose of this document is to demonstrate the wider applicability of the JUNIPER programming model to existing, commercial Big Data systems. The JUNIPER approach, as detailed in previous deliverables (D2.1 and D2.2), describes a unique programming model for large-scale data processing systems based around the benefits of real-time analysis and the use of model-driven development to improve development, deployment and testing.

The approach implements a message-passing communications model that is discussed in deliverables D2.1 and D2.2. This model is based on MPI, common in high performance computing, but less common in general purpose computing. This deliverable demonstrates how the JUNIPER approach is designed to have complementary APIs layered above MPI to implement the programming models of common Big Data frameworks, such as Hadoop, Spark, or Storm.

This deliverable shows both how this is achieved, and the benefits of doing so.

1 Introduction

This document constitutes deliverable *D3.7 – Hadoop with Real-Time Enhancements* of work package 3 of the JUNIPER project.

The purpose of this document is to demonstrate the wider applicability of the JUNIPER programming model to existing, commercial Big Data systems. The JUNIPER approach, as detailed in previous deliverables, describes a unique programming model for large-scale data processing systems based around the benefits of real-time analysis and the use of model-driven engineering to improve development, deployment and testing.

The programming model has the following main features:

- A message-passing communications model, the implementation of which is automatically built by the model-driven engineering code generation during deployment. This allows for a developer to write highly portable software without worrying about deployment targets, and allows post deployment testing and scaling to occur without altering software.
- Locality primitives which allow developers to portably describe the way in which the threads and data of their system should be mapped to a target cloud or HPC, with the aim of increasing both performance and predictability of the deployed system.
- Architecture patterns which expose pertinent details about the target hardware to allow software to maximise its exploitation of the architecture of the current server, but also to retain portability.
- FPGA-based acceleration to increase performance and predictability.

The message-passing communications model (discussed in deliverables D2.1 and D2.2) is based on MPI. This is common in high performance computing, but less common in general purpose computing.

In order to allow for wider appear and exploitation of the platform, this deliverable demonstrates how the JUNIPER approach is designed to have complementary APIs layered above it to implement the programming models of common Big Data frameworks, such as Hadoop, Spark and Storm.

Section 1.1 discusses how programming models can layer on top of the JUNIPER platform. Section 2 outlines the Java 8 Stream API. Section 3 details the requirements of a stream API suitable for real-time Big Data. Our proposed extensions to the existing API are in Section 4, and are implemented in section 5. Section 6 then defines the Hadoop model and how it can be implemented in JUNIPER. Section 7 discusses Spark and Storm. Section 8 evaluates these approaches and section 9 outlines the additional features in JUNIPER that can further improve application performance. Finally, section 10 concludes.

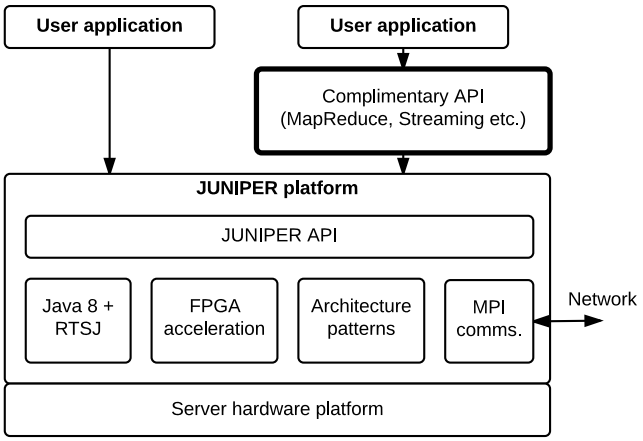

Figure 1: Layering programming models on top of the JUNIPER approach

1.1 Layering programming models

The JUNIPER approach is designed to be extensible to new Big Data techniques. Rather than being tied to a specific technology (such as Hadoop), the platform is designed to provide an efficient set of services which could be used by any approach. Java libraries can be provided that support different Big Data programming models, including new models that are developed in the future.

This is illustrated in figure 1. The user can use the JUNIPER approach directly (as shown on the left of the figure), or they can use a JUNIPER extension API to adopt a different programming model that they are perhaps more familiar with (as shown on the right).

In this document four extension APIs will be described:

- Extensions for compute nodes and groups,
- Drop-in replacement extensions for streams,
- Distribution of data extensions for streams, and
- Distributed Collections.

2 Java 8 Streams

This section provides a recap for the data streaming facilities provided by Java 8. From this we can describe the requirements of the JUNIPER extensions to this model in section 3, while section 4 uses these requirements to discuss the design and implementation of these extensions.

A *stream* is a sequence of data elements that can be processed by a *pipeline* of operations. Streams can be generated from several sources, including:

- Collections, by calling the `stream` method if the desired stream should be sequential, or the `parallelStream` method for a parallel stream;
- Arrays, by calling the `Arrays.stream` method;
- Factory methods in the `Stream` class;
- Files, by calling the `BufferedReader.lines` method.

After retrieving the stream from a source, a pipeline of aggregate operations (formerly called bulk data operations) can be performed on it, consisting of zero or more intermediate operations followed by a terminal operation. From the programmer's perspective, an intermediate operation returns a new stream of processed elements from the given stream, and a terminal operation returns a non-stream result.

Streams, pipelines and operations have the following properties and restrictions:

- Pipelines are evaluated lazily. Only enough elements are consumed, or "pulled" through the pipeline, as required by the terminal operation.
- Pipelines are linear. There is a single stream source, and there is no branching mechanism for routing elements to different downstream operations.
- Streams can be traversed at most once. To use the data source again, a new stream has to be created.
- Operations must not change the data source.

Java 8 Streams further classify intermediate operations as stateless and stateful, depending on whether the operation needs to hold any state as data passes through. For example, `map` is a stateless operation as each element can be processed independently of another. However, the `distinct` operation (remove all duplicate elements from the Stream) is stateful because it must keep track of all encountered elements.

2.1 Java Stream API

A list of all intermediate and terminal operations defined in Java 8 Streams is provided below. Full details of the Java 8 Stream API can be found at [10].

```
1 public interface Stream<T> extends BaseStream<T, Stream<T>> {
2   // Intermediate operations
3   public Stream<T> distinct();
4   public Stream<T> filter(Predicate<? super T> predicate);
5   public <R> Stream<R> flatMap(
```

```
6       Function<? super T,? extends Stream<? extends R>> mapper);
7     public DoubleStream flatMapToDouble(
8       Function<? super T,? extends DoubleStream> mapper);
9     public IntStream flatMapToInt(
10      Function<? super T,? extends IntStream> mapper);
11    public LongStream flatMapToLong(
12      Function<? super T,? extends LongStream> mapper);
13    public Stream<T> limit(long maxSize);
14    public <R> Stream<R> map(Function<? super T,? extends R> mapper);
15    public DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper);
16    public IntStream mapToInt(ToIntFunction<? super T> mapper);
17    public LongStream mapToLong(ToLongFunction<? super T> mapper);
18    public Stream<T> peek(Consumer<? super T> action);
19    public Stream<T> skip(long n);
20    public Stream<T> sorted();
21    public Stream<T> sorted(Comparator<? super T> comparator);
22    public Object[] toArray();
23    public <A> A[] toArray(IntFunction<A[]> generator);
24
25    // Terminal operations
26    public boolean allMatch(Predicate<? super T> predicate);
27    public boolean anyMatch(Predicate<? super T> predicate);
28    public <R,A> R collect(Collector<? super T,A,R> collector);
29    public <R> R collect(Supplier<R> supplier,
30      BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner);
31    public long count();
32    public Optional<T> findAny();
33    public Optional<T> findFirst();
34    public void forEach(Consumer<? super T> action);
35    public void forEachOrdered(Consumer<? super T> action);
36    public Optional<T> max(Comparator<? super T> comparator);
37    public Optional<T> min(Comparator<? super T> comparator);
38    public boolean noneMatch(Predicate<? super T> predicate);
39    public Optional<T> reduce(BinaryOperator<T> accumulator);
40    public T reduce(T identity, BinaryOperator<T> accumulator);
41    public <U> U reduce(U identity, BiFunction<U,? super T,U> accumulator,
42      BinaryOperator<U> combiner);
43
44    // ...
45  }
```

2.2 Implementation of Streams

This section outlines how Java 8 Streams are implemented by default.

2.2.1 The fork-join framework

Introduced in Java 7, the fork-join framework allows programmers to specify tasks that can be subdivided and executed in parallel on multicore systems. Such tasks are submitted to a fork-join pool, which consists of a set of worker threads. Each worker thread has a task queue, and when empty,

steals tasks from other threads' queues. To simplify usage of the framework, a common fork-join pool is defined, with the number of worker threads defaulting to one less than the number of cores on the system. This common pool receives tasks from executing pipelines. In the context of Java Streams, a task is the pipeline that operates on part of the stream covered by a spliterator.

2.2.2 Parallel execution of pipelines

While an iterator is sufficient for executing a sequential pipeline, a *spliterator* is needed to operate on a parallel pipeline. A spliterator recursively partitions the stream by "splitting" itself to create child spliterators, allowing threads to then traverse the multiple spliterators in parallel.

The spliterator methods of interest are shown below:

```
1 public interface Spliterator<T> {
2   public default void forEachRemaining(Consumer<? super T> action);
3   public Spliterator<T> trySplit();
4
5   ...
6 }
```

To execute a pipeline in parallel, a spliterator covering the entire associated stream is first created. Recursive partitioning is achieved by creating child spliterators from parent spliterators when the `trySplit` method is called, as shown in Figure 2.

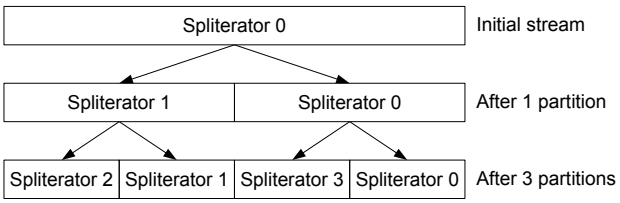

Figure 2: Recursive partitioning of a stream.

Partitioning stops when the `trySplit` method decides that further partitioning is inefficient, impossible or, if the stream's size is known in advance, when the default work granularity of 4 tasks per core is reached (this is to account for uneven workloads or when worker threads are blocked [9]). Each task is then submitted to the common fork-join pool for execution. When a task is executed, the data elements in the corresponding spliterator are traversed with the `forEachRemaining` method. Thus, the same element is moved through the entire pipeline by the same worker thread.

2.3 Example

The following method (from [5]) illustrates the use of Java 8 Streams. It counts the number of words in a collection of lines and prints each word together with its occurrence on a separate line.

```
1  static void wordcount(Collection<String> lines) {
2    // Word boundary regular expression
3    Pattern pat = Pattern.compile("\\s+");
4    Map<String, Long> words = lines
5      // Get parallel stream from collection of lines
6      .parallelStream()
7      // Replace line with words
8      .flatMap(line -> Arrays.stream(pat.split(line)))
9      // Count words
10     .collect(Collectors.groupingBy(
11       w -> w, TreeMap::new, Collectors.counting()));
12   // Get collection of (word, count) pairs
13   Set<Map.Entry<String, Long>> entries = words.entrySet();
14   entries
15     // Get sequential stream
16     .stream()
17     // Print each (word, count) pair
18     .forEach(e -> {
19       System.out.println(e.getKey() + "\t" + e.getValue()); });
20 }
```

Lines 3 to 10 establish a stream that outputs a `TreeMap` of words and their occurrences. The `flatMap` operation takes a line of text (`String`) as input and outputs a stream of words (`Strings`) backed by the array returned from the `split` method. The `collect` operation accumulates new words in a `TreeMap` and keeps a count of their occurrences.

The `entrySet` method turns the resulting `TreeMap` into a `Set` of *(word, count)* pairs. Finally, a stream of *(word, count)* pairs is established and the `forEach` operation prints each pair to standard output.

3 Requirements from real-time Big Data

Since Java 8 Streams exist within a single JVM, and JVMs tend to only support individual SMP or ccNUMA machines, using them to target Big Data systems requires an extension of the programming model to support computations involving multiple nodes. A primary requirement of the extended model is to be a drop-in replacement of the existing model. This implies that the properties of Java 8 Streams (for example, the lazy evaluation of pipelines) should be preserved as far as possible unless they conflict with requirements for executing in a distributed environment.

Streams and pipelines can be distributed among compute nodes independently (see Figure 3). A data source in a *distributed stream* is spread over or accessed by multiple nodes, while operations in a *distributed pipeline* span multiple nodes. Both methods potentially speed up processing, depending on the workload. An I/O-bound workload will see a speed improvement on a distributed stream, while a CPU-bound workload will benefit more on a distributed pipeline. Big Data computations are usually I/O-bound [7], thus emphasis should be on a distributed stream model. However, support for a distributed pipeline model must be available.

The JUNIPER communications model described in deliverable D2.1 already gives the developer a way to express a graph of communications in a distributed system and therefore is likely to make a good base for the implementation of distributed pipelines.

3.1 Distributed Streams

A Distributed Stream facilitates parallelism by having a replicated pipeline that operates on different parts of a dataset, which is usually located in multiple nodes. There are several reasons for using Distributed Streams:

- The dataset is too large to fit on disk in one node.
- It is too slow to read the entire dataset from a single node (due to limited bandwith, for example).

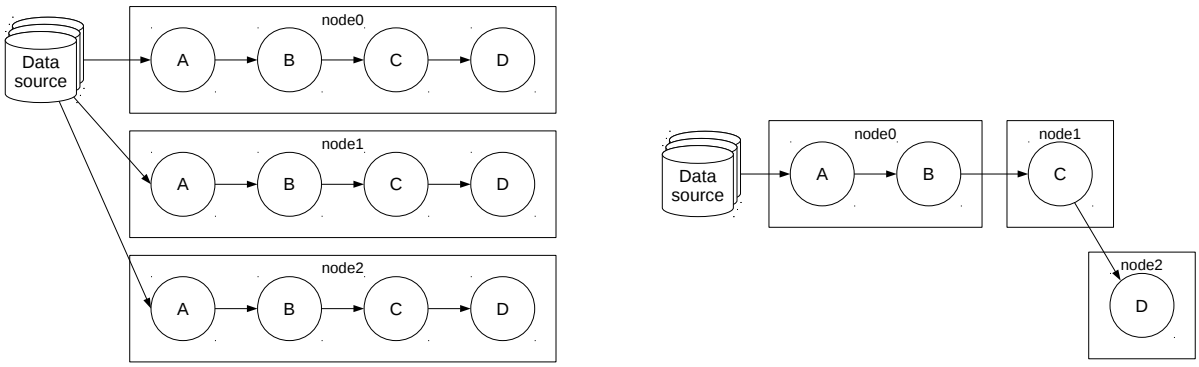

Figure 3: Conceptual model of a distributed stream (left) and a distributed pipeline (right). In the extended model, boxes indicate compute nodes, and the arrows indicate data flow from data source to pipeline operations.

- The dataset is located in storage nodes with specialised hardware, allowing fast parallel data access.

To maintain compatibility, Distributed Streams must be a drop-in replacement for Java 8 Streams. For example, the following which computes the sum of squares of a stream of integers:

```
1 long total = stream.map(n -> n * n).sum();
```

must not need modification regardless of whether the `stream` variable refers to a stream of integers or a distributed stream of integers. It will always compute the sum of the entire stream, distributed or not. Each node executing the program gets the same return value from the terminal operation.

At certain points in the pipeline, it may be necessary for data to be distributed and gathered over the compute nodes to do partitioning and aggregation, such as in the MapReduce shuffle stage. Java 8 Streams support gathering of data with the general-purpose terminal operations `collect` and `reduce`, but there are currently no operations for distributing data according to a given algorithm (as there is no requirement to do this in the current Java 8 streams). Therefore, low-level pipeline operations which facilitate the transfer of data using inter-node communication are needed for distributed streams.

These operations must also be general enough so that they can be used in a distributed pipeline model (to send data from one segment of the pipeline to another across nodes).

To facilitate communication across nodes, there also needs to be a system of identifying and grouping nodes. The programmer must be allowed to specify which node or group of nodes a computation takes place on, and the destination of operations that distribute data.

3.2 Lazy evaluation and distributing data

The Distributed Streams model does not require that all participating compute nodes execute exactly the same operations in the evaluation of a pipeline. For example, a pipeline can read data on a given set of input nodes, filter it on a different set of compute nodes, and then store it on a further different set of output nodes. This is described using the `distribute` operation introduced in section 4 which addresses the distribution of data requirement. `distribute` can be viewed as moving the evaluation of part of the pipeline from one subset of the target cluster to another. It is also essential for splitting and joining multiple streams, a requirement discussed in section 3.1.

However, the presence of `distribute` causes a problem with the lazy evaluation of pipelines. Consider the simple pipeline in figure 4. In this pipeline, the a operation is executing on multiple nodes in the target cluster, then after a `distribute` the c operation is only executing on a single node. This is a common pattern when a lot of high complexity processing is performed to generate a relatively small amount of output, such as searching a large dataset. Under lazy evaluation, c determines the rate at which data items are pulled through the pipeline. c still executes the evaluation in parallel (because it is a parallel stream) but now the programmer must ensure that sufficient threads are spawned on c to keep enough data items in flight so that all of the instances of a are kept busy. A similar problem exists in reverse, when a single instance of a is distributed to many instances of c.

This is problematic for the following reasons:

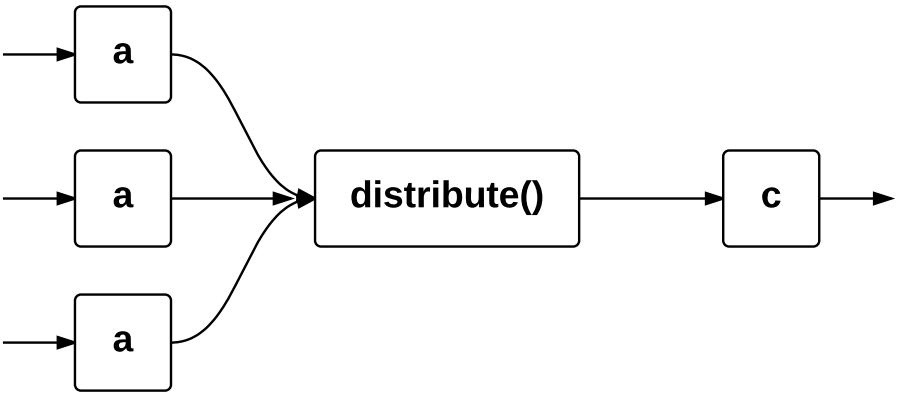

Figure 4: A simple pipeline section illustrating the use of `distribute`.

1. It creates an unnecessary dependency in otherwise unrelated software. Now, the number of threads spawned in c affects the concurrency potential of a.

2. It leaks an abstraction detail. The programmer must now worry about deployment concerns when programming c, when they only should have to consider algorithmic issues.

It should be highlighted that this is not simply a problem of matching data rates and balancing pipeline stages, which remains a problem with this model (as with Hadoop, Spark and Storm). This is a problem with expressing the potential concurrency throughout a pipeline. The entire purpose of `distribute` is to signify that the potential concurrency has changed because a different set of target nodes is now evaluating the pipeline.

Consequentially, the solution employed is to view `distribute` as terminating the current pipeline, and starting another. This allows a different amount of parallelism to be expressed in the new stream, and ensures that each section is evaluated as efficiently as possible.

3.3 Distributed Collections and Distributed Stored Collections

Deliverable D2.2 proposed the concept of Stored Collections, which extend Java 8 (in-memory) collections to efficiently read large datasets. Stored Collections are an example of the real-time technologies that are provided by the JUNIPER platform, integrating with JUNIPER kernel support to allow high priority tasks to reserve disk access bandwidth. This means that lower priority parts of the system cannot interfere, allowing for greater QoS guarantees.

Distributed forms of in-memory collections and Stored Collections are needed for caching intermediate results from Distributed Streams. Thus recomputing data is avoided at the expense of memory or disk space usage. To support this, it must be possible to save data to these collections from a Distributed Stream.

Additionally, it should be possible to extend a non-distributed collection without needing to reimplement the distributed version from scratch.

3.4 Summary of requirements

This section has shown that there are four main requirements:

1. Distributed Collections and Distributed Stored Collections are needed for storing intermediate results.
2. Distributed Streams must be a drop-in replacement for Java 8 Streams.
3. New eager operations for distributing data across nodes are needed, such as the distribute operation described in section 4.
4. Support for splitting and combining streams is needed.

From these we can also derive a further requirement, which is that the API extensions must provide a mechanism for the discovery and grouping of nodes in the cluster. This is not normally covered by Java, but is required so that distributed operations can be targeted at specific subsets of the cluster.

4 Extensions

This section addresses the requirements identified in section 3 and proposes a number of extensions to the existing JUNIPER programming model.

4.1 Compute nodes and groups

Before generalised distribution of data can be supported, a method of identifying compute nodes and groups of compute nodes is needed for decisions such as which node to send data to. We therefore define the `ComputeNode` class and the `ComputeGroup` class, which is a `List` of `ComputeNodes`:

```
1  public class ComputeNode {
2    public String getName();
3    public boolean isSelf();
4
5    public static ComputeNode getSelf();
6    public static ComputeNode findByName(String name);
7  }
8
9  public class ComputeGroup extends ArrayList<ComputeNode> {
10   public ComputeGroup();
11   public ComputeGroup(Collection<ComputeNode> nodes);
12   public ComputeGroup(ComputeNode node);
13
14   public static ComputeGroup getCluster();
15 }
```

Each compute node in the cluster is represented by a `ComputeNode` and has a unique name. There are also methods to get the current node, or a specific node by its name. A compute group is a list of compute nodes and is represented by a `ComputeGroup`. The entire cluster can be retrieved by calling the `getCluster` method. Standard Java `List` methods can be called to modify a compute group's members. Complete details of the API can be found in appendix A.

4.2 Distributed Collections

To orthogonally support in-memory, on-disk, non-distributed and distributed forms of collections, we define the following interfaces:

Properties	In-memory	On-disk
Non-distributed	`Collection`	`StoredCollection`
Distributed	`DistibutedCollection`	`DistributedStoredCollection`

The distributed versions of collections need replacement methods to return Distributed Streams. They also need to know the participating compute nodes. Thus, the Distributed Collection interface is defined as follows:

```
1  public interface DistributedCollection<E> extends Collection<E> {
2    public ComputeGroup getComputeGroup();
3    public DistributedStream<E> stream();
```

```
4   public DistributedStream<E> parallelStream();
5 }
```

The Distributed Stored Collection interface simply extends its parent interfaces:

```
1 public interface DistributedStoredCollection<E>
2   extends StoredCollection<E>, DistributedCollection<E> {
3 }
```

To save data into one of these collections over a compute group, the `localCollect` operator can be used together with the appropriate collector, supplier, accumulator and/or combiner parameters (e.g. the supplier allocates a new `DistributedCollection`, etc.). Thus data on each node is stored locally, avoiding unnecessary and costly communication between nodes.

4.3 Distributed Streams – Drop-in replacement extensions

To make Distributed Streams a drop-in replacement for Java 8 Streams, we propose a new `DistributedStream` interface that extends the existing `Stream` interface. The full definition of this interface can be found in appendix A.

```
1  public interface DistributedStream<T> extends Stream<T> {
2    public DistributedStream<T> distinct();
3    public DistributedStream<T> filter(Predicate<? super T> predicate);
4    public <R> DistributedStream<R> flatMap(
5      Function<? super T,? extends Stream<? extends R>> mapper);
6    public DistributedDoubleStream flatMapToDouble(
7      Function<? super T,? extends DoubleStream> mapper);
8    public DistributedIntStream flatMapToInt(
9      Function<? super T,? extends IntStream> mapper);
10   public DistributedLongStream flatMapToLong(
11     Function<? super T,? extends LongStream> mapper);
12   public DistributedStream<T> limit(long maxSize);
13   public <R> DistributedStream<R> map(
14     Function<? super T,? extends R> mapper);
15   public DistributedDoubleStream mapToDouble(
16     ToDoubleFunction<? super T> mapper);
17   public DistributedIntStream mapToInt(
18     ToIntFunction<? super T> mapper);
19   public DistributedLongStream mapToLong(
20     ToLongFunction<? super T> mapper);
21   public DistributedStream<T> parallel();
22   public DistributedStream<T> peek(Consumer<? super T> action);
23   public DistributedStream<T> sequential();
24   public DistributedStream<T> skip(long n);
25   public DistributedStream<T> sorted();
26   public DistributedStream<T> sorted(Comparator<? super T> comparator);
27
28   // ...
29 }
```

All operations that return a `Stream` need to be overridden by those that return a `DistributedStream`. The `DistributedStream` interface provides versions of the Stream operations like `distinct`, `flatMap`, etc. that return `DistributedStreams` instead of `Streams`.

Primitive-type stream interfaces (`IntStream`, `LongStream` and `DoubleStream`) will have similar interface definitions (`DistributedIntStream`, `DistributedLongStream` and `DistributedDoubleStream`). These interfaces are also defined in appendix A.

In a distributed environment, stateful intermediate operations potentially need to know the elements in every participating node in order to give a correct output. This may imply transferring all elements to a single node or doing remote comparisons, both of which result in heavy network traffic and/or increased memory usage. To address this issue, we provide new operations that deal only with local data (see section 4.4 for further discussion).

4.4 Distributed Streams – Distribution of data extensions

A functional interface for user-specified partitioning of data is defined as follows:

```
1  @FunctionalInterface
2  public interface Partitioner<T> {
3    public int partition(T data);
4  }
```

Similar interfaces for primitive-type partitioners (`IntPartitioner`, `LongPartitioner` and `DoublePartitioner`) for the appropriate primitive-type Distributed Streams are also defined and can be found in appendix A.

The `partition` method accepts a data element and returns an index representing a node in the compute group. For ease of use, it is not important for the programmer to know the size of the compute group, so if the index is out of range it will be wrapped around by the framework. This makes the partitioner suitable for range partitioning if the compute group size is known, as well as load balancing methods such as hash-based partitioning.

With a method to partition data, we propose the addition of several variants of the `distribute` method that transfer data from one compute group to another, and of methods to modify and retrieve the compute group:

```
1  public interface DistributedStream<T> extends Stream<T> {
2    public DistributedStream<T> distribute();
3    public DistributedStream<T> distribute(Partitioner<? super T> p);
4    public DistributedStream<T> distribute(ComputeGroup grp);
5    public DistributedStream<T> distribute(
6      ComputeGroup grp, Partitioner<? super T> p);
7    public DistributedStream<T> distribute(ComputeNode node);
8    public DistributedStream<T>[] distribute(ComputeGroup[] grps);
9    public DistributedStream<T>[] distribute(
10     ComputeGroup[] grps, Partitioner<? super T> p);
11
12   public ComputeGroup getComputeGroup();
13   public void setComputeGroup(ComputeGroup grp);
14
```

```
15    // ...
16  }
```

Parameters change the behaviour of `distribute` as follows:

- If used without parameters, `distribute` sends data to the same compute group according to a default hash-based partitioner.
- If a partitioner is given, it is used in place of the default hash-based partitioner.
- If a compute group/node is given, data is partitioned and sent to that group/node instead. The nodes in the specified compute group do not have to be part of the initial compute group.
- If an array of compute groups is given, data is distributed to each group in the collection as described above, with every group receiving the same elements.

A `distribute` operation returns either a new Distributed Stream or an array of new Distributed Streams (in the case of splitting a stream) consisting of the same data elements which may have been moved across nodes. Combining streams is currently being worked on.

To complete the extensions, the behaviour of some terminal operations need to be addressed. Due to the drop-in replacement requirement, terminal operations return the same result on all participating nodes. However, for operations such as `collect`, a large result requires significant network communication to replicate data elements on each node. This may be undesirable or unnecessary. Hence, we propose the addition of local variants of these operations that prevent data from being replicated on other nodes:

```
1  public interface DistributedStream<T> extends Stream<T>
2  {
3    // ...
4
5    public <R, A> R localCollect(Collector<? super T, A, R> collector);
6    public <R> R localCollect(Supplier<R> supplier,
7      BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner);
8    public DistributedStream<T> localDistinct();
9    public DistributedStream<T> localForEach(Consumer<? super T> action);
10   public DistributedStream<T> localLimit(long maxSize);
11   public DistributedStream<T> localPeek(Consumer<? super T> action);
12   public Optional<T> localReduce(BinaryOperator<T> accumulator);
13   public T localReduce(T identity, BinaryOperator<T> accumulator);
14   public <U> U localReduce(U identity,
15     BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner);
16   public DistributedStream<T> localSkip(long n);
17   public DistributedStream<T> localSorted();
18   public DistributedStream<T> localSorted(Comparator<? super T> comparator);
19
20   // ...
21 }
```

Both variants behave identically on single-node clusters. The local operations can be used to construct more efficient pipelines. On-disk caching of results can be achieved by having the `localCollect` operation accumulate data into a Distributed Stored Collection.

Examples showing the use of Distributed Streams can be found in section 6.3.

5 Prototype implementation

This section describes how the library is implemented. The JUNIPER project work packages do not require maturation of these extensions. They exist to demonstrate the wide applicability of the platform for a range of Big Data paradigms. However, they are very useful for real-world use of the platform and so York intends to continue implementation and academic exploitation of this work.

5.1 Compute nodes and groups

Our implementation uses MPI [4] to communicate between nodes. Each compute node is identified by an integer identical to its global MPI communicator (`COMM_WORLD`) rank. The name of each node is the string "`node`" concatenated with its rank (in other implementations, this can be read from a configuration file, for example). The `ComputeNode` class in our implementation is also the program's entry point. The `main` method initialises MPI, discovers the other nodes in the cluster and runs the actual program. It also cleans up after the program finishes.

As defined in section 4.1, a compute group is an `ArrayList` of compute nodes. Thus it supports all `ArrayList` operations. The main addition is the `getCluster` method, which creates a new compute group containing all nodes in the cluster as discovered by `ComputeNode.main`. The first node in each group is designated as the root.

This implementation assumes no node failures; this will be addressed in future work.

5.2 Distributed Collections

To carry out evaluations in section 8, we implemented a `DistributedStringStoredCollection` using the `StringStoredCollection` class previously written in [5] as follows:

```
1 public class DistributedStringStoredCollection extends StringStoredCollection
2   implements DistributedCollection<String> {
3   ...
4 }
```

The implementation assumes that the dataset is distributed over the entire cluster, and that the files can be identified using the same path prefix and a suffix that varies with each node (this is to guarantee unique filenames for single multicore tests where the dataset is distributed over files in the same directory).

5.3 Distributed Streams – Distribution of data extensions

5.3.1 Default partitioner

The default hash-based partitioner in our implementation computes the hash by calling the `Object.hashCode` method.

5.3.2 The `distribute` operation

As mentioned in section 3.2, the operation terminates the existing pipeline and starts a new one. The following algorithm describes the core of all `distribute` operation variants:

The existing pipeline is terminated with a `localForEach` operation, which sends each element to the appropriate destination node. (The MPI implementation serialises each element. Thus, our implementation can only send objects that implement the `Serializable` interface.) After all elements are sent, an end-of-data marker is sent to all participating nodes. We use a null object message to represent this marker.

Concurrently, a new pipeline is created and converts incoming messages into data elements. It determines that no more data is available when it has received end-of-data markers from all participating nodes.

Since stream computations block until the terminal operator has output a result, extra threads are created to work around the blocking and keep the two pipelines executing concurrently. This incurs overheads as the OS may repeatedly schedule and deschedule the threads.

5.4 Distributed Streams – Drop-in replacement extensions

We give details of how some of the Distributed Stream operations are implemented to satisfy the drop-in replacement requirement.

5.4.1 The `reduce` operation

There are a number of `reduce` operations for each stream type, but all have implementations similar to the following:

1. A reduction is performed on local data elements with the `localReduce` operation.
2. The local result is sent to the first node in the compute group.
3. The first node receives and accumulates all local results, and sends the final result to the other nodes.
4. The other nodes receive the final result. All nodes return the same value.

5.4.2 The `allMatch` operation

The `allMatch` operation can be expressed as a reduction with the result being the logical-AND of local `allMatch` operations on each participating node.

5.4.3 The `count` operation

The `count` operation can be expressed as a reduction with the result being the sum of local `count` operations on each participating node.

5.4.4 The `distinct` operation

The `distinct` operation can be expressed in terms of the `distribute` operation followed by the `localDistinct` operation.

```
1 stream
2     ...
3     .distribute()
4     .localDistinct()
5     ...
```

The `distribute` operation sends elements with the same hash value (which includes all identical elements) to the same node. The `localDistinct` operation then removes duplicate elements within each node.

5.4.5 The `forEach` operation

To be a drop-in replacement, each participating node needs to perform the specified action on every element in the Distributed Stream. Thus, each node broadcasts its local data elements to other nodes, thereby ensuring that each local stream contains elements from all nodes. However, this is unlikely to be efficient. The `localForEach` operation avoids broadcasting elements and is intended for programmers to optimise their implementations.

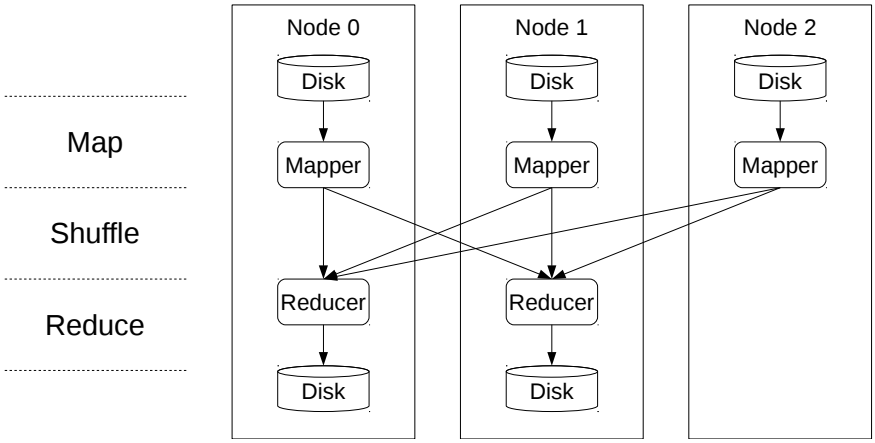

Figure 5: The three stages of a MapReduce computation. Arrows indicate the flow of data. Note that the number of mappers and reducers do not have to be equal.

6 Mapping MapReduce to JUNIPER

Section 5 described how the API extensions are implemented. The following sections will now show how the programming models of three popular Big Data approaches (Hadoop, Spark, and Storm) can be expressed in terms of this model.

Evaluation of the implementation and these comparisons can be found in section 8.

6.1 MapReduce and Hadoop

MapReduce [6] is a popular batch processing model for Big Data computations, allowing computations on a dataset that is typically partitioned over a cluster of nodes. Hadoop [1] is a popular open source implementation of MapReduce. This section first gives an overview of the programming model (including Hadoop-specific details where necessary), then demonstrates how it can be implemented with Distributed Streams.

Before a MapReduce computation takes place, the required data needs to be distributed over the cluster. Hadoop achieves this with the *Hadoop Distributed File System* (HDFS) [12], which is overlaid on the host file system and manages the distribution of datasets across a cluster. Files in HDFS are divided into blocks (usually of 64MB) and replicated on different nodes by default. To avoid data coherency issues, files can only be written to once.

MapReduce requires programmers to implement *mappers* and *reducers*, each of which normally run on a node in the cluster. With reference to Hadoop, a MapReduce computation consists of three stages (also see Figure 5).

1. The *map* stage: Input data from a given file in HDFS is processed by a set of mappers, which output key-value pairs as a result.

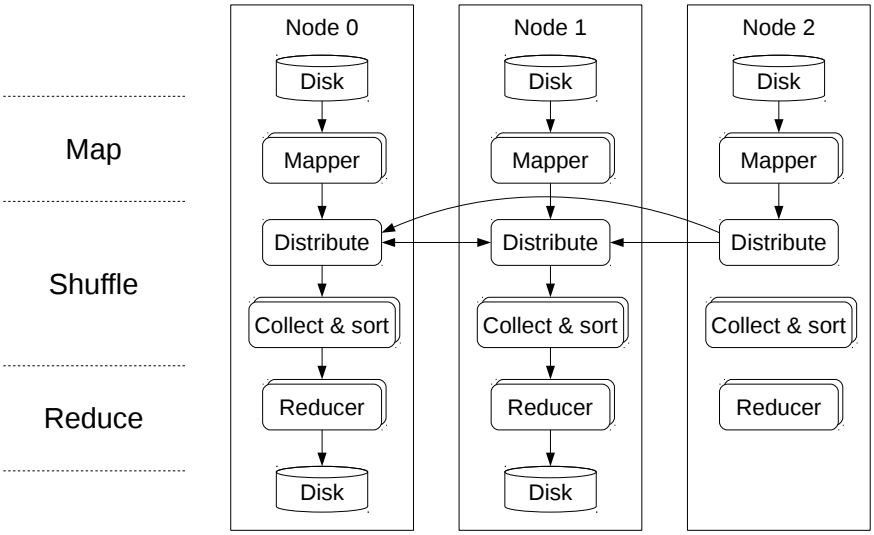

Figure 6: Expressing a MapReduce computation in terms of distributed streams. Arrows indicate the flow of data. The mapper, reducer, local collect and local sort may span multiple stream operations.

2. The *shuffle* stage: The key-value pairs are collected over a set of reducers, with the same keys sent to the same reducer. For each key, the values are collected and sorted in a key-*value-list* pair.

3. The *reduce* stage: Each reducer processes all given key-*value-list* pairs and outputs the result in a local HDFS file. The full result of the computation is the concatenation of all partial results.

If the dataset is spread across the mapper nodes, Hadoop can optimise execution times by having each mapper node work on its local data.

6.2 Distributed Stream implementation

With Java 8 streams, performing MapReduce computations across a cluster was not possible as the framework operated within a single node. Distributed streams solve this problem by defining operations to transfer data across nodes. In MapReduce, the shuffle stage is where data transfer is needed. This stage can be broken down into sub-stages and implemented with distributed streams as follows:

1. The `distribute` operation transfers data (in key-value form) between compute nodes such that those with the same key are sent to the same node.

2. The `localCollect` operation on each node accumulates incoming data into a collection of key-*value-list* pairs. The value lists are optionally sorted. Since `localCollect` is a terminal operation, a new stream consisting of elements in the collection is created and passed to the reduce stage.

Figure 6 shows in general how a MapReduce computation can be represented with pipeline operations.

Special cases such as summing, counting and collecting are built into Distributed Streams. Hence the resulting code is more concise if such operations are used.

6.3 Illustrative example

We return to the *word-count* application introduced in section 2.3, which outputs a sorted list of words together with their frequencies, as an example of using Java 8 Streams and Stored Collections for MapReduce computations. Since Distributed Streams and Distributed Stored Collections are drop-in replacements, the same algorithm can be used on an input text file distributed over the cluster, and the computation also occurs cluster-wide. Performance can be improved, for example, by using `localForEach` to output only the elements on the local node:

```
1  void wordcount(DistributedCollection<String> lines) {
2    Pattern pat = Pattern.compile("\\s+");
3    Map<String, Long> words = lines
4      .parallelStream()
5      // Map stage
6      .flatMap(line -> Arrays.stream(pat.split(line)))
7      // Shuffle stage
8      .collect(Collectors.groupingBy(
9        w -> w, TreeMap::new, Collectors.counting()));
10   Set<Map.Entry<String, Long>> entries = words.entrySet();
11   entries
12     .stream()
13     // Reduce stage
14     .localForEach(e -> {
15       System.out.println(e.getKey() + "\t" + e.getValue()); });
16 }
```

Since the Distributed Collection has information on the location of the dataset in the form of a compute group, only the nodes in that group will take part in the computation.

If only a subset of nodes are needed for reduction, the pipeline can be modified to distribute the elements to the required subset. Using the example above, the addition of a distribute operation in line 8 funnels the data for reduction into a single node:

```
1  void wordcount(DistributedCollection<String> lines) {
2    Pattern pat = Pattern.compile("\\s+");
3    Map<String, Long> words = lines
4      .parallelStream()
5      // Map stage
6      .flatMap(line -> Arrays.stream(pat.split(line)))
7      // Shuffle stage -- send to first node in compute group
8      .distribute(lines.getComputeGroup().get(0))
9      .collect(Collectors.groupingBy(
10       w -> w, TreeMap::new, Collectors.counting()));
11   Set<Map.Entry<String, Long>> entries = words.entrySet();
12   entries
13     .stream()
14     // Reduce stage
15     .localForEach(e -> {
```

```
16          System.out.println(e.getKey() + "\t" + e.getValue()); });
17  }
```

7 Mapping in-memory streaming to JUNIPER

Though suitable for many applications, the MapReduce model is inflexible due to the fixed stage sequence. More recently, focus has shifted to in-memory stream-based models for Big Data processing, with the Spark [2] and Storm [8] frameworks being prominent examples of such models. We focus on these frameworks in this section, comparing their models with that of Distributed Streams.

7.1 Spark and comparisons with Distributed Streams

Spark removes some of MapReduce's limitations by allowing pipeline-based data processing. Instead of implementing mappers and reducers, programmers specify a pipeline of operations on a *Resilient Distributed Dataset* (RDD). An operation either returns a new RDD (a *transformation*, which can be chained) or returns a value (an *action*, which terminates the pipeline).

Since both Spark and Distributed Streams are stream-based programming models, there are some similarities. For example, Spark pipelines are also lazily evaluated, and their Java syntax resembles that of Java 8 streams. Also, transformations are analogous to intermediate operations and actions are equivalent to terminal operations.

To demonstate the similarities between Spark's and Distributed Streams' operations, the following table lists a number of Spark operations together with the equivalent ones in Distributed Streams:

Spark operation	Distributed Stream equivalent	Description
`.countByKey()`	`.collect(Collectors.groupingBy(e -> e.getKey(), Collectors.counting()))`	Counts the occurrence of each key and returns a map of key-count pairs.
`.filter(p)`	`.filter(p)`	Removes elements in the Distributed Stream that do not satisfy the predicate.
`.foreach(f)`	`.forEach(f)`	Execute a function over each element.
`.groupByKey()`	`.collect(Collectors.groupingBy(e -> e))`	Groups key-value pairs into key-*value-list* pairs.
`.map(f)`	`.map(f)`	Replaces each element with those from the mapping function.
`.reduce(f)`	`.reduce(f)`	Reduces the elements to a single value using a binary function.
`.take(n)`	`.limit(n).collect(Collectors.toList())`	Returns the first `n` elements in a `List`.

There are also significant differences between the models. Spark was designed specifically for Big Data applications, whereas distributed streams have to maintain compatibility with Java 8 streams. RDDs can be reused, unlike Java 8 Streams, and Spark allows caching of data in memory for frequently-used RDDs to avoid recomputing data. Also, Java 8 streams are conceptually separate from Java collections (which can be the source of a stream), but Spark RDDs do not have such a distinction. Depending on its position in the pipeline, an RDD may contain data from HDFS blocks,

or transformed data cached in memory, or even information on how the data should be processed (to support lazy evaluation). This allows the entire pipeline to be lazy even across compute nodes.

Since Spark is a high-level framework, it does not have operations specifically for distributing data across nodes. This is done by the various transformations and actions when required.

7.2 Storm and comparisons with Distributed Streams

Storm executes *topologies* which run indefinitely on a cluster. A topology defines a directed acyclic graph of nodes and data *streams* (vertices). A stream consists of an unbounded sequence of tuples. A node is either a *spout* (data source – a Twitter feed, for example) or *bolt* (consumes and processes streams, and may emit new streams).

Storm is also a stream-based programming model, but it is eager and emphasises task-parallel computations, setting it apart from Distributed Streams which are mainly lazy and data-parallel. Nodes in Storm normally run different computations which are then joined together with streams. Hence it is closer to a distributed pipeline model (see section 3) where each node processes a segment of the pipeline and does not have knowledge of the entire pipeline. On the other hand, a Distributed Stream is primarily replicated pipeline-based and makes use of compute groups to partition the cluster for different data-parallel computations.

7.3 Distributed Stream implementation for Spark

Due to the similarities in programming models, parts of the Spark and Distributed Stream pipelines can be almost identical. However, a Distributed Stream implementation will need to be aware of the following due to differences in the programming models:

- A feature of Spark is that it can provide caching and reuse of computed values in a pipeline. This is not currently automated by Distributed Streams. The programmer must implement this manually through the use of `localCollect` and distributed collections.
- Since Spark has the concept of a driver program which defines the pipeline and submits work to the master node, actions such as `collect` send data to the driver instead of to all processes on participating nodes. To achieve a similar effect of sending all data to one compute node, the `distribute(`*`node`*`)` operation can be used.
- Like Distributed Stream operations, a number of Spark transformations and actions require significant inter-node communication when dealing with datasets spanning multiple nodes and should be avoided if there are still many elements in the dataset.
- If the underlying data source has no redundancy (we believe this is rare as common distributed file systems such as HDFS and Lustre have this property), Distributed Stored Collections will need to implement it.

7.4 Distributed Stream implementation for Storm

Since Storm uses a distributed pipeline model, the programmer can achieve a similar result by splitting the pipeline into several segments and executing the `distribute` operation at segment bound-

aries. For nodes that send data to multiple destinations, the `distribute` variant that accepts a collection of compute groups can be used.

7.5 Illustrative example

To demonstrate the similarities and differences between Spark and Distributed Streams, we refer to the *word-count* example in section 6.3:

```
1  static void wordcount(Collection<String> lines) {
2    Pattern pat = Pattern.compile("\\s+");
3    lines
4      .parallelStream()
5      .flatMap(line -> Arrays.stream(pat.split(line)))
6      .collect(Collectors.groupingBy(
7        w -> w, TreeMap::new, Collectors.counting()))
8      .entrySet()
9      .stream()
10     .forEach(e -> {
11       System.out.println(e.getKey() + "\t" + e.getValue()); });
12 }
```

The corresponding Spark code [3] (using Java 8) is as follows:

```
1  static void wordcount(JavaRDD<String> lines) {
2    Pattern pat = Pattern.compile("\\s+");
3    List<Tuple2<String, Integer>> result = lines
4      .flatMap(line -> Arrays.asList(pat.split(line)))
5      .mapToPair(s -> new Tuple2<String, Integer>(s, 1)) // Convert to (word, 1)
6      .reduceByKey((i1, i2) -> i1 + i2) // Add pairs with same words together
7      .collect(); // Save as list of (word, count) pairs
8    for (Tuple2<?,?> t : result) // Print each (word, count) pair
9      System.out.println(t._1() + "\t" + t._2());
10 }
```

After replacing each line of text with the individual words (using `flatMap`), each word is converted to a *(word, 1)* pair. The `reduceByKey` transformation reduces values *(word, M)* and *(word, N)* to *(word, M + N)*. The pipeline ends with `collect` which saves the remaining pairs into a `List`. Finally, the list contents are printed to standard output.

For Storm, we split the pipeline into two bolts which are outlined below:

```
1  public class Splitter implements IRichBolt {
2    // Private variables are initialised in the prepare() method
3    private OutputCollector collector;
4
5    @Override public vold execute(Tuple line) {
6      Pattern pat = Pattern.compile("\\s+");
7      for (String word: pat.split(line.getString(0)))
8        collector.emit(new Values(word)); // Send each word to next bolt
9      collector.ack(line);
10   }
11
12   // ...
```

```
13 }
14
15 public class Counter implements IRichBolt {
16   // Private variables are initialised in the prepare() method
17   private OutputCollector collector;
18   private TreeMap<String, Integer> wordcount;
19
20   @Override public vold execute(Tuple t) {
21     String word = t.getString(0);
22     if (wordcount.containsKey(word))
23       wordcount.put(word, wordcount.get(word) + 1); // Update occurrence
24     else
25       wordcount.put(word, 1); // Add new word
26     collector.ack(t);
27   }
28
29   // ...
30 }
```

The bolts can be incorporated into a topology with the following code:

```
1 TopologyBuilder builder = new TopologyBuilder();
2 builder.setSpout("input", ...);
3 builder.setBolt("split", new Splitter()).shuffleGrouping("input");
4 builder.setBolt("count", new Counter()).shuffleGrouping("split");
5 // ...
```

When run, the `Splitter` accepts lines of text and outputs individual words. Concurrently, the `Counter` adds each word from the `Splitter` into a `TreeMap` or updates its occurrence if already present. Since Spark topologies run indefinitely, an external signal, special marker or timeout is needed to indicate the end of input and that the `TreeMap` is ready to be output.

8 Initial evaluation

The purpose of this preliminary evaluation is to demonstrate that the extended JUNIPER APIs described in this document allow efficient description of both map-reduce and distributed streaming processing. Direct comparisons of end-to-end execution times for JUNIPER, Spark and Hadoop are often not appropriate because each system performs different styles of computation, at different times, and with different data distribution, replication, and fault tolerance guarantees.

Therefore this section attempts to remove initial distribution from the comparison and focus solely on the computation by pre-distributing the input data. This is how Hadoop (and Spark on HDFS) works normally, but it is not required by Storm or the JUNIPER Distributed Streams approach.

8.1 Experimental setup

All tests were carried out on a cluster with one master and six compute nodes:

- Intel Core 2 Duo E6600 with 2GB RAM (master)
- Intel Core 2 Duo E6400 with 2GB RAM
- Intel Core 2 Duo E6400 with 4GB RAM
- Intel Core 2 Duo E8400 with 8GB RAM
- Intel Core 2 Duo E8400 with 4GB RAM $\times$ 3

Each node has a single 7200RPM hard disk attached. Linux is installed on the cluster: the master node runs Ubuntu 12.04 and all compute nodes run Debian 7. All nodes are connected with gigabit ethernet via a switch.

For Distributed Streams, input data is distributed equally between all compute nodes and resides on their local filesystems.

For the Hadoop tests, version 1.2.1 is used. Input data is copied to HDFS before computation begins. The master node acts as the HDFS namenode as well as the jobtracker, while the compute nodes are each HDFS datanodes and tasktrackers.

For the Spark tests, version 1.1.1 is used, with the same HDFS configuration as above.

8.2 Tests

Execution times for word-count were measured:

- For Distributed Streams, the existing implementation based on Java 8 Streams and Collections (in section 2.3) was used.
- For Hadoop, the implementation in the `hadoop-examples` JAR file (included in standard Hadoop installations) was used.
- For Spark, the implementation in the `spark-examples` JAR file (included in standard Spark installations) was used.

The input dataset is a text file derived from [11] and repeated to give the following sizes:

Input size	45MB		450MB		4.5GB	
Library/framework	Time	Std. dev.	Time	Std. dev.	Time	Std. dev.
Distributed Streams	13.6	0.309	20.3	0.278	79.3	1.13
Hadoop	20.7	0.497	62.1	4.90	248	3.06
Spark	19.9	0.969	57.0	0.879	423	1.48

Table 1: Word-count execution times and standard deviations (in seconds).

- 45MB
- 450MB
- 4.5GB

30 runs were carried out with each input size. Timing begins after data is distributed to the compute nodes and ends after the computation results are written to disk. Since only a minimal amount of data (not more than 8MB) is written to disk, the results are therefore representative of computation times alone.

8.3 Results and evaluation

Table 1 shows the mean execution times and standard deviations obtained from the tests described in section 8.2. Distributed Streams are between 1.4 and 3.1 times faster and are more predictable (they have smaller standard deviations) than Hadoop or Spark for the input sizes tested. They also scale as well as Hadoop as input sizes increase.

The results indicate that Distributed Streams are lightweight compared to Hadoop and Spark. This is evident from execution times of the smallest input dataset, where communication overheads are more significant. For this dataset, a Distributed Stream job takes at least 6.3 seconds shorter to complete than a Hadoop or Spark job. Furthermore, it can be seen that even in this stock example the range of observed execution times is lower with the JUNIPER approach. This is purely because, as a thinner layer, less unpredictable overhead is introduced. Section 9 discusses how the other facilities of the JUNIPER approach could then be added to this to further increase predictability.

To reiterate, the purpose of these initial tests is not to demonstrate that the JUNIPER approach is faster, as this is not directly comparable. The Distributed Streams extension described in this document does not provide any fault tolerance above that which is already provided by the underlying MPI implementation. Instead, these results illustrate that as a thin layer, the JUNIPER API is a suitable base from which it is possible to efficiently implement many different Big Data processing models.

Since the current implementation of Distributed Streams is a proof-of-concept and not heavily optimised, we believe that there is room for improvements in efficiency and predictability. We are also confident that performance can be maintained even after reliability issues have been addressed.

9 Further exploitation of the JUNIPER platform

In this deliverable we have shown how the Distributed Streams API extension allows the programming models of Hadoop, Spark and Storm to be conveniently expressed using the JUNIPER platform. This is one advantage of using the JUNIPER approach, but to further improve on their applications' real-time behaviour, programmers can also take advantage of additional features of the JUNIPER platform.

Locales (described in deliverable D2.1) allow the programmer to better target large-scale ccNUMA and NUMA servers. Whilst this document has only shown the use of standard pipeline operations, real-world data processing applications will define their own pipeline stages. For greater real-time performance these stages can use the JUNIPER hardware APIs and Locales to control thread and data placement throughout the individual nodes of the cluster.

Another advantage of using Locales is that a Locale can use the JUNIPER API to request disk bandwidth reservations. Reservations allow the program to reserve an amount of disk access time for high priority tasks. This increases the worst-case response time of high priority tasks at the cost of analysis pessimism in lower-priority tasks. Reservations and response time analysis is discussed in deliverables D3.2 and D5.5.

The Real-Time Specification for Java is supported by the JUNIPER platform and allows the developer to express their software in a way which is more amenable to real-time systems. The RTSJ allows for better static analysis of the input software, and can better control memory usage throughout the application.

Related to the use of the RTSJ, the JUNIPER system software supports more advanced deadline-based scheduling policies which can be exploited by the developer. This is discussed in deliverable D3.1.

10 Conclusions

Extending Java 8 Streams to distribute data-parallel computation over a cluster enables Big Data applications to be written with JUNIPER's programming API. From the comparisons above, we believe that the programming model provided by Distributed Streams and augmented with distributed in-memory and on-disk collections is flexible enough as a base to implement the common Big Data paradigms. With the use of real-time features in Distributed Stored Collections, for example, such applications can meet timing requirements in addition to running "fast enough".

Work is in progress to fully implement Distributed Streams and Distributed Stored Collections, as well as to optimise their performance. In the future, we plan to address reliability issues such as node failures and unreliable data sources, and to further quantify the real-time benefits provided by the JUNIPER approach to such applications.

A Distributed Stream API

This Appendix contains the JavaDoc for the JUNIPER Distributed Stream API extensions and all related classes.

A.1 `DistributedStream`

```
1  package dstream;
2
3  import java.util.*;
4  import java.util.function.*;
5  import java.util.stream.*;
6
7  /**
8   * A distributed extension of Java 8 Streams.
9   * The associated pipeline is replicated on each participating compute node.
10  * @see java.util.stream.Stream
11  */
12 public interface DistributedStream<T> extends Stream<T>
13 {
14   /**
15    * Returns the current Distributed Stream's compute group.
16    * This group determines the participating nodes in the computation.
17    * @return Current Distributed Stream's compute group.
18    */
19   public ComputeGroup getComputeGroup();
20
21   /**
22    * Changes the current Distributed Stream's compute group.
23    * @param grp New compute group.
24    */
25   public void setComputeGroup(ComputeGroup grp);
26
27   // Data distribution operations
28
29   /**
30    * Sends data elements between nodes in the current compute group
31    * according to a hash-based partitioner.
32    * A stateful eager intermediate operation.
33    * Hash function is the Object.hashCode method.
34    * Elements with the same hash value are sent to the same destination.
35    * Suitable for MapReduce-style shuffling of data.
36    * @return Distributed Stream consisting of all elements.
37    */
38   public DistributedStream<T> distribute();
39
40   /**
41    * Sends data elements between nodes in the current compute group
42    * according to the specified partitioner.
43    * A stateful eager intermediate operation.
44    * @param p Programmer-defined partitioner.
```

```
45      * @return Distributed Stream consisting of all elements.
46      */
47     public DistributedStream<T> distribute(Partitioner<? super T> p);
48
49     /**
50      * Sends data elements from nodes in the current compute group to
51      * another compute group according to a hash-based partitioner.
52      * A stateful eager intermediate operation.
53      * Suitable for MapReduce-style shuffling of data.
54      * @param grp Destination compute group.
55      * @return Distributed Stream consisting of all elements.
56      */
57     public DistributedStream<T> distribute(ComputeGroup grp);
58
59     /**
60      * Sends data elements from nodes in the current compute group to
61      * another compute group according to the specified partitioner.
62      * A stateful eager intermediate operation.
63      * @param grp Destination compute group.
64      * @param p Programmer-defined partitioner.
65      * @return Distributed Stream consisting of all elements.
66      */
67     public DistributedStream<T> distribute(ComputeGroup grp,
68       Partitioner<? super T> p);
69
70     /**
71      * Sends data elements from nodes in the current compute group to
72      * the specified node.
73      * A stateful eager intermediate operation.
74      * @param node Destination compute node.
75      * @return Distributed Stream consisting of all elements.
76      */
77     public DistributedStream<T> distribute(ComputeNode node);
78
79     /**
80      * Sends data elements from nodes in the current compute group to
81      * multiple compute groups according to a hash-based partitioner.
82      * A stateful eager intermediate operation.
83      * @param grps Array of destination compute groups.
84      * @return Array of Distributed Streams each consisting of all elements.
85      */
86     public DistributedStream<T>[] distribute(ComputeGroup[] grps);
87
88     /**
89      * Sends data elements from nodes in the current compute group to
90      * multiple compute groups according to the specified partitioner.
91      * A stateful eager intermediate operation.
92      * @param grps Destination compute group.
93      * @param p Programmer-defined partitioner.
94      * @return Array of Distributed Streams each consisting of all elements.
95      */
96     public DistributedStream<T>[] distribute(ComputeGroup[] grps,
97       Partitioner<? super T> p);
```

```
98
99    // Local operations
100
101   /**
102    * Accumulates local data elements  on each node into a container.
103    * Equivalent to executing collect() on each local stream.
104    * A terminal operation.
105    * @param collector An operation which includes supplier, accumulator
106    * and combiner functions. @see java.util.stream.Collector
107    * @return The resulting container.
108    */
109   public <R, A> R localCollect(Collector<? super T, A, R> collector);
110
111   /**
112    * Accumulates local data elements on each node into a container.
113    * Equivalent to executing collect() on each local stream.
114    * A terminal operation.
115    * @param supplier Function returning a new container.
116    * @param accumulator Function to add a new element into a container.
117    * @param combiner Function for combining two containers.
118    * @return The resulting container.
119    */
120   public <R> R localCollect(Supplier<R> supplier,
121     BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner);
122
123   /**
124    * Returns the number of data elements in the local stream.
125    * Equivalent to executing count() on each local stream.
126    * A terminal operation.
127    * @return The number of local data elements.
128    */
129   public long localCount();
130
131   /**
132    * Removes duplicate elements in the local stream.
133    * Equivalent to executing distinct() on each local stream.
134    * An intermediate operation.
135    * @return Distributed Stream without duplicate local data elements.
136    */
137   public DistributedStream<T> localDistinct();
138
139   /**
140    * Executes an action on each local element.
141    * Encounter order is not preserved.
142    * Equivalent to executing forEach() on each local stream.
143    * A terminal operation.
144    * @param action Action to apply on each element. Must be non-interfering.
145    */
146   public void localForEach(Consumer<? super T> action);
147
148   /**
149    * Keeps only the specified number of elements in each local stream.
150    * Equivalent to executing limit() on each local stream.
```

```
151      * An intermediate operation.
152      * @param maxSize Maximum number of elements to keep.
153      * @return Distributed Stream with each local stream
154      * truncated to maxSize elements.
155      */
156     public DistributedStream<T> localLimit(long maxSize);
157
158     /**
159      * Performs an action on each local element and returns the same
160      * Distributed Stream.
161      * Equivalent to executing peek() on each local stream.
162      * An intermediate operation.
163      * @param action A non-interfering action.
164      * @return The same Distributed Stream.
165      */
166     public DistributedStream<T> localPeek(Consumer<? super T> action);
167
168     /**
169      * Accumulates local elements into a single value of the same type.
170      * Equivalent to executing reduce() on each local stream.
171      * A terminal operation.
172      * @param accumulator Associative accumulating function.
173      * @return An Optional of either the reduced value or an empty value.
174      */
175     public Optional<T> localReduce(BinaryOperator<T> accumulator);
176
177     /**
178      * Accumulates local elements into a single value of the same type.
179      * Equivalent to executing reduce() on each local stream.
180      * A terminal operation.
181      * @param identity The accumulating function's identity value.
182      * @param accumulator Associative accumulating function.
183      * @return The reduced value.
184      */
185     public T localReduce(T identity, BinaryOperator<T> accumulator);
186
187     /**
188      * Accumulates local elements into a single value.
189      * Equivalent to executing reduce() on each local stream.
190      * A terminal operation.
191      * @param identity The accumulating function's identity value.
192      * @param accumulator Function to accumulate elements into the value.
193      * @param combiner Function to combine two values.
194      * @return The reduced value.
195      */
196     public <U> U localReduce(U identity,
197       BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner);
198
199     /**
200      * Removes the first n elements in each local stream.
201      * Equivalent to executing skip() on each local stream.
202      * An intermediate operation.
203      * @param n Number of local elements to skip.
```

```
204      * @return Distributed Stream with the elements removed.
205      */
206     public DistributedStream<T> localSkip(long n);
207
208     /**
209      * Sorts all elements in each local stream.
210      * Equivalent to executing sorted() on each local stream.
211      * An intermediate operation.
212      * @return Distributed Stream with the elements locally sorted.
213      */
214     public DistributedStream<T> localSorted();
215
216     /**
217      * Sorts all elements in each local stream using the specified comparator.
218      * Equivalent to executing sorted() on each local stream.
219      * An intermediate operation.
220      * @param comparator Function to compare two values.
221      * @return Distributed Stream with the elements locally sorted.
222      */
223     public DistributedStream<T> localSorted(Comparator<? super T> comparator);
224
225     /**
226      * Collects all local elements into an array.
227      * Equivalent to executing toArray() on each local stream.
228      * A terminal operation.
229      * @return Array containing all local elements.
230      */
231     public Object[] localToArray();
232
233     /**
234      * Collects all local elements into an array.
235      * Equivalent to executing toArray() on each local stream.
236      * A terminal operation.
237      * @param generator Array allocation function.
238      * @return Array containing all local elements.
239      */
240     public <A> A[] localToArray(IntFunction<A[]> generator);
241
242     // Overrides
243
244     /**
245      * Removes duplicate elements in the Distributed Stream.
246      * A stateful eager intermediate operation.
247      * @return Distributed Stream without duplicate data elements.
248      */
249     @Override public DistributedStream<T> distinct();
250
251     /**
252      * Removes elements in the Distributed Stream that do not satisfy the
253      * specified predicate.
254      * An intermediate operation.
255      * @param predicate Boolean function that decides whether an element
256      * should remain in the Distributed Stream.
```

```
257     * @return Distributed Stream where all elements satisfy the predicate.
258     */
259    @Override public DistributedStream<T> filter(
260      Predicate<? super T> predicate);
261
262    /**
263     * Replaces each element of the Distributed Stream with elements in the
264     * stream returned by the specified mapping function.
265     * An intermediate operation.
266     * @param mapper Mapping function returning a stream.
267     * @return Distributed Stream with the applied mapping.
268     */
269    @Override public <R> DistributedStream<R> flatMap(
270      Function<? super T,? extends Stream<? extends R>> mapper);
271
272    /**
273     * Replaces each element of the Distributed Stream with elements in the
274     * stream returned by the specified mapping function.
275     * An intermediate operation.
276     * @param mapper Mapping function returning a stream of doubles.
277     * @return Distributed DoubleStream with the applied mapping.
278     */
279    @Override public DistributedDoubleStream flatMapToDouble(
280      Function<? super T,? extends DoubleStream> mapper);
281
282    /**
283     * Replaces each element of the Distributed Stream with elements in the
284     * stream returned by the specified mapping function.
285     * An intermediate operation.
286     * @param mapper Mapping function returning a stream of integers.
287     * @return Distributed IntStream with the applied mapping.
288     */
289    @Override public DistributedIntStream flatMapToInt(
290      Function<? super T,? extends IntStream> mapper);
291
292    /**
293     * Replaces each element of the Distributed Stream with elements in the
294     * stream returned by the specified mapping function.
295     * An intermediate operation.
296     * @param mapper Mapping function returning a stream of long integers.
297     * @return Distributed LongStream with the applied mapping.
298     */
299    @Override public DistributedLongStream flatMapToLong(
300      Function<? super T,? extends LongStream> mapper);
301
302    /**
303     * Keeps only the specified number of elements in the Distributed Stream.
304     * An intermediate operation.
305     * @param maxSize Maximum number of elements to keep.
306     * @return Distributed Stream truncated to maxSize elements.
307     */
308    @Override public DistributedStream<T> limit(long maxSize);
309
```

```
310   /**
311    * Replaces each element of the Distributed Stream with values from the
312    * specified mapping function.
313    * An intermediate operation.
314    * @param mapper Mapping function returning values of an arbitrary type.
315    * @return Distributed Stream with the applied mapping.
316    */
317   @Override public <R> DistributedStream<R> map(
318     Function<? super T,? extends R> mapper);
319
320   /**
321    * Replaces each element of the Distributed Stream with values from the
322    * specified mapping function.
323    * An intermediate operation.
324    * @param mapper Mapping function returning values of type double.
325    * @return Distributed Stream with the applied mapping.
326    */
327   @Override public DistributedDoubleStream mapToDouble(
328     ToDoubleFunction<? super T> mapper);
329
330   /**
331    * Replaces each element of the Distributed Stream with values from the
332    * specified mapping function.
333    * An intermediate operation.
334    * @param mapper Mapping function returning values of type integer.
335    * @return Distributed Stream with the applied mapping.
336    */
337   @Override public DistributedIntStream mapToInt(
338     ToIntFunction<? super T> mapper);
339
340   /**
341    * Replaces each element of the Distributed Stream with values from the
342    * specified mapping function.
343    * An intermediate operation.
344    * @param mapper Mapping function returning values of type long integer.
345    * @return Distributed Stream with the applied mapping.
346    */
347   @Override public DistributedLongStream mapToLong(
348     ToLongFunction<? super T> mapper);
349
350   /**
351    * Returns a parallel Distributed Stream with an otherwise identical state.
352    * An intermediate operation.
353    * @return A parallel Distributed Stream.
354    */
355   @Override public DistributedStream<T> parallel();
356
357   /**
358    * Performs an action on each element on all nodes and returns the same
359    * Distributed Stream.
360    * An intermediate operation.
361    * @param action A non-interfering action.
362    * @return The same Distributed Stream.
```

```
363       */
364     @Override public DistributedStream<T> peek(Consumer<? super T> action);
365
366     /**
367      * Returns a sequential Distributed Stream with an otherwise identical
368      * state.
369      * An intermediate operation.
370      * @return A sequential Distributed Stream.
371      */
372     @Override public DistributedStream<T> sequential();
373
374     /**
375      * Removes the first n elements in the Distributed Stream.
376      * A stateful eager intermediate operation.
377      * @param n Number of local elements to skip.
378      * @return Distributed Stream with the elements removed.
379      */
380     @Override public DistributedStream<T> skip(long n);
381
382     /**
383      * Sorts all elements in the Distributed Stream.
384      * A stateful eager intermediate operation.
385      * @return Distributed Stream with the elements sorted.
386      */
387     @Override public DistributedStream<T> sorted();
388
389     /**
390      * Sorts all elements in the Distributed Stream using the specified
391      * comparator.
392      * A stateful eager intermediate operation.
393      * @param comparator Function to compare two values.
394      * @return Distributed Stream with the elements locally sorted.
395      */
396     @Override public DistributedStream<T> sorted(Comparator<? super T> comparator);
397 }
```

A.2 DistributedDoubleStream

```
1  package dstream;
2
3  import java.util.*;
4  import java.util.function.*;
5  import java.util.stream.*;
6
7  /**
8   * A distributed extension of Java 8 Streams for double primitive type.
9   * The associated pipeline is replicated on each participating compute node.
10  * @see java.util.stream.DoubleStream
11  */
12 public interface DistributedDoubleStream extends DoubleStream
13 {
14   /**
15    * Returns the current Distributed Stream's compute group.
```

```
16      * This group determines the participating nodes in the computation.
17      * @return Current Distributed Stream's compute group.
18      */
19     public ComputeGroup getComputeGroup();
20
21     /**
22      * Changes the current Distributed Stream's compute group.
23      * @param grp New compute group.
24      */
25     public void setComputeGroup(ComputeGroup grp);
26
27     // Data distribution operations
28     /**
29      * Sends data elements between nodes in the current compute group
30      * according to a hash-based partitioner.
31      * A stateful eager intermediate operation.
32      * Hash function is the Object.hashCode method.
33      * Elements with the same hash value are sent to the same destination.
34      * Suitable for MapReduce-style shuffling of data.
35      * @return DistributedDoubleStream consisting of all elements.
36      */
37     public DistributedDoubleStream distribute();
38
39     /**
40      * Sends data elements between nodes in the current compute group
41      * according to the specified partitioner.
42      * A stateful eager intermediate operation.
43      * @param p Programmer-defined partitioner.
44      * @return DistributedDoubleStream consisting of all elements.
45      */
46     public DistributedDoubleStream distribute(DoublePartitioner p);
47
48     /**
49      * Sends data elements from nodes in the current compute group to
50      * another compute group according to a hash-based partitioner.
51      * A stateful eager intermediate operation.
52      * Suitable for MapReduce-style shuffling of data.
53      * @param grp Destination compute group.
54      * @return DistributedDoubleStream consisting of all elements.
55      */
56     public DistributedDoubleStream distribute(ComputeGroup grp);
57
58     /**
59      * Sends data elements from nodes in the current compute group to
60      * another compute group according to the specified partitioner.
61      * A stateful eager intermediate operation.
62      * @param grp Destination compute group.
63      * @param p Programmer-defined partitioner.
64      * @return DistributedDoubleStream consisting of all elements.
65      */
66     public DistributedDoubleStream distribute(
67       ComputeGroup grp, DoublePartitioner p);
68
```

```
69    /**
70     * Sends data elements from nodes in the current compute group to
71     * the specified node.
72     * A stateful eager intermediate operation.
73     * @param node Destination compute node.
74     * @return DistributedDoubleStream consisting of all elements.
75     */
76    public DistributedDoubleStream distribute(ComputeNode node);
77
78    /**
79     * Sends data elements from nodes in the current compute group to
80     * multiple compute groups according to a hash-based partitioner.
81     * A stateful eager intermediate operation.
82     * @param grps Array of destination compute groups.
83     * @return Array of DistributedDoubleStreams each consisting of all elements.
84     */
85    public DistributedDoubleStream[] distribute(ComputeGroup[] grps);
86
87    /**
88     * Sends data elements from nodes in the current compute group to
89     * multiple compute groups according to the specified partitioner.
90     * A stateful eager intermediate operation.
91     * @param grps Destination compute group.
92     * @param p Programmer-defined partitioner.
93     * @return Array of DistributedDoubleStreams each consisting of all elements.
94     */
95    public DistributedDoubleStream[] distribute(ComputeGroup[] grps,
96      DoublePartitioner p);
97
98    // Local operations
99
100   /**
101    * Returns the mean of all local data elements.
102    * Equivalent to executing average() on each local stream.
103    * A terminal operation.
104    * @return The mean of local data elements, or empty if no elements.
105    */
106   public OptionalDouble localAverage();
107
108   /**
109    * Accumulates local data elements on each node into a container.
110    * Equivalent to executing collect() on each local stream.
111    * A terminal operation.
112    * @param supplier Function returning a new container.
113    * @param accumulator Function to add a new element into a container.
114    * @param combiner Function for combining two containers.
115    * @return The resulting container.
116    */
117   public <R> R localCollect(Supplier<R> supplier,
118     ObjDoubleConsumer<R> accumulator, BiConsumer<R, R> combiner);
119
120   /**
121    * Returns the number of data elements in the local stream.
```

```
122     * Equivalent to executing count() on each local stream.
123     * A terminal operation.
124     * @return The number of local data elements.
125     */
126    public long localCount();
127
128    /**
129     * Removes duplicate elements in the local stream.
130     * Equivalent to executing distinct() on each local stream.
131     * An intermediate operation.
132     * @return DistributedDoubleStream without duplicate local data elements.
133     */
134    public DistributedDoubleStream localDistinct();
135
136    /**
137     * Keeps only the specified number of elements in each local stream.
138     * Equivalent to executing limit() on each local stream.
139     * An intermediate operation.
140     * @param maxSize Maximum number of elements to keep.
141     * @return DistributedDoubleStream with each local stream
142     * truncated to maxSize elements.
143     */
144    public DistributedDoubleStream localLimit(long maxSize);
145
146    /**
147     * Returns the largest encountered value of a local data element.
148     * Equivalent to executing max() on each local stream.
149     * A terminal operation.
150     * @return The largest local data element, or empty if no elements.
151     */
152    public OptionalDouble localMax();
153
154    /**
155     * Returns the smallest encountered value of a local data element.
156     * Equivalent to executing min() on each local stream.
157     * A terminal operation.
158     * @return The smallest local data element, or empty if no elements.
159     */
160    public OptionalDouble localMin();
161
162    /**
163     * Performs an action on each local element and returns the same
164     * DistributedDoubleStream.
165     * Equivalent to executing peek() on each local stream.
166     * An intermediate operation.
167     * @param action A non-interfering action.
168     * @return The same DistributedDoubleStream.
169     */
170    public DistributedDoubleStream localPeek(DoubleConsumer action);
171
172    /**
173     * Accumulates local elements into a single value of the same type.
174     * Equivalent to executing reduce() on each local stream.
```

```
175      * A terminal operation.
176      * @param op Associative accumulating function.
177      * @return Either the reduced value or an empty value if no elements.
178      */
179     public OptionalDouble localReduce(DoubleBinaryOperator op);
180
181     /**
182      * Accumulates local elements into a single value of the same type.
183      * Equivalent to executing reduce() on each local stream.
184      * A terminal operation.
185      * @param identity The accumulating function's identity value.
186      * @param op Associative accumulating function.
187      * @return The reduced value.
188      */
189     public double localReduce(double identity, DoubleBinaryOperator op);
190
191     /**
192      * Removes the first n elements in each local stream.
193      * Equivalent to executing skip() on each local stream.
194      * An intermediate operation.
195      * @param n Number of local elements to skip.
196      * @return DistributedDoubleStream with the elements removed.
197      */
198     public DistributedDoubleStream localSkip(long n);
199
200     /**
201      * Sorts all elements in each local stream.
202      * Equivalent to executing sorted() on each local stream.
203      * An intermediate operation.
204      * @return DistributedDoubleStream with the elements locally sorted.
205      */
206     public DistributedDoubleStream localSorted();
207
208     /**
209      * Returns the sum of all elements in each local stream.
210      * Equivalent to executing sum() on each local stream.
211      * A terminal operation.
212      * @return Sum of local elements.
213      */
214     public double localSum();
215
216     /**
217      * Returns statistics (average, sum, etc.) for each local stream.
218      * Equivalent to executing summaryStatistics() on each local stream.
219      * A terminal operation.
220      * @return DoubleSummaryStatistics containing statistics about each
221      * local stream.
222      */
223     public DoubleSummaryStatistics localSummaryStatistics();
224
225     /**
226      * Collects all local elements into an array.
227      * Equivalent to executing toArray() on each local stream.
```

```
228      * A terminal operation.
229      * @return Array containing all local elements.
230      */
231     public double[] localToArray();
232
233     // Overrides
234
235     /**
236      * Converts each element into a Double object.
237      * An intermediate operation.
238      * @return Distributed Stream of converted elements.
239      */
240     public DistributedStream<Double> boxed();
241
242     /**
243      * Removes duplicate elements in the DistributedDoubleStream.
244      * A stateful eager intermediate operation.
245      * @return DistributedDoubleStream without duplicate data elements.
246      */
247     public DistributedDoubleStream distinct();
248
249     /**
250      * Removes elements in the DistributedDoubleStream that do not satisfy the
251      * specified predicate.
252      * An intermediate operation.
253      * @param predicate Boolean function that decides whether an element
254      * should remain in the Distributed Stream.
255      * @return DistributedDoubleStream where all elements satisfy the predicate.
256      */
257     public DistributedDoubleStream filter(DoublePredicate predicate);
258
259     /**
260      * Replaces each element of the DistributedDoubleStream with elements in the
261      * stream returned by the specified mapping function.
262      * An intermediate operation.
263      * @param mapper Mapping function returning a stream.
264      * @return DistributedDoubleStream with the applied mapping.
265      */
266     public DistributedDoubleStream flatMap(
267       DoubleFunction<? extends DoubleStream> mapper);
268
269     /**
270      * Keeps only the specified number of elements in the DistributedDoubleStream.
271      * An intermediate operation.
272      * @param maxSize Maximum number of elements to keep.
273      * @return DistributedDoubleStream truncated to maxSize elements.
274      */
275     public DistributedDoubleStream limit(long maxSize);
276
277     /**
278      * Replaces each element of the DistributedDoubleStream with values from the
279      * specified mapping function.
280      * An intermediate operation.
```

```
281      * @param mapper Mapping function returning values of type integer.
282      * @return DistributedDoubleStream with the applied mapping.
283      */
284     public DistributedDoubleStream map(DoubleUnaryOperator mapper);
285
286     /**
287      * Replaces each element of the DistributedDoubleStream with values from the
288      * specified mapping function.
289      * An intermediate operation.
290      * @param mapper Mapping function returning values of type long integer.
291      * @return DistributedIntStream with the applied mapping.
292      */
293     public DistributedIntStream mapToInt(DoubleToIntFunction mapper);
294
295     /**
296      * Replaces each element of the DistributedDoubleStream with values from the
297      * specified mapping function.
298      * An intermediate operation.
299      * @param mapper Mapping function returning values of type long integer.
300      * @return DistributedLongStream with the applied mapping.
301      */
302     public DistributedLongStream mapToLong(DoubleToLongFunction mapper);
303
304     /**
305      * Replaces each element of the DistributedDoubleStream with values from the
306      * specified mapping function.
307      * An intermediate operation.
308      * @param mapper Mapping function returning values of an arbitrary type.
309      * @return Distributed Stream with the applied mapping.
310      */
311     public <U> DistributedStream<U> mapToObj(DoubleFunction<? extends U> mapper);
312
313     /**
314      * Returns a parallel DistributedDoubleStream with an otherwise identical
315      * state.
316      * An intermediate operation.
317      * @return A parallel DistributedDoubleStream.
318      */
319     public DistributedDoubleStream parallel();
320
321     /**
322      * Performs an action on each element on all nodes and returns the same
323      * DistributedDoubleStream.
324      * An intermediate operation.
325      * @param action A non-interfering action.
326      * @return The same DistributedDoubleStream.
327      */
328     public DistributedDoubleStream peek(DoubleConsumer action);
329
330     /**
331      * Returns a sequential DistributedDoubleStream with an otherwise identical
332      * state.
333      * An intermediate operation.
```

```
334     * @return A sequential DistributedDoubleStream.
335     */
336    public DistributedDoubleStream sequential();
337
338    /**
339     * Removes the first n elements in the DistributedDoubleStream.
340     * A stateful eager intermediate operation.
341     * @param n Number of local elements to skip.
342     * @return DistributedDoubleStream with the elements removed.
343     */
344    public DistributedDoubleStream skip(long n);
345
346    /**
347     * Sorts all elements in the DistributedDoubleStream.
348     * A stateful eager intermediate operation.
349     * @return DistributedDoubleStream with the elements sorted.
350     */
351    public DistributedDoubleStream sorted();
352  }
```

A.3 DistributedIntStream

```
1   package dstream;
2
3   import java.util.*;
4   import java.util.function.*;
5   import java.util.stream.*;
6
7   /**
8    * A distributed extension of Java 8 Streams for int primitive type.
9    * The associated pipeline is replicated on each participating compute node.
10   * @see java.util.stream.IntStream
11   */
12  public interface DistributedIntStream extends IntStream
13  {
14    /**
15     * Returns the current Distributed Stream's compute group.
16     * This group determines the participating nodes in the computation.
17     * @return Current Distributed Stream's compute group.
18     */
19    public ComputeGroup getComputeGroup();
20
21    /**
22     * Changes the current Distributed Stream's compute group.
23     * @param grp New compute group.
24     */
25    public void setComputeGroup(ComputeGroup grp);
26
27    // Data distribution operations
28
29    /**
30     * Sends data elements between nodes in the current compute group
31     * according to a hash-based partitioner.
```

```
32      * A stateful eager intermediate operation.
33      * Hash function is the Object.hashCode method.
34      * Elements with the same hash value are sent to the same destination.
35      * Suitable for MapReduce-style shuffling of data.
36      * @return DistributedIntStream consisting of all elements.
37      */
38     public DistributedIntStream distribute();
39
40     /**
41      * Sends data elements between nodes in the current compute group
42      * according to the specified partitioner.
43      * A stateful eager intermediate operation.
44      * @param p Programmer-defined partitioner.
45      * @return DistributedIntStream consisting of all elements.
46      */
47     public DistributedIntStream distribute(IntPartitioner p);
48
49     /**
50      * Sends data elements from nodes in the current compute group to
51      * another compute group according to a hash-based partitioner.
52      * A stateful eager intermediate operation.
53      * Suitable for MapReduce-style shuffling of data.
54      * @param grp Destination compute group.
55      * @return DistributedIntStream consisting of all elements.
56      */
57     public DistributedIntStream distribute(ComputeGroup grp);
58
59     /**
60      * Sends data elements from nodes in the current compute group to
61      * another compute group according to the specified partitioner.
62      * A stateful eager intermediate operation.
63      * @param grp Destination compute group.
64      * @param p Programmer-defined partitioner.
65      * @return DistributedIntStream consisting of all elements.
66      */
67     public DistributedIntStream distribute(ComputeGroup grp, IntPartitioner p);
68
69     /**
70      * Sends data elements from nodes in the current compute group to
71      * the specified node.
72      * A stateful eager intermediate operation.
73      * @param node Destination compute node.
74      * @return DistributedIntStream consisting of all elements.
75      */
76     public DistributedIntStream distribute(ComputeNode node);
77
78     /**
79      * Sends data elements from nodes in the current compute group to
80      * multiple compute groups according to a hash-based partitioner.
81      * A stateful eager intermediate operation.
82      * @param grps Array of destination compute groups.
83      * @return Array of DistributedIntStreams each consisting of all elements.
84      */
```

```
85    public DistributedIntStream[] distribute(ComputeGroup[] grps);
86
87    /**
88     * Sends data elements from nodes in the current compute group to
89     * multiple compute groups according to the specified partitioner.
90     * A stateful eager intermediate operation.
91     * @param grps Destination compute group.
92     * @param p Programmer-defined partitioner.
93     * @return Array of DistributedIntStreams each consisting of all elements.
94     */
95    public DistributedIntStream[] distribute(
96      ComputeGroup[] grps, IntPartitioner p);
97
98    // Local operations
99
100   /**
101    * Returns the mean of all local data elements.
102    * Equivalent to executing average() on each local stream.
103    * A terminal operation.
104    * @return The mean of local data elements, or empty if no elements.
105    */
106   public OptionalDouble localAverage();
107
108   /**
109    * Accumulates local data elements on each node into a container.
110    * Equivalent to executing collect() on each local stream.
111    * A terminal operation.
112    * @param supplier Function returning a new container.
113    * @param accumulator Function to add a new element into a container.
114    * @param combiner Function for combining two containers.
115    * @return The resulting container.
116    */
117   public <R> R localCollect(Supplier<R> supplier,
118     ObjIntConsumer<R> accumulator, BiConsumer<R, R> combiner);
119
120   /**
121    * Returns the number of data elements in the local stream.
122    * Equivalent to executing count() on each local stream.
123    * A terminal operation.
124    * @return The number of local data elements.
125    */
126   public long localCount();
127
128   /**
129    * Removes duplicate elements in the local stream.
130    * Equivalent to executing distinct() on each local stream.
131    * An intermediate operation.
132    * @return DistributedIntStream without duplicate local data elements.
133    */
134   public DistributedIntStream localDistinct();
135
136   /**
137    * Keeps only the specified number of elements in each local stream.
```

```
138     * Equivalent to executing limit() on each local stream.
139     * An intermediate operation.
140     * @param maxSize Maximum number of elements to keep.
141     * @return DistributedIntStream with each local stream
142     * truncated to maxSize elements.
143     */
144    public DistributedIntStream localLimit(long maxSize);
145
146    /**
147     * Returns the largest encountered value of a local data element.
148     * Equivalent to executing max() on each local stream.
149     * A terminal operation.
150     * @return The largest local data element, or empty if no elements.
151     */
152    public OptionalInt localMax();
153
154    /**
155     * Returns the smallest encountered value of a local data element.
156     * Equivalent to executing min() on each local stream.
157     * A terminal operation.
158     * @return The smallest local data element, or empty if no elements.
159     */
160    public OptionalInt localMin();
161
162    /**
163     * Performs an action on each local element and returns the same
164     * DistributedIntStream.
165     * Equivalent to executing peek() on each local stream.
166     * An intermediate operation.
167     * @param action A non-interfering action.
168     * @return The same DistributedIntStream.
169     */
170    public DistributedIntStream localPeek(IntConsumer action);
171
172    /**
173     * Accumulates local elements into a single value of the same type.
174     * Equivalent to executing reduce() on each local stream.
175     * A terminal operation.
176     * @param op Associative accumulating function.
177     * @return Either the reduced value or an empty value if no elements.
178     */
179    public OptionalInt localReduce(IntBinaryOperator op);
180
181    /**
182     * Accumulates local elements into a single value of the same type.
183     * Equivalent to executing reduce() on each local stream.
184     * A terminal operation.
185     * @param identity The accumulating function's identity value.
186     * @param op Associative accumulating function.
187     * @return The reduced value.
188     */
189    public int localReduce(int identity, IntBinaryOperator op);
190
```

```
191     /**
192      * Removes the first n elements in each local stream.
193      * Equivalent to executing skip() on each local stream.
194      * An intermediate operation.
195      * @param n Number of local elements to skip.
196      * @return DistributedIntStream with the elements removed.
197      */
198     public DistributedIntStream localSkip(long n);
199
200     /**
201      * Sorts all elements in each local stream.
202      * Equivalent to executing sorted() on each local stream.
203      * An intermediate operation.
204      * @return DistributedIntStream with the elements locally sorted.
205      */
206     public DistributedIntStream localSorted();
207
208     /**
209      * Returns the sum of all elements in each local stream.
210      * Equivalent to executing sum() on each local stream.
211      * A terminal operation.
212      * @return Sum of local elements.
213      */
214     public int localSum();
215
216     /**
217      * Returns statistics (average, sum, etc.) for each local stream.
218      * Equivalent to executing summaryStatistics() on each local stream.
219      * A terminal operation.
220      * @return IntSummaryStatistics containing statistics about each
221      * local stream.
222      */
223     public IntSummaryStatistics localSummaryStatistics();
224
225     /**
226      * Collects all local elements into an array.
227      * Equivalent to executing toArray() on each local stream.
228      * A terminal operation.
229      * @return Array containing all local elements.
230      */
231     public int[] localToArray();
232
233     // Overrides
234
235     /**
236      * Converts each element into a double.
237      * An intermediate operation.
238      * @return DistributedDoubleStream of converted elements.
239      */
240     public DistributedDoubleStream asDoubleStream();
241
242     /**
243      * Converts each element into a long integer.
```

```
244      * An intermediate operation.
245      * @return DistributedLongStream of converted elements.
246      */
247     public DistributedLongStream asLongStream();
248
249     /**
250      * Converts each element into an Integer object.
251      * An intermediate operation.
252      * @return Distributed Stream of converted elements.
253      */
254     public DistributedStream<Integer> boxed();
255
256     /**
257      * Removes duplicate elements in the DistributedIntStream.
258      * A stateful eager intermediate operation.
259      * @return DistributedIntStream without duplicate data elements.
260      */
261     public DistributedIntStream distinct();
262
263     /**
264      * Removes elements in the DistributedIntStream that do not satisfy the
265      * specified predicate.
266      * An intermediate operation.
267      * @param predicate Boolean function that decides whether an element
268      * should remain in the Distributed Stream.
269      * @return DistributedIntStream where all elements satisfy the predicate.
270      */
271     public DistributedIntStream filter(IntPredicate predicate);
272
273     /**
274      * Replaces each element of the DistributedIntStream with elements in the
275      * stream returned by the specified mapping function.
276      * An intermediate operation.
277      * @param mapper Mapping function returning a stream.
278      * @return DistributedIntStream with the applied mapping.
279      */
280     public DistributedIntStream flatMap(IntFunction<? extends IntStream> mapper);
281
282     /**
283      * Keeps only the specified number of elements in the DistributedIntStream.
284      * An intermediate operation.
285      * @param maxSize Maximum number of elements to keep.
286      * @return DistributedIntStream truncated to maxSize elements.
287      */
288     public DistributedIntStream limit(long maxSize);
289
290     /**
291      * Replaces each element of the DistributedIntStream with values from the
292      * specified mapping function.
293      * An intermediate operation.
294      * @param mapper Mapping function returning values of type integer.
295      * @return DistributedIntStream with the applied mapping.
296      */
```

```
297     public DistributedIntStream map(IntUnaryOperator mapper);
298
299     /**
300      * Replaces each element of the DistributedIntStream with values from the
301      * specified mapping function.
302      * An intermediate operation.
303      * @param mapper Mapping function returning values of type double.
304      * @return DistributedDoubleStream with the applied mapping.
305      */
306     public DistributedDoubleStream mapToDouble(IntToDoubleFunction mapper);
307
308     /**
309      * Replaces each element of the DistributedIntStream with values from the
310      * specified mapping function.
311      * An intermediate operation.
312      * @param mapper Mapping function returning values of type long integer.
313      * @return DistributedLongStream with the applied mapping.
314      */
315     public DistributedLongStream mapToLong(IntToLongFunction mapper);
316
317     /**
318      * Replaces each element of the DistributedIntStream with values from the
319      * specified mapping function.
320      * An intermediate operation.
321      * @param mapper Mapping function returning values of an arbitrary type.
322      * @return Distributed Stream with the applied mapping.
323      */
324     public <U> DistributedStream<U> mapToObj(IntFunction<? extends U> mapper);
325
326     /**
327      * Returns a parallel DistributedIntStream with an otherwise identical
328      * state.
329      * An intermediate operation.
330      * @return A parallel DistributedIntStream.
331      */
332     public DistributedIntStream parallel();
333
334     /**
335      * Performs an action on each element on all nodes and returns the same
336      * DistributedIntStream.
337      * An intermediate operation.
338      * @param action A non-interfering action.
339      * @return The same DistributedIntStream.
340      */
341     public DistributedIntStream peek(IntConsumer action);
342
343     /**
344      * Returns a sequential DistributedIntStream with an otherwise identical
345      * state.
346      * An intermediate operation.
347      * @return A sequential DistributedIntStream.
348      */
349     public DistributedIntStream sequential();
```

```
350
351     /**
352      * Removes the first n elements in the DistributedIntStream.
353      * A stateful eager intermediate operation.
354      * @param n Number of local elements to skip.
355      * @return DistributedIntStream with the elements removed.
356      */
357     public DistributedIntStream skip(long n);
358
359     /**
360      * Sorts all elements in the DistributedIntStream.
361      * A stateful eager intermediate operation.
362      * @return DistributedIntStream with the elements sorted.
363      */
364     public DistributedIntStream sorted();
365 }
```

A.4 DistributedLongStream

```
1  package dstream;
2
3  import java.util.*;
4  import java.util.function.*;
5  import java.util.stream.*;
6
7  /**
8   * A distributed extension of Java 8 Streams for long primitive type.
9   * The associated pipeline is replicated on each participating compute node.
10  * @see java.util.stream.LongStream
11  */
12 public interface DistributedLongStream extends LongStream
13 {
14     /**
15      * Returns the current Distributed Stream's compute group.
16      * This group determines the participating nodes in the computation.
17      * @return Current Distributed Stream's compute group.
18      */
19     public ComputeGroup getComputeGroup();
20
21     /**
22      * Changes the current Distributed Stream's compute group.
23      * @param grp New compute group.
24      */
25     public void setComputeGroup(ComputeGroup grp);
26
27     // Data distribution operations
28
29     /**
30      * Sends data elements between nodes in the current compute group
31      * according to a hash-based partitioner.
32      * A stateful eager intermediate operation.
33      * Hash function is the Object.hashCode method.
34      * Elements with the same hash value are sent to the same destination.
```

```
35      * Suitable for MapReduce-style shuffling of data.
36      * @return DistributedLongStream consisting of all elements.
37      */
38     public DistributedLongStream distribute();
39
40     /**
41      * Sends data elements between nodes in the current compute group
42      * according to the specified partitioner.
43      * A stateful eager intermediate operation.
44      * @param p Programmer-defined partitioner.
45      * @return DistributedLongStream consisting of all elements.
46      */
47     public DistributedLongStream distribute(LongPartitioner p);
48
49     /**
50      * Sends data elements from nodes in the current compute group to
51      * another compute group according to a hash-based partitioner.
52      * A stateful eager intermediate operation.
53      * Suitable for MapReduce-style shuffling of data.
54      * @param grp Destination compute group.
55      * @return DistributedLongStream consisting of all elements.
56      */
57     public DistributedLongStream distribute(ComputeGroup grp);
58
59     /**
60      * Sends data elements from nodes in the current compute group to
61      * another compute group according to the specified partitioner.
62      * A stateful eager intermediate operation.
63      * @param grp Destination compute group.
64      * @param p Programmer-defined partitioner.
65      * @return DistributedLongStream consisting of all elements.
66      */
67     public DistributedLongStream distribute(
68       ComputeGroup grp, LongPartitioner p);
69
70     /**
71      * Sends data elements from nodes in the current compute group to
72      * the specified node.
73      * A stateful eager intermediate operation.
74      * @param node Destination compute node.
75      * @return DistributedLongStream consisting of all elements.
76      */
77     public DistributedLongStream distribute(ComputeNode node);
78
79     /**
80      * Sends data elements from nodes in the current compute group to
81      * multiple compute groups according to a hash-based partitioner.
82      * A stateful eager intermediate operation.
83      * @param grps Array of destination compute groups.
84      * @return Array of DistributedLongStreams each consisting of all elements.
85      */
86     public DistributedLongStream[] distribute(ComputeGroup[] grps);
87
```

```
88    /**
89     * Sends data elements from nodes in the current compute group to
90     * multiple compute groups according to the specified partitioner.
91     * A stateful eager intermediate operation.
92     * @param grps Destination compute group.
93     * @param p Programmer-defined partitioner.
94     * @return Array of DistributedLongStreams each consisting of all elements.
95     */
96    public DistributedLongStream[] distribute(
97      ComputeGroup[] grps, LongPartitioner p);
98
99    // Local operations
100
101   /**
102    * Returns the mean of all local data elements.
103    * Equivalent to executing average() on each local stream.
104    * A terminal operation.
105    * @return The mean of local data elements, or empty if no elements.
106    */
107   public OptionalDouble localAverage();
108
109   /**
110    * Accumulates local data elements on each node into a container.
111    * Equivalent to executing collect() on each local stream.
112    * A terminal operation.
113    * @param supplier Function returning a new container.
114    * @param accumulator Function to add a new element into a container.
115    * @param combiner Function for combining two containers.
116    * @return The resulting container.
117    */
118   public <R> R localCollect(Supplier<R> supplier,
119     ObjLongConsumer<R> accumulator, BiConsumer<R, R> combiner);
120
121   /**
122    * Returns the number of data elements in the local stream.
123    * Equivalent to executing count() on each local stream.
124    * A terminal operation.
125    * @return The number of local data elements.
126    */
127   public long localCount();
128
129   /**
130    * Removes duplicate elements in the local stream.
131    * Equivalent to executing distinct() on each local stream.
132    * An intermediate operation.
133    * @return DistributedLongStream without duplicate local data elements.
134    */
135   public DistributedLongStream localDistinct();
136
137   /**
138    * Keeps only the specified number of elements in each local stream.
139    * Equivalent to executing limit() on each local stream.
140    * An intermediate operation.
```

```
141      * @param maxSize Maximum number of elements to keep.
142      * @return DistributedLongStream with each local stream
143      * truncated to maxSize elements.
144      */
145     public DistributedLongStream localLimit(long maxSize);
146
147     /**
148      * Returns the largest encountered value of a local data element.
149      * Equivalent to executing max() on each local stream.
150      * A terminal operation.
151      * @return The largest local data element, or empty if no elements.
152      */
153     public OptionalLong localMax();
154
155     /**
156      * Returns the smallest encountered value of a local data element.
157      * Equivalent to executing min() on each local stream.
158      * A terminal operation.
159      * @return The smallest local data element, or empty if no elements.
160      */
161     public OptionalLong localMin();
162
163     /**
164      * Performs an action on each local element and returns the same
165      * DistributedLongStream.
166      * Equivalent to executing peek() on each local stream.
167      * An intermediate operation.
168      * @param action A non-interfering action.
169      * @return The same DistributedLongStream.
170      */
171     public DistributedLongStream localPeek(LongConsumer action);
172
173     /**
174      * Accumulates local elements into a single value of the same type.
175      * Equivalent to executing reduce() on each local stream.
176      * A terminal operation.
177      * @param op Associative accumulating function.
178      * @return Either the reduced value or an empty value if no elements.
179      */
180     public OptionalLong localReduce(LongBinaryOperator op);
181
182     /**
183      * Accumulates local elements into a single value of the same type.
184      * Equivalent to executing reduce() on each local stream.
185      * A terminal operation.
186      * @param identity The accumulating function's identity value.
187      * @param op Associative accumulating function.
188      * @return The reduced value.
189      */
190     public long localReduce(long identity, LongBinaryOperator op);
191
192     /**
193      * Removes the first n elements in each local stream.
```

```
194      * Equivalent to executing skip() on each local stream.
195      * An intermediate operation.
196      * @param n Number of local elements to skip.
197      * @return DistributedLongStream with the elements removed.
198      */
199     public DistributedLongStream localSkip(long n);
200
201     /**
202      * Sorts all elements in each local stream.
203      * Equivalent to executing sorted() on each local stream.
204      * An intermediate operation.
205      * @return DistributedLongStream with the elements locally sorted.
206      */
207     public DistributedLongStream localSorted();
208
209     /**
210      * Returns the sum of all elements in each local stream.
211      * Equivalent to executing sum() on each local stream.
212      * A terminal operation.
213      * @return Sum of local elements.
214      */
215     public long localSum();
216
217     /**
218      * Returns statistics (average, sum, etc.) for each local stream.
219      * Equivalent to executing summaryStatistics() on each local stream.
220      * A terminal operation.
221      * @return LongSummaryStatistics containing statistics about each
222      * local stream.
223      */
224     public LongSummaryStatistics localSummaryStatistics();
225
226     /**
227      * Collects all local elements into an array.
228      * Equivalent to executing toArray() on each local stream.
229      * A terminal operation.
230      * @return Array containing all local elements.
231      */
232     public long[] localToArray();
233
234     // Overrides
235
236     /**
237      * Converts each element into a double.
238      * An intermediate operation.
239      * @return DistributedDoubleStream of converted elements.
240      */
241     public DistributedDoubleStream asDoubleStream();
242
243     /**
244      * Converts each element into a Long object.
245      * An intermediate operation.
246      * @return Distributed Stream of converted elements.
```

```
247       */
248     public DistributedStream<Long> boxed();
249
250     /**
251       * Removes duplicate elements in the DistributedIntStream.
252       * A stateful eager intermediate operation.
253       * @return DistributedLongStream without duplicate data elements.
254       */
255     public DistributedLongStream distinct();
256
257     /**
258       * Removes elements in the DistributedLongStream that do not satisfy the
259       * specified predicate.
260       * An intermediate operation.
261       * @param predicate Boolean function that decides whether an element
262       * should remain in the Distributed Stream.
263       * @return DistributedLongStream where all elements satisfy the predicate.
264       */
265     public DistributedLongStream filter(LongPredicate predicate);
266
267     /**
268       * Replaces each element of the DistributedLongStream with elements in the
269       * stream returned by the specified mapping function.
270       * An intermediate operation.
271       * @param mapper Mapping function returning a stream.
272       * @return DistributedLongStream with the applied mapping.
273       */
274     public DistributedLongStream flatMap(
275       LongFunction<? extends LongStream> mapper);
276
277     /**
278       * Keeps only the specified number of elements in the DistributedLongStream.
279       * An intermediate operation.
280       * @param maxSize Maximum number of elements to keep.
281       * @return DistributedLongStream truncated to maxSize elements.
282       */
283     public DistributedLongStream limit(long maxSize);
284
285     /**
286       * Replaces each element of the DistributedLongStream with values from the
287       * specified mapping function.
288       * An intermediate operation.
289       * @param mapper Mapping function returning values of type long.
290       * @return DistributedLongStream with the applied mapping.
291       */
292     public DistributedLongStream map(LongUnaryOperator mapper);
293
294     /**
295       * Replaces each element of the DistributedLongStream with values from the
296       * specified mapping function.
297       * An intermediate operation.
298       * @param mapper Mapping function returning values of type double.
299       * @return DistributedDoubleStream with the applied mapping.
```

```
300      */
301     public DistributedDoubleStream mapToDouble(LongToDoubleFunction mapper);
302
303     /**
304      * Replaces each element of the DistributedLongStream with values from the
305      * specified mapping function.
306      * An intermediate operation.
307      * @param mapper Mapping function returning values of type long integer.
308      * @return DistributedIntStream with the applied mapping.
309      */
310     public DistributedIntStream mapToInt(LongToIntFunction mapper);
311
312     /**
313      * Replaces each element of the DistributedLongStream with values from the
314      * specified mapping function.
315      * An intermediate operation.
316      * @param mapper Mapping function returning values of an arbitrary type.
317      * @return Distributed Stream with the applied mapping.
318      */
319     public <U> DistributedStream<U> mapToObj(LongFunction<? extends U> mapper);
320
321     /**
322      * Returns a parallel DistributedLongStream with an otherwise identical
323      * state.
324      * An intermediate operation.
325      * @return A parallel DistributedLongStream.
326      */
327     public DistributedLongStream parallel();
328
329     /**
330      * Performs an action on each element on all nodes and returns the same
331      * DistributedLongStream.
332      * An intermediate operation.
333      * @param action A non-interfering action.
334      * @return The same DistributedLongStream.
335      */
336     public DistributedLongStream peek(LongConsumer action);
337
338     /**
339      * Returns a sequential DistributedLongStream with an otherwise identical
340      * state.
341      * An intermediate operation.
342      * @return A sequential DistributedLongStream.
343      */
344     public DistributedLongStream sequential();
345
346     /**
347      * Removes the first n elements in the DistributedLongStream.
348      * A stateful eager intermediate operation.
349      * @param n Number of local elements to skip.
350      * @return DistributedLongStream with the elements removed.
351      */
352     public DistributedLongStream skip(long n);
```

```
353
354     /**
355      * Sorts all elements in the DistributedLongStream.
356      * A stateful eager intermediate operation.
357      * @return DistributedLongStream with the elements sorted.
358      */
359     public DistributedLongStream sorted();
360 }
```

A Distributed Collection API

```
1  package util;
2
3  import dstream.*;
4  import java.util.*;
5
6  /**
7   * Represents a set of collections on participating compute nodes containing
8   * data that is part of a distributed dataset.
9   */
10 public interface DistributedCollection<E> extends Collection<E>
11 {
12   /**
13    * Returns the group of nodes on which the Distributed Collection's
14    * data resides.
15    * This group is used as the initial compute group for Distributed
16    * Streams backed by the Distributed Collection
17    * @return Distributed Collection's compute group.
18    */
19   public ComputeGroup getComputeGroup();
20
21   /**
22    * Returns a sequential Distributed Stream backed by this
23    * Distributed Collection.
24    * @return A new Distributed Stream.
25    */
26   @Override
27   public default DistributedStream<E> stream()
28   {
29     DistributedStream<E> s = DistributedStreamSupport.stream(
30       spliterator(), false);
31     s.setComputeGroup(getComputeGroup());
32     return s;
33   }
34
35   /**
36    * Returns a parallel Distributed Stream backed by this
37    * Distributed Collection.
38    * @return A new Distributed Stream.
39    */
40   @Override
41   public default DistributedStream<E> parallelStream()
42   {
43     DistributedStream<E> s = DistributedStreamSupport.stream(
44       spliterator(), true);
45     s.setComputeGroup(getComputeGroup());
46     return s;
47   }
48 }
```

A Partitioner API

A.1 Partitioner

```
1  package dstream;
2
3  /**
4   * Function for partitioning data over a predefined compute group.
5   */
6  @FunctionalInterface
7  public interface Partitioner<T>
8  {
9    /**
10    * Decides which compute node the data element should be sent to.
11    * @param data Data element for consideration.
12    * @return Integer which will be converted into an index of the
13    * compute node in the compute group. Indexes that are out of bounds
14    * will be wrapped around.
15    */
16   public int partition(T data);
17 }
```

A.2 DoublePartitioner

```
1  package dstream;
2
3  /**
4   * Function for partitioning data over a predefined compute group.
5   */
6  @FunctionalInterface
7  public interface DoublePartitioner
8  {
9    /**
10    * Decides which compute node the data element should be sent to.
11    * @param data Data element for consideration.
12    * @return Integer which will be converted into an index of the
13    * compute node in the compute group. Indexes that are out of bounds
14    * will be wrapped around.
15    */
16   public int partition(double data);
17 }
```

A.3 IntPartitioner

```
1  package dstream;
2
3  /**
4   * Function for partitioning data over a predefined compute group.
5   */
6  @FunctionalInterface
7  public interface IntPartitioner
```

```
8  {
9    /**
10    * Decides which compute node the data element should be sent to.
11    * @param data Data element for consideration.
12    * @return Integer which will be converted into an index of the
13    * compute node in the compute group. Indexes that are out of bounds
14    * will be wrapped around.
15    */
16   public int partition(int data);
17 }
```

A.4 LongPartitioner

```
1  package dstream;
2
3  /**
4   * Function for partitioning data over a predefined compute group.
5   */
6  @FunctionalInterface
7  public interface LongPartitioner
8  {
9    /**
10    * Decides which compute node the data element should be sent to.
11    * @param data Data element for consideration.
12    * @return Integer which will be converted into an index of the
13    * compute node in the compute group. Indexes that are out of bounds
14    * will be wrapped around.
15    */
16   public int partition(long data);
17 }
```

A Compute node and group API

A.1 `ComputeNode`

```
1  package dstream;
2
3  /**
4   * Represents a compute node in the cluster.
5   * Compute nodes can be grouped together with compute groups.
6   * @see dstream.ComputeGroup
7   */
8  public class ComputeNode
9  {
10   /**
11    * Program entry point on every node.
12    * MPI initialisation and finalisation is done here.
13    * @param argv Program arguments. argv[0] is the actual class to run which
14    * must have a static main() method.
15    */
16   public static void main(String[] argv);
17
18   /**
19    * Returns the compute node's name.
20    * @return String containing the node's name.
21    */
22   public String getName()
23   {
24     return name;
25   }
26
27   /**
28    * Tells whether we are executing on this node.
29    * @return True if we are executing on this node, false otherwise.
30    */
31   public boolean isSelf()
32   {
33     return this == thisNode;
34   }
35
36   /**
37    * Returns the node we are executing on.
38    * @return The currently executing node.
39    */
40   public static ComputeNode getSelf()
41   {
42     return thisNode;
43   }
44
45   /**
46    * Returns the node whose name matches the specified string.
47    * @param name Name to search for.
48    * @return Node with the matching name.
```

```
49      */
50    public static ComputeNode findByName(String name)
51    {
52      return thisNode.nodes.get(name);
53    }
54  }
```

A.2 ComputeGroup

```
1   package dstream;
2
3   /**
4    * Represents a group of compute nodes.
5    * @see dstream.ComputeNode
6    */
7   public class ComputeGroup extends ArrayList<ComputeNode> // A group of compute nodes
8   {
9     /**
10     * Constructor that creates an empty compute group.
11     */
12    public ComputeGroup()
13    {
14      super();
15      ...
16    }
17
18    /**
19     * Constructor that creates a compute group from a collection of
20     * compute nodes.
21     * @param nodes Collection of compute nodes. Can also be an existing
22     * compute group since it is a collection of compute nodes.
23     */
24    public ComputeGroup(Collection<ComputeNode> nodes)
25    {
26      this();
27      addAll(nodes);
28    }
29
30    /**
31     * Constructor that creates a compute group consisting of a single
32     * compute node.
33     * @param node Compute node that will be part of the new group.
34     */
35    public ComputeGroup(ComputeNode node)
36    {
37      this();
38      add(node);
39    }
40
41    /**
42     * Returns a compute group consisting of all nodes in the cluster.
43     * @return New compute group with all nodes in cluster.
44     */
```

```
45    public static ComputeGroup getCluster()
46    {
47      return new ComputeGroup(cluster);
48    }
49  }
```

References

[1] Apache Software Foundation. Apache Hadoop. http://hadoop.apache.org/, accessed 2013/09/01.

[2] Apache Software Foundation. Apache Spark – Lightning-Fast Cluster Computing. http://spark.incubator.apache.org/, accessed 2013/10/03.

[3] Apache Software Foundation. Spark Examples. http://spark.apache.org/examples.html, accessed 2014/10/22.

[4] Bryan Carpenter, Aamir Shafi, and Mark Baker. MPJ Express Project. http://www.mpjexpress.org/, accessed 2014/11/16.

[5] Yu Chan, Ian Gray, and Andy Wellings. Exploiting Multicore Architectures in Big Data Applications: The JUNIPER Approach. In *Proceedings of 7th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2014)*, January 2014.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[7] Adam Jacobs. The pathologies of big data. *Queue*, 7(6):10:10–10:19, July 2009.

[8] Nathan Marz. Storm – Distributed and fault-tolerant realtime computation. http://storm-project.net/, accessed 2013/10/03.

[9] Oracle Corporation. AbstractTask.java. http://hg.openjdk.java.net/jdk8/tl/jdk/file/tip/src/share/classes/java/util/stream/AbstractTask.java, accessed 2014/05/05.

[10] Oracle Corporation. Stream (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html, accessed 2014/11/17.

[11] Marcel Proust. Remembrance of Things Past. http://alarecherchedutempsperdu.com/text.html, accessed 2013/09/04.

[12] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.