# Changing Engines in Midstream: A Java Stream Computational Model for Big Data Processing

Xueyuan Su, Garret Swart, Brian Goetz, Brian Oliver, Paul Sandoz
Oracle Corporation
{First.Last}@oracle.com

## ABSTRACT

With the addition of lambda expressions and the *Stream* API in Java 8, Java has gained a powerful and expressive query language that operates over in-memory collections of Java objects, making the transformation and analysis of data more convenient, scalable and efficient. In this paper, we build on Java 8 Stream and add a *DistributableStream* abstraction that supports federated query execution over an extensible set of distributed compute engines. Each query eventually results in the creation of a materialized result that is returned either as a local object or as an engine defined distributed Java Collection that can be saved and/or used as a source for future queries. Distinctively, DistributableStream supports the changing of compute engines both between and within a query, allowing different parts of a computation to be executed on different platforms. At execution time, the query is organized as a sequence of pipelined stages, each stage potentially running on a different engine. Each node that is part of a stage executes its portion of the computation on the data available locally or produced by the previous stage of the computation. This approach allows for computations to be assigned to engines based on pricing, data locality, and resource availability. Coupled with the inherent laziness of stream operations, this brings great flexibility to query planning and separates the semantics of the query from the details of the engine used to execute it. We currently support three engines, Local, Apache Hadoop MapReduce and Oracle Coherence, and we illustrate how new engines and data sources can be added.

## 1. INTRODUCTION

In this paper, we introduce *DistributableStream*, a Java API that enables programmers to write distributed and federated queries on top of a set of pluggable compute engines. Queries are expressed in a concise and easy to understand way as illustrated in the WordCount example shown in Program 1. The framework supports engines that are disk or memory based, local or distributed, pessimistic or optimistic fault handling. Distinctively it can also federate multi-stage queries over multiple engines to allow for data access to be localized, or resource utilization to be optimized. Since DistributableStream is a pure Java library with an efficient local implementation, it can also scale down for processing small amounts of data within a JVM.

**Program 1** WordCount

```
public static Map<String, Integer> wordCount(
  DistributableStream<String> stream) {
    return stream
      .flatMap(s -> Stream.of(s.split("\\s+")))
      .collect(DistributableCollectors
        .toMap(s -> s, s -> 1, Integer::sum)); }
```

The contributions of our work include:

- A DistributableStream interface for processing Big Data across various platforms. The abstraction frees developers from low-level platform-dependent details and allows them to focus on the algorithmic part of their design. At execution time, stream applications are translated into query plans to be executed on the configured compute engines.
- A pluggable Engine abstraction. This design separates engine-specific configuration from the stream computational model and the modular design makes adding new JVM-enabled engines straightforward, supporting negotiable push or pull-based data movement across engine boundaries. Applications developed with the DistributableStream API can be reused on different engines without rewriting any algorithmic logic.
- Three engine implementations – two distributed engines, Apache Hadoop and Oracle Coherence, and a local engine, layered on the Java 8 Stream thread pool where local computations are executed in parallel on multiple cores – serve to validate and commercialize this design. Our implementation approach allows for both high-level query planner and low-level compiler optimizations. On the high level, we use explicit data structures to describe the processing steps and allow an execution planner to manipulate these data structures to optimize the movement of data and state between the engines. On the low level, each portion of a DistributableStream is assembled as a local Java 8 Stream which arranges its code to make Java Just-in-Time based compilation optimizations applicable.


*Proceedings of the VLDB Endowment,* Vol. 7, No. 13

- Implementations of Java 8 Streams for various external data sources, including *InputFormatStream* for accessing HDFS and other Hadoop data sources and *NamedCacheStream* for accessing Oracle Coherence; and Java 8 Stream Collector implementations, such as *OutputFormatCollector* and *NamedCacheCollector*, for storing the results of stream computations into Apache Hadoop and Oracle Coherence.

- Federated query model: Data streams connect fusible operators within a single engine pipeline and operators running in parallel on different engines, forming a clean and easy to understand federated query model. The ability to exploit multiple engines within a single query allows us to most efficiently exploit disk arrays, distributed memory, and local memory for performing bulk ETL, iterative jobs, and interactive analytics, by applying the appropriate compute engines for each processing stage. For example, combining Hadoop with a distributed memory engine (Coherence or Spark) and an SMP, we can read input data from HDFS, cache and efficiently process it in memory, and perform the final reduction on a single machine.

- Applications are the proof of any programming system and WordCount is not enough. For that reason we show this interface at work for distributed reservoir sampling, PageRank, and k-means clustering. These examples demonstrate the expressiveness and conciseness of the DistributableStream programming model. We also run these examples on different combinations of engines and compare their performance.

A variety of features are incomplete as of this writing, including a global job optimizer, job progress monitoring, and automatic cleanup of temporary files and distributed objects dropped by client applications. In addition we have plans to support other engines and data sources, such as Spark, Tez, and the Oracle Database, high on our list.

Our discussion starts with an overview of Java 8 Stream (§2) and DistributableStream(§3). We then present detailed discussions on the design and implementation of DistributableStream (§4), example stream applications (§5), and experimental evaluation (§6). Finally, we explore work that are promising for future development (§7), survey related work (§8), and conclude the paper (§9).

# 2. STREAM IN JAVA 8

Java SE 8 [12] is the latest version of the Java platform and includes language-level support for lambda expressions and the Stream API that provides efficient bulk operations on Java Collections and other data sources.

## 2.1 Prerequisite Features for Supporting Stream

### 2.1.1 Lambda Expressions

A Java lambda expression is an anonymous method consisting of an argument list with zero or more formal parameters with optionally inferred types, the arrow token, and a body consisting of a single expression or a statement block. `(a,b) -> a+b` is a simple example that computes the sum of two arguments, given that the enclosing environment provides sufficient type information to infer the types of `a`, `b`.

Lambda expressions are lifted into object instances by a *Functional Interface*. A functional interface is an interface that contains only one abstract method. Java 8 predefines some functional interfaces in the package *java.util.function*, including *Function*, *BinaryOperator*, *Predicate*, *Supplier*, *Consumer*, and others. Concise representation of a lambda expression depends on *target typing*. When the Java compiler compiles a method invocation, it has a data type for each argument. The data type expected is the target type. Target typing for a lambda expression is used to infer the argument and return type of a lambda expression. This allows types to be elided in the definition of the lambda expression, making the expression less verbose and more generic.

The Java 8 Stream API is designed from the ground up for use with lambda expressions, and encourages a stateless programming style which provides maximum flexibility for the implementation.

### 2.1.2 Dynamic Compilation

The Just-in-Time compiler is an important technique used to optimize the performance of programs at runtime, inlining methods, removing redundant loads, and eliminating dead code. The design of the Java 8 Stream API plays naturally into the strength of such dynamically-compiling runtime, exploiting its ability to identify and dynamically optimize critical loops. This design allows the performance of streams to continue to improve as the Java runtime environment improves, resulting in highly efficient stream execution.

An important design principle is to ensure the classes for methods invoked in critical loops can be determined by the JIT when the loop is compiled, as this allows the inner methods to be inlined and the loop to be intelligently unrolled.

## 2.2 The Stream Computational Model

Java Stream is an abstraction that represents a sequence of elements that support sequential and parallel aggregate operations. A stream is not a data structure that stores elements but instead it conveys elements from a source through a pipeline of computational operations. An operation on a stream produces a result without modifying its source.

A stream pipeline consists of a source, zero or more intermediate operations, and a terminal operation. The elements of a stream can only be visited once during the life of a stream. If data is needed as input to several operations, a new stream must be generated to revisit the elements of the source, or the data must be buffered for later use.

### 2.2.1 Streams: Data in Motion

A stream is an abstract representation of possibly unbounded elements from an array, a Collection, a data structure, a generator function, an I/O channel, and so on. In contrast with lazy data sets such as Pig Latin relations [4] and Spark RDDs [5], streams conceptually represent data in motion. As we will see in Section 3, our stream model not only allows data to easily flow between different components within a compute engine, but also to conveniently flow from one engine into another. This makes the federated query model conceptually clean and easy to understand.

### 2.2.2 Stream Transforms: Intermediate Operations

Stream transforms, also called intermediate operations in the API, return a new stream from a stream and are processed lazily. A lazy operation is deferred until the final

result is requested. Lazy operations avoid unnecessary early materialization and offer potential optimization opportunities. For example, filtering, mapping, and reducing in a filter-map-reduce pipeline can be fused into a single pass over the data with minimal intermediate state. Laziness also avoids unnecessary stream traversal so that a query like "find the first item that matches the predicate" may return before examining all elements in the stream.

Commonly used intermediate operations include *filter*, *map*, *flatMap*, *distinct*, and *sorted*. Note that sorted is a blocking operation but not a terminal operation. The sorted operation requires incorporating state from previously seen elements when processing new elements.

### 2.2.3 Collectors and Terminal Operations

Terminal operations trigger the traversal of data items and consume the stream. Two commonly used terminal operations are *reduce* and *collect*. Reduce is a classical functional *fold* operation, performing a reduction on the elements of a stream using an associative accumulation function. However, users are more likely to use the collect operation, which extends the notion of reduction to support mutable result containers such as lists, maps, and buffers, as is common in Java programs.

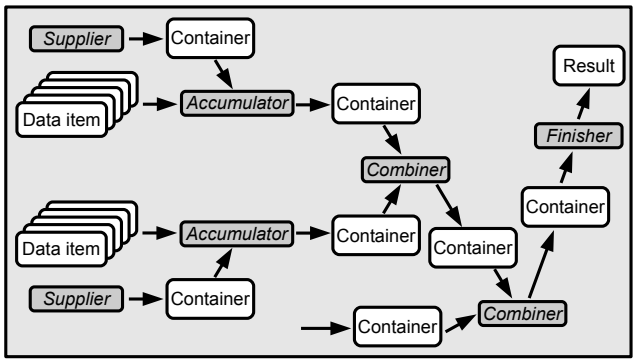


**Figure 1: Parallel data reduction using a Collector.**

The most general use of collect performs the mutable reduction using a *Collector*. A Collector is a mutable reduction operator that accumulates input elements into a mutable result container, optionally transforming the accumulated result into a final representation after all input elements have been processed. A Collector is defined by four functions: a *Supplier* that creates and returns a new mutable result container, an *Accumulator* that folds a value into a result container, a *Combiner* that merges two result containers representing partial results, and an optional *Finisher* that transforms the intermediate result into the final representation. By default the finisher is an identity function. With the new syntax of double colons for method reference capture, the following code creates a collector for collecting widgets into a *TreeSet*,

```
Collector.of(
  TreeSet::new,
  TreeSet::add,
  (left, right) ->
    { left.addAll(right); return left; });
```

Collecting data with a Collector is semantically equivalent to the following,

```
R container = collector.supplier().get();
for (T t : data)
  collector
    .accumulator()
    .accept(container, t);
return collector.finisher().apply(container);
```

However, as illustrated in Figure 1, the collector can partition the input, perform the reduction on the partitions in parallel, and then use the combiner function to combine the partial results. There is a rich library of Collector building blocks in Java SE 8 for composing common queries.

### 2.2.4 Spliterator and Parallelism

Java 8 Streams are built on top of *Spliterators*, which combine sequential iteration with recursive decomposition. Spliterators can invoke consumer callbacks on the elements of the stream. It is also possible to decompose a Spliterator, partitioning elements into another Spliterator that can be consumed in parallel. The results of processing each Spliterator can be combined inside of a Collector. Collection implementations in the JDK have already been furnished with a reasonable Spliterator implementation and most application developers do not need to work directly with Spliterators, instead writing to the Stream abstraction built on top. The following is a query that computes the sum weight of all red widgets in parallel,

```
widgets
  .parallelStream()
  .filter(b -> b.getColor() == RED)
  .mapToInt(b -> b.getWeight())
  .sum();
```

Note that a query on a parallel stream over the same data collection is not guaranteed to perform better than on a sequential stream. There are many factors that affect the performance of parallel execution, including whether the underlying data source can be efficiently split, how balanced the splitting is, and what is the relative cost of the accumulator and combiner functions. In order to achieve performance improvement, one must carefully design the Spliterator as well as the stream operators and collectors.

## 2.3 Challenges Not Addressed By Stream

The Java 8 Stream API provides expressive and efficient ways of processing data inside a JVM, either sequentially or in parallel. However, there are many challenges brought by data sets that cannot fit in a single machine and thus require distributed storage and processing. These challenges include but are not limited to

- Querying against distributed data sources like HDFS, Coherence, and distributed Databases;
- Shipping functions and results among multiple nodes;
- Storing results in distributed form other than single-JVM Java Collections;
- Federating multiple compute engines for query processing and global query optimization.

In the following sections, we present the distributable version of the Stream API and describe how we address these challenges in our design and implementation.

# 3. DISTRIBUTABLE STREAM

## 3.1 The Basic Idea

*DistributableStream* is a stream-like interface that provides similar filter/map/collect operations that can be applied to distributed data sets. At the high level, DistributableStream works by serializing the whole operation pipeline and shipping them to worker nodes. Each worker node is responsible for performing a local portion of the DistributableStream, usually from the data partitions that are locally resident on this node. To do this, the stream pipeline is deserialized and applied directly to the local stream. Shipping the stream pipeline to the place where data resides enables efficient stream execution. Finally, these partial answers from local streams are connected by some form of distributed collection to form a global view of the data. Additional streams can be generated from such distributed collections to be consumed by downstream applications. See Figure 2 for an illustration of this paradigm.

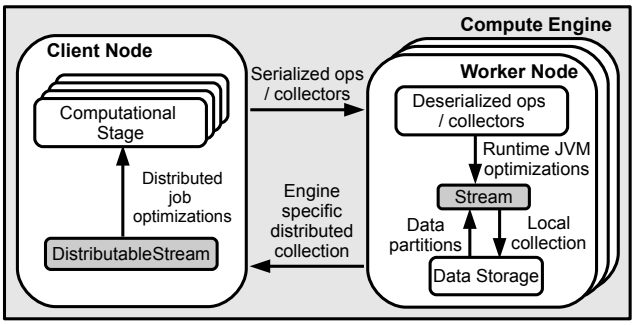


**Figure 2: The workflow of DistributableStream.**

We rely on *serialization* to achieve function shipping described above. Serialization is the process of converting a graph of object instances into a linear stream of bytes, which can be stored to a file or sent over the network. A clone of the original objects can be obtained by deserializing the bytes. To facilitate this, we define *Serializable* subclasses of the Java 8 *java.utils.function* interfaces. By using these interfaces as the arguments for DistributableStream, we ensure that the lambda expressions can be serialized and the functions can be shipped to the node where the data resides. We also define a *Serializable* subclass for the Collector interface as *DistributableCollector* and use this type in DistributableStream. A companion factory called *DistributableCollectors* is provided to create commonly used collectors appropriate for DistributableStream.

## 3.2 The Engine Abstraction

An important feature of the DistributableStream framework is its support for multiple compute engines and federated queries that over them. The *Engine* interface provides a convenient abstraction for this purpose. Each implementation of DistributableStream has a corresponding instance of the Engine interface that controls which compute engine is to be used to execute the computation. Modern data centers have many clusters, supporting multiple engines. An instance of Engine refers to one engine running on one cluster. Calling the

```
DistributableStream.withEngine(Engine engine)
```

method switches the underlying compute engine from the current one to the one specified in the argument, and returns an instance of the DistributableStream associated with the new engine. The Engine class is also used for passing platform specific parameters, as well as providing an avenue for two compute engines to negotiate the data movement between them. The Engine abstraction makes the stream computational model succinct and clean.

Currently we have support for three engines, *LocalEngine*, *MapReduceEngine*, and *CoherenceEngine*. Supporting new engines is convenient as the only requirement is to implement the Engine interface. We plan to add the support for more distributed platforms such as Apache Spark and Apache Tez in the future.

## 3.3 Extended Forms of Collect

In the standard Stream library, the collect() operation is a terminal operation – it consumes a data set and produces a summary result. However, more flexibility is needed in a distributed environment, so we extend the default terminating collect in two ways in DistributableStream.

A method `collectToStream()` is added to the interface that provides for a collecting operator that is blocking but not terminating. CollectToStream() returns another DistributableStream so that we can build longer running operations, reduce startup time and avoid materialization of intermediate results. This approach also allows for optimizations span beyond collect call boundaries.

We also define a subclass of Collector called *ByKeyCollector*. It is introduced to instruct the implementation that collection can be done independently for each key allowing for less communication during the collection process. To do this we insist that the result of such a collection is a *java.util.Map*. Instead of a more general combiner, used in the base class, a *merger* is used to combine just the values associated with the same key. By-key collect is especially important when implementing the non-terminating collectToStream() method on a distributed compute engine.

## 3.4 Engine Specific Distributed Collections

To facilitate the representation of distributed data sets used for creating DistributableStreams and as the output of a DistributableCollector, we introduce engine specific distributed collections. These are distributed materializations of Java *List* and *Map*. Unlike lazily materialized Pig Latin relations and Spark RDDs, they are always materialized. We provide this distributed abstraction so that multiple engines can be supported using the same Java 8 program. These collections are not required to support normal operations such as Map.get(), instead they are DistributableStream factories, providing access to the data they represent. These distributed collections can be named independently of the JVM using engine specific naming scheme. Collections have a preferred engine, and by default, DistributableStreams created from such a collection will use that engine. We also allow collections associated with one engine to be accessed by another engine. If these engines are running on the same cluster, as enabled by Apache Hadoop YARN [2], this can be done without network transfers.

## 3.5 Creation of a DistributableStream

In summary, there are three ways to create a DistributableStream, (i) Use the Engine instance methods to create

a DistributableStream over a persistent engine specific data set, (ii) Build a DistributableStream from an engine specific distributed collection created by DistributableStream.collect(), (iii) Use the result of DistributableStream.collectToStream().

# 4. DESIGN AND IMPLEMENTATION

In this section, we discuss our design and implementation in greater detail. We will cover how we integrate DistributableStream with the native Java 8 Stream, and how we translate stream computation into execution plans over Hadoop MapReduce and Coherence, respectively.

## 4.1 DistributableStream and Stream

We considered three potential relationships for these two interfaces, (i) Have Stream implement DistributableStream, (ii) Have DistributableStream implement Stream, (iii) Provide efficient conversion operators. We chose (iii) but it is instructive to consider the alternatives.

From a type-theory point of view (i) is the most attractive, as Stream is logically a restriction of DistributableStream that is bound to a local compute engine. This runs into a series of technical problems. First of all Java does not support parameter contra-variance for method overrides. This means that a method with a parameter restricted in the base class, for example serializable functions and collectors, cannot be overridden with a method that removes the parameter restriction. An alternative to contra-variance is polymorphism, two methods with the same name that accept different parameters, for example Stream instead of overriding the operator methods taking serializable functions and collectors, defines new methods that accept nonserializable functions and collectors. When matching polymorphic functions, Java picks the most restrictive function that the actual argument can be coerced into. This means that potentially serializable functions will match the DistributableStream methods but not the Stream methods. Unfortunately the Java implementation of Serializability adds additional overhead to the creation of serializable instances which we do not want to pay in the local case.

Alternative (ii) can be made to work but it has problems because for a DistributableStream to implement Stream, it must check that its parameters are Serializable at runtime. If the parameter is not Serializable, it must either fail or move to the local engine which does not require serializability. Of course a DistributableStream can add polymorphic operator methods that accept the serializable functions and collectors, and as long as the arguments are Serializable, these will be called in preference to the Stream methods. But this is error prone as it will be easy for a programmer to write a nondistributable computation thinking it is distributable, only to find out at runtime there is a problem.

This is why we ended up with the option (iii) of converting efficiently between DistributableStream and Stream. We do this by defining an local engine called *LocalEngine*. LocalEngine is a singleton class that represents an engine that runs within this JVM and can be used to process DistributableStream computations. We provide a factory method in LocalEngine that accepts a Stream and returns a DistributableStream associated with the local engine. The resulting DistributableStream is implemented by projecting each method onto the local Stream hidden inside the LocalEngine implementation of DistributableStream.

We handle the other direction by defining

```
DistributableStream.stream()
```

This pulls a local stream out of a distributable stream to allow its result to be examined locally. If the DistributableStream is managed by LocalEngine, then it can just return the underlying Stream object directly. Otherwise it can call withEngine(LocalEngine.getInstance()).stream() to perform the equivalent operation.

Local Java 8 Streams are used to process both small and large amounts of local data so we need to make both Stream creation and per item processing very efficient. On the other hand, DistributableStreams need to be understood by the distributed execution planner which must optimize the movement of data and state between engines. Therefore, our design supports the optimization of both use cases. The local streams are structured to enable runtime JVM optimizations while the distributed tasks can be efficiently managed by the DistributableStream library.

## 4.2 Mapping Streams into Job Plans

One interesting challenge in designing DistributableStream is to translate stream computations into job plans. We break stream computations into stages at the points where shuffle is required. Shuffling data is generally needed in blocking and terminal operators. For example, the following query is mapped into two stages, "filter-flatMap-collectToStream" followed by "map-collect".

```
distributableStream
  .filter(predicate)
  .flatMap(firstMapper)
  .collectToStream(firstCollector)
  .map(secondMapper)
  .collect(secondCollector);
```

We implement the translation lazily, similar to the way Java 8 Stream implements operation pipelines. When an intermediate operation is called on a DistributableStream, the transformation is stored in a data structure referred to by the new instance of DistributableStream returned. Only when a terminal operation is called are both the transformations and the terminal operation serialized and shipped to the appropriate engine's worker nodes. This lazy approach allows the execution of stream operations to be delayed until a concrete result is needed, leaving opportunities for operation pipelining and job plan optimization.

For those compute engines that already support shuffle operation, such as Hadoop MapReduce, we rely on it to shuffle the data at the end of each stage. For underlying engines without explicit shuffle operations, such as Coherence, the Engine implementation will implement the shuffle itself.

## 4.3 MapReduceStream

*MapReduceStream* was our first attempt to implement DistributableStream on a distributed engine. It supports the execution of stream operations on Hadoop clusters. To support MapReduceStream, there is no need to recompile Hadoop source code with JDK8. The only requirement is to install the Java 8 runtime environment (JRE8).

The first challenge we addressed in MapReduceStream was function shipping. We provide three options, all using Java serialization to serialize the pipeline and the functions it references, to a sequence of bytes. (i) When the pipeline being shipped does not reference large objects, we stringify the serialized pipeline and insert it into the Hadoop job

*Configuration* object. (ii) When the pipeline does reference large objects, this becomes inefficient so instead we store the serialized pipeline into a file and add it to the Hadoop *DistributedCache*. (iii) If the objects being referenced are very large, we require the programmer to use a distributed collection class with a smart serializer and deserializer so that the reference to the distributed collection rather than the actual data is included in the serialized pipeline.

Another challenge for MapReduceStream is dealing with the strong typing in Hadoop. It is required that the input and output key/value instances match the classes specified in the job configuration. While this is quite useful to guarantee the correct job behavior, it is tedious and error-prone to configure such parameters for jobs with multiple rounds of map and reduce. Initially we provided a way for users to embed these parameters to the Collector and made the framework responsible for setting the job configuration. This approach made stream application code less clean and too specific to the MapReduceEngine – other engines may not have such a strong typing requirement and we did not want to pollute the DistributableCollector interface. We decided to remove this Hadoop-specific constraint from the interface and instead require users to specify the classes in the Engine configuration at each step of the stream computation. For improved usability, we also provide a wrapper class that can wrap any *Serializable* or *Writable* objects. The framework can wrap and unwrap the data items if the classes are set to this wrapper type. This saves effort in configuring each computational stage at the cost of efficiency due to wrapping and unwrapping.

Stream computations are broken into stages by blocking and terminal operations. Each stage of MapReduceStream is implemented by a single Hadoop job. Hadoop jobs are only submitted when a terminal operation is called.

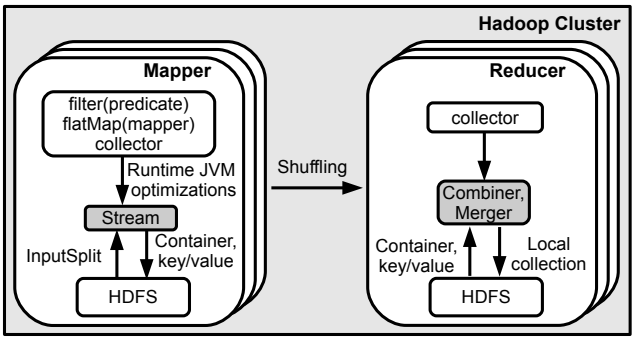


**Figure 3: Translating a stage to a Hadoop job.**

Figure 3 shows how a single stage of a stream computation is mapped to a Hadoop job in greater detail. In particular, by assembling local stream from an *InputSplit* of a mapper, we are able to apply the pipelined operations to data items in one pass. This gives the same result as the native Hadoop *ChainMapper* that runs several map phases inside a single mapper, with the advantages of being inline-able, avoiding configuring parameters for each map phase separately, and providing in-memory partial merging opportunity before collecting records to the MapOutputBuffer. Note that collecting into containers can happen at both mapper and reducer side, with the distinction that the mapper side never applies the optional finisher embedded in the collector.

We rely on the Hadoop framework to shuffle mapper output to reducers. If the collector is a ByKeyCollector, individual records are shuffled by key and efficient merging of values associated with the same key is performed on the reducer side; otherwise, all containers are sent to a single reducer that combines them in pairs. At the end of this stage, all local collections coming out of the reducers are logically put together to form an engine specific distributed collection. Then the next stage continues from there.

MapReduceEngine includes factories to generate MapReduceStreams based on a Hadoop Configuration object, which must contain properly configured parameters such as InputFormat and InputPath. Three methods `keyStream(conf)`, `valueStream(conf)`, and `entryStream(conf)` return a DistributableStream of keys, values, and entries, respectively.

## 4.4 CoherenceStream

Oracle Coherence [16] is an in-memory Java object store equipped with computational capability. Our design and implementation of *CoherenceStream* has benefited from the lessons we have learned from MapReduceStream. We borrow ideas from MapReduceStream and adapt them to fit requirements in the Coherence environment.

The first special treatment for CoherenceStream was to add the support for streams without keys. The current version of Coherence provides distributed in-memory maps but not lists. Therefore, storing stream elements into a Coherence NamedCache requires that the framework generate unique keys for these items. One solution is to use machine-specific auto-incrementing integers as the primary key. The key structure we use contains an optional thread id, a machine id, and an integer. Another approach is to use a Coherence provided primary key generator that allows for optimized data loading.

Another challenge comes from the fact that Coherence uses its own object format, the portable object format (POF), to do object serialization. POF is language independent and very efficient. Usually (de)serialization of POF object is several times faster than the standard Java serialization and the result is much smaller. Standard Java types have their default serializer implemented in Coherence. However, user-defined types have to either implement the *PortableObject* interface themselves or register their own serializers that implement the *PofSerializer* interface. This means that Writables that come with Hadoop and user-defined Serializables that are not standard in Java cannot be serialized directly into Coherence. We provide serializers for commonly used Writable types and we also provide a wrapper class that translates objects between the different serialization methods. However, we recommend that users write their own serializers for their customized types if the performance penalty caused by wrapping becomes significant.

Coherence does not have a similar framework like Hadoop for accepting jobs, assigning mappers and reducers, and shuffling data between them. Instead, we provide our own implementation to achieve the same functionalities. This framework is responsible for assigning an id to each job submission, starting necessary services for the job execution, and keeping track of the job status and output.

The actual computation is implemented using distributed *Invocation Services*. An invocation service allows the execution of single-pass agents called *Invocable objects* (i) on a given node, (ii) in parallel on a given set of nodes, or

(iii) in parallel on all nodes of the grid. We take a two-step approach similar to MapReduceStream to implement a single stage of the stream computation. For each stage, the framework first starts an invocation service to run the pipeline of intermediate operations and collect the results into an intermediate *MapperOutputCache*. We call this step the *MapperService* which has functionality corresponding to a Hadoop Map task. Upon the completion of the MapperService, the framework starts another invocation service, the *ReducerService*, to read from the intermediate cache and write the final results into the output cache.

We have applied some optimization techniques to avoid unnecessary data movement in CoherenceStream. A NamedCache in Coherence is partitioned and distributed across a set of member nodes in the grid by the framework. A MapperService is started on the member nodes that own the input cache, and we use a filter so that each Map task processes only the locally available data from the input cache. In Hadoop terminology, all Map tasks are data-local. The *OutputCollector* accumulator is called by the MapperService and buffers entries in local memory and flushes them to the MapperOutputCache when a pre-configured threshold is reached. The MapperOutputCache is implemented with partition affinity so that all entries associated with the same key are stored in the same partition, thus eliminating the need to reshuffle the data before starting the ReducerService. Like MapperService, the ReducerService is started on member nodes storing the MapperOutputCache and, with the help of a filter, each of them only processes its local partitions. At the end of the ReducerService execution, the final results are written to their local nodes with partition affinity. See Figure 4 for an illustration of CoherenceStream that corresponds to the MapReduceStream example in Figure 3. The ReducerService could also execute the map and collect associated with the next stage of the computation but we have not implemented this yet.

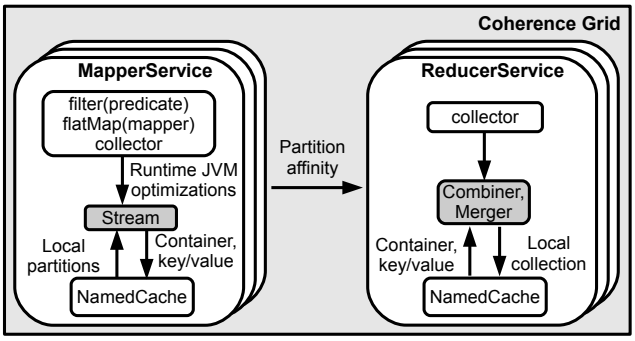


**Figure 4: Translating a stage to invocation services.**

We define a *Configuration* class to store job related configuration parameters. We also provide factory methods in CoherenceEngine to generate CoherenceStreams based on such a Configuration object. Three methods `keyStream(conf)`, `valueStream(conf)`, and `entryStream(conf)` return a DistributableStream of keys, values, and entries, respectively.

## 4.5 Changing Engines in Midstream

To support federated query, the DistributableStream framework relies on the Engine abstraction to separate the engine-specific details from the computational model. The Engine abstraction also provides an avenue for two compute engines adjacent in the computation to negotiate the data movement at their boundary. The two approaches for data movement between engines is shown in Figure 5. By default, data movement follows the push-based model where the upstream engine directly writes the result to the downstream engine. But in some cases, having the upstream engine materialize the result itself and the downstream engine to pull from it reduces the data transfer cost. Let us take a look at the following example,

```
CoherenceEngine
  .entryStream(conf)
  .filter(predicate)
  .withEngine(mapReduceEngine);
```

Instead of having Coherence to write entries into HDFS files, we ask Hadoop to directly pull the data from the NamedCache via the *NamedCacheInputFormat*. This strategy saves two passes over the entire disk resident data set.

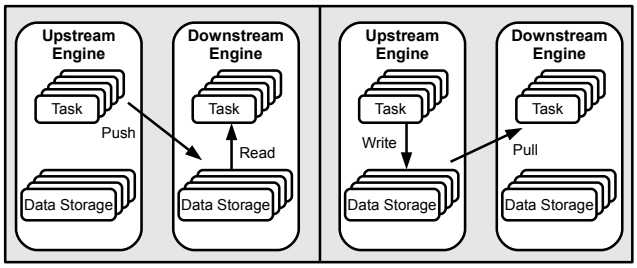


**Figure 5: Data movement between engines.**

There is also an optimization called "short-circuiting", as illustrated by Figure 6, available to the special case where the downstream engine simply pulls data from the upstream engine without having the latter to do any preprocessing. In this case, if the downstream engine is able to read directly from the upstream storage, there is no need to run a separate job in the upstream engine just for data movement.

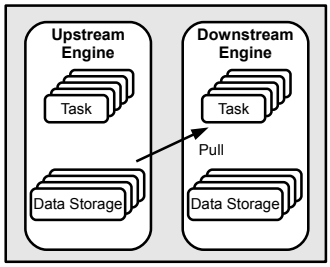


**Figure 6: Short-circuiting without upstream job.**

A useful application of short-circuiting is the pulling of external data specified by a Hadoop InputFormat directly into the Coherence in-memory cache. This enables us to pull data from Hadoop into memory similar to the creation of a HadoopRDD in Spark.

```
MapReduceEngine
  .entryStream(conf)
  .withEngine(coherenceEngine);
```

We have implemented factories for creating local streams from various data sources, for example, *InputFormatStream* from HDFS files or Hive tables and *NamedCacheStream* from Coherence NamedCaches. We have the corresponding implementations *OutputFormatCollector* and *NamedCacheCollector* for collecting to Hadoop and Coherence. We use these streams inside of our Engine implementations but they are also useful for programmers writing nondistributed applications that want to access distributed data sets.

# 5. EXAMPLE STREAM APPLICATIONS

## 5.1 Distributed Reservoir Sampling

Reservoir sampling is a family of randomized algorithms for uniformly sampling $k$ items from a base stream of $n$ items, where $n$ is either very large or unknown in advance. A simple $O(n)$ algorithm maintains a reservoir list of size $k$ and runs in a single pass over the base stream. The algorithm works by storing the first $k$ items in the reservoir. For the $i$-th item where $k < i \leq n$, the algorithm includes it in the reservoir with probability $k/i$. If this happens, a random item from the reservoir is removed to keep the total as $k$.

To sample from a distributed stream in parallel, we extend the basic algorithm and choose at most $k$ samples uniformly from each partition and then combine these samples by choosing a subset so that each item from the base stream has equal probability $k/n$ of being chosen. To achieve this goal, we implement a *SampleContainer* where the `put(T item)` method implements the basic reservoir sampling algorithm above and the `putAll(SampleContainer<T> other)` method combines two containers by choosing elements from the partitions with weights computed from their respective sizes. The distributed reservoir sampling algorithm is implemented as a collector where the supplier creates SampleContainers, the accumulator puts an item into a SampleContainer, the combiner combines two SampleContainers, and the finisher extracts the reservoir list after the final combining. To facilitate the creation of such collectors, we provide a factory `SamplingCollector.get(k)` to return a reservoir sampler with the appropriate size reservoir.

**Program 2** Distributed Reservoir Sampling

```
public class SamplingCollector {
  public static <T> DistributableCollector<
    T, SampleContainer<T>, List<T>> get(int k) {
      return DistributableCollector
        .of(() -> new SampleContainer<T>(k),
            SampleContainer::put,
            SampleContainer::putAll,
            SampleContainer::getList); } }

public static <T> List<T> distReservoirSample (
  DistributableStream<T> stream) {
    return stream
      .collect(SamplingCollector.get(k)); }
```

## 5.2 PageRank

PageRank is a graph algorithm for measuring the importance of webpages. On a directed graph, the algorithm iteratively updates the rank of a vertex by computing a weighted sum of ranks from its incoming neighbors.

**Program 3** PageRank

```
// Page is a class that represents the format
// <URL, <[neighbor-URL, neighbor-URL], rank>>
public static DistributableStream<Page> iterate(
  DistributableStream<Page> stream, float damp) {
    return stream
      .flatMap(p -> {
        final float rank = p.getRank();
        p.setRank(1 - damp);
        List<String> nbrs = p.getNeighbors();
        final int size = nbrs.size();
        return Stream.concat(
          // contribution from incoming neighbors
          nbrs.stream()
            .map(nbr -> new Page(nbr,
              emptyList, damp * rank / size)),
          // contribution from damping factor
          Stream.of(p)); })
      .collectToStream(
        DistributableCollectors
          .toMap(
            // clustering based on URL
            p -> p.getURL(),
            p -> p.getValue(),
            (left, right) -> {
              // reconstructing neighbor list
              mergeNeighbors(left, right);
              // updating rank
              left.setRank(
                left.getRank() + right.getRank());
              return left; })); }

public static DistributableStream<Page> pageRank(
  DistributableStream<Page> stream,
  float damp, int iteration) {
    return Stream
      .iterate(stream, s -> iterate(s, damp))
      .skip(iteration - 1).findFirst().get(); }
```

Assume a webpage is represented in the format `<URL, <[neighbor-URL, neighbor-URL], rank>>`. In one iteration of the algorithm, it parses every page so that an entry `<neighbor-URL, <[], contribution>>` is emitted for each of its neighbors, where contribution is calculated as the product of its rank and the damping factor, divided by the neighbor size. A special entry `<URL, <[neighbor-URL, neighbor-URL], 1 - damp>>` is also emitted for this page so that the list of neighbors can be reconstructed for next iteration. Entries are then by-key collected by their URLs and their page ranks updated. Note how we take advantage of the non-terminating collectToStream() method to build larger multi-stage computations.

## 5.3 K-Means Clustering

K-means clustering is a popular clustering method in machine learning and data mining. Given $n$ vertices in the space, the algorithm assigns each vertex to the cluster whose centroid is closest to it. At each iteration the centroids are updated using the mean of the vertices assigned to it.

In each iteration, the stream supplier returns a stream of all vertices in the space. The vertices are by-key collected

**Program 4** K-Means Clustering

```
public static List<Vertex> kMeans(
  Supplier<DistributableStream<Vertex>> supplier,
  List<Vertex> centroids, int maxIter,
  double threshold) {
    do {} while (
      supplier
        .get()  // generating vertex stream
        .collect(DistributableCollectors
          .toMap( // clustering by cluster id
            v -> closestCentroid(v, centroids),
            v -> new Pair(v, 1),
            (left, right) -> {
              left.setVert(
                sum(left.getVert(),
                  right.getVert()));
              left.setCnt(
                left.getCnt() + right.getCnt());
              return left; }))
        .entrySet()
        .parallelStream() // SMP processing
        .collect( // updating means
          Collector.of(
            () -> new DoubleWritable(0),
            (delta, entry) -> {
              int id = entry.getKey();
              Vertex centroid = mean(
                entry.getValue());
              delta.set(delta.get() +
                distance(centroid,
                  centroids.get(id)));
              centroids.set(id, centroid); },
            (left, right) -> {
              left.set(left.get()
                + right.get());
              return left; },
            delta -> delta.get())) > threshold
        && maxIter-- > 0 )
    return centroids; }
```

into a map, where the key is the id of the cluster to which the vertex is closest and the value is a pair consisting of the vertex coordinate and 1. The merger merges two pairs returning a new pair that represents the vector sum and the vertex count. The clustering step is done with a distributed compute engine. After that, the means for each cluster are computed by dividing the vector sum by the vertex count to create the new centroids of the $k$ clusters. The new centroids and the termination condition are evaluated by the LocalEngine running inside the client JVM. This stage could have been executed using a second reduce step on the iteration engine, but this is less efficient if $k$ is sufficiently small and the client machine is sufficiently large.

To bootstrap this algorithm, we need to give it a stream supplier and a list of initial centroids. Hadoop is used to parse and filter the raw data. The filtered input is then written into a Coherence NamedCache to take advantage of the fast in-memory store for the iterative computations. The initial $k$ centroids are obtained using our distributed reservoir sampling algorithm described in Section 5.1. The driver for k-means clustering is written in Program 5.

**Program 5** Driver for K-Means Clustering

```
List<Vertex> centroids =
  MapReduceEngine.valueStream(hConf)
    .map(parser)
    .withEngine(CoherenceEngine.get(cConf))
    .collect(SamplingCollector.get(k));
return kMeans(
  () -> CoherenceEngine.valueStream(cConf),
    centroids, maxIter, threshold);
```

# 6. EVALUATION

We evaluate our implementation through a series of experiments in an Oracle Big Data Appliance [15], where each node is equipped with 2 × eight-core Intel Xeon processors, 64GB memory, and 12 × 4TB 7200RPM disks. Nodes are connected by InfiniBand switches for high network throughput. We use Cloudera CDH 5.0.2 Hadoop distribution, Oracle Coherence 12.1.2, and Java SE 8u5 as the testing environment. Coherence cache server instances are started by a long running map-only Hadoop job.

We first compare a DistributableStream program with a native program for the same engine. In this test we compare MapReduceStream with a native Hadoop job. We run WordCount on 45GB Wikipedia dumps. We compare the stream implementation using Java immutable types shown in Program 1 and another using Hadoop Writable types. For comparison, we compare both the standard WordCount example in the Hadoop package and another using Java immutable types. The four implementations are executed using the same hardware with equal amount of resources.

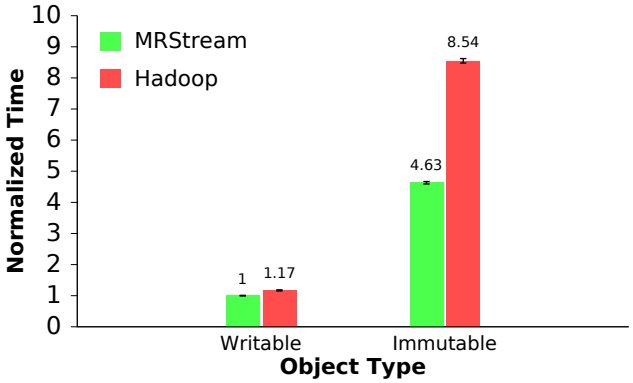


**Figure 7: WordCount.**

The normalized job completion times are shown in Figure 7. MapReduceStream outperforms Hadoop by 17% for Writable types, and by 84% for immutable types. This is because MapReduceStream performs in-memory partial merging before collecting records to MapOutputBuffer. The saving is substantial for immutable types, which incur higher serialization and garbage collection costs.

One of the most important features of DistributableStream is the ability to perform federated queries on top of multiple compute engines for query optimization. We run k-means clustering algorithm shown in Section 5 to test the benefit of using Coherence over Hadoop as the compute engine for iterations. About 45GB of raw data representing one

billion vertices is first parsed and filtered by the Hadoop engine, and gets written into HDFS or Coherence, depending on the compute engine to be used for the iterative steps. Then the compute engine chosen for iterations is used to cluster vertices into one thousand clusters. At the end of each iteration, the LocalEngine is used for updating the list of centroids and evaluating the termination condition.

Previous work has shown that by avoiding disk IOs, in-memory processing can outperform Hadoop by up to 20× in iterative applications [19]. We take a different approach in our evaluation: by caching the input data and all intermediate results in the OS cache, we avoid most disk IOs during the job execution. The number of iterations is varied to see how the job completion time is affected by the increased workload. We configure the parameters so that Hadoop and Coherence have equal amounts of available resources.

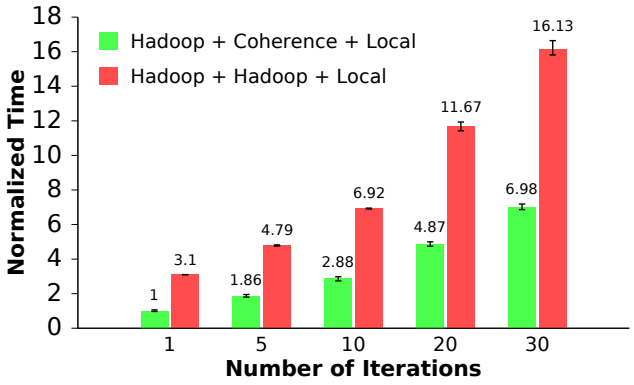


**Figure 8: K-means clustering.**

The normalized job completion times are shown in Figure 8. We observe that for evaluating each iteration, Hadoop is outperformed by Coherence by 2.3× to 3.1×, even without the extra disk IOs. This is partly because accessing the OS cache is slower than accessing the Java heap and also because Coherence stores data in memory as Java objects which saves deserialization cost.

# 7. FURTHER DISCUSSIONS

## 7.1 Operation Pipeline Improvement

Consider again the example shown in Section 4.2. In our current MapReduceStream implementation, two consecutive Hadoop jobs are needed for the two stream stages, resulting in the "map-reduce-map-reduce" pattern. Spark and Tez, with their ability to perform "map-reduce-reduce" operations, can apply the second map operation in the same reducer processes in the first stage, right after they finish reducing on each key. This will allow intermediate operations from the next stage of the computation to be pipelined with the terminal operation in the previous stage, saving one pass over the data set.

## 7.2 Job Planner and Optimizer

The ability to change engines in midstream offers great flexibility in designing stream applications. Choosing the right engine based on price, data locality, and available resources is critical in performance optimization.

**Small Data Sets:** Some applications with small data sets fit within a single JVM. Running such applications in a local engine avoids unnecessary costs of starting distributed services and worker instances and sometimes can be more efficient than a distributed engine. As we provide DistributableStream with an efficient local implementation, applications written with this API can easily scale down for processing small amounts of data efficiently.

**Iterative Algorithms:** Many graph and machine learning algorithms apply computations iteratively until a convergence condition is reached. Loading the input to the fast in-memory store and running iterations there avoids expensive IOs and can lead to a huge performance improvement.

**Huge Data Sets or CPU-Bound Applications:** Some applications work on huge amount of data that just cannot fit in memory. Other applications, such as natural language processing and video encoding, are CPU-bound and benefit more from increased computing resources rather than faster IO. Memory is also much more expensive than disk: extra money spent on memory might be better used to buy more or faster processors.

**Resource Availability:** Data centers are generally not built out as one uniform cluster. Data centers have a variety of different clusters, purchased at different times, running different software, storing different data, with different amounts of memory, disk and computational resources. The ability to run a federated computation allows the programmer to exploit the available resources. For example, a memory based engine, like Coherence, may be a cache layer shared by multiple applications. It can make sense to ship data and offload some work to another cluster which has more resources available to it.

**Automatic Engine Assignment:** In the current implementation, we do not have a computation optimizer to automatically choose the right engine for each stage in a stream application. By including the withEngine() method in DistributableStream interface, we provide a convenient means for an optimizer to take charge. Programmers now can manually design the execution plan by explicitly calling the withEngine() method. We give such an example for the k-means clustering algorithm in Section 5. Using our Engine framework, programmers can write programs that do not hard code the engine and instead read the engine class name and configuration from the command line.

Without careful application design, the flexibility provided by our framework can lead to poor performance. Let us assume some input data resides in a Hadoop cluster. Consider the two examples shown in Figure 9.

The first approach first generates an InputFormatStream that represents the input data, and then wraps it into a LocalStream which is a DistributableStream with the LocalEngine, and finally uses withEngine() call to switch back to the MapReduceEngine.

```
LocalEngine
  .of(InputFormatStream.valueStream(conf))
  .withEngine(mapReduceEngine);
```

This approach is semantically identical to the second approach shown as the direct path in Figure 9,

```
MapReduceEngine.valueStream(conf);
```

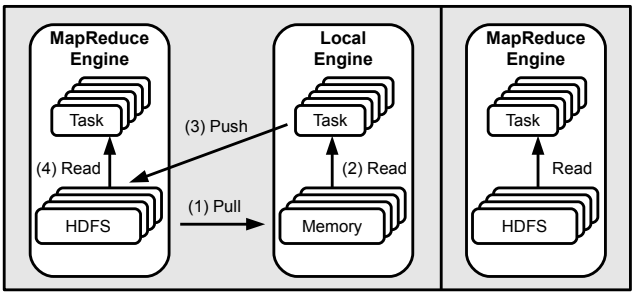


**Figure 9: Improper engine assignment.**

However, the first one suffers from the fact that all stream elements go through the local JVM before being consumed by the processing tasks in the Hadoop cluster, resulting in a less efficient program. While this example might seem artificial, given the flexibility our framework offers to the programmers, we could end up seeing a few similar cases. This is similar to what we have discussed in Section 2.2.4 on parallelism in local streams – making a stream parallel is not always beneficial, and sometimes can be harmful.

Therefore, we emphasize here that, carefully choosing the right source representation and the engine assignment is important in designing stream applications. The users need to have a basic performance cost model in order to use the tools effectively. While good visualization and monitoring tools are useful for programmers to approach the right design, an automatic job planner and optimizer is a more attractive way of achieving the same goal.

### 7.3 Job Progress Monitoring and Workspace Management

The chance of failure during a job execution grows with the scale of the distributed environment. Each compute engine usually has its own way of monitoring job progress and tolerating task failures during the job execution. However, as the execution of a DistributableStream application can span across multiple compute engines, job progress monitoring also needs to go across their boundaries.

Assume a stream application consists of stages on several engines. On the top level, a centralized service, ZooKeeper [8] for example, can be responsible for keeping track of the unique global job id. Engine-specific job information can be stored under the global job id by the individual Engine implementations. Task-level progress monitoring and fault tolerance within each compute engine are still handled by individual engines. When a portion of the job on a specific engine fails entirely, the centralized service takes the charge to recover or fail the job.

Consider data movement between two engines. The pull-based model works by providing a means for tasks in the downstream engine to read from the data storage in the upstream engine. Temporary results from failed tasks in the upstream engine will not move across the engine boundary because they are automatically handled by the first engine. On the other hand, the push-based model usually works by side-effects, instructing upstream engine to write directly into downstream data storage. Without proper output commitment, partial results from failed upstream engine tasks could sneak into the downstream storage. One technique for avoiding problems is to generate repeatable primary keys. In this case, entries generated by a restarted task will overwrite the partial results from the failed task, even if they have made through into the downstream storage.

Once the execution of the entire job is complete, intermediate results inside individual compute engines need to be cleaned up and temporary files deleted. The centralized service can, in a similar way as Cascading and Tez, instruct every compute engine to handle its own piece. It could also leave the users to do the cleanup themselves, in case that some of the results were named for later reuse.

Currently, in the MapReduceStream implementation, temporary files are written under a temporary directory which gets cleaned up periodically, while in the CoherenceStream intermediate caches for MapperServices are destroyed upon the completion of ReducerServices.

## 8. RELATED WORK

**Language Integrated Query:** We like SQL, but not all Java programmers use SQL and implementing certain applications is more natural in Java. Therefore, instead of integrating SQL style declarative query as LINQ [14] and OptiQL [13], DistributableStream takes a similar approach to Dryad [11] and Spark [5] to adopt functional programming style for the language integrated interface. This allows application developers to live on the stream abstraction with its well-defined data-parallel operations and still enjoy the flexibility of procedural languages. While one could consider building yet another SQL compiler on top of DistributableStream, like Hive [3] on Hadoop or Shark [17] on Spark, we are more interested in extending existing SQL engines with table functions defined using Java computations.

Cascading [9] is an application development platform for building applications on Hadoop and most recently Tez [7]. Similar to DistributableStream, Cascading provides an abstraction layer between the compute engine APIs and developers, so that applications developed with Cascading APIs can be reused without rewriting any business logic when the underlying engine is replaced. However, it is not clear to us that Cascading supports changing engines within the same application. By reusing the patterns and concepts from the standard Java 8 Stream interface and supporting changing engines in midstream, DistributableStream could be a better choice to help Java programmers quickly and conveniently adopt the API for processing big data sets.

**Massively Parallel Processing Systems:** MapReduce [10], Hadoop [1], Dryad [11], Spark [5], Storm [6], and Tez [7] are examples of recent data-parallel MPP systems. These systems store data as individual splits and run parallel tasks responsible for processing their own data splits. DistributableStream, on the other hand, does not provide its own data-parallel MPP infrastructure. Our focus here is to provide a clean computational model and API for federating different MPP systems both between and within a query. The goal is to facilitate the development of simple, generic, and efficient applications across an extensible set of compute engines to process distributed data sets.

**Distributed Streaming Computation:** Apache Storm [6] is a distributed realtime computation system for processing streams of data. Computations in Storm are specified by topology transformations in DAGs. Storm defines

a spout abstraction, which is responsible for feeding messages into the topology for processing, for integration with new queuing systems. Spark Streaming [18] is another distributed stream processing system that has tight integration with Spark and combines streaming with batch and interactive queries. Although the engine implementations we currently have for DistributableStream do not support real-time streaming, the DistributableStream API itself can be extended with tee and window functions to support Storm and Spark Streaming.

**Resilient Distributed Datasets (RDDs):** Combining Coherence and Hadoop allows us to cache, store, and process data that originally resides in HDFS in a similar way as RDDs in Spark. Both Spark and Coherence allow storing data at different levels depending on whether they are stored as Java objects or as serialized bytes, whether they are stored in memory or on disk, and whether they are replicated or not. RDDs also support storing the transformation recipe to allow lineage-based recovery for fault tolerance, while the current Coherence implementation does not include this feature. We have developed a preliminary implementation that allows us to use RDDs as a data source and Spark as a compute engine for stream computations. Detailed evaluations are planned for future work.

# 9. CONCLUSIONS

Java and other JVM-based languages play an important role in the Hadoop and Big Data ecosystem. Java is a more natural implementation language for certain applications and we have seen these JVM-based distributed systems complement the SQL based analytical systems for processing unstructured data and for algorithmic programming.

While the flexibility of Java has been appreciated in the Hadoop ecosystem, there are some features in high demand. First, expressiveness and conciseness brought by functional and declarative languages are convenient for application development, which were not well supported by older versions of Java. Second, many JVM-based distributed systems have their own distinct APIs. This forces companies and application programmers to split their effort and investment among several frameworks, hoping that their bet on today's frameworks will not become obsolete tomorrow with newly emerged systems. A single API that can be supported over multiple engines can be seen as insurance against obsolescence. Third, the demand for improved performance can never be satisfied. It is preferred that a framework can be organized so that it can easily benefit both from high-level job planning and low-level JVM optimizations.

To address this demand, we present DistributableStream, a Java computational model that enables programmers to write generic, distributed and federated queries on top of an extensible set of compute engines, allowing data to flow between them so that different stages of the computation can be carried out on their respective optimized engines. By reusing the patterns and concepts from the standard Java Stream interface, DistributableStream provides a friendly way of integrating queries into the programming language, making Java a more expressive tool for Big Data processing. With the Engine abstraction, the platform-specific parameters are separated from the computational model, freeing programmers from low-level details as they design the algorithmic part of their applications. The abstraction also makes it convenient to run existing applications developed on the DistributableStream framework on new engines, leveraging the investment in past development. The DistributableStream approach, translating computations into stages and decomposing data into local streams, allows for efficient execution that benefits from both job optimization and intelligent Java runtimes.

We validate this API by showing three implementations – a local SMP engine, a distributed file based engine and a distributed in-memory engine – and demonstrate the usefulness of DistributableStream with examples and performance evaluations. We are considering incorporating this work in a JSR and hope that others will write applications using the API, support the API on additional engines, and extend the computational model supported by the API.

# 10. ACKNOWLEDGEMENT

We thank the Oracle Java team for their work on the Java 8 Stream API and Michel Benoliel for his inspiring implementation of MapReduce on Coherence. We also thank the anonymous referees for their useful suggestions that significantly improved the quality of this work.

# 11. REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org/.
[2] Apache Hadoop YARN. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.
[3] Apache Hive. http://hive.apache.org/.
[4] Apache Pig. http://pig.apache.org/.
[5] Apache Spark. http://spark.apache.org/.
[6] Apache Storm. http://storm.incubator.apache.org/.
[7] Apache Tez. http://tez.incubator.apache.org/.
[8] Apache ZooKeeper. http://zookeeper.apache.org/.
[9] Cascading. http://www.cascading.org/.
[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of OSDI*, pages 137–150, 2004.
[11] The Dryad project. http://research.microsoft.com/en-us/projects/Dryad/.
[12] JDK 8 project. https://jdk8.java.net/.
[13] H. Lee, K. J. Brown, A. K. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, 2011.
[14] LINQ: language integrated query. http://msdn.microsoft.com/en-us/library/bb397926.aspx/.
[15] Oracle Big Data Appliance. http://www.oracle.com/us/products/database/big-data-appliance/.
[16] Oracle Coherence. http://www.oracle.com/technetwork/middleware/coherence/.
[17] Shark. http://shark.cs.berkeley.edu/.
[18] Spark streaming. http://spark.apache.org/streaming/.
[19] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of NSDI*, pages 15–28, 2012.