BigData TECHCON

Spark

Spark Streaming

BigData Techcon SF

October 28, 2014

Dean Wampler, Ph.D

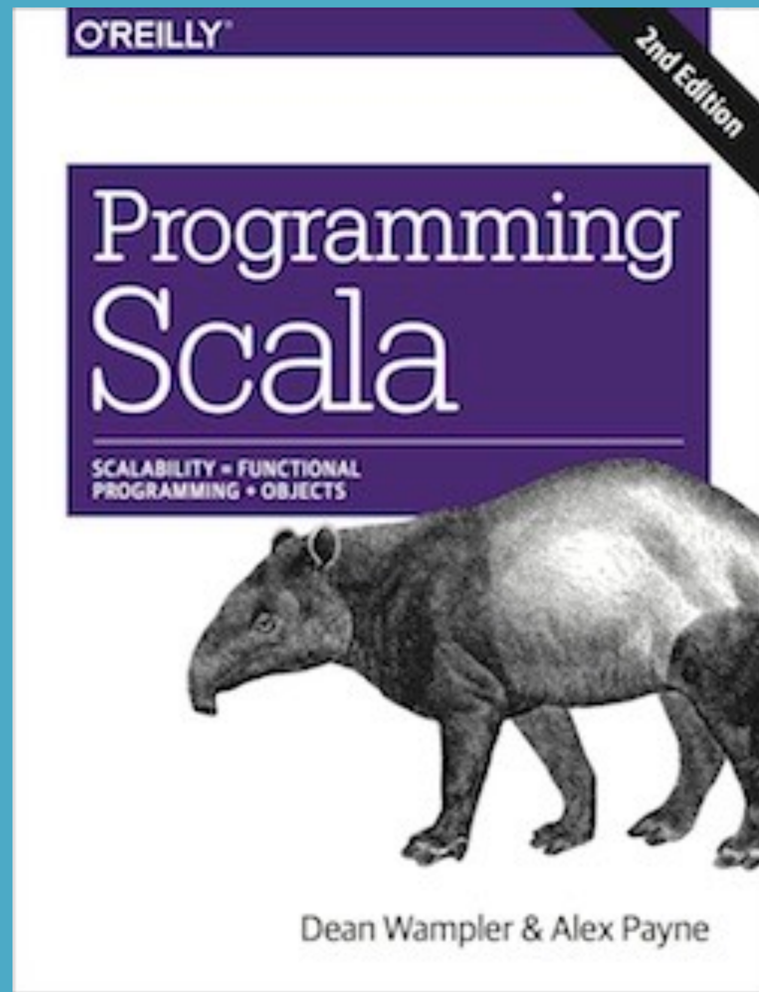
dean.wampler@typesafe.com

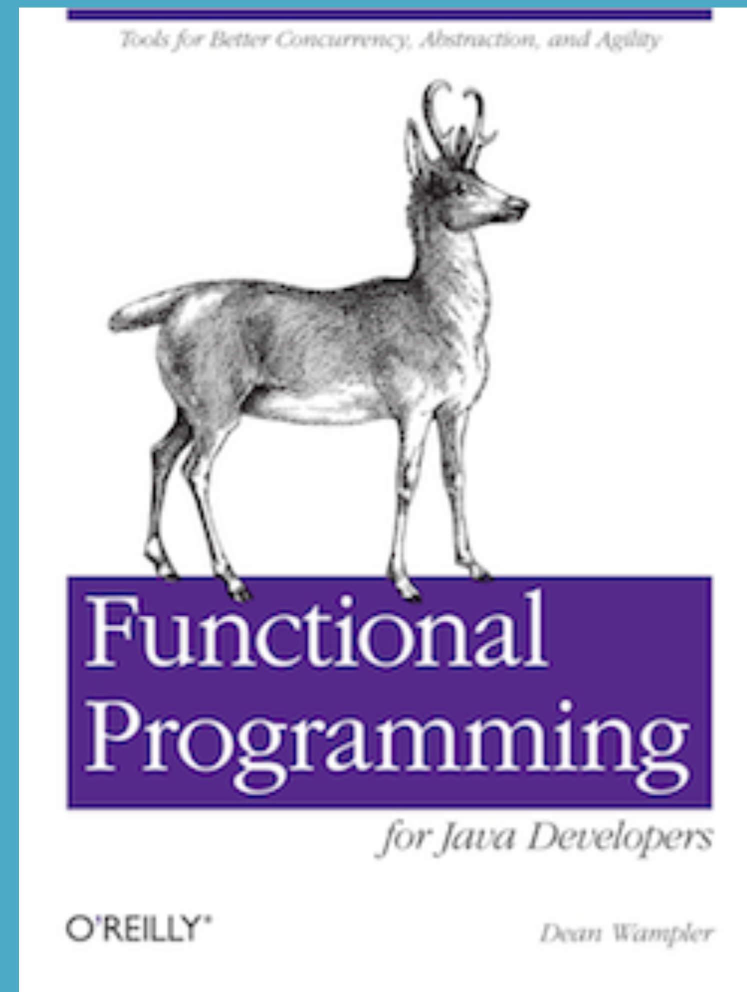
Typesafe

Monday, October 27, 14

The Spark Logo is the property of the Apache Foundation.

Photo: Fog on the Fyris River, Uppsala, Sweden





dean.wampler@typesafe.com
polyglotprogramming.com/talks
@deanwampler

Typesafe provides products and services for building Reactive, Big Data applications

typesafe.com/reactive-big-data

Outline

- Introduction: Quick Spark overview
- Event-stream Processing in Spark
- Internals of Spark Streaming
- Connecting to Other Kinds of Sources

We'll take periodic 10-15 minute breaks and a 45-minute lunch break.

Monday, October 27, 14

Photo: Fog and Bridge on the Fyris River, Uppsala, Sweden

Why Spark?

Monday, October 27, 14

Why Spark? (1/2)

- Flexible, composable programming model.
- Concise, powerful API.
- Excellent performance for complex jobs.
- Supports event-streaming applications.

Monday, October 27, 14

Most of the major Hadoop vendors have embraced Spark as the replacement for MapReduce because it is strong in all the ways that MapReduce is weak.

Why Spark? (2/2)

- Efficient support for iterative, machine learning, and graph algorithms.
- Scales from a single laptop to a large cluster.

Monday, October 27, 14

You'll continue to write large-Hadoop applications as before, but much more easily and with better performance, but Spark also opens new options.

Why Scala?

- Spark is implemented in Scala.
- Scala is ideal for Big Data applications.

  – http://www.hakkalabs.co/articles/three-reasons-data-eng-learn-scala
  – http://polyglotprogramming.com/papers/WhyScalaIsTakingOverTheBigDataWorld.pdf (PDF)

I've been advocating for Scala as the best tool for this type of work for a while. It's true that Java 8 has added some of the important features needed, like lambdas and more functionally-oriented collection semantics, but Java's backwards compatibility constraints limit what it can do to fully embrace functional programming and approach the concision of Scala. To be fair, Clojure would be just as good an option for the JVM, but it hasn't seen the uptake or library support that Scala has seen. As the blog post in the first link says, the subset of Scala you'll use most of the time has the same concise, approachable feel that languages like Python provide, but with much better performance at scale.

Why Scala?

- Spark is implemented in Scala.
- Scala is ideal for Big Data applications.

- ... but you can use Java, Python, and R (forthcoming) if you prefer.

Typesafe

Monday, October 27, 14

I've been advocating for Scala as the best tool for this type of work for a while. It's true that Java 8 has added some of the important features needed, like lambdas and more functionally-oriented collection semantics, but Java's backwards compatibility constraints limit what it can do to fully embrace functional programming and approach the concision of Scala. To be fair, Clojure would be just as good an option for the JVM, but it hasn't seen the uptake or library support that Scala has seen. As the blog post in the first link says, the subset of Scala you'll use most of the time has the same concise, approachable feel that languages like Python provide, but with much better performance at scale.

Spark History

Over the last twenty years, Internet giants like Amazon, Google, Yahoo!, eBay, and Twitter invented new tools for working with data sets of unprecedented size, far beyond what traditional tools could handle. They started the _Big Data_ revolution, characterized by the ability to store and analyze these massive data sets with acceptable performance, at drastically reduced costs.

Spark History

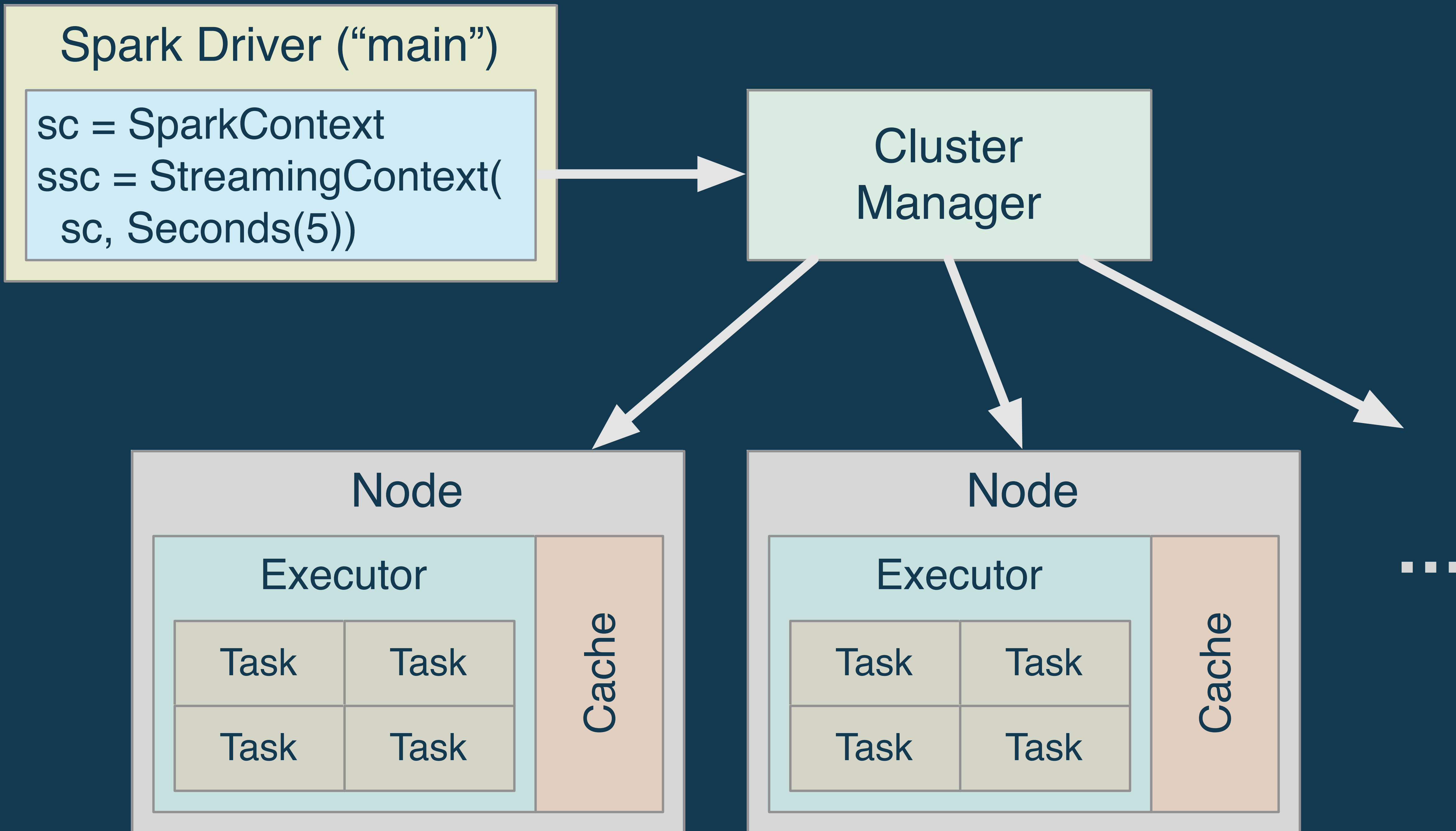
- Started in 2009 as a Berkeley AMPLab research project.
  - Matei Zaharia's Ph.D. research.
- Part of the BDAS project.
  - https://amplab.cs.berkeley.edu/software/
- Now a top-level Apache project:
  - http://spark.apache.org

Spark Concepts

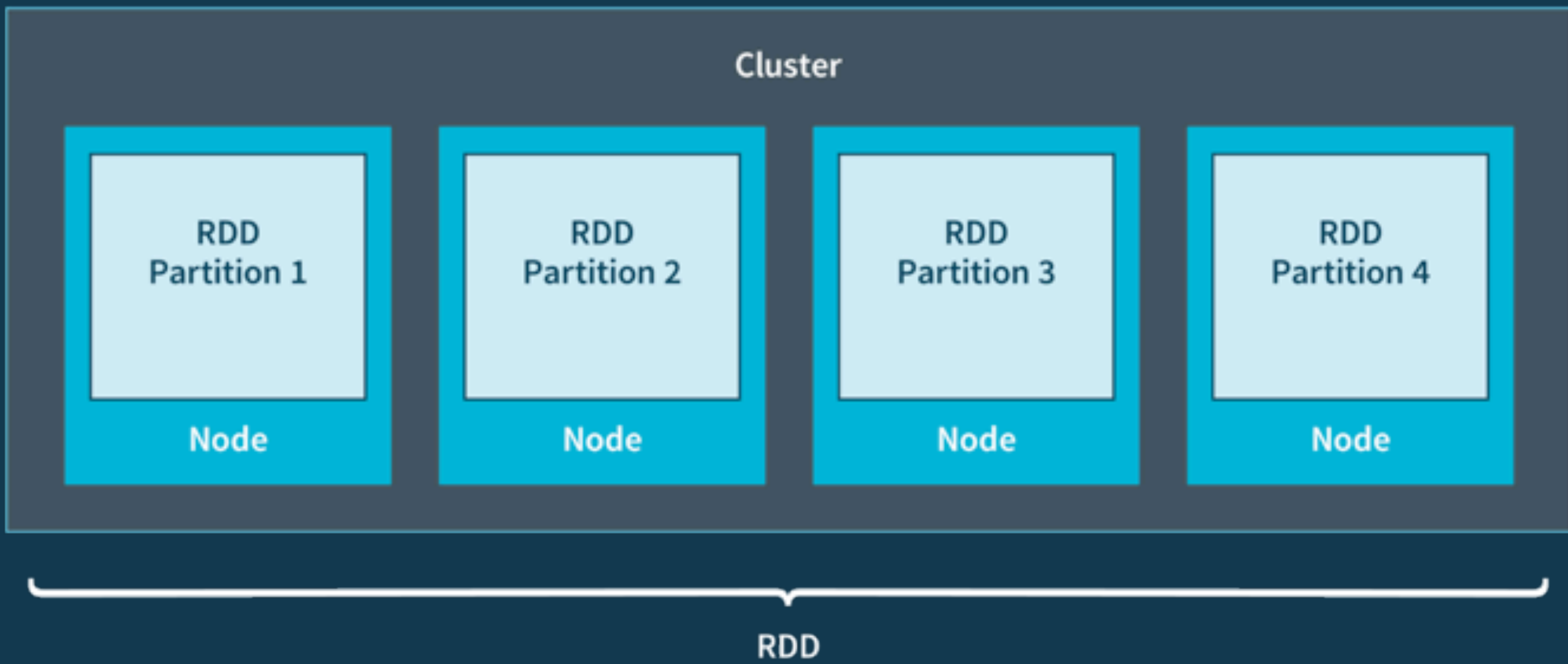
Spark Clusters

- Several Deployment Modes
  - Standalone
  - Mesos
  - Hadoop YARN
  - EC2
  - Cassandra (and other DBs soon?)
- (We'll discuss details later.)

Spark Clusters



Resilient Distributed Datasets



Spark's core abstraction!

Reference Links

- Spark Documentation
  - http://spark.apache.org/docs/latest/
- Spark "Scaladocs"
  - http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.package
- Research Paper on Spark
  - https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

Typesafe

Reference Links

- Spark Streaming Programming Guide
  - http://spark.apache.org/docs/latest/streaming-programming-guide.html
- Spark Distro Streaming Examples
  - https://github.com/apache/spark/tree/master/examples/src/main/scala/org/apache/spark/examples/streaming
- Chris Fregly's talks on Slideshare

Scaladocs

http://spark.apache.org/docs/latest/api/scala

Monday, October 27, 14

Spark Core API: Crash Course

Monday, October 27, 14

Photo: Trail on the Fyris River, Uppsala, Sweden

Word Count

Load a corpus of documents (in parallel), tokenize into words, count the occurrence of each word.

Typesafe

This algorithm is the “hello world” of the Big Data world, because it’s easy to understand, so you can focus on learning an API and how it works.

WordCount.scala

Adapted from Typesafe training...

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
object WordCount {
 def main(args: Array[String]) = {
   val inputPath  = args(0) // e.g., "data/corpus"
   val outputPath = args(1) // e.g., "output/word-count"
   val master     = args(2) // e.g., "local[*]", “yarn-client”
   val sc = new SparkContext(master, "Word Count")
   try {
    ...
   } finally {
    sc.stop
   }
 }
}
```

WordCount.scala

```scala
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
object WordCount {
  def
    val inputPath  = args(0) // e.g., "data/corpus"
    val outputPath = args(1) // e.g., "output/word-count"
    val master     = args(2) // e.g., "local[*]", "yarn-client"
    val sc = new SparkContext(master, "Word Count")
    try {
     ...
    } finally {
     sc.stop
    }
  }
}
```

Common imports for all core apps

WordCount.scala

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
object WordCount {
 def main(args: Array[String]) = {
   val inputPath  = args(0) // e.g., "data/corpus"
   val                                 utput/word-count"
   val                                 ocal[*]", “yarn-client”
   val                                 ord Count")
   try {
    ...
   } finally {
    sc.stop
   }
 }
}
```

Singleton object, used to hold main

WordCount.scala

```scala
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
object WordCount {
 def main(args: Array[String]) = {
   val inputPath  = args(0) // e.g., "data/corpus"
   val outputPath = args(1) // e.g., "output/word-count"
   val master     = args(2) // e.g., "local[*]", “yarn-client”
   val sc = new SparkContext(master, "Word Count")
   try {
    ...
   } finally {
    sc.stop
   }
 }
}
```

Specify input, output, and which Spark “master” to use.

WordCount.scala

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
object WordCount {
 def main(args: Array[String]) = {
   val inputPath  = args(0) // e.g., "data/corpus"
   val outputPath = args(1) // e.g., "output/word-count"
   val master     = args(2) // e.g., "local[*]", “yarn-client”
   val sc = new SparkContext(master, "Word Count")
   try {
    ...
   } finally {
    sc.stop
   }
 }
}
```

Construct a SparkContext

WordCount.scala

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
object WordCount {
 def main(args: Array[String]) = {
   val inputPath  = args(0) // e.g., "data/corpus"
   val outputPath = args(1) // e.g., "output/word-count"
   val master     = args(2) // e.g., "local[*]", "yarn-client"
   val sc = new SparkContext(master, "Word Count")
   try {
    ...
   } finally {
    sc.stop
   }
 }
}
```

Use try/finally for cleanup

WordCount.scala

```scala
try {
  val input = sc.textFile(inputPath)
    .map(line => line.toLowerCase)
  val wc = input
    .flatMap(line => line.split("""\W+"""))
    .map(word => (word, 1))
    .reduceByKey((n1, n2) => n1 + n2)
  wc.saveAsTextFile(outputPath)
}
```

WordCount.scala

```
try {
  val input = sc.textFile(inputPath)
    .map(line => line.toLowerCase)
  val wc = input
    .flatMap(l
    .map(word
    .reduceByKey((n1, n2) => n1 + n2)
  wc.saveAsTextFile(outputPath)
}
```

Input the lines and convert to lower case.

WordCount.scala

```scala
try {
  val input = sc.textFile(inputPath)
    .map(line => line.toLowerCase)
  val wc = input
    .flatMap(line => line.split("""\W+"""))
    .map(word => (word, 1))
    .reduceByKey((n1, n2) =>
  wc.saveAsTextFile(outputPath)
}
```

Split into words

WordCount.scala

```scala
try {
  val input = sc.textFile(inputPath)
    .map(line => line.toLowerCase)
  val wc = input
    .flatMap(line => line.split("""\W+"""))
    .map(word => (word, 1))
    .reduceByKey((n1, n2) => n1 + n2)
  wc.saveAsTextFile(
}
```

Create tuples with words and seed counts.

WordCount.scala

```scala
try {
  val input = sc.textFile(inputPath)
    .map(line => line.toLowerCase)
  val wc = input
    .flatMap(line => line.split("""\W+"""))
    .map(word => (word, 1))
    .reduceByKey((n1, n2) => n1 + n2)
  wc.saveAsTextFile(outputPath)
}
```

Group equal words and sum counts

WordCount.scala

```scala
try {
  val input = sc.textFile(inputPath)
    .map(line => line.toLowerCase)
  val wc = input
    .flatMap(line => line.split("""\W+"""))
    .map(word => (word, 1))
    .reduceByKey((n1, n2) => n1 + n2)
  wc.saveAsTextFile(outputPath)
}
```

Can be abbreviated:

_ + _

WordCount.scala

```
try {
  val input = sc.textFile(inputPath)
    .map(line => line.toLowerCase)
  val wc = input
    .flatMap(line => line.split("""\W+"""))
    .map(word => (word, 1))
    .reduceByKey((n1, n2) => n1 + n2)
  wc.saveAsTextFile(outputPath)
}
```

Write (word,n) results

Demo

Look at the Output

- For some input corpus and output location:

```
$ ls -o output/word-count
-rw-r--r--  1 me       0 Aug 27 12:08 _SUCCESS
-rw-r--r--  1 me  156502 Aug 27 12:08 part-00000
-rw-r--r--  1 me  166101 Aug 27 12:08 part-00001
$ head output/word-count/part-00000
(some,21)
(words,3)
(that,4)
(were,2)
(found,10)
...
$
```


The output path is interpreted as a directory, following Hadoop conventions. A marker file _SUCCESS is written after output to the “partition” files was completed and one or more partition files with the actual data.

Scalable Abstractions

- The following alternative code works for both Scala collections and Spark!

```
...
  .map(line => line.toLowerCase)
  .flatMap(line => line.split("""\W+"""))
  .groupBy(word => word)
  .map { case (word, group) => (word, group.size) }
...
```

Typesafe

Monday, October 27, 14

Another implementation, which works IDENTICALLY for both Spark and the normal Scala Collections API. However, it is actually inefficient, because you build up these groups of stuff only to simply size each group and then discard it. The Spark API adds the methods we've used in the examples, which are more efficient for the special case of counting things.

Spark Streaming

Monday, October 27, 14

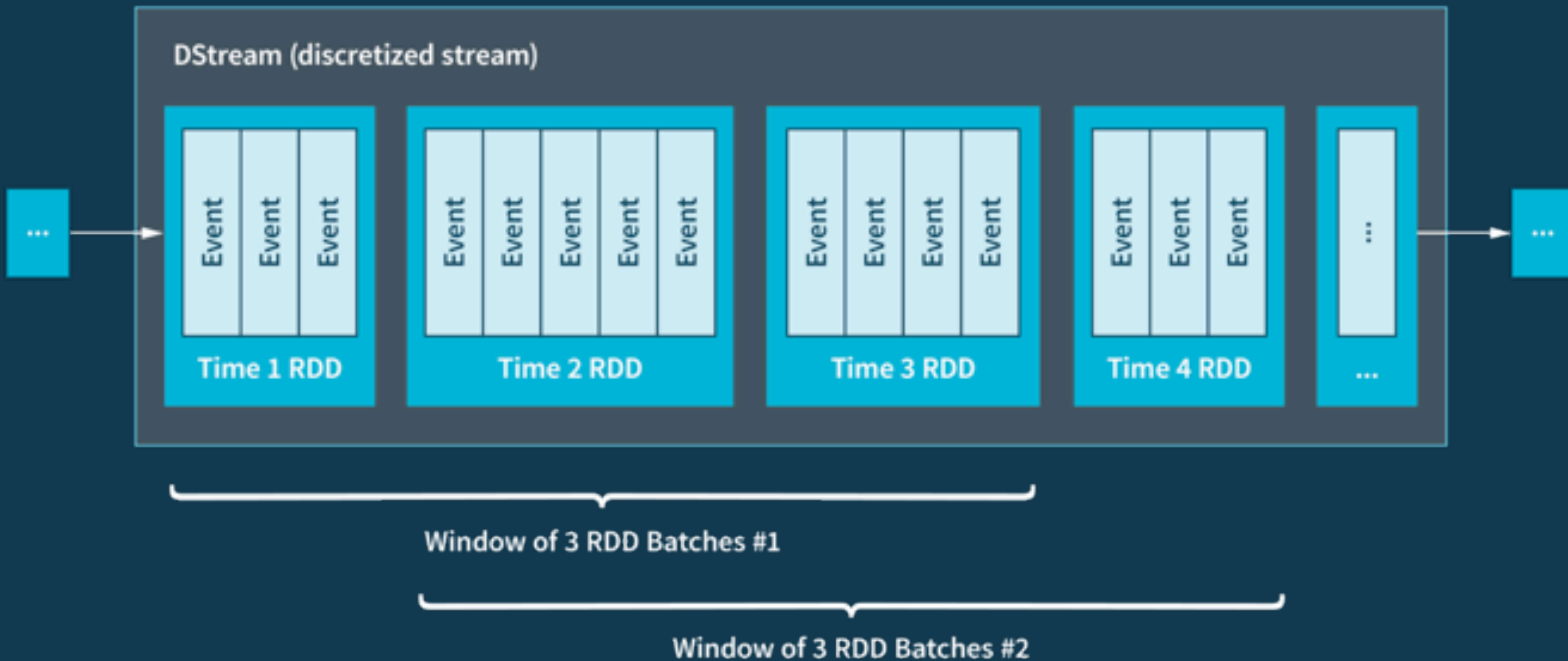
Photo: Restaurant and Cathedral on the Fyris River, Uppsala, Sweden

Event Handling

- Spark Streaming captures time slices of events.
  - DStream (discretized stream): sequence of batches.
    - A Receiver listens for the data.
  - Batch: one time interval of data, stored in an RDD.
  - Time intervals typically 1/2 to 60+ seconds.
    - Depends on the rate of data, etc.

A Receiver connects to the data source and handles input. EACH RECEIVER REQUIRES A DEDICATED CORE!

Event Handling



Event Handling

- API
  - All the RDD functions, plus "window" functions through the DStream wrapper.

Use Cases

- ETL Pipeline
  - Ingest, cleanse, transform data for storage or downstream processing.
- Trends, "Online" Machine Learning
  - Train models as data arrives.
  - Running trends and statistics.
- Component of the Lambda Architecture

Spark Streaming vs. Storm?

- Spark Streaming:
  - Doesn't handle individual events.
  - ✓ Rich operations over batches.
- Storm, message queues, etc.
  - ✓ Per-event handling.
  - Limited operations on events.

Event Sources (1/2)

We'll try the first two.

- Watch a directory for new files.
  - Existing files are ignored!
  - New files must appear atomically, i.e., by moving them there. Don't "slow write" these files!
- Read a socket.
- Create a DStream from a queue of RDDs (for testing). See streamingContext.queueStream.
- Read messages from an Akka actor.

Monday, October 27, 14

The streaming subpackages contain utilities for ingesting streams from the listed sources. More will probably be added in subsequent releases. When you use a queue of RDDs, they are treated as the batches to process.

Event Sources (2/2)

- Advanced sources: Sub-packages of org.apache.spark.streaming:
  - Flume, Kafka, Kinesis, MQTT, Twitter, ZeroMQ, ...
  - Each requires a separate jar. See http://spark.apache.org/docs/latest/streaming-programming-guide.html#linking
  - Implement your own custom receiver!
    - See http://spark.apache.org/docs/latest/streaming-custom-receivers.html

Monday, October 27, 14

The streaming subpackages contain utilities for ingesting streams from the listed sources. More will probably be added in subsequent releases.

SparkStreaming.scala

Just the interesting bits.

```scala
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming.dstream.{
  InputDStream, DStream}
import org.apache.spark.streaming.scheduler.{
  StreamingListener, StreamingListenerReceiverError,
  StreamingListenerReceiverStopped}
...
```

imports...

SparkStreaming.scala

Just the interesting bits.

```
...
  val sc  = new SparkContext(...)
  val ssc = new StreamingContext(sc, Seconds(1))
...
```

A StreamingContext that specifies the batch size in seconds.

SparkStreaming.scala

```scala
...
  object EndOfStreamListener extends StreamingListener {
    override def onReceiverError(
        error: StreamingListenerReceiverError):Unit = {
      println(s"Receiver Error: $error. Stopping...")
      shutdown()
    }
    override def onReceiverStopped(
        stopped: StreamingListenerReceiverStopped):Unit ={
      println(s"Receiver Stopped: $stopped. Stopping...")
      shutdown()
    }
  }
  ssc.addStreamingListener(new EndOfStreamListener(ssc))
...
```

Stream event listener.

SparkStreaming.scala

```scala
...
  try {
    val lines =
      if (useDirectory) ssc.textFileStream(inputPath)
      else ssc.socketTextStream(hostname, port)

      // Word Count...
      val words = lines.flatMap(line => line.split("\\W+"))
      val pairs = words.map(word => (word, 1))
      val wordCounts = pairs.reduceByKey(_ + _)

      wordCounts.print()  // print a few counts...
      ...
```

SparkStreaming.scala

```scala
...
  try {
    val lines =
      if (useDirectory) ssc.textFileStream(inputPath)
      else ssc.socketTextStream(hostname, port)

      // Word Count...
      val words = lines.flatM
      val pairs = words.map(word => (word, 1))
      val wordCounts = pairs.reduceByKey(_ + _)

      wordCounts.print()  // print a few counts...
      ...
```

Read either from new files in a directory or from a socket.

SparkStreaming.scala

```scala
...
  try {
    val lines =
      if (useDirectory) ssc.textFileStream(inputPath)
      else ssc.socketTextStream(hostname, port)

      // Word Count...
      val words = lines.flatMap(line => line.split("\\W+"))
      val pairs = words.map(word => (word, 1))
      val wordCounts = pairs.reduceByKey(_ + _)

      wordCounts.print()  // Word count on each batch!
      ...
```


Monday, October 27, 14

SparkStreaming.scala

```scala
...
  try {
    val lines =
      if (useDirectory) ssc.textFileStream(inputPath)
      else ssc.socketTextStream(hostname, port)

      // Word Count...
      val words = lines.flatMap(line => line.split("\\W+"))
      val pairs = words.map(word => (word, 1))
      val wordCounts = pairs.reduceByKey(_ + _)

      wordCounts.print()  // print a few counts...
      ...
```

Diagnostic console messages

SparkStreaming.scala

```scala
...
  try {
    ...
    // Generates a separate subdirectory for each interval!!
    wordCounts.saveAsTextFiles(out, "txt")

    ssc.start()
    ssc.awaitTermination()
  } finally {
    ssc.stop()
  }
}
...
```

SparkStreaming.scala

```scala
...
  try {
    ...
    // Generates a separate subdirectory for each interval!!
    wordCounts.saveAsTextFiles(out, "txt")

    ssc.start()
    ssc.awaitTermination()
  } finally {
    ssc.stop()
  }
}
...
```

Write the output on each batch iteration

SparkStreaming.scala

```
...
  try {
    ...
    // Generates a separate subdirectory for each interval!!
    wordCounts.saveAsTextFiles(out, "txt")

    ssc.start()
    ssc.awaitTermination()
  } finally {
    ssc.stop()
  }
}
...
```

Explicitly start the pipeline and wait for it to terminate.

SparkStreaming.scala

```
...
  try {
    ...
    // Generates a separate subdirectory for each interval!!
    wordCounts.saveAsTextFiles(out, "txt")

    ssc.start()
    ssc.awaitTermination()
  } finally {
    ssc.stop()
  }
}
...
```

Shutdown both the Streaming and Spark contexts.

SparkStreaming.scala

- You can also use SparkSQL combined with Spark Streaming!

```scala
import org.apache.spark.sql.SQLContext
...
val sqlc = new SQLContext(sc)
import sqlc._
...
wordCount.window(Seconds(window), Seconds(slide))
  .foreachRDD { wordcount =>
    wordcount.registerTempTable("wordcount")
    val topWCs = sql("
      SELECT * FROM wordcount ORDER BY _2 DESC LIMIT 10")
  }
```

Demo

Internals

Photo: Pond surface, Virginia Tech campus, Blacksburg, Va., USA

Use of Cores

Core Allocation

- Except for when reading from files, allocate at least one core per DStream, plus 1 additional core.
  - Use local[*], when possible.
- See http://spark.apache.org/docs/latest/streaming-programming-guide.html#input-dstreams

Monday, October 27, 14

Processing each stream will consume a core, so you need another one for the rest of your computation!
Recall from a previous slide that each DStream has a Receiver object that handles connecting to the data source and ingesting data. It's the object that needs a dedicated core. Hence if you have several DStreams/Receivers, you'll need that many cores, plus at least one more for the other processing.

However, using file stream doesn't require a receiver, so it doesn't require a dedicated core per DStream.

Core Failover

- Actually, you need additional standby cores for failover.
- If a core is lost (e.g., a node is lost), Spark will reconstruct the stream on another core.
- This core is now unavailable for other work.

- Can lead to livelock!

See this blog post about failure handling and the number of cores. http://metabroadcast.com/blog/design-your-spark-streaming-cluster-carefully

Checkpointing and Other Control Operations

RDD Control vs. Transformation Methods

- We've used the transformation methods:
  - E.g., map, flatmap, filter, and more to come.
  - (Mostly) lazy execution.
- There are also control methods:
  - E.g., cache, checkpoint, persist, unpersist, and coalesce.
  - Also dependencies, getCheckpointFile, getStorageLevel, isCheckpointed, repartition,

"Transformation" and "Control" are my terms. The boundaries are loose, but the intent is to categorize methods that focus on the data (transform) vs. those that manipulate how the system stores the data.
I won't talk about all of these...

RDD Control Methods

- RDD.persist(storageLevel)
  - –Store depending on storage level
    - Default - MEMORY_ONLY.
    - Other options include memory and disk, off-heap (Tachyon) and custom.
  - –Performed when the RDD is computed.
  - –Remember lineage, so lost partitions can be reconstructed.

```
scala> tupleVects.persist(MEMORY_AND_DISK)
```

RDD Control Methods

- RDD.unpersist
  - Remove from storage (memory, disk, etc.).

```
scala> tupleVects.unpersist()
```

Typesafe

RDD Control Methods

- RDD.cache
  - Identical to RDD.persist(MEMORY_ONLY)
  - Keep in memory (spill bits to disk if too big).
  - Remember lineage, so lost partitions can be reconstructed.

```
scala> tupleVects.cache
```

RDD Control Methods

- RDD.checkpoint
  - Materialize an RDD (once evaluated).
  - Save to the file system (e.g., HDFS).
  - Forgets the lineage
    - Truncating long lineages important for streaming. (more on that in the streaming section.)
  - Call before computing with the RDD.

```
scala> sc.setCheckpointDir("output/checkpoints")
scala> tupleVects.checkpoint
```


Monday, October 27, 14

Setting the checkpoint dir is required. There isn't a default.

RDD Control Methods

- RDD.checkpoint
  - The files are used to reconstruct lost partitions.
  - Delete these files when no longer needed!
    - Done automatically in Streaming, but not for non-streaming apps.

Streaming Resiliency

- Since streams don't have source data files to reconstruct lost RDD partitions (in the general case...), streaming uses aggressive checkpointing and in-memory data replication to improve resiliency.
  - Frequent checkpointing keeps RDD lineages down to a reasonable size.
  - Otherwise they build up quickly as batches are processed.

Stream checkpointing

- For DStreams, checkpoint
  - also writes a metadata file, e.g., which RDD file names correspond to the DStream.
  - The metadata checkpoint is written for every batch.
  - The data checkpoint is configured with DStreams.checkpoint(duration).
    - duration must be n*batch interval for some n.
  - ...

Stream checkpointing

- Without the call to DStreams.checkpoint (duration):
  - If the batch interval is > 10 seconds, checkpointing defaults to every batch.
  - < 10 seconds, defaults to closest n, where n*batch interval > 10 seconds.

So, whether or not the default it used, checkpointing always happens at every one or more batch intervals.

Resilience Characteristics of Different Sources

Buffered Streams

- Supports some replay and flow control (backpressure).
- Examples: Flume, some message queues, Akka

Batched Streams

- Improved throughput.
- May get repeated data for error-recovery replays.
- Examples: Flume, some message queues.

Streams with Checkpointing

- (The stream source itself has a checkpoint capability.)
- Replay from a checkpoint.
- Examples: Some message queues.

Other Resiliency Tips

Read from HDFS

- Stream from files staged in HDFS.
- Rely on the resiliency of HDFS.
- The files are your data backups.
- Delete the files when you know the data is processed.

Replication

- Replicate sources.
- Replicate listeners.

Checkpoints

- Tune frequency of checkpoints.
  - Frequent checkpoints keep a current "backup", but increase overhead.
- Use a reliable file system (e.g., HDFS).
- But purge old files to conserve space.

Node Crash??

- Recall that RDDs will reconstruct lost partitions.
  - Using checkpoint files.
  - Using original file sources and RDD lineage, when available.

Tuning Tips

Tuning Batch Interval

- Keep job processing time per batch less than the batch interval.
- Choose interval to trade off:
  - Lower latency (smaller)
  - Lower overhead (higher)

Tuning Checkpoint Interval

- Rule of thumb: ~5-10x the batch interval.
- Choose interval to trade off:
  - Lower data loss (smaller)
  - Lower overhead (higher)

Tuning Number of Input Streams

- To improve throughput, can you break large streams into smaller ones, processed in parallel?
  - Replace one stream for two Message Queue topics into two streams with one topic each.

Others

- Use RDD.repartition to adjust the parallelism.
  - Higher parallelism leads to lower "wall clock" time,
  - ... but watch for Amdahl's law.
- Property spark.streaming.unpersist=true
  - Let runtime decide how to manage persistence.

Data Source Integration

Typesafe

Monday, October 27, 14

Supported Integrations

- Link in separate modules:
  - Kafka - spark-streaming-kafka_2.10
  - Flume - spark-streaming-flume_2.10
  - Kinesis - spark-streaming-kinesis-asl_2.10
  - Twitter - spark-streaming-twitter_2.10
  - ZeroMQ - spark-streaming-zeromq_2.10
  - MQTT - spark-streaming-mqtt_2.10
- See the examples maven pom.xml.

Monday, October 27, 14

The Kinesis module is covered by the Apache Software License, too.

The last bullet refers to the Spark distribution's examples.

For Example: Kafka

- Steps:
  - Add spark-streaming-kafka_2.10 to your dependencies (Maven, SBT, etc.)
  - Use the KafkaUtils:

```
import org.apache.spark.streaming.kafka._

val kafkaStream = KafkaUtils.createStream(
  streamingContext, zookeeperQuorum,
  groupIdOfConsumer, perTopicNumberOfKafkaPartitionsToConsume)
```

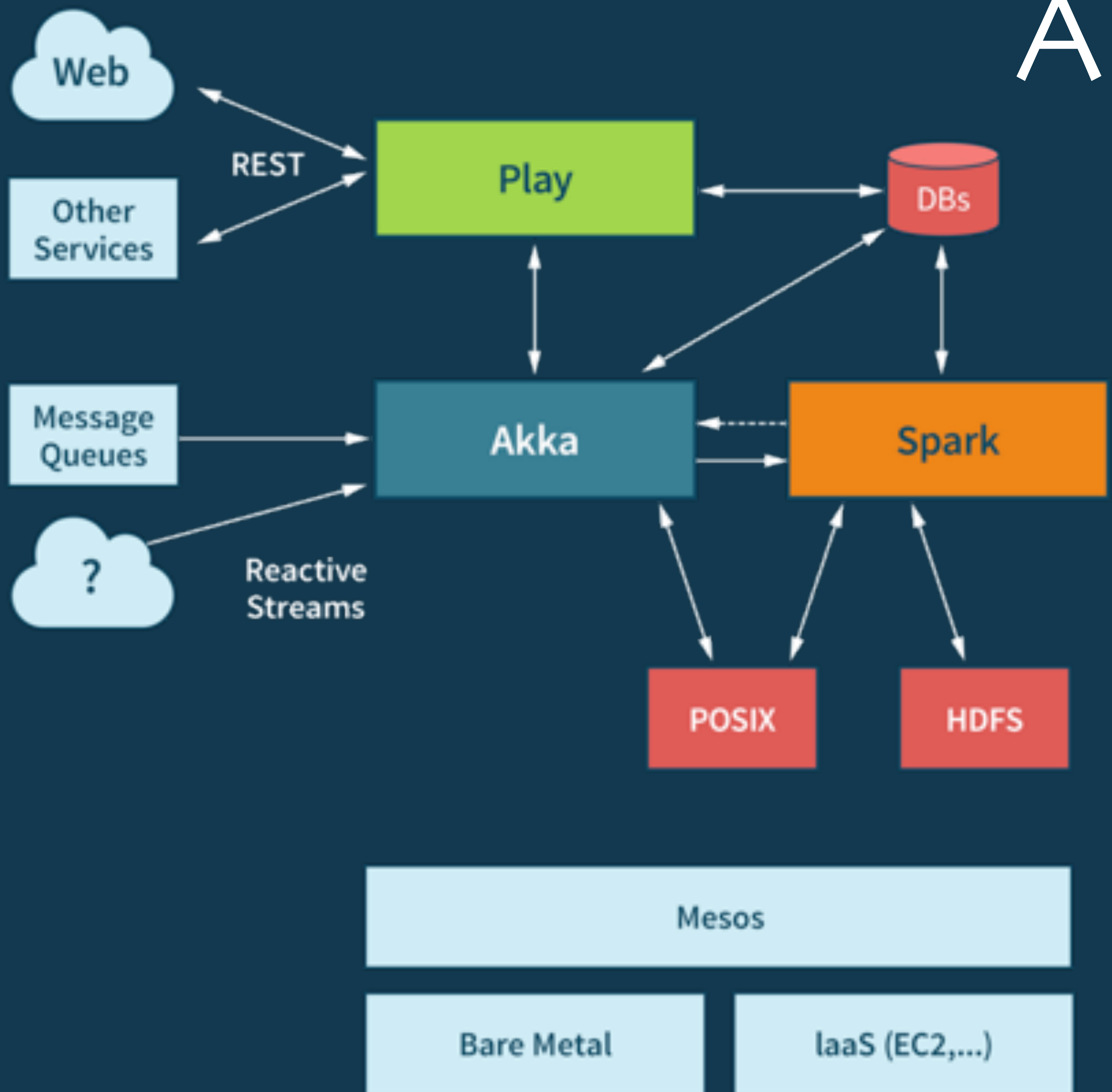
  - Link jar with your code, Kafka library, and transitive dependencies.

For Further Information

- Kafka Integration:
  - http://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/

Spark Streaming and the Typesafe Reactive Platform

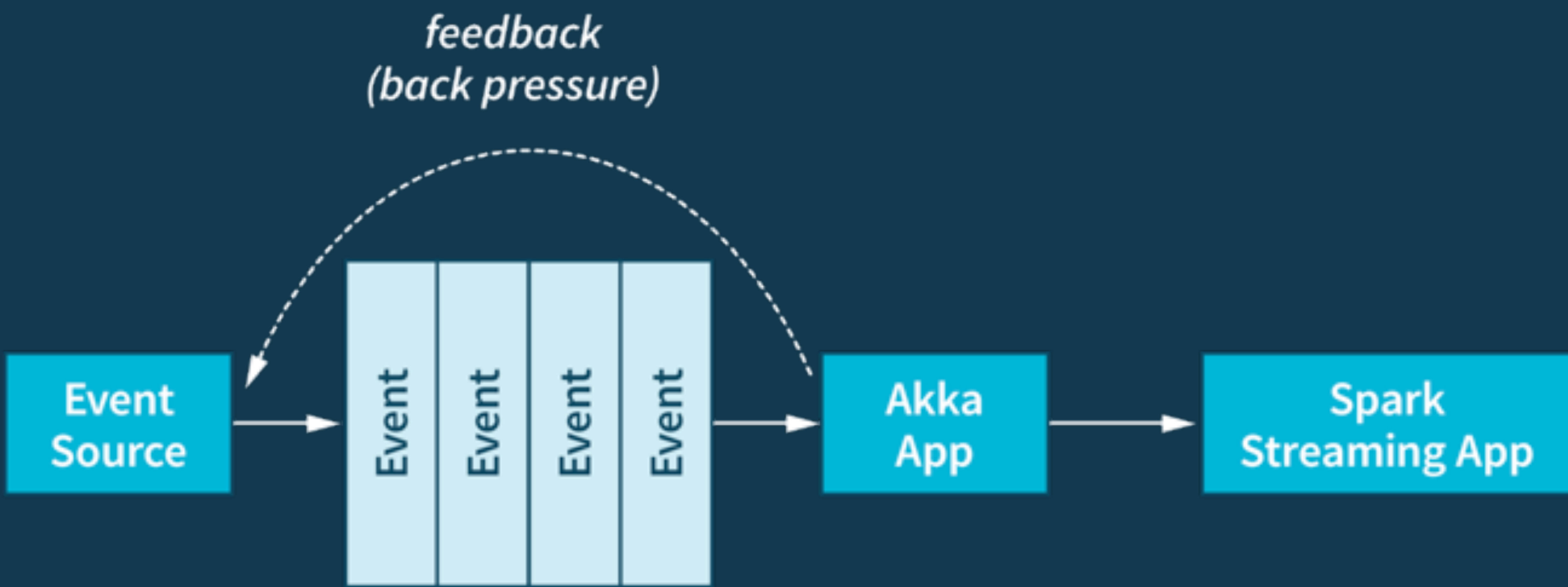
Event Streaming with Akka + Spark



Monday, October 27, 14

A particular scenario: you have an event-handling system written in Akka, which is adding support for "reactive streams", event streams with dynamic flow control through feedback from the consumer to the producer ("backpressure"). Akka can manage these data flows and implement many tasks, but when you want more sophisticated data analytics, such as incrementally training machine learning algorithms, analyzing data graphs, working with structured ("semi-relational") data, then Akka can stream events to Spark for these "on-line" (real-time) analytics. You can also run separate, batch-mode spark apps for offline analytics.

Akka Streams + Spark Streaming



Monday, October 27, 14

Spark Streaming uses the RDD model, but wraps RDDs into a "discretized stream", DStream. Each time slice, (typically 1 or a few seconds) contains all the events received in that time delta, stored in an RDD. You get all the usual RDD operations, plus window functions, such as computing the moving average of some value over the last three deltas, as shown.

Monday, October 27, 14

Photo: San Francisco Bay

Spark Streaming

- Flexible computation model for batch mode and streaming.
- High performance.
- Integrates with a variety of sources.
- Flexibly clustering options.

Typesafe

http://typesafe.com/reactive-big-data

dean.wampler@typesafe.com