Four Things to Know about Reliable Spark Streaming

Dean Wampler, Typesafe

Tathagata Das, Databricks

Agenda for today

- The Stream Processing Landscape
- How Spark Streaming Works - A Quick Overview
- Features in Spark Streaming that Help Prevent Data Loss
- Design Tips for Successful Streaming Applications

The Stream Processing Landscape

Stream Processors

Stream Storage

kafka
amazon web services™
S3
hadoop
HDFS
riak

Stream Sources

ØMQ

MQTT

kafka

hadoop
HDFS

amazon
web services™ | S3

riak

databricks™

Typesafe

How Spark Streaming Works: A Quick Overview

Spark Streaming

Scalable, fault-tolerant stream processing system

High-level API

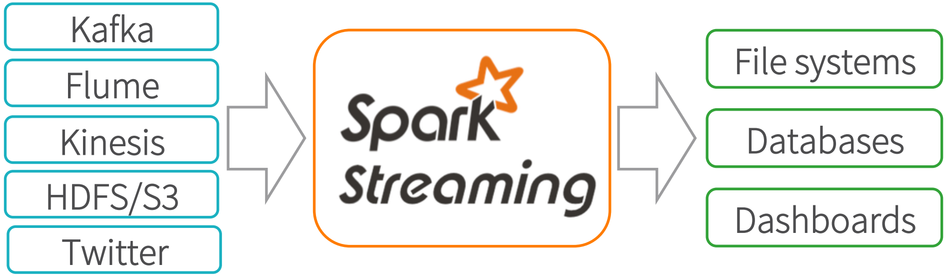
joins, windows, …
often 5x less code

Fault-tolerant

Exactly-once semantics,
even for stateful ops

Integration

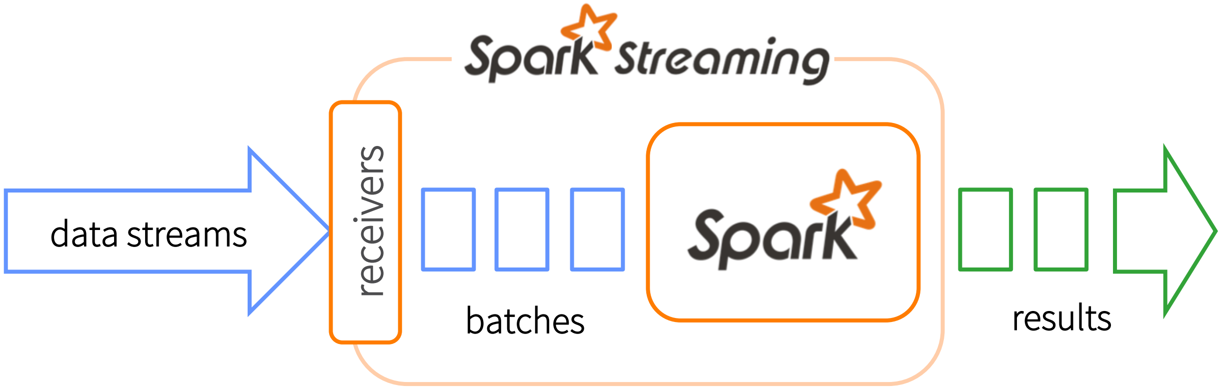
Integrates with MLlib, SQL,
DataFrames, GraphX



Spark Streaming

Receivers receive data streams and chop them up into batches

Spark processes the batches and pushes out the results



Spark Streaming

data streams

receivers

batches

Spark

results

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

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

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

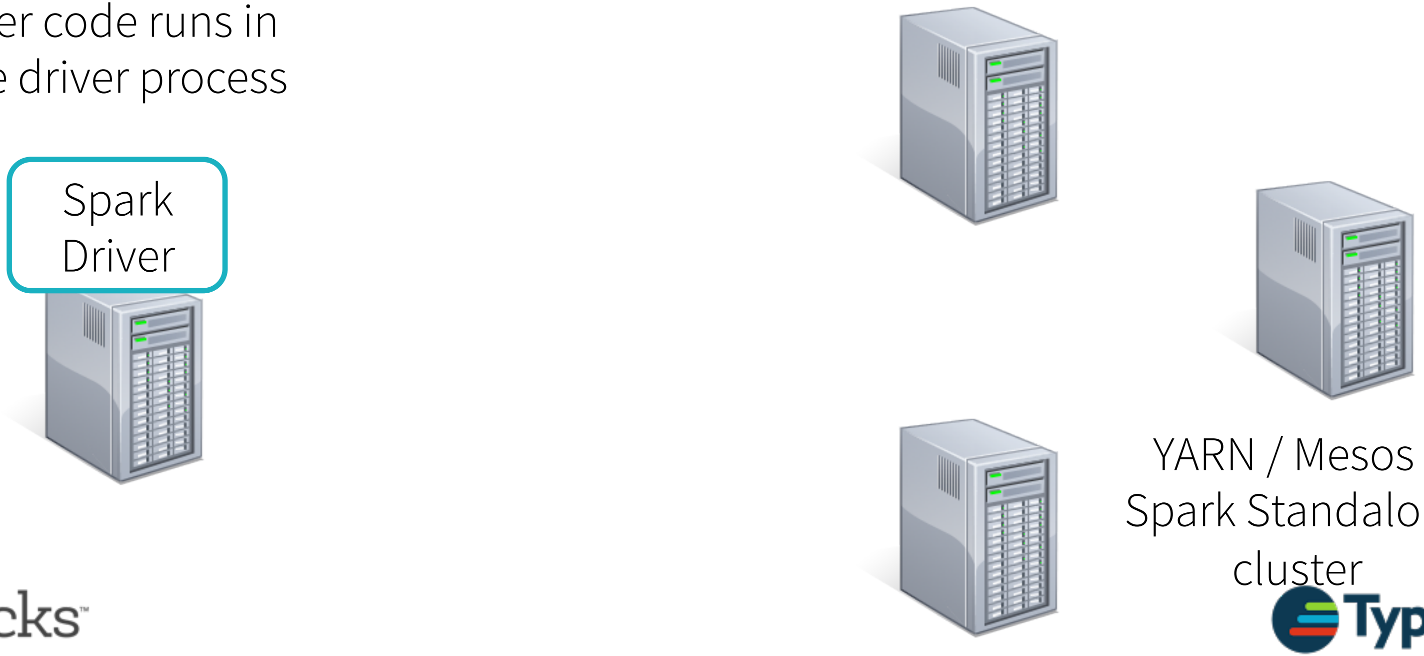
Word Count with Kafka

```
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()
  }
}
```

Features in Spark Streaming that Help Prevent Data Loss

A Deeper View of Spark Streaming

Any Spark Application

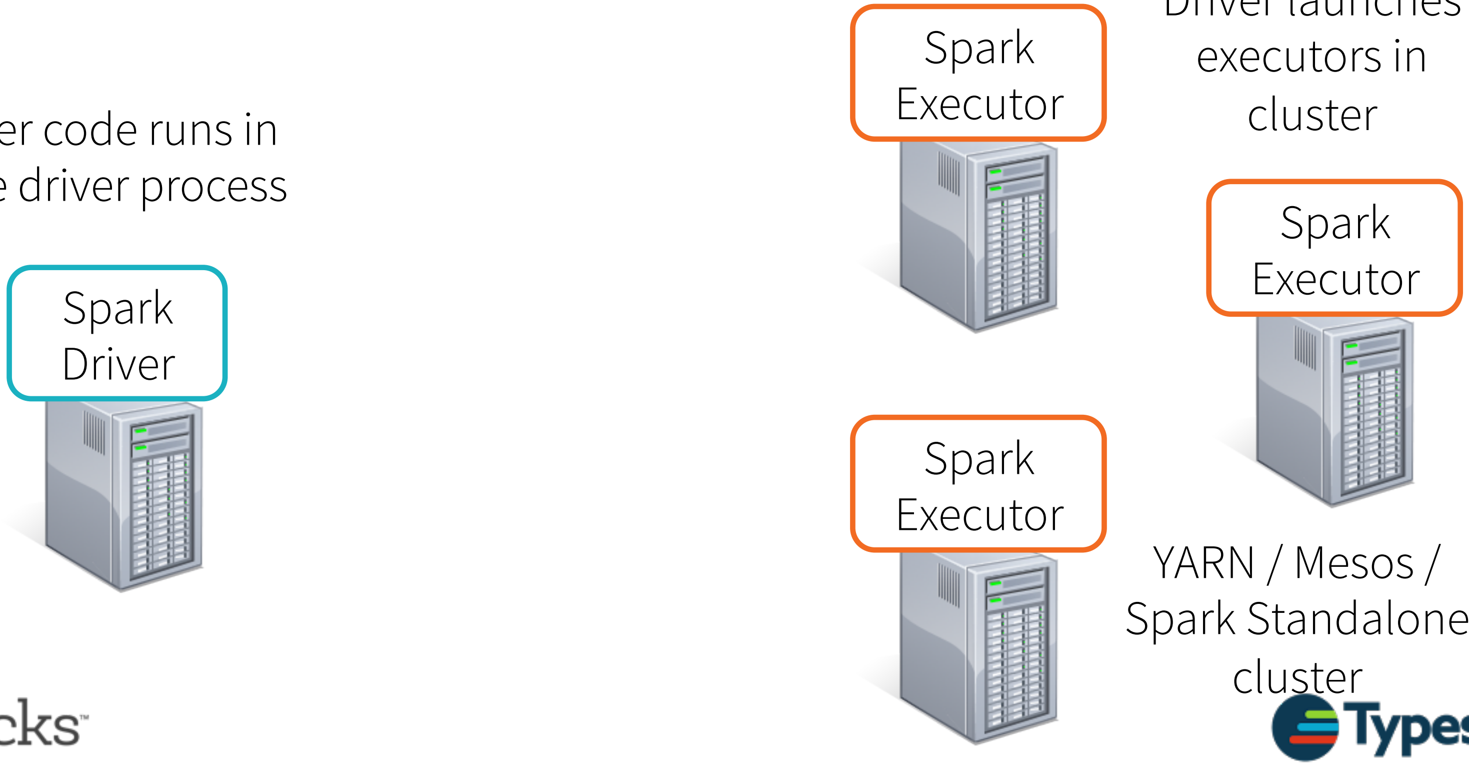


User code runs in the driver process

Spark Driver

YARN / Mesos / Spark Standalone cluster

Any Spark Application



User code runs in the driver process

Spark Driver

Spark Executor

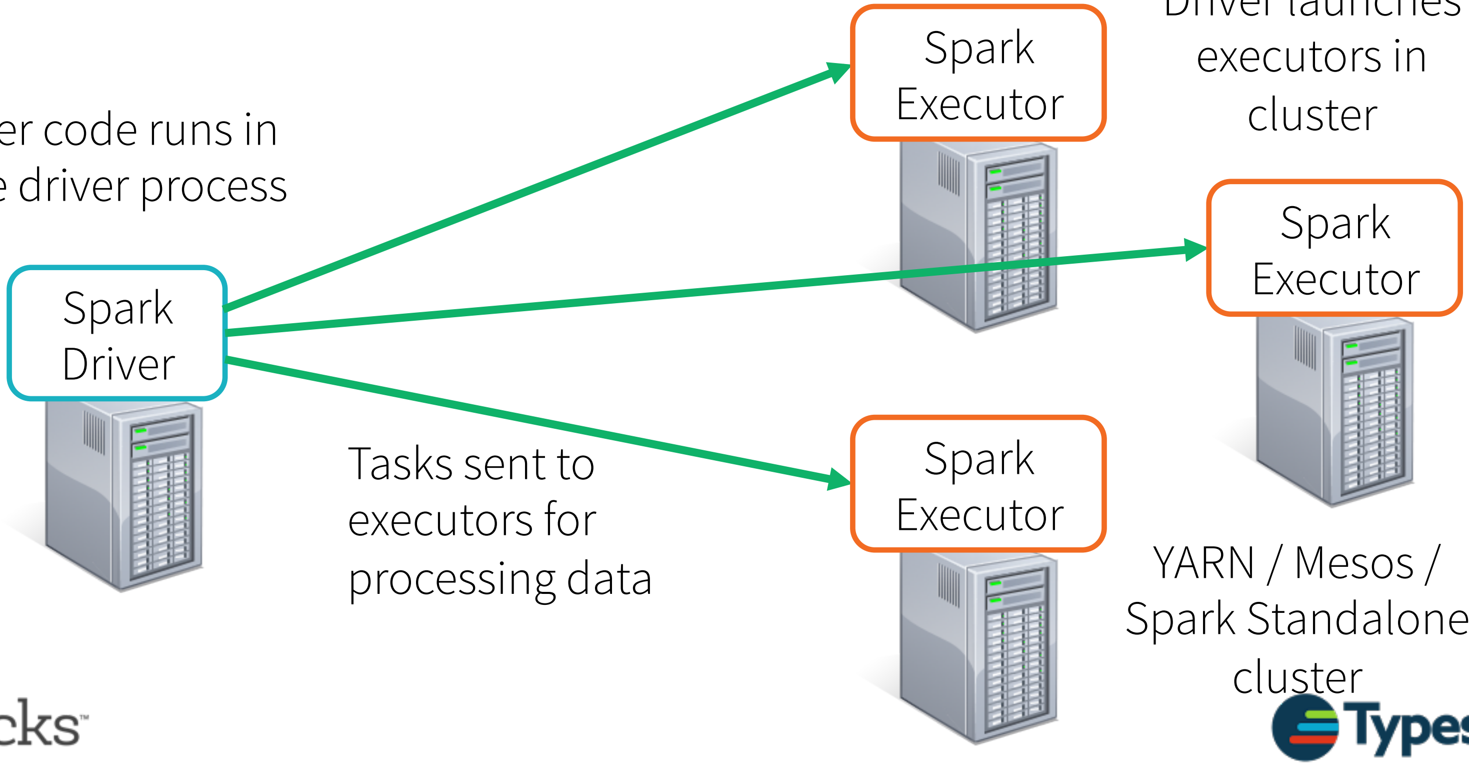
Driver launches executors in cluster

Spark Executor

Spark Executor

YARN / Mesos / Spark Standalone cluster

Any Spark Application



User code runs in the driver process

Spark Driver

Tasks sent to executors for processing data

Spark Executor

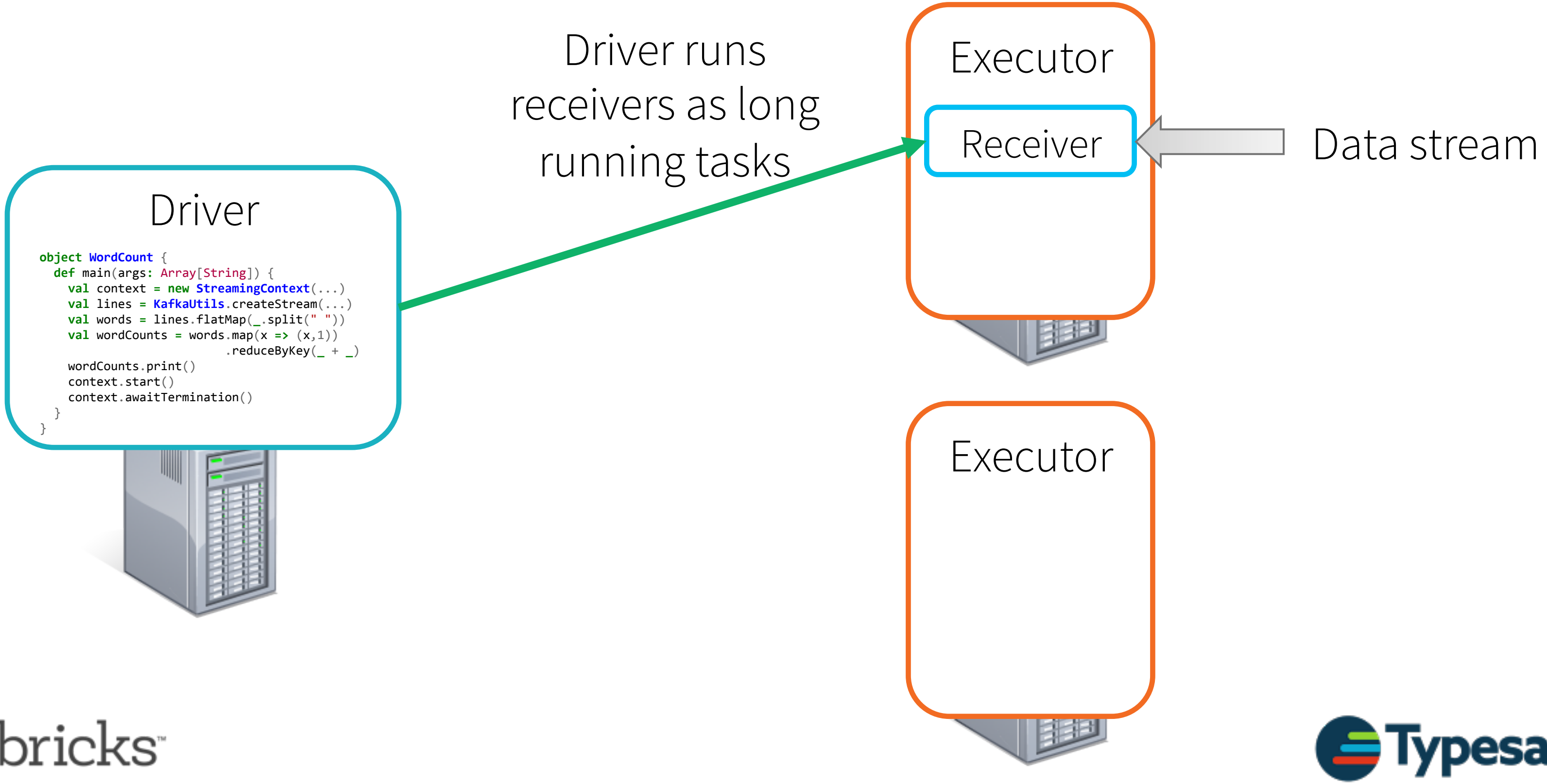
Spark Executor

Spark Executor

Driver launches executors in cluster

YARN / Mesos / Spark Standalone cluster

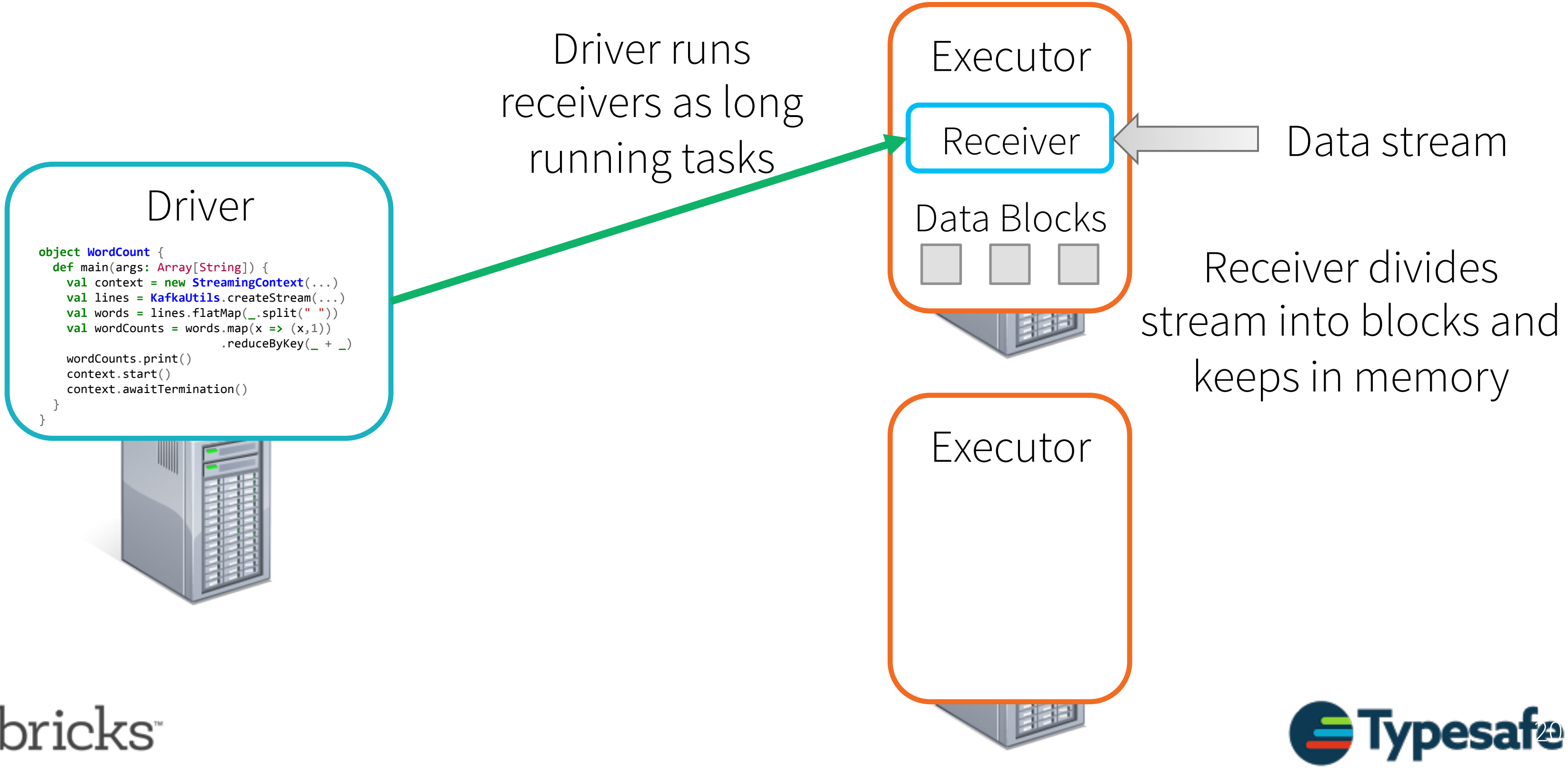
Spark Streaming Application: Receive data



Driver runs receivers as long running tasks

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

Spark Streaming Application: Receive data



Driver

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

Driver runs receivers as long running tasks

Executor

Receiver

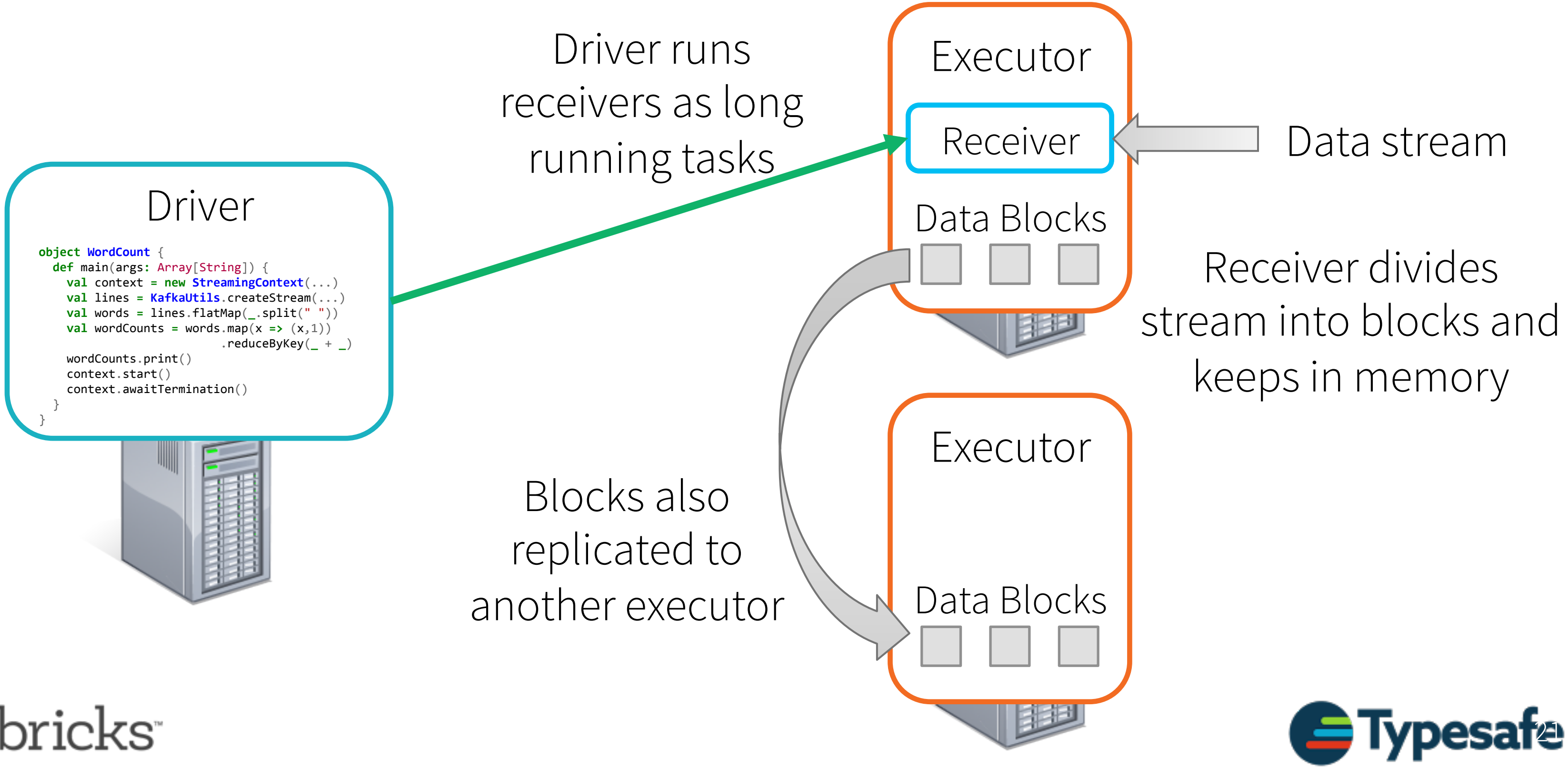
Data stream

Data Blocks

Receiver divides stream into blocks and keeps in memory

Executor

Spark Streaming Application: Receive data



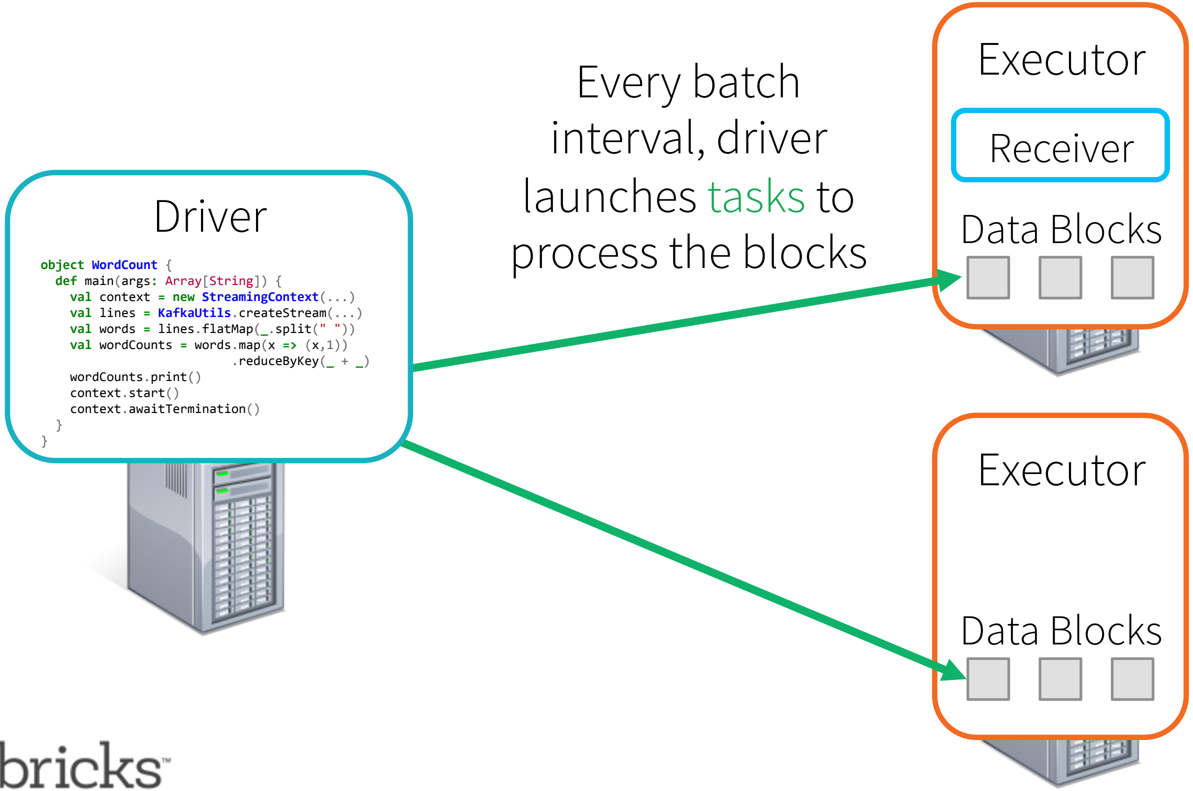
```
object WordCount {
  def main(args: Array[String]) {
    val context = new StreamingContext(...)
    val lines = KafkaUtils.createStream(...)
    val words = lines.flatMap(_.split(" "))
    val wordCounts = words.map(x => (x,1))
                          .reduceByKey(_ + _)
    wordCounts.print()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

Driver runs receivers as long running tasks

Receiver divides stream into blocks and keeps in memory

Blocks also replicated to another executor

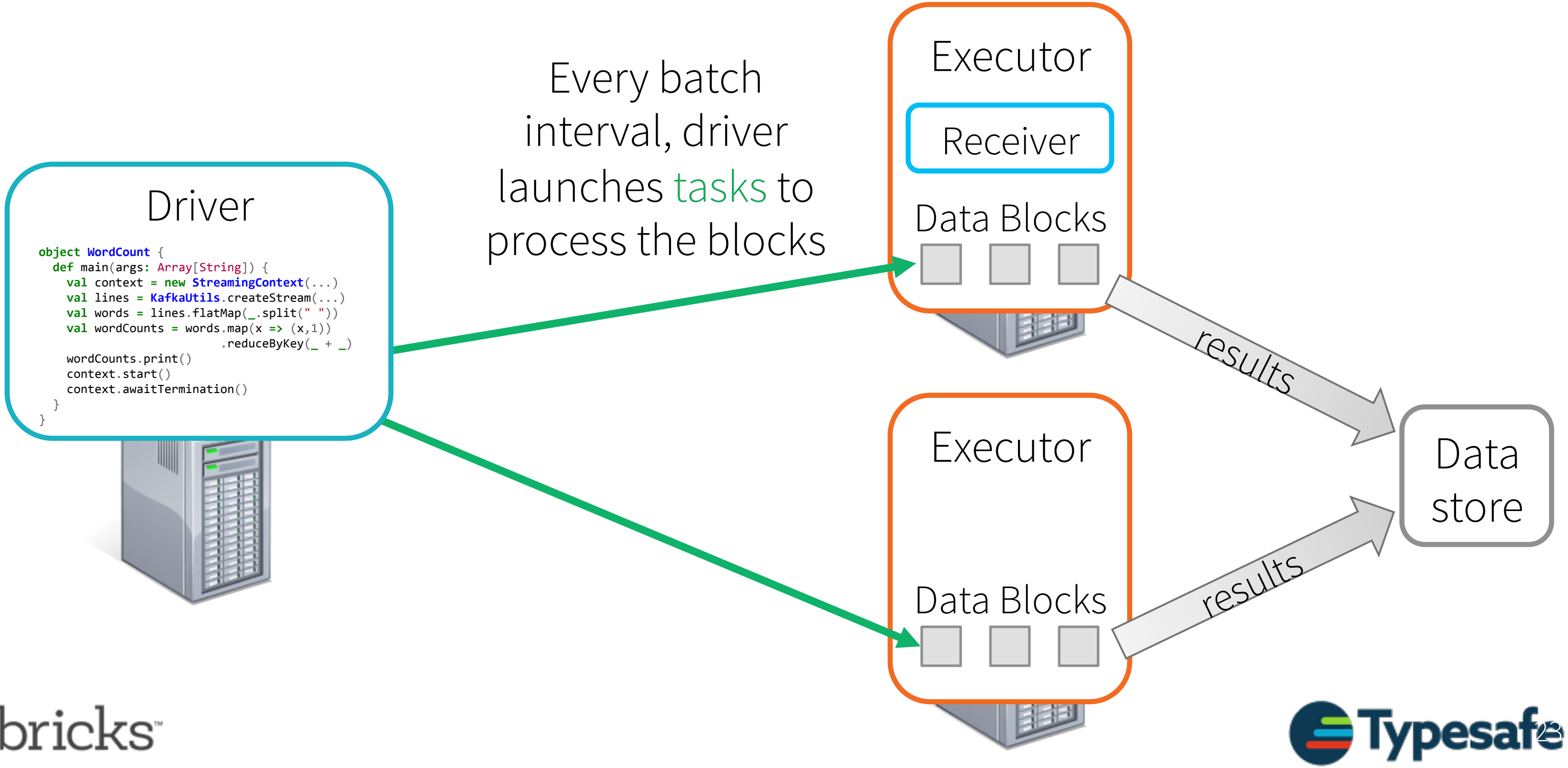
Spark Streaming Application: Process data



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

Every batch interval, driver launches tasks to process the blocks

Spark Streaming Application: Process data



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

Every batch interval, driver launches tasks to process the blocks

Fault Tolerance and Reliability

Failures? Why care?

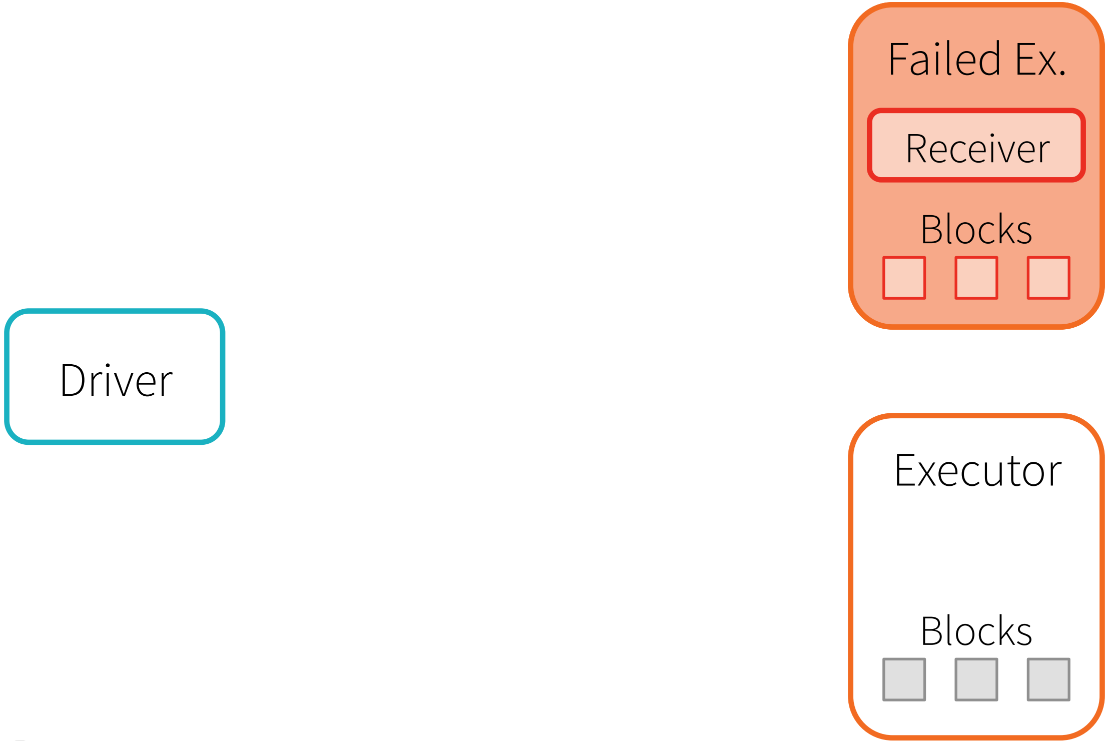
Many streaming applications need zero data loss guarantees despite any kind of failures in the system

- 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

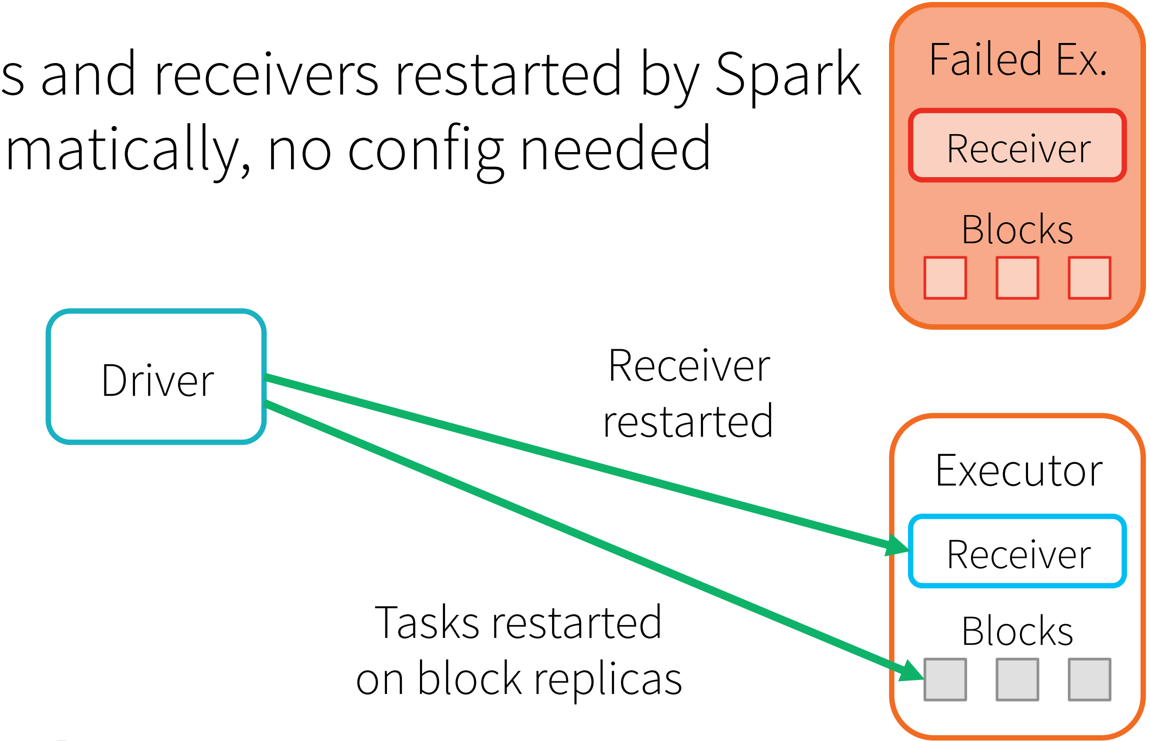
What if an executor fails?



If executor fails, receiver is lost and all blocks are lost

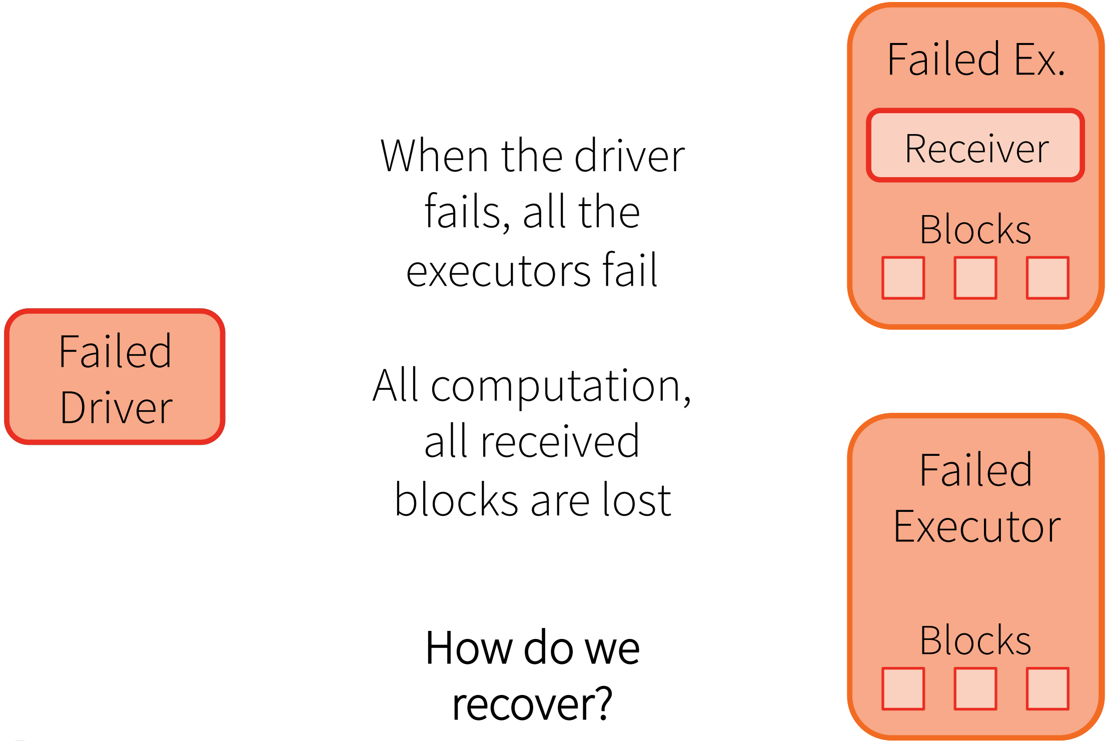
What if an executor fails?

Tasks and receivers restarted by Spark automatically, no config needed



If executor fails, receiver is lost and all blocks are lost

What if the driver fails?



Failed Driver

When the driver fails, all the executors fail

All computation, all received blocks are lost

How do we recover?

Failed Ex.

Receiver

Blocks

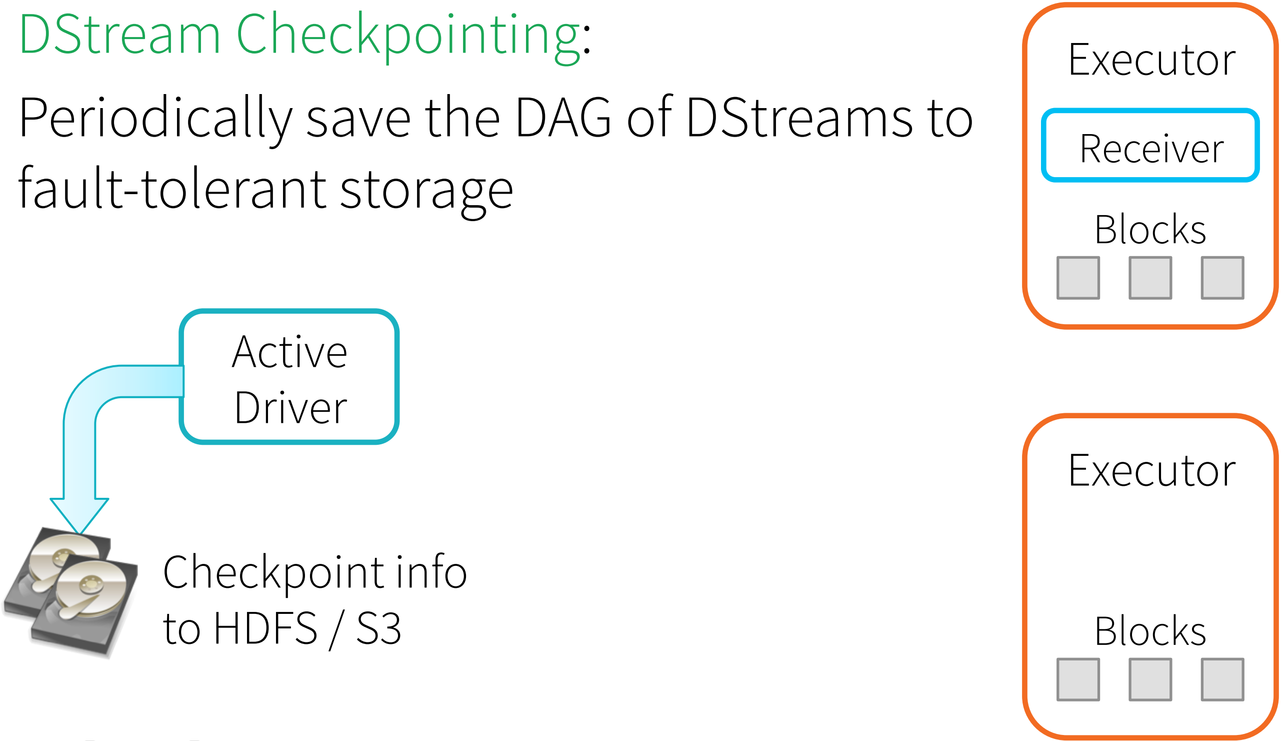
Failed Executor

Blocks

Recovering Driver w/ DStream Checkpointing

DStream Checkpointing:

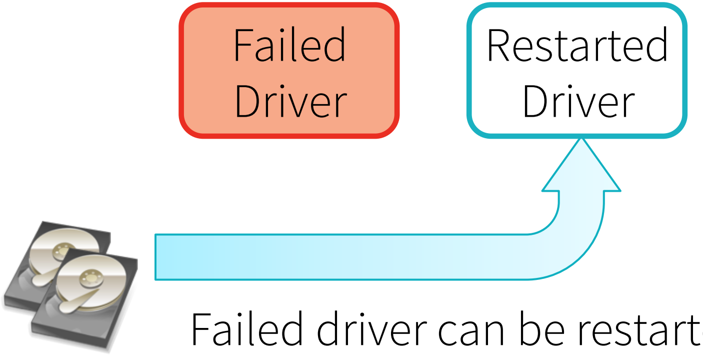
Periodically save the DAG of DStreams to fault-tolerant storage



Recovering Driver w/ DStream Checkpointing

DStream Checkpointing:

Periodically save the DAG of DStreams to fault-tolerant storage

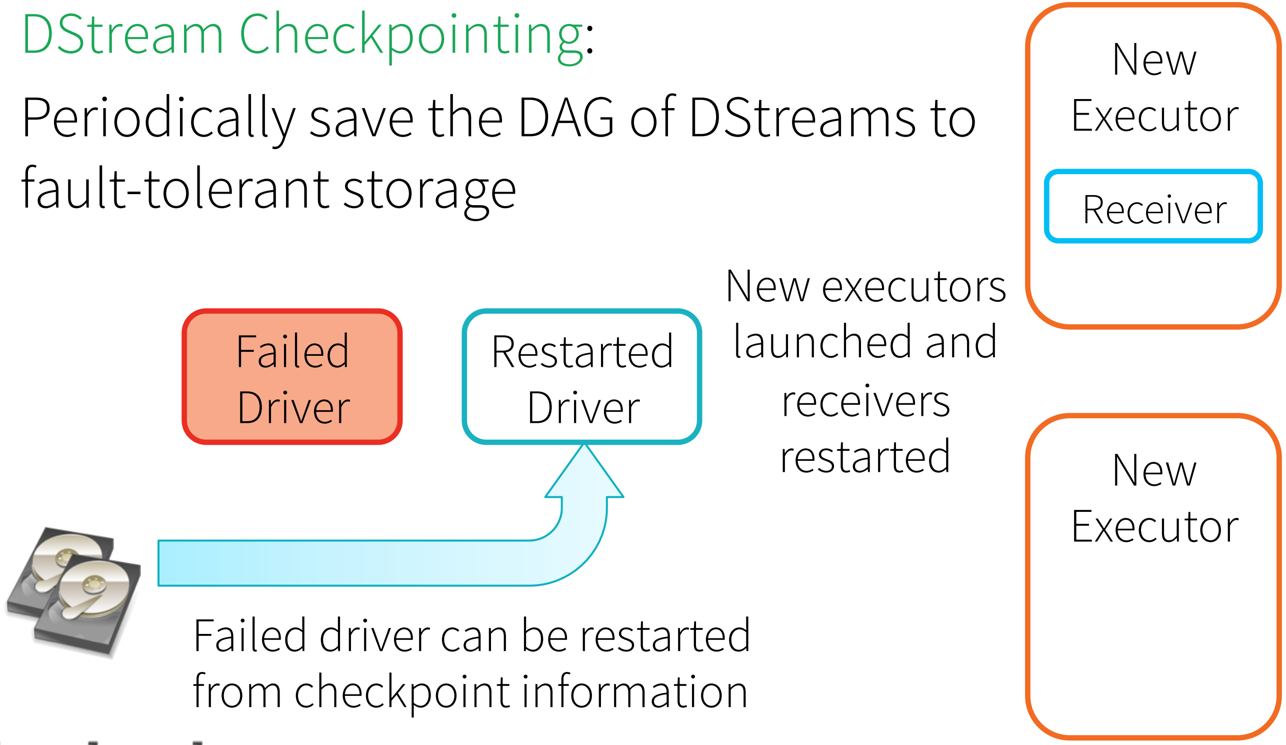


Failed driver can be restarted from checkpoint information

Recovering Driver w/ DStream Checkpointing

DStream Checkpointing:

Periodically save the DAG of DStreams to fault-tolerant storage



Failed driver can be restarted from checkpoint information

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 applications or use the “--supervise” flag.

Restructuring code for Checkpointing

Create
+
Setup

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

Start

```
context.start()
```

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()
```

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():

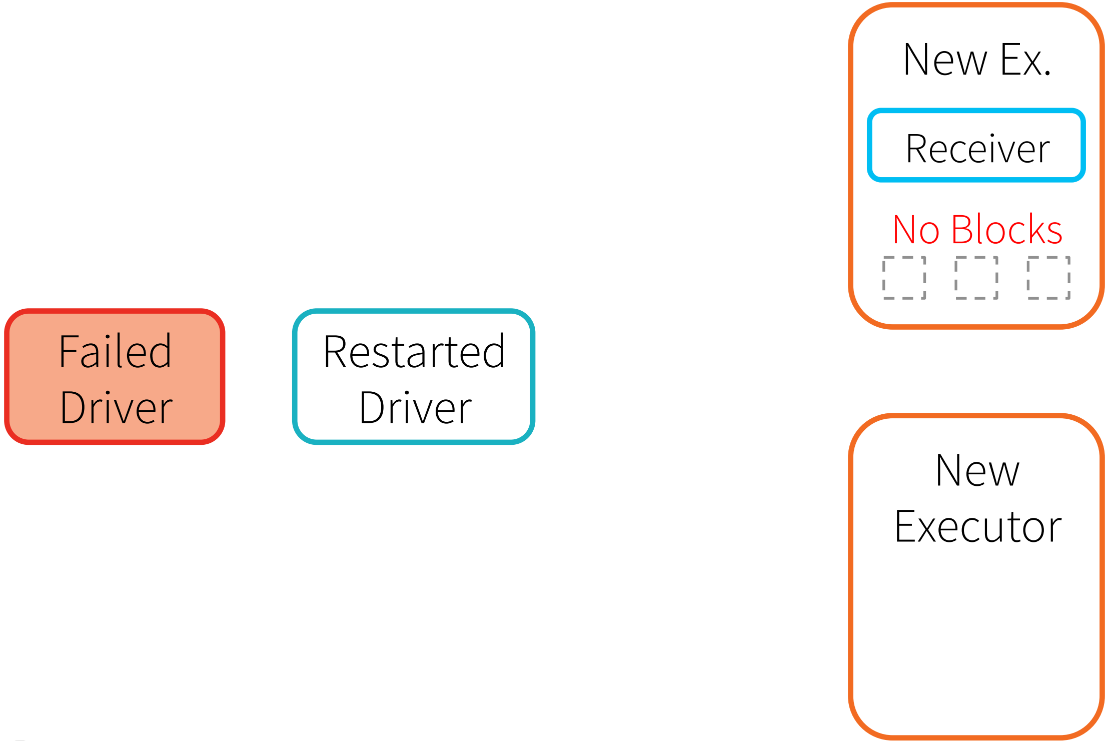
```
If HDFS directory has checkpoint info
        recover context from info
else
        call creatingFunc() to create
        and setup a new context
```

```
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()
```

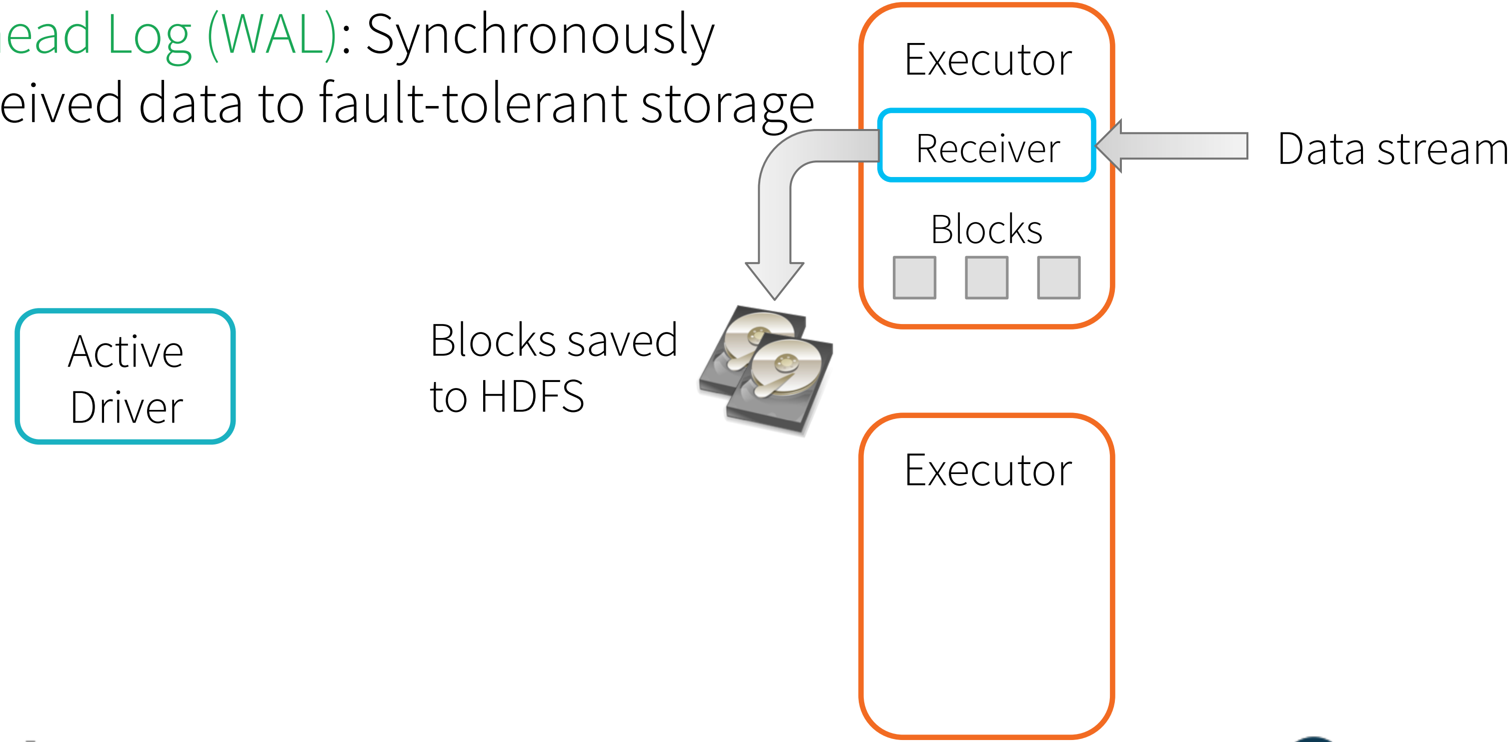
Received blocks lost on Restart!



In-memory blocks of buffered data are lost on driver restart

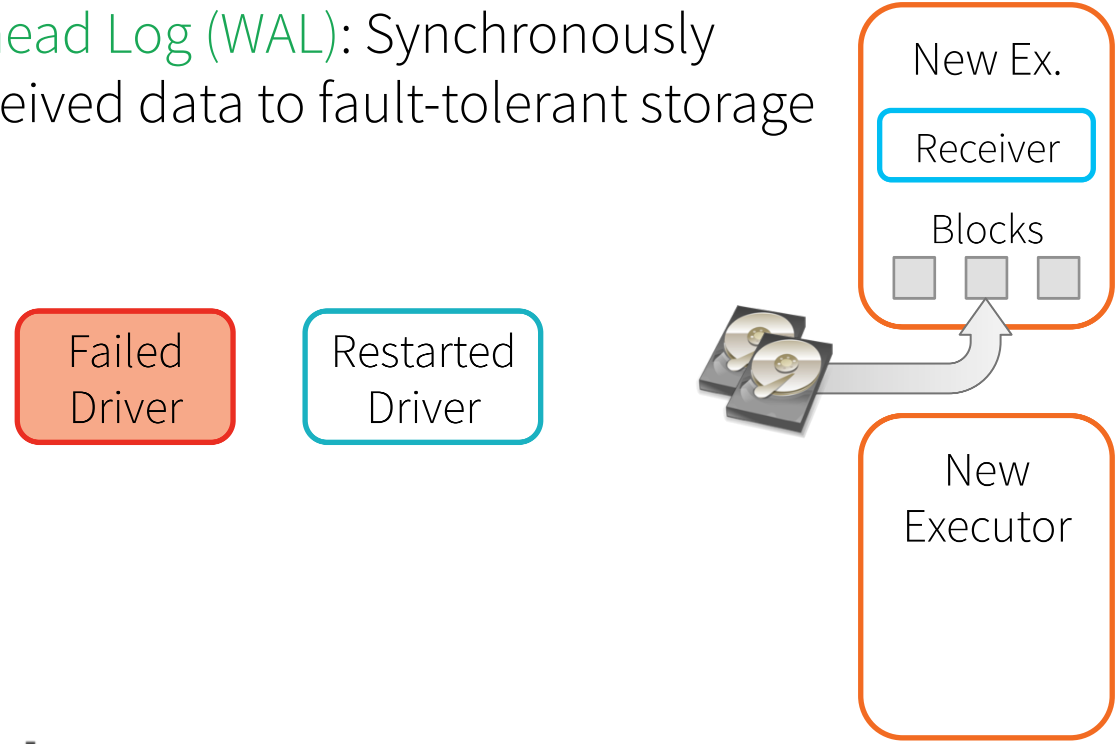
Recovering data with Write Ahead Logs

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



Recovering data with Write Ahead Logs

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



Blocks recovered from Write Ahead Log

Recovering data with Write Ahead Logs

1. Enable checkpointing, logs written in checkpoint directory

3. 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

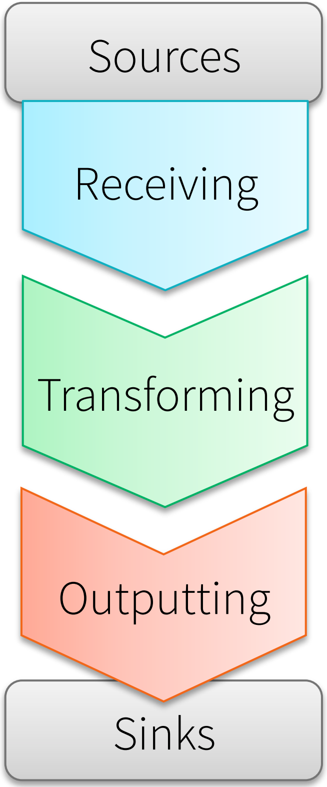
5. Disable in-memory replication (already replicated by HDFS)

   Use `StorageLevel.MEMORY_AND_DISK_SER` for input DStreams

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



Sources

Receiving

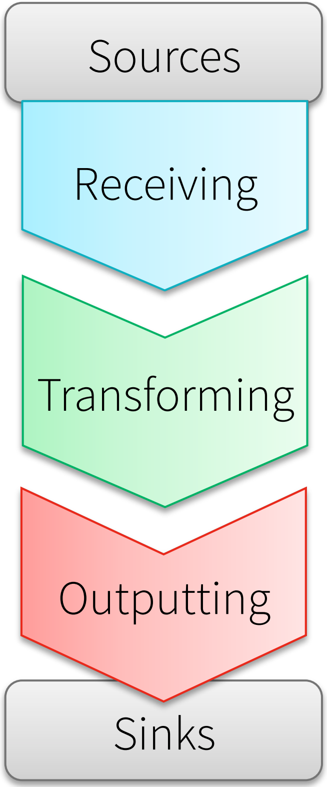
Transforming

Outputting

Sinks

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

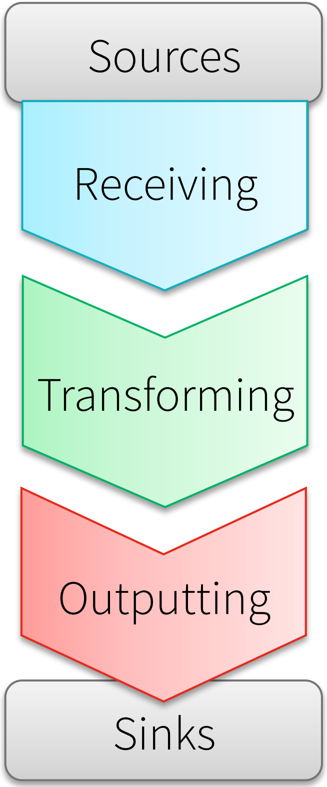
Fault-tolerance Semantics



Exactly once, as long as received data is not lost

End-to-end semantics:
At-least once

Fault-tolerance Semantics

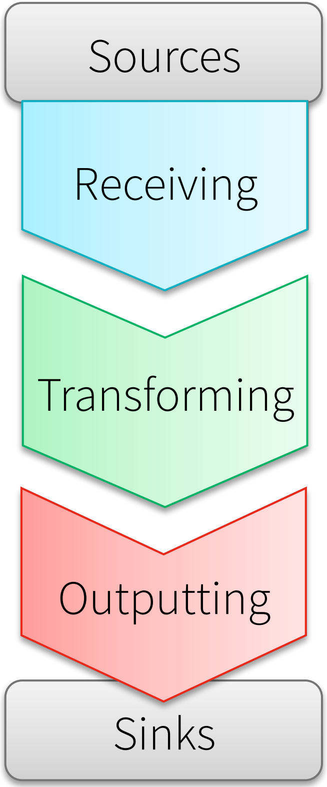


Exactly once, as long as received data is not lost

Exactly once, if outputs are idempotent or transactional

End-to-end semantics:
At-least once

Fault-tolerance Semantics



Sources

Receiving

Transforming

Outputting

Sinks

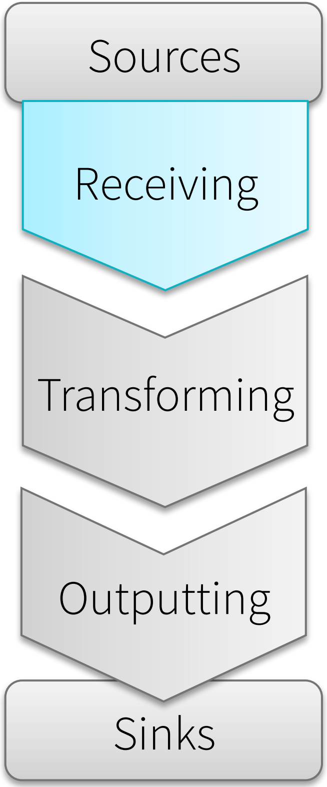
At least once, w/ Checkpointing + WAL + Reliable receivers

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

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

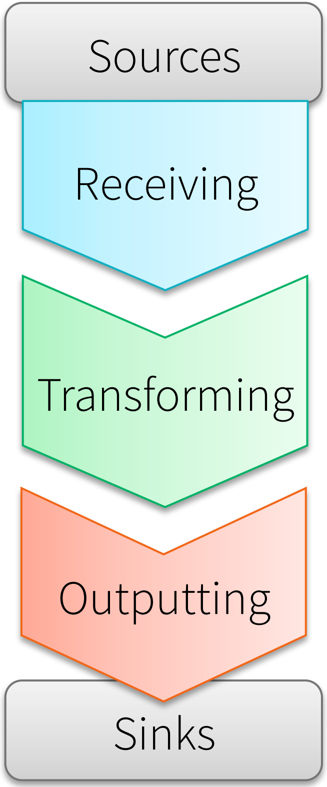
No need to create multiple DStreams and union them

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

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!

Design Tips for Successful Streaming Applications

Areas for consideration

- Enhance resilience with additional components.
- Mini-batch vs. per-message handling.
- Exploit Reactive Streams.

Mini-batch vs. per-message handling

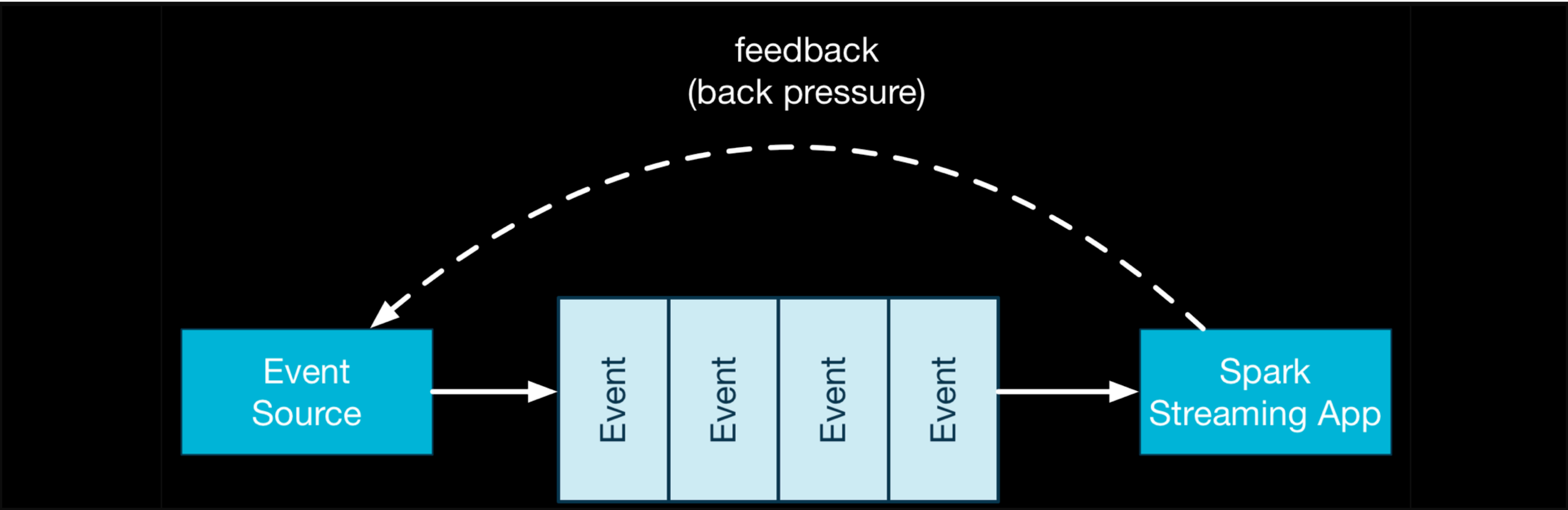
- Use Storm, Akka, Samza, etc. for handling individual messages, especially with sub-second latency requirements.
- Use Spark Streaming's mini-batch model for the Lambda architecture and highly-scalable analytics.

Enhance Resiliency with Additional Components.

- Consider Kafka or Kinesis for resilient buffering in front of Spark Streaming.
  - Buffer for traffic spikes.
  - Re-retrieval of data if an RDD partition is lost and must be reconstructed from the source.
- Going to store the raw data anyway?
  - Do it first, then ingest to Spark from that storage.

Exploit Reactive Streams

- Spark Streaming v1.5 will have support for back pressure to more easily build end-to-end reactive applications

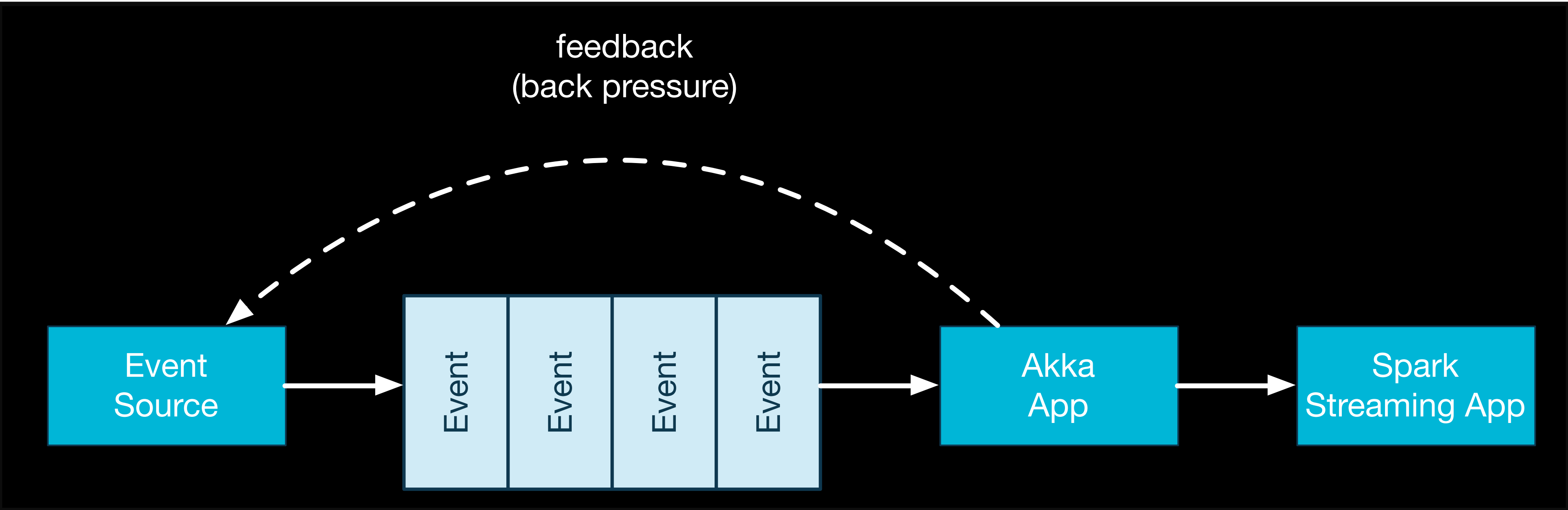


Exploit Reactive Streams

- Spark Streaming v1.5 will have support for back pressure to more easily build end-to-end reactive applications
- Backpressure from consumer to producer:
  - Prevents buffer overflows.
  - Avoids unnecessary throttling.

Exploit Reactive Streams

- Spark Streaming v1.4? Buffer with Akka Streams:



Exploit Reactive Streams

- Spark Streaming v1.4 has a rate limit property:
  - **spark.streaming.receiver.maxRate**
  - Consider setting it for long-running streaming apps with a variable input flow rate.
- Have a graph of Reactive Streams? Consider using an Akka app to buffer the data fed to Spark Streaming over a socket (until 1.5…).

Thank you!

Dean Wampler, Typesafe

Tathagata Das, Databricks

What to do next?

Databricks is available as a hosted platform on AWS with a monthly subscription.

Start with a free trial

Typesafe now offers certified support for Spark, Mesos & DCOS, read more about it

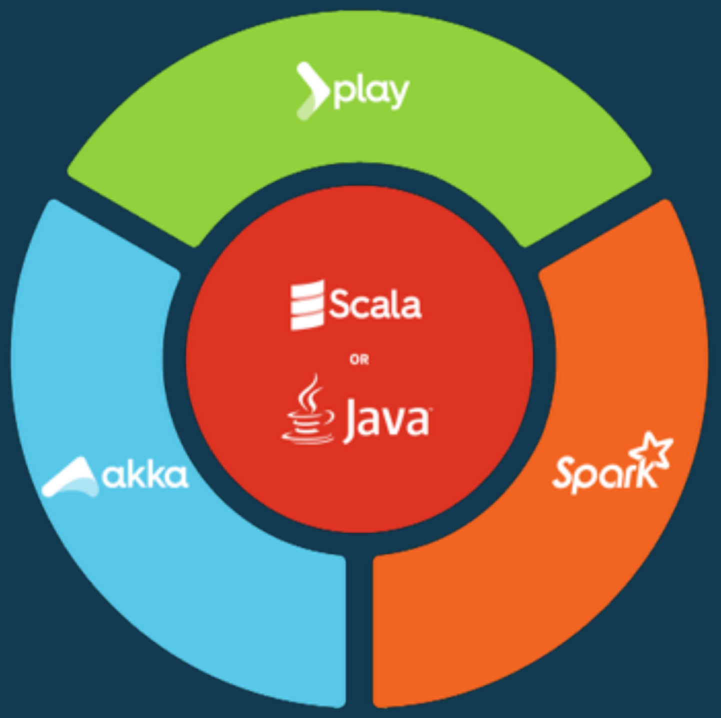
READ MORE

databricks™

Typesafe

REACTIVE PLATFORM

Full Lifecycle Support for Play, Akka, Scala and Spark



Give your project a boost with Reactive Platform:

- **Monitor Message-Driven Apps**
- **Resolve Network Partitions Decisively**
- **Integrate Easily with Legacy Systems**
- **Eliminate Incompatibility & Security Risks**
- **Protect Apps Against Abuse**
- **Expert Support from Dedicated Product Teams**

Enjoy learning? See about the availability of on-site training for Scala, Akka, Play and Spark!

CONTACT US

Learn more about our offers