Recipes for Running Spark Streaming Apps in Production

Tathagata "TD" Das

@tathadas

Spark Summit 2015

databricks™

Spark Streaming

Scalable, fault-tolerant stream processing system

High-level API

joins, windows, …
often 5x less code

Fault-tolerant

Exactly-once semantics,
even for stateful ops

Integration

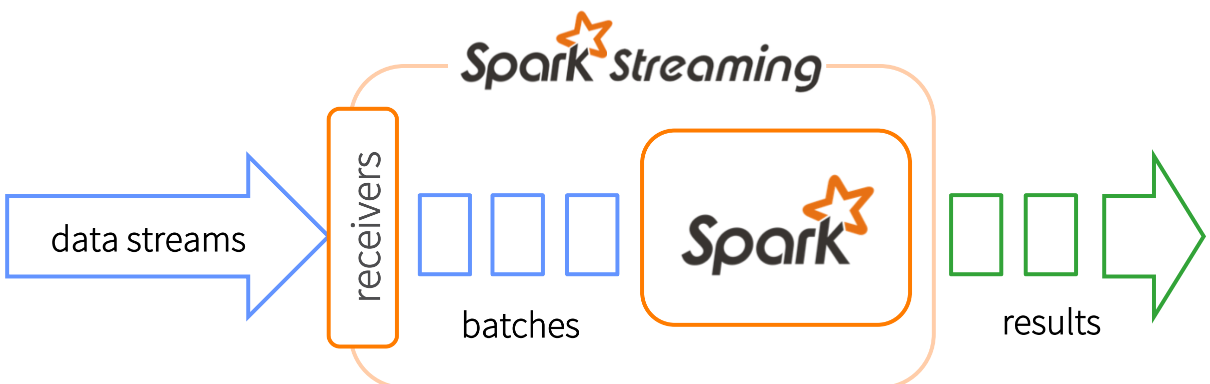
Integrates with MLlib, SQL,
DataFrames, GraphX

databricks

Spark Streaming

Receivers receive data streams and chop them up into batches

Spark processes the batches and pushes out the results



databricks

Word Count with Kafka

```
val context = new StreamingContext(conf, Seconds(1))

val lines = KafkaUtils.createStream(context, ...)
```

entry point of streaming functionality

create DStream from Kafka data

databricks

Word Count with Kafka

```
val context = new StreamingContext(conf, Seconds(1))

val lines = KafkaUtils.createStream(context, ...)

val words = lines.flatMap(_.split(" "))
```

split lines into words

databricks™

Word Count with Kafka

```
val context = new StreamingContext(conf, Seconds(1))

val lines = KafkaUtils.createStream(context, ...)

val words = lines.flatMap(_.split(" "))

val wordCounts = words.map(x => (x, 1))
                      .reduceByKey(_ + _)

wordCounts.print()

context.start()
```

count the words

print some counts on screen

start receiving and transforming the data

databricks

Word Count with Kafka

```scala
object WordCount {
  def main(args: Array[String]) {
    val context = new StreamingContext(new SparkConf(), Seconds(1))
    val lines = KafkaUtils.createStream(context, ...)
    val words = lines.flatMap(_.split(" "))
    val wordCounts = words.map(x => (x,1)).reduceByKey(_ + _)
    wordCounts.print()
    context.start()
    context.awaitTermination()
  }
}
```

Got it working on a small Spark cluster on little data

What's next??

How to get it production-ready?

Fault-tolerance and Semantics

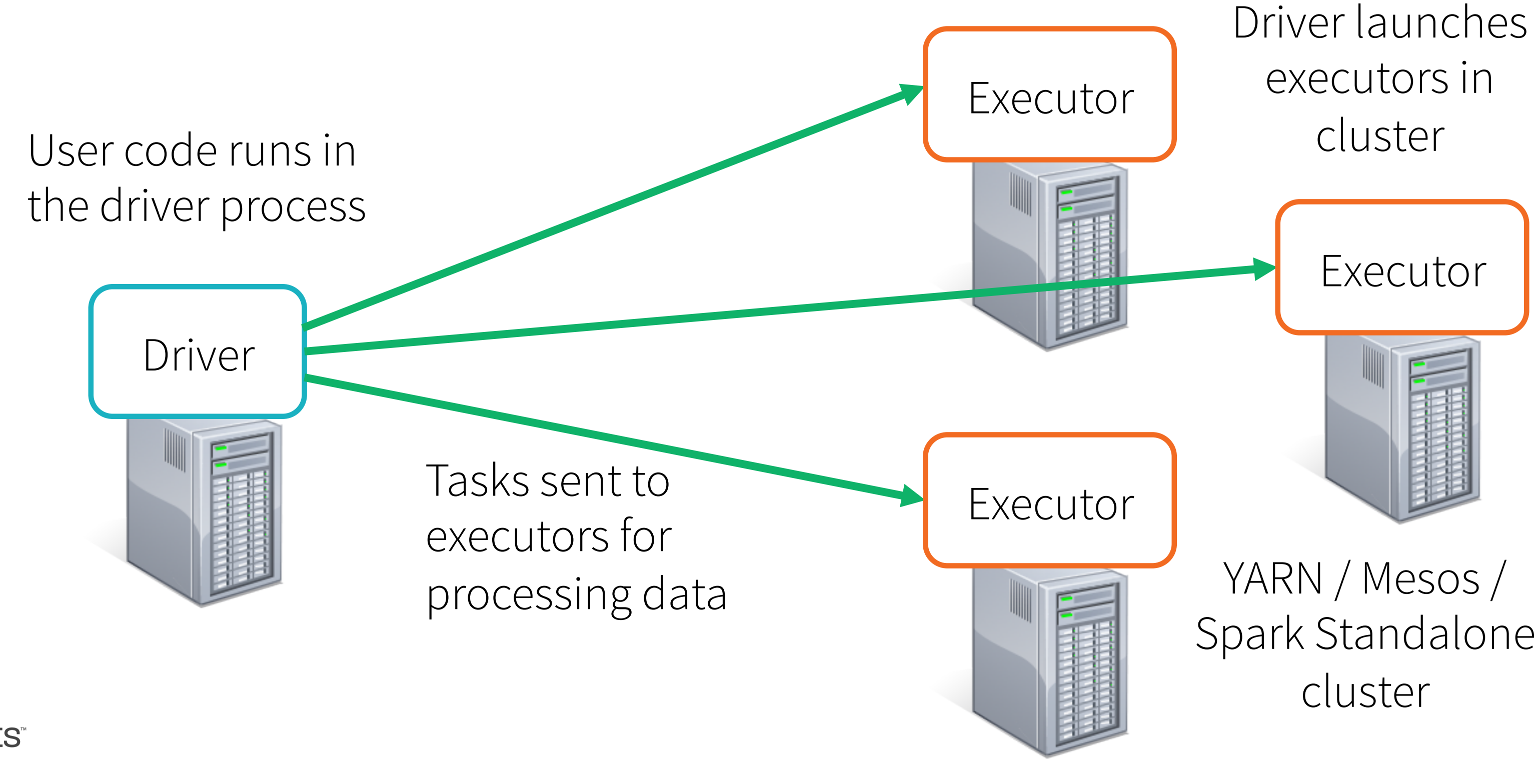
Performance and Stability

Monitoring and Upgrading

Deeper View of Spark Streaming

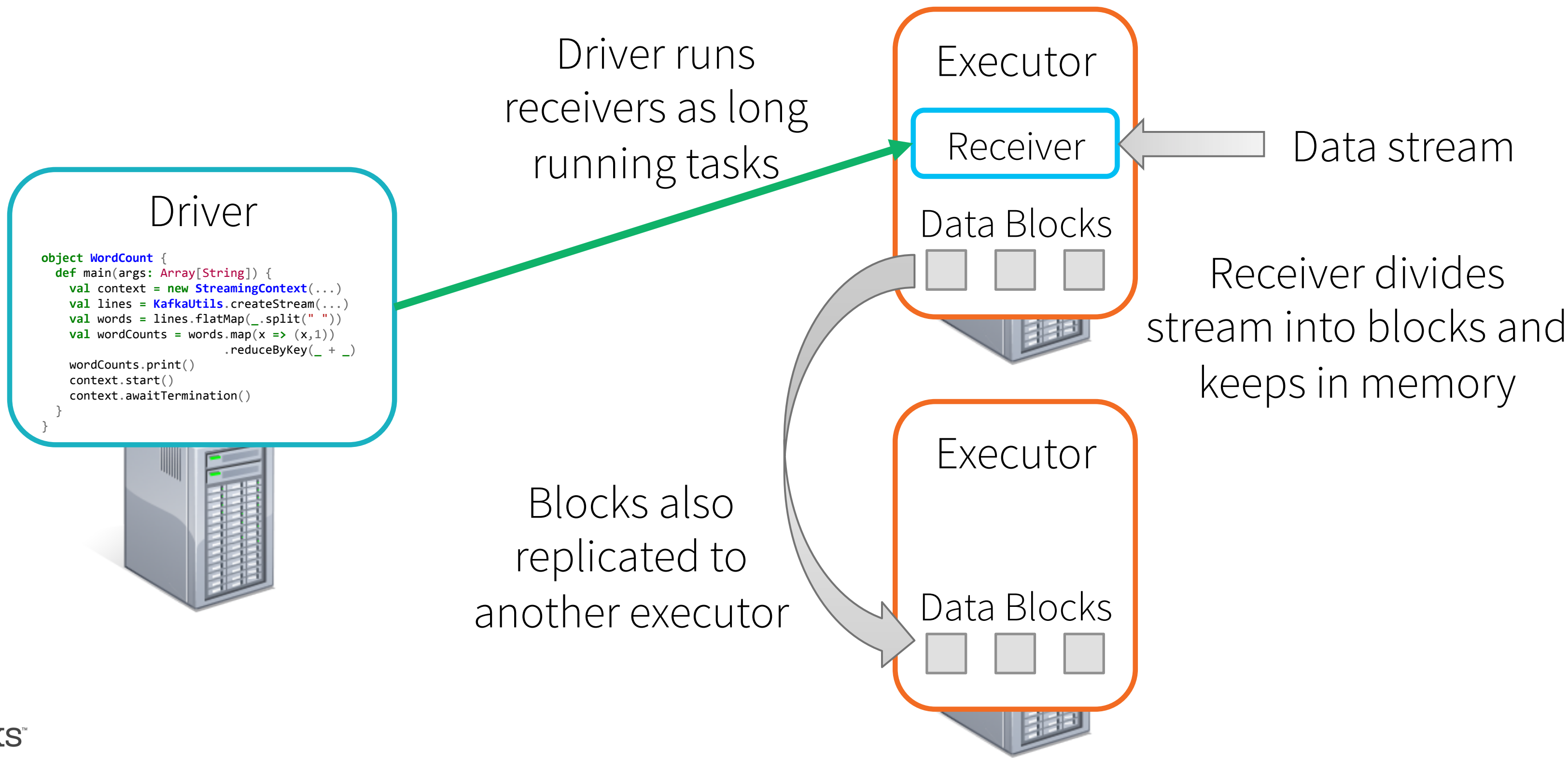
databricks™

Any Spark Application

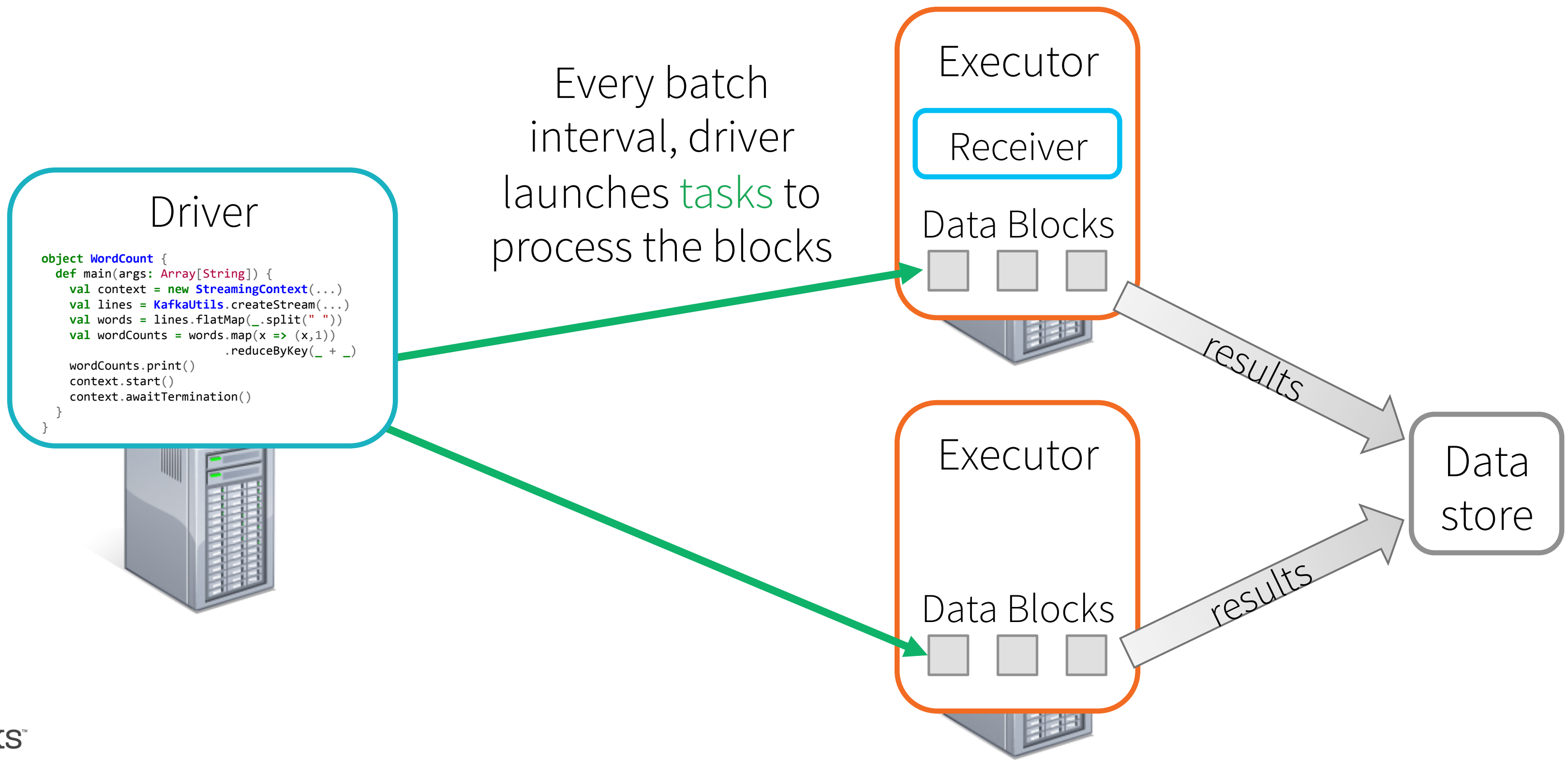


databricks

Spark Streaming Application: Receive data



Spark Streaming Application: Process data



databricks

Fault-tolerance and Semantics

Performance and Stability

Monitoring and upgrading

Failures? Why care?

Many streaming applications need zero data loss guarantees despite any kind of failures in the system

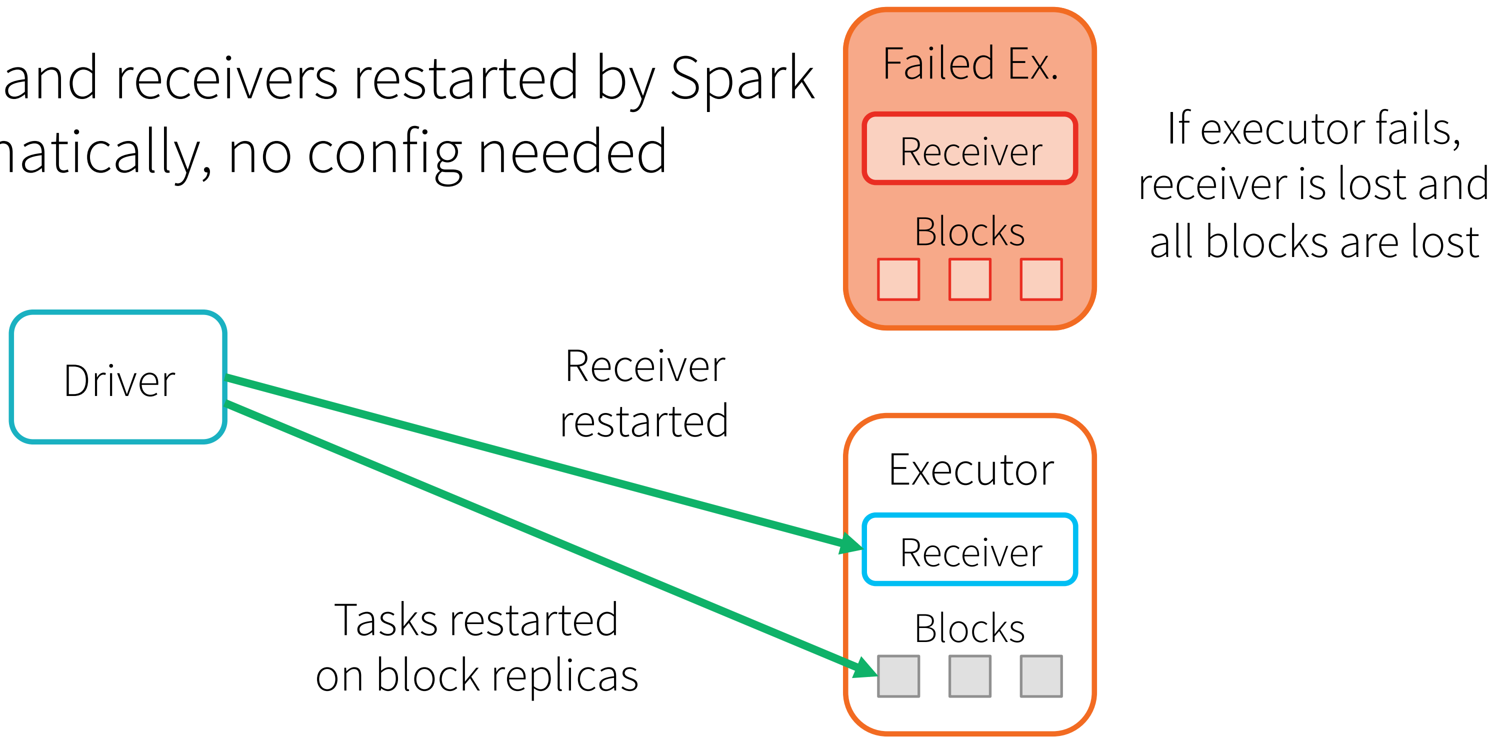
- At least once guarantee – every record processed at least once
- Exactly once guarantee – every record processed exactly once

Different kinds of failures – executor and driver

Some failures and guarantee requirements need additional configurations and setups

What if an executor fails?

Tasks and receivers restarted by Spark automatically, no config needed



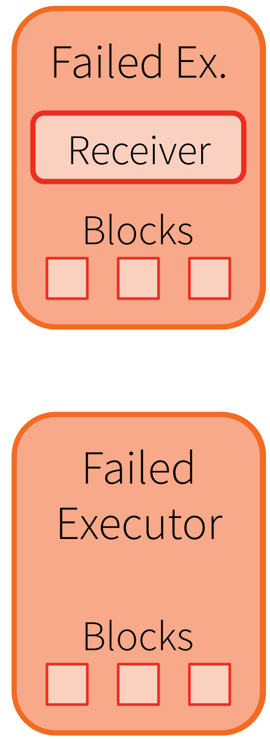
databricks

What if the driver fails?

When the driver fails, all the executors fail

All computation, all received blocks are lost

How do we recover?

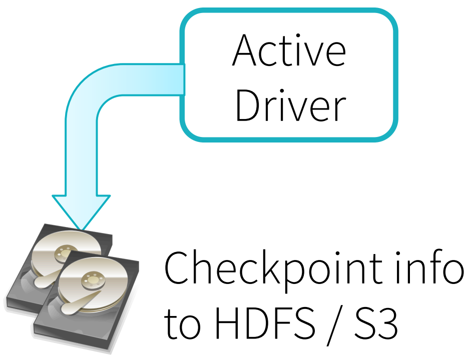


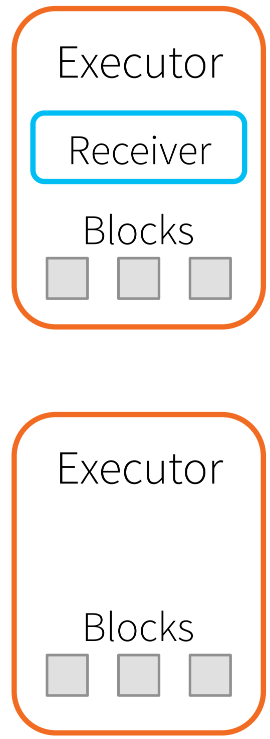
databricks

Recovering Driver with Checkpointing

DStream Checkpointing:

Periodically save the DAG of DStreams to fault-tolerant storage





Recovering Driver w/ DStream Checkpointing

DStream Checkpointing:

Periodically save the DAG of DStreams to fault-tolerant storage

Failed Driver

Restarted Driver

New executors launched and receivers restarted

New Executor

Receiver

New Executor

Failed driver can be restarted from checkpoint information

databricks

Recovering Driver w/ DStream Checkpointing

1. Configure automatic driver restart
   All cluster managers support this

2. Set a checkpoint directory in a HDFS-compatible file system
   ```
   streamingContext.checkpoint(hdfsDirectory)
   ```

3. Slightly restructure of the code to use checkpoints for recovery

Configurating Automatic Driver Restart

Spark Standalone – Use spark-submit with “cluster” mode and “--supervise”

See http://spark.apache.org/docs/latest/spark-standalone.html

YARN – Use spark-submit in “cluster” mode

See YARN config “yarn.resourcemanager.am.max-attempts”

Mesos – Marathon can restart Mesos applications

Restructuring code for Checkpointing

Create
+
Setup

```
val context = new StreamingContext(...)
val lines = KafkaUtils.createStream(...)
val words = lines.flatMap(...)
...
```

```
def creatingFunc(): StreamingContext = {
   val context = new StreamingContext(...)
   val lines = KafkaUtils.createStream(...)
   val words = lines.flatMap(...)
   ...
   context.checkpoint(hdfsDir)
}
```

Put all setup code into a function that returns a new StreamingContext

Start

```
context.start()
```

```
val context =
StreamingContext.getOrCreate(
  hdfsDir, creatingFunc)
context.start()
```

Get context setup from HDFS dir OR *create* a new one with the function

Restructuring code for Checkpointing

StreamingContext.getOrCreate():

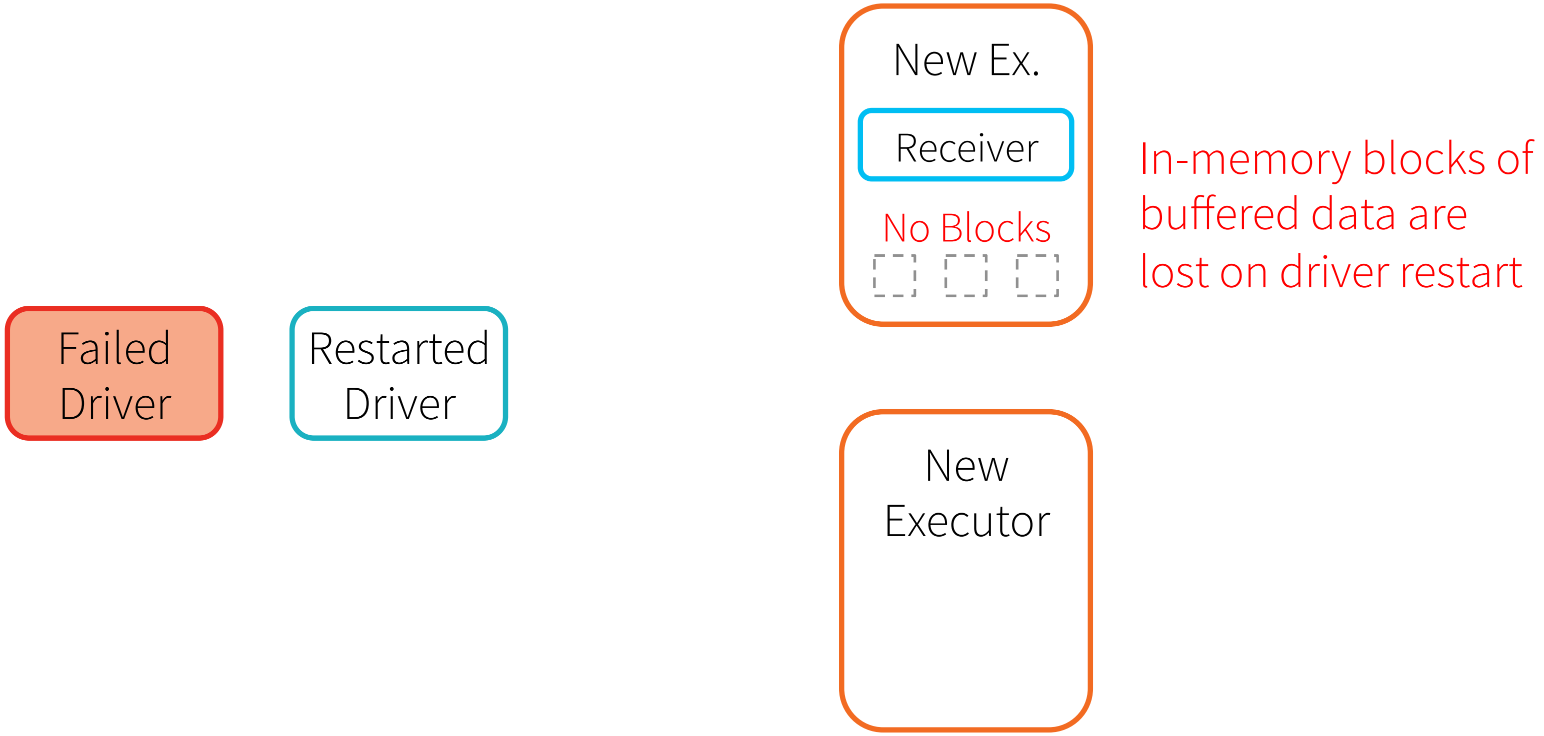
If HDFS directory has checkpoint info
    recover context from info
else
    call **`creatingFunc()`** to create
    and setup a new context

```
def creatingFunc(): StreamingContext = {
   val context = new StreamingContext(...)
   val lines = KafkaUtils.createStream(...)
   val words = lines.flatMap(...)
   ...
   context.checkpoint(hdfsDir)
}
```

Restarted process can figure out whether to recover using checkpoint info or not

```
val context =
StreamingContext.getOrCreate(
  hdfsDir, creatingFunc)
context.start()
```

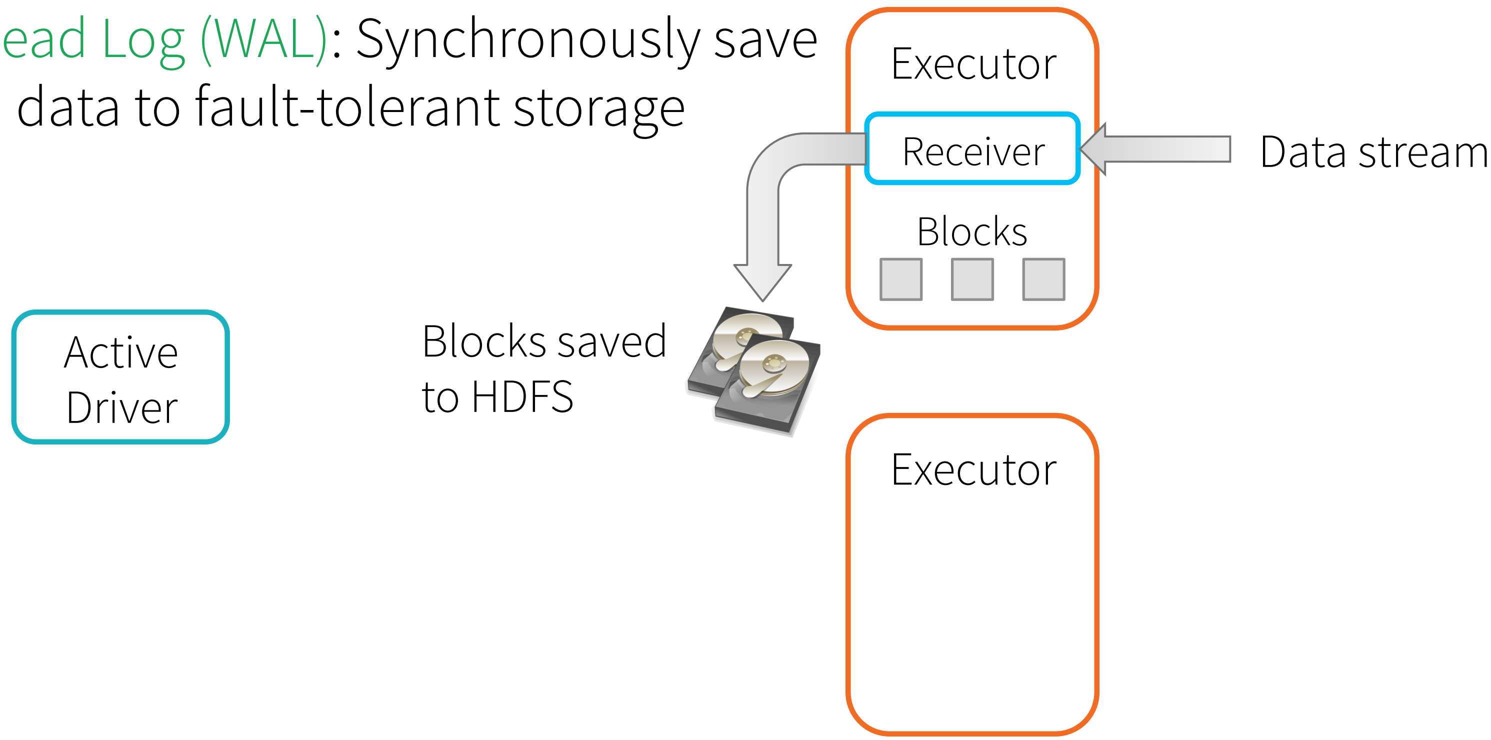
Received blocks lost on Restart!



databricks

Recovering data with Write Ahead Logs

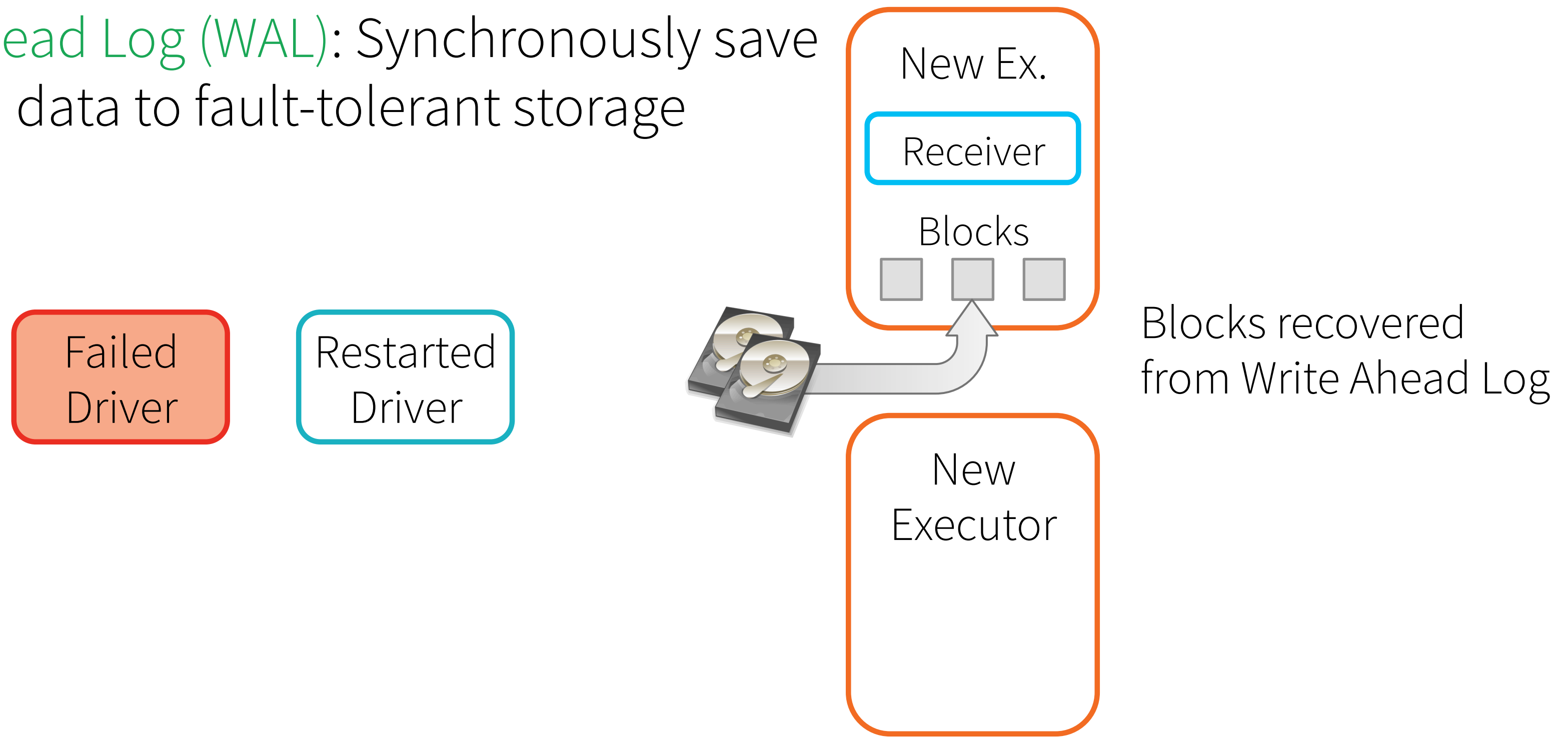
Write Ahead Log (WAL): Synchronously save received data to fault-tolerant storage



databricks

Recovering data with Write Ahead Logs

Write Ahead Log (WAL): Synchronously save received data to fault-tolerant storage



Recovering data with Write Ahead Logs

1. Enable checkpointing, logs written in checkpoint directory

2. Enabled WAL in SparkConf configuration
   `sparkConf.set("spark.streaming.receiver.writeAheadLog.enable", "true")`

3. Receiver should also be *reliable*
   Acknowledge source only after data saved to WAL
   Unacked data will be replayed from source by restarted receiver

4. Disable in-memory replication (already replicated by HDFS)
   Use `StorageLevel.MEMORY_AND_DISK_SER` for input DStreams

databricks™

RDD Checkpointing

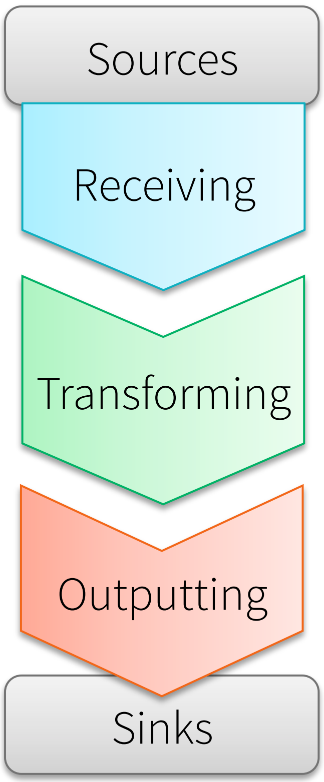
Stateful stream processing can lead to long RDD lineages

Long lineage = bad for fault-tolerance, too much recomputation

RDD checkpointing saves RDD data to the fault-tolerant storage to limit lineage and recomputation

More: http://spark.apache.org/docs/latest/streaming-programming-guide.html#checkpointing

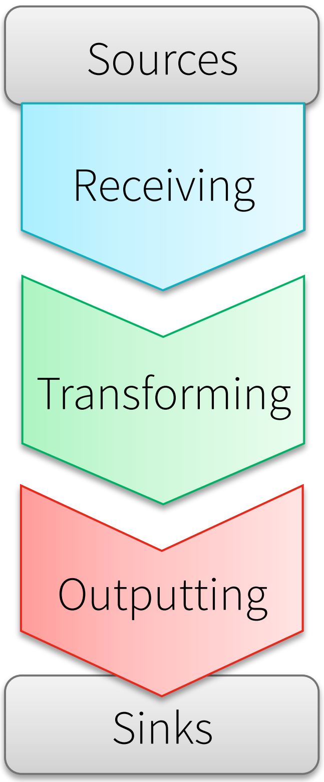
Fault-tolerance Semantics



Zero data loss = every stage processes each event at least once despite any failure

databricks

Fault-tolerance Semantics



At least once, w/ Checkpointing + WAL + Reliable receivers

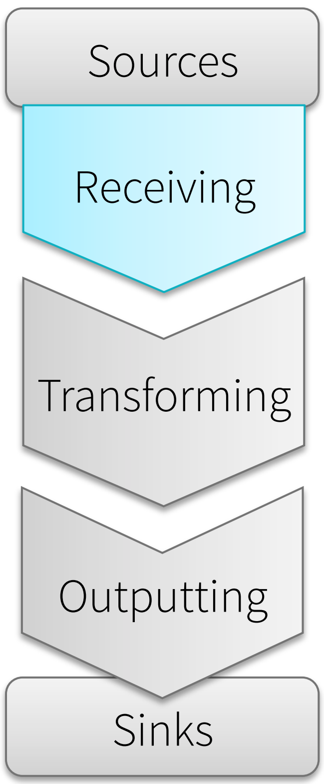
Exactly once, as long as received data is not lost

Exactly once, if outputs are idempotent or transactional

End-to-end semantics:
At-least once

databricks

Fault-tolerance Semantics



Exactly once receiving with new Kafka Direct approach

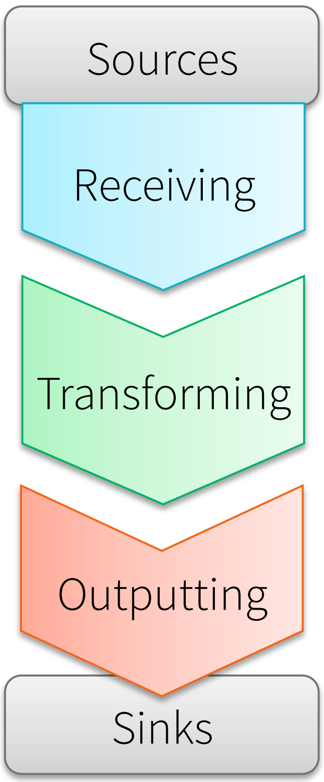
- Treats Kafka like a replicated log, reads it like a file
- Does not use receivers
- No need to create multiple DStreams and union them
- No need to enable Write Ahead Logs

```
val directKafkaStream = KafkaUtils.createDirectStream(...)
```

https://databricks.com/blog/2015/03/30/improvements-to-kafka-integration-of-spark-streaming.html
http://spark.apache.org/docs/latest/streaming-kafka-integration.html

databricks

Fault-tolerance Semantics



Exactly once receiving with new Kafka Direct approach

Exactly once, as long as received data is not lost

Exactly once, if outputs are idempotent or transactional

End-to-end semantics:
Exactly once!

NETFLIX

The Netflix Tech Blog

Wednesday, March 11, 2015

Can Spark Streaming survive Chaos Monkey?

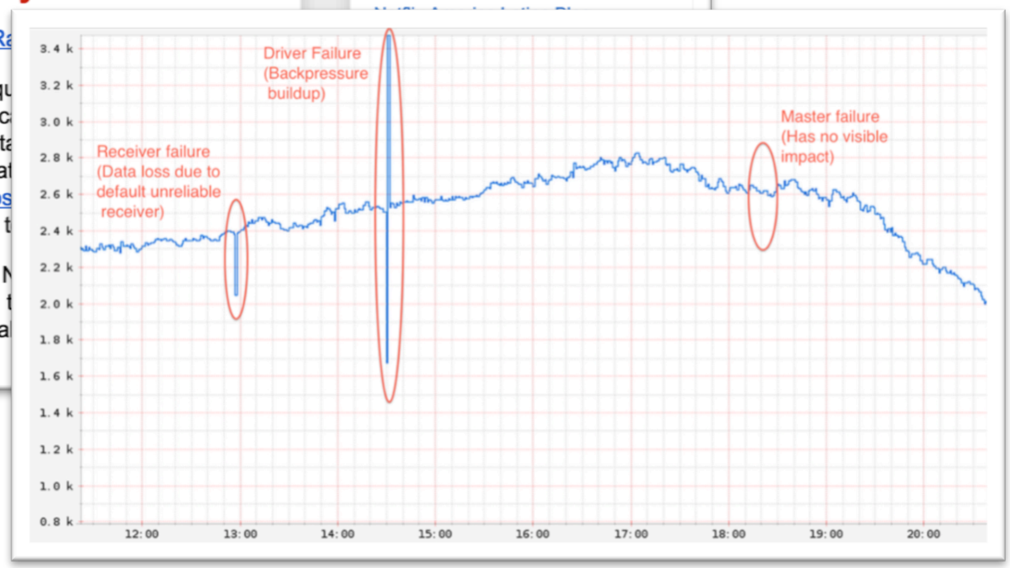
by Bharat Venkat, Prasanna Padmanabhan, Antony Arokiasamy, Ra

Netflix is a data-driven organization that places emphasis on the qu
processed. In our previous blog post, we highlighted our use c
processing in the context of online recommendations and data
Streaming as our choice of stream processor, we set out to evaluat
story for Spark Streaming in the AWS cloud environment. A Chaos
which randomly terminated instances or processes, was employed t

Spark on Amazon Web Services (AWS) is relevant to us as N
primarily out of the AWS cloud. Stream processing systems need t
be tolerant to failures. Instances on AWS are ephemeral, which ma
Spark's resiliency.

Links

Netflix US & Canada Blog



http://techblog.netflix.com/2015/03/can-spark-streaming-survive-chaos-monkey.html

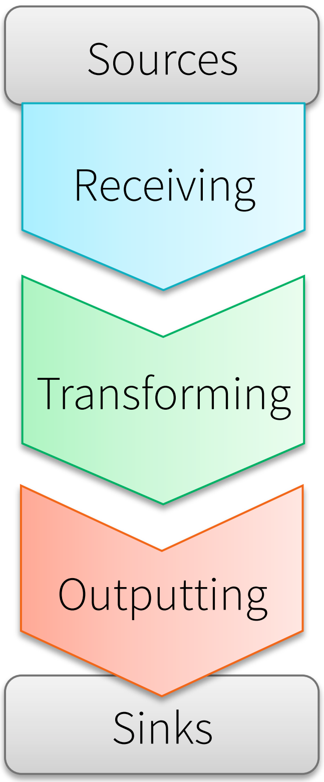
Fault-tolerance and Semantics

Performance and Stability

Monitoring and Upgrading

databricks

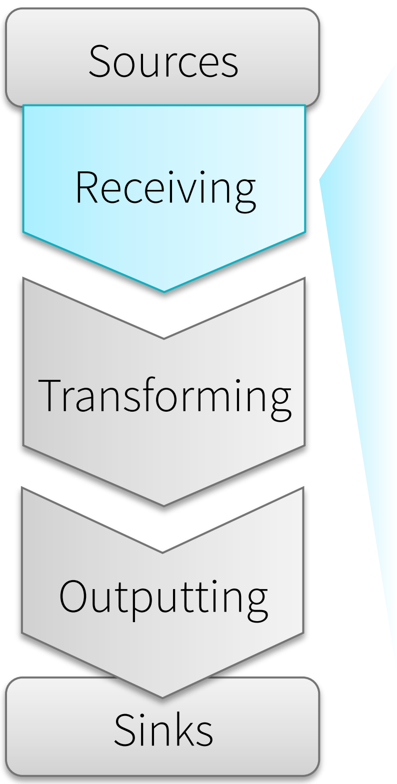
Achieving High Throughput



High throughput achieved by sufficient parallelism at all stages of the pipeline

databricks

Scaling the Receivers



Sources must be configured with parallel data streams

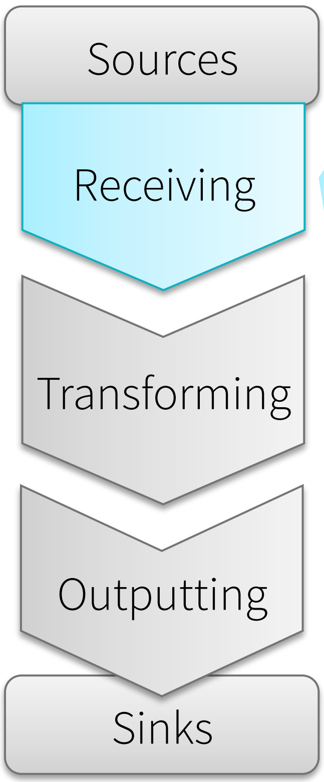
- #partitions in Kafka topics, #shards in Kinesis streams, …

Streaming app should have multiple receivers that receive the data streams in parallel

- Multiple input DStreams, each running a receiver
- Can be unioned together to create one DStream

```
val kafkaStream1 = KafkaUtils.createStream(...)
val kafkaStream2 = KafkaUtils.createStream(...)
val unionedStream = kafkaStream1.union(kafkaStream2)
```

Scaling the Receivers



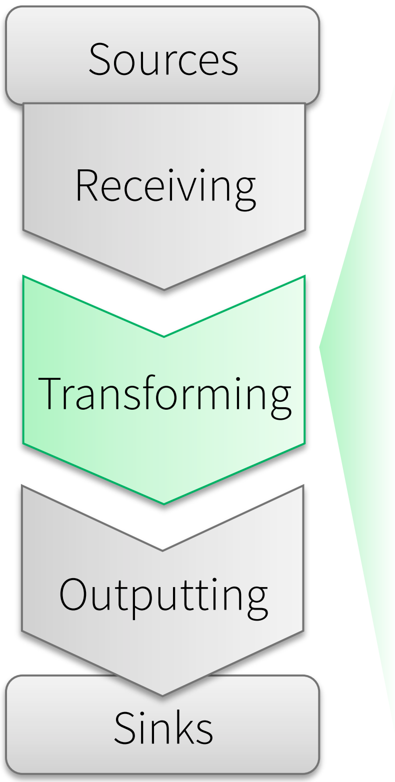
Sufficient number of executors to run all the receivers

- Absolute necessity: #cores > #receivers
- Good rule of thumb: #executors > #receivers, so that no more than 1 receiver per executor, and network is not shared between receivers

Kafka Direct approach does not use receivers

- Automatically parallelizes data reading across executors
- Parallelism = # Kafka partitions

Stability in Processing



For stability, must process data as fast as it is received

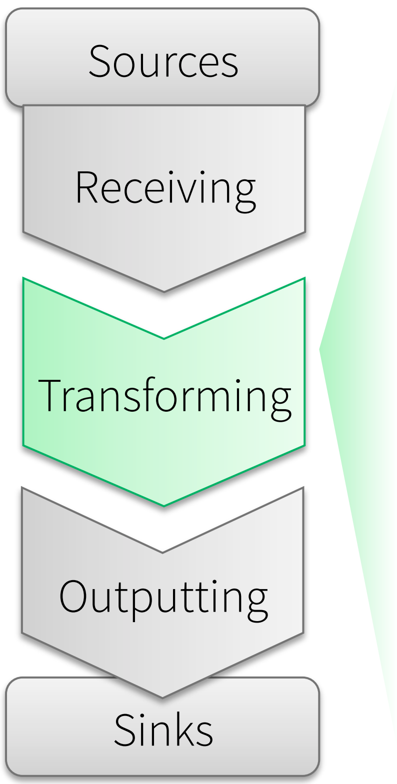
Must ensure avg batch processing times < batch interval

Previous batch is done by the time next batch is received

Otherwise, new batches keeps queueing up waiting for previous batches to finish, scheduling delay goes up

databricks

Reducing Batch Processing Times



More receivers!

Executor running receivers do lot of the processing

Repartition the received data to explicitly distribute load

```
unionedStream.repartition(40)
```

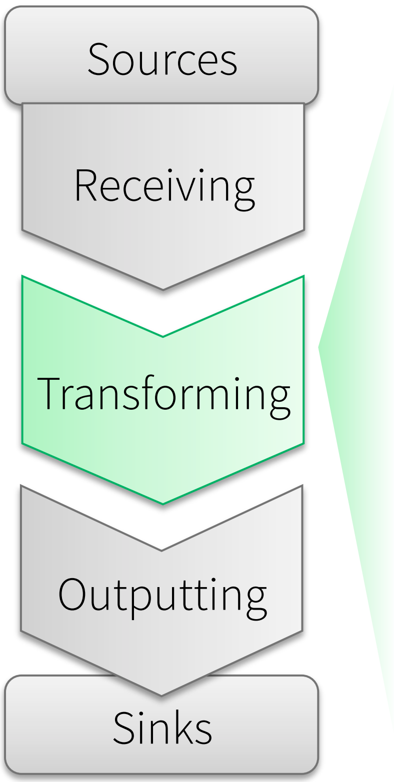
Set #partitions in shuffles, make sure its large enough

```
transformedStream.reduceByKey(reduceFunc, 40)
```

Get more executors and cores!

databricks

Reducing Batch Processing Times



Use Kryo serialization to serialization costs

- Register classes for best performance
- See configurations spark.kryo.*
  http://spark.apache.org/docs/latest/configuration.html#compression-and-serialization

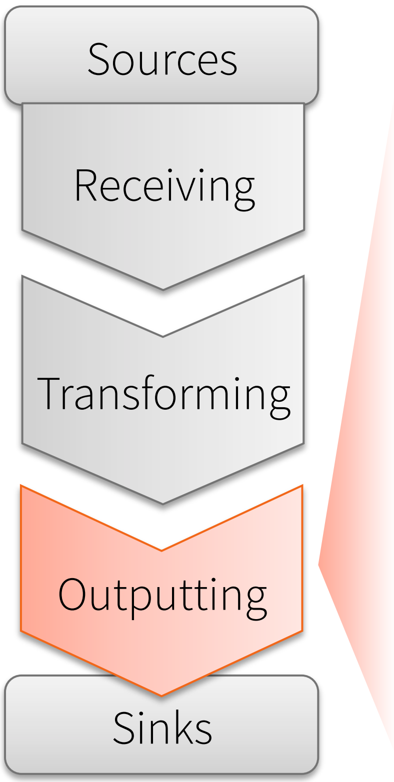
Larger batch durations improve stability

- More data aggregated together, amortized cost of shuffle

Limit ingestion rate to handle data surges

- See configurations spark.streaming.*maxRate*
  http://spark.apache.org/docs/latest/configuration.html#spark-streaming

Speeding up Output Operations



Write to data stores efficiently

foreach: inefficient

```
dataRDD.foreach { event =>
  // open connection
  // insert single event
  // close connection
}
```

foreachPartition: efficient

```
dataRDD.foreachPartition { partition =>
  // open connection
  // insert all events in partition
  // close connection
}
```

foreachPartition + connection pool: more efficient

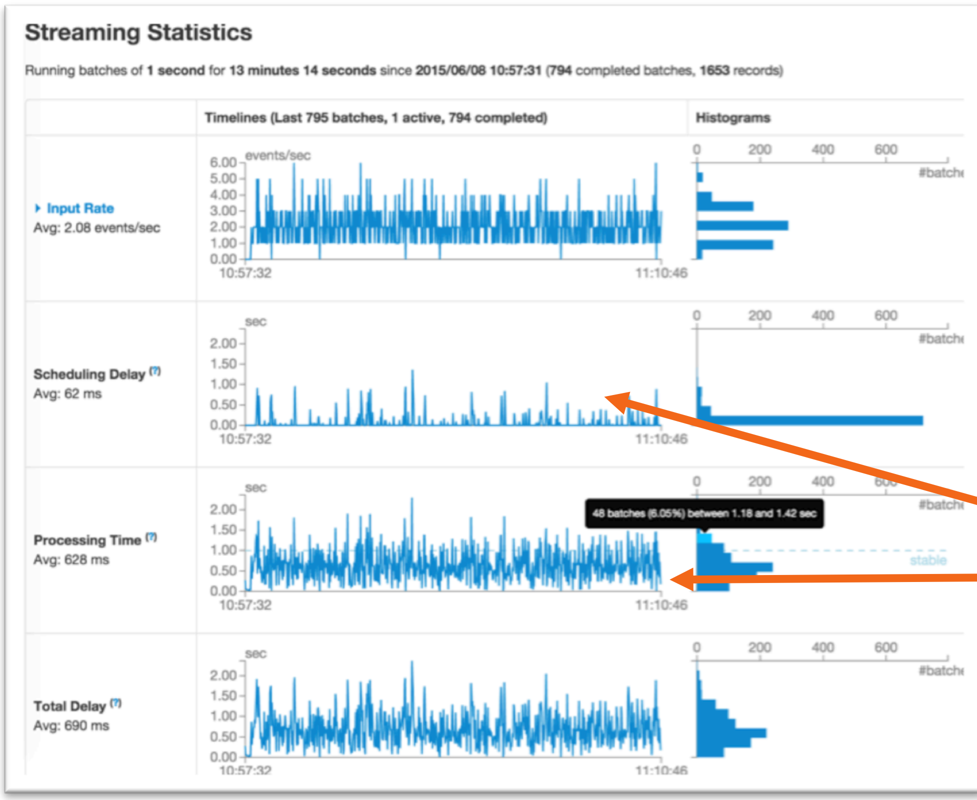
```
dataRDD.foreachPartition { partition =>
  // initialize pool or get open connection from pool in executor
  // insert all events in partition
  // return connection to pool
}
```

Fault-tolerance and Semantics

Performance and Stability

Monitoring and Upgrading

Streaming in Spark Web UI



Stats over last 1000 batches

New in Spark 1.4

For stability

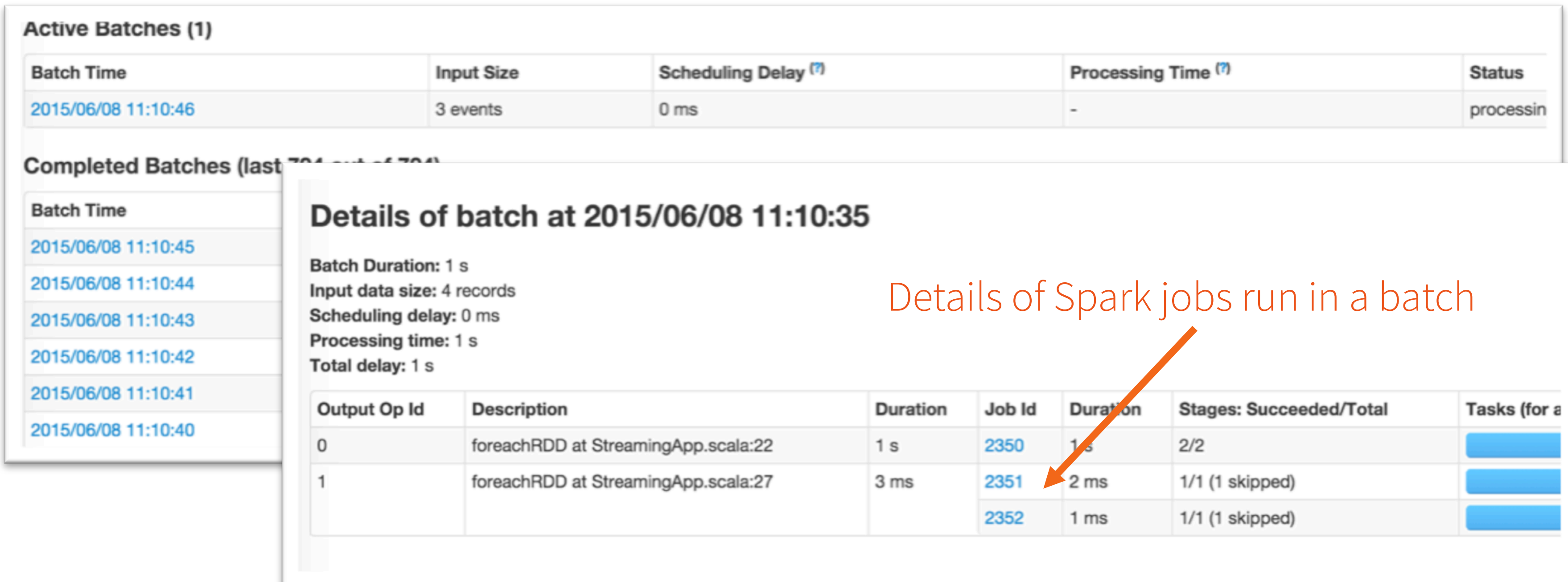
Scheduling delay should be approx 0

Processing Time approx < batch interval

databricks

Streaming in Spark Web UI

Details of individual batches



Active Batches (1)

Batch Time	Input Size	Scheduling Delay (?)	Processing Time (?)	Status
2015/06/08 11:10:46	3 events	0 ms	-	processin

Completed Batches (last

Batch Time
2015/06/08 11:10:45
2015/06/08 11:10:44
2015/06/08 11:10:43
2015/06/08 11:10:42
2015/06/08 11:10:41
2015/06/08 11:10:40

Details of batch at 2015/06/08 11:10:35

Batch Duration: 1 s
Input data size: 4 records
Scheduling delay: 0 ms
Processing time: 1 s
Total delay: 1 s

Output Op Id	Description	Duration	Job Id	Duration	Stages: Succeeded/Total	Tasks (for a
0	foreachRDD at StreamingApp.scala:22	1 s	2350	1 s	2/2	
1	foreachRDD at StreamingApp.scala:27	3 ms	2351	2 ms	1/1 (1 skipped)	
			2352	1 ms	1/1 (1 skipped)	

databricks

Operational Monitoring

Streaming app stats published through Codahale metrics

- Ganglia sink, Graphite sink, custom Codahale metrics sinks
- Can see long term trends, across hours and days

Configure the metrics using `$SPARK_HOME/conf/metrics.properties`

- Need to compile Spark with Ganglia LGPL profile for Ganglia support

(see http://spark.apache.org/docs/latest/monitoring.html#metrics)

Programmatic Monitoring

StreamingListener – Developer interface to get internal events

onBatchSubmitted, onBatchStarted, onBatchCompleted,
onReceiverStarted, onReceiverStopped, onReceiverError

Take a look at StreamingJobProgressListener (private class) for inspiration

Upgrading Apps

1. Shutdown your current streaming app gracefully

   Will process all data before shutting down cleanly

   ```
   streamingContext.stop(stopGracefully = true)
   ```

2. Update app code and start it again

Cannot upgrade from previous checkpoints if code changes or Spark version changes

Much to say I have ... but time I have not

Memory and GC tuning

Using SQLContext

DStream.transform operation

…

Refer to online guide

http://spark.apache.org/docs/latest/streaming-programming-guide.html


Spark 1.3.1 Overview Programming Guides API Docs Deploying

Spark Streaming Programming Guide

Overview

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughp streams. Data can be ingested from many sources like Kafka, Flume, Twitter, ZeroMQ, Kine algorithms expressed with high-level functions like `map`, `reduce`, `join` and `window`. Finally, pr

Thank you

May the stream be with you

databricks™