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

VLDB’14
September 1st - 5th, 2014
DistributableStream for Big Data Processing
Motivation
Motivation
DistributableStream
Next Steps

Challenges We Try to Address
Clarifications

Big Data Space

Many data sources!
Many compute engines!
Many tools to learn, use, and maintain!

- TEZ
- HIVE
- hadoop
- HDFS
- Spark
- STORM
- Apache HBASE
- Cassandra
- Oracle Database 12c
- Oracle Fusion Middleware Coherence
Usability

Simple computational model.
Friendly programming interface.
School kids can process Big Data!

Daddy's Hadoop app will take over the world!

Cool! I hope to do the same in Java 101.
Portability

A single API supported over multiple engines. Reuse applications developed for old engines. Leverage the investment in past development.

They also want your Hadoop app on Spark.

Manager

Sure... Maybe in 6 months?

Developer

Customers
Various data processing requirements.
Varied engine capabilities.
Price, data locality, and resource availability.
A Java stream computational model and interface for Big Data processing
Clarifications

Q: Why Java?
A: User friendliness, big user base, broad adoption in Hadoop ecosystem (with other JVM-based languages), ...

Q: Why not SQL?
A: We certainly love SQL – but not all Java programmers use SQL, less natural to implement certain applications in a declarative language, one can build a SQL compiler on top, ...

Q: Yet another data-parallel MPP system?
A: No. A clean computational model and API for federating different MPP systems both between and within a query.
DistributableStream for Big Data Processing
DistributableStream is an abstraction that supports generic, distributed and federated queries on top of an extensible set of compute engines.
Concise Yet Expressive

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));}
**Motivation**

**DistributableStream**

**Next Steps**

---

**Generic Programming on Distributed Engines**

**WordCount**

```java
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));
}
```

---

**Thread Pool**

**Hadoop MapReduce**

**Oracle Coherence**

**Apache Spark**

---

**Thread Pool**

**Hadoop MapReduce**

**Oracle Coherence**

---

**Apache Spark**

**Oracle Coherence**

---

**DistributableStream for Big Data Processing**
Stream Stages On Respective Optimized Engines

- Initial Parsing & Filtering
- Iterating
- Updating Summary & Evaluating Termination Condition

Hadoop
MapReduce

Apache
Spark

Oracle
Coherence

Thread Pool

Task Queue
Completed Tasks

Initial Parsing & Filtering

Thread Pool
Model

DistributableStream for Big Data Processing
A stream represents a sequence of elements that support sequential and parallel aggregate operations.

A stream pipeline consists of a source, zero or more intermediate operations, and a terminal operation.
An intermediate operation returns a new stream from a stream and is processed lazily.

Commonly used intermediate operations include filter, map, flatMap, distinct, and so on.
A terminal operation triggers the traversal of data items and consumes the stream.

Two commonly used terminal operations are reduce and collect.
Collect method usually works with a Collector.

A Collector is defined by a Supplier, an Accumulator, a Combiner, and an optional Finisher.
We extend the Stream model to allow the use of distributed engines for processing distributed data sets.
Design & Implementation
Function shipping via Java serialization.

Assemble local streams from local data partitions.

Engine specific immutable distributed collections.
Break stream computations into stages at the points where shuffle is required.

PageRank:

Page \xrightarrow{flatMap} collectToStream

... \xrightarrow{flatMap} collectToStream

\xrightarrow{flatMap} collect \xrightarrow{Page}
Engine interface for separating low-level details from the computational model and negotiating data/state movement between engines.

Each compute engine needs to implement the Engine interface.

Engine parameters are configured in the engine configuration object.
DistributableStream for Big Data Processing

Motivation
DistributableStream
Next Steps

Configuring Engine Parameters

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

MapReduce Engine

```java
job.setOutputKeyClass(String.class);
job.setOutputValueClass(Integer.class);
FileInputFormat.addInputPath(job, new Path(args[0]));
...
DistributableStream<String> stream =
    MapReduceEngine
        .valueStream(job.getConfiguration())
        .map(s -> s.toString());
wordCount(stream);
```
Changing Engines

DistributableStream.withEngine(Engine) switches the underlying engine and returns an instance of the DistributableStream associated with the new engine.

\[ \text{dstream.map(...).filter(...).withEngine(\textit{engine})} \]
Data Movement When Changing Engines

Push vs. Pull

Push

Upstream Engine
Task
Data Storage
Push

Downstream Engine
Task
Data Storage
Read

Pull

Upstream Engine
Task
Data Storage
Write

Downstream Engine
Task
Data Storage
Pull

Push vs. Pull

27 Xueyuan Su etc. DistributableStream for Big Data Processing
**Push** is the default option. 

Upstream engine writes to downstream storage.
Pull is usually used when upstream engine cannot write to downstream storage.

When upstream engine is in memory, pulling from it saves disk access costs.
Special Pull: Short-Circuiting

**Short-circuiting** enables downstream engine to pull from upstream storage without running tasks there.

**Use case:** Hadoop InputFormat $\rightarrow$ Coherence in-memory cache, similar to HadoopRDD in Spark.
Example Applications

WordCount
Distributed Reservoir Sampling
PageRank
K-Means Clustering

Refer to the paper for actual code
Performance
Oracle Big Data Appliance (BDA):
- Each node has:
  - 2 × eight-core Intel Xeon processors.
  - 64GB memory.
  - 12 × 4TB 7200RPM disks.
  - InfiniBand interconnections.
- Cloudera Hadoop CDH 5.0.2.
- Oracle Coherence 12.1.2.
- Java SE 8u5.
- Recompile Hadoop and Coherence source code with JDK8, install JRE8.
Overhead?
**Stream vs. Native Implementations**

**WordCount** implemented with DistributableStream and native Hadoop.

Input: 45GB Wikipedia dumps.

Writable and Java immutable types for comparison.
Native Hadoop implementation is slower, $1.17 \times$ for Writables, and $1.84 \times$ for immutables.

Partial in-memory merging before MapOutputBuffer.
Why Federation?
K-means implemented with DistributableStream.

Input: 45GB raw data representing one billion vertices.

Hadoop MR for parsing raw input, \{Hadoop MR, Coherence\} for iterations, Local SMP for updating centroids and evaluating termination condition.

Most disk IOs are avoided during the job execution by caching the input data and all intermediate results in the OS cache.
Iterations w/ Hadoop MR is slower, $2.3 \times$ to $3.1 \times$. Accessing OS cache is slower than Java heap. In-memory Java objects avoid deserialization cost.
Next?
More Data Sources, such as Databases.

More Compute Engines, such as Apache Spark and Tez.
Long Term

Job Planner and Optimizer, for automatical engine assignment.

Job Progress Monitor, for fault tolerance across engines.

Java JIT Optimization, for low-level JVM tuning.

API Extension, for supporting DAGs.
We need your help!

A JSR (Java Specification Request). An OpenJDK project.
DistributableStream is an abstraction that supports generic, distributed and federated queries on top of an extensible set of compute engines.

You are welcome to make contributions!
Thank You!

Questions?
Backup Slides
A wrapper DistributableStream implementation that wraps a Java 8 Stream inside.

Operations are delegated to the wrapped Stream.
Pipelining several map phases inside a single mapper similar to the **ChainMapper** but w/o the need for configuring parameters for each phase separately.

Partial in-memory merging before MapOutputBuffer.
Portable object format (POF).

Invocation services.

Use of filter and partition affinity for performance.
By Engine instance methods over a persistent engine specific data set, e.g.,
MapReduceEngine.valueStream(conf)

From an Engine specific distributed collection, e.g.,
coherenceDistMap.entryStream()

Use the result of
DistributableStream.collectToStream(collector)