Spark Streaming: Best Practices

Prakash Chockalingam
@prakash573

databricks™

Who am I ?

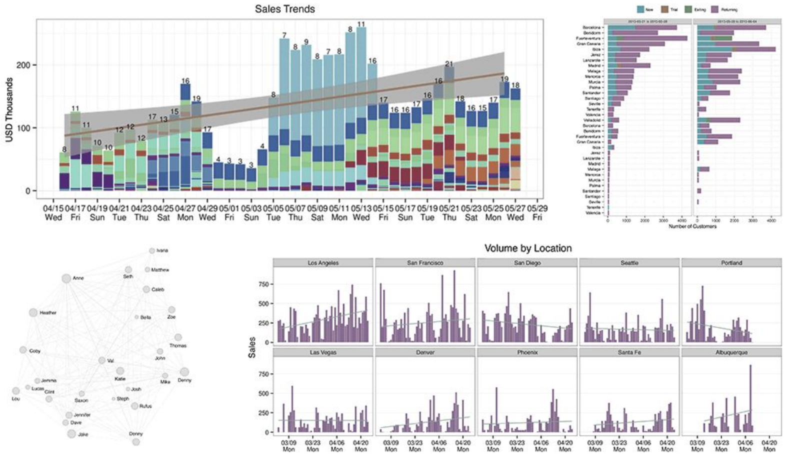
- Solutions Architect / Product Manager at Databricks
- Formerly Netflix, Personalization Infrastructure
- Formerly Yahoo!, Personalized Ad Targeting

About Databricks

Founded by creators of Spark in 2013

Cloud enterprise data platform

- Managed Spark clusters
- Interactive data science
- Production pipelines
- Data governance, security, …



Agenda

- Introduction to Spark Streaming
- Lifecycle of a Spark streaming app
- Aggregations and best practices
- Operationalization tips
- Key benefits of Spark streaming

What is Spark Streaming?

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

Integrate with MLlib, SQL, DataFrames, GraphX

kixer

NETFLIX

produban

CISCO

RelayHealth

DATASTAX

kelkoo

SIGMOID ANALYTICS

FAiMDATA

AsiaInfo

intel

virdata

CLOUDPHYSICS

stratio

sharethrough

cloudera

viadeo

guavus

Pinterest

PEARSON

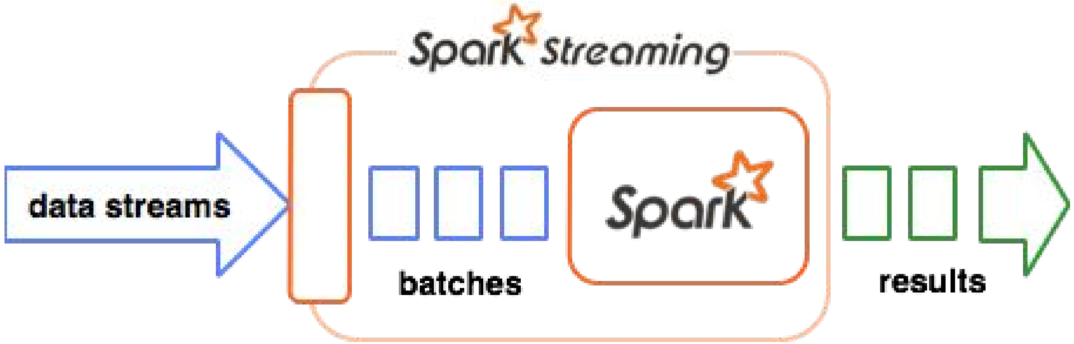
Atigeo

OOYALA

VELOCIDATA

How does it work?

- *Receivers* receive data streams and chops them in to batches.
- Spark processes the batches and pushes out the results



Word Count

Entry point

Batch Interval

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

val lines = context.socketTextStream(...)
```

DStream: represents a data stream

Word Count

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

val lines = context.socketTextStream(...)

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

Transformations: transform data to create new DStreams

Word Count

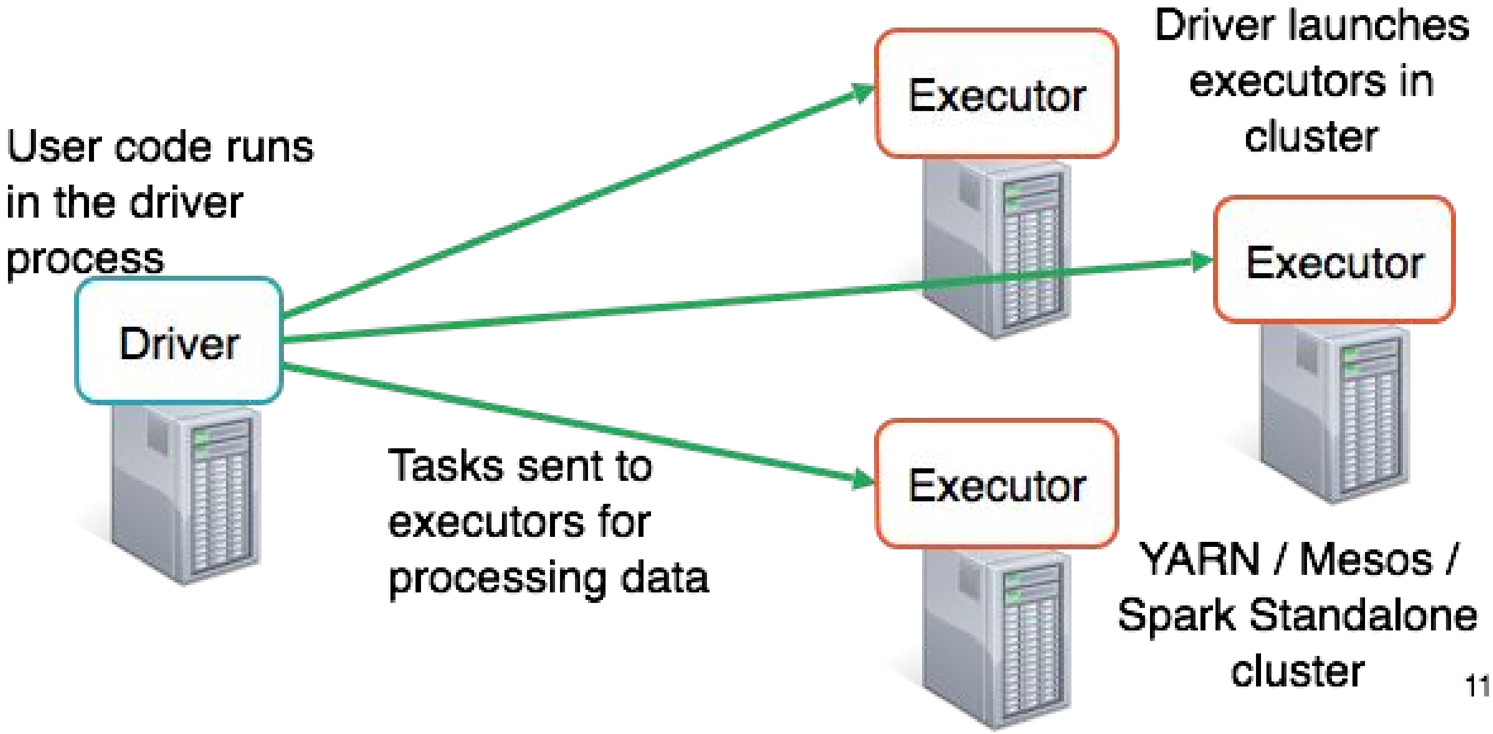
```
val context = new StreamingContext(conf, Seconds(1))

val lines = context.socketTextStream(...)

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

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

wordCounts.print()

context.start()
```

Print the DStream contents on screen

Start the streaming job

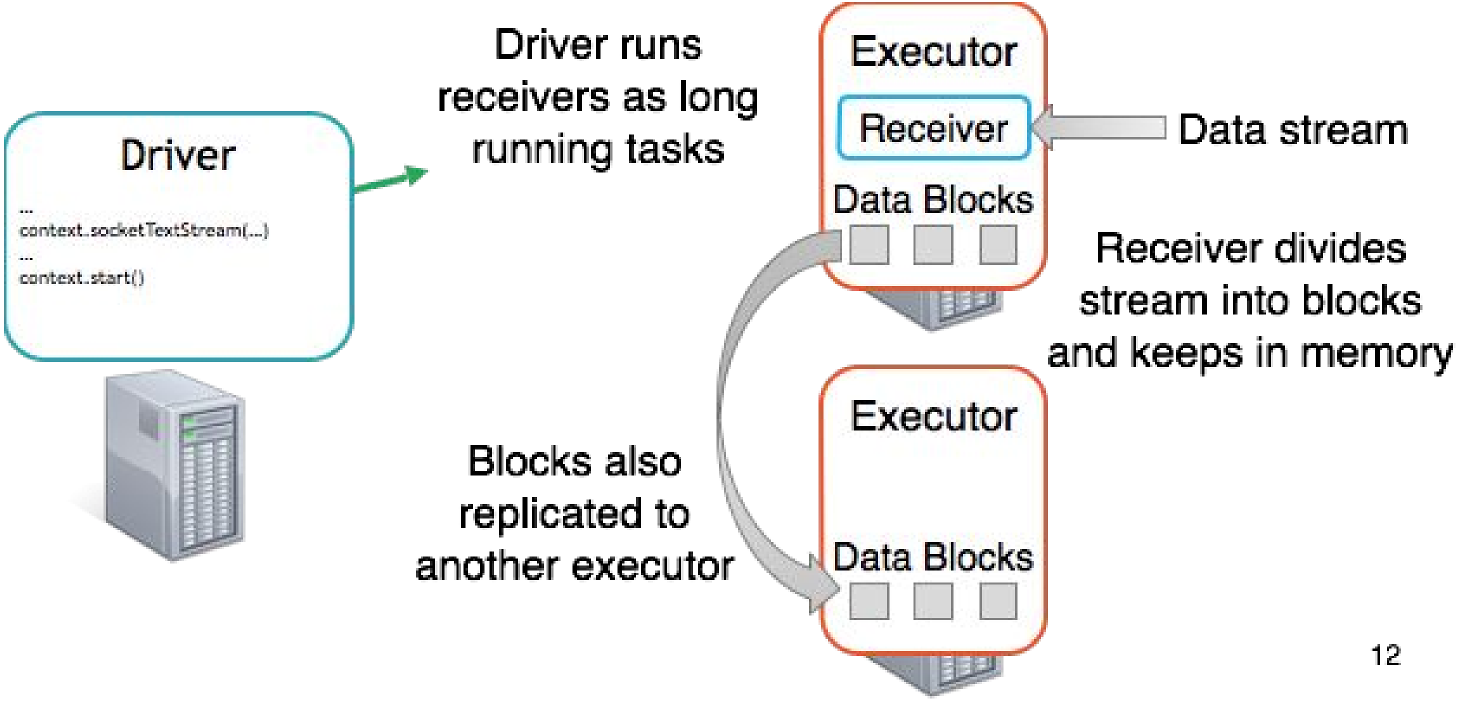
Lifecycle of a streaming app

Execution in any Spark Application

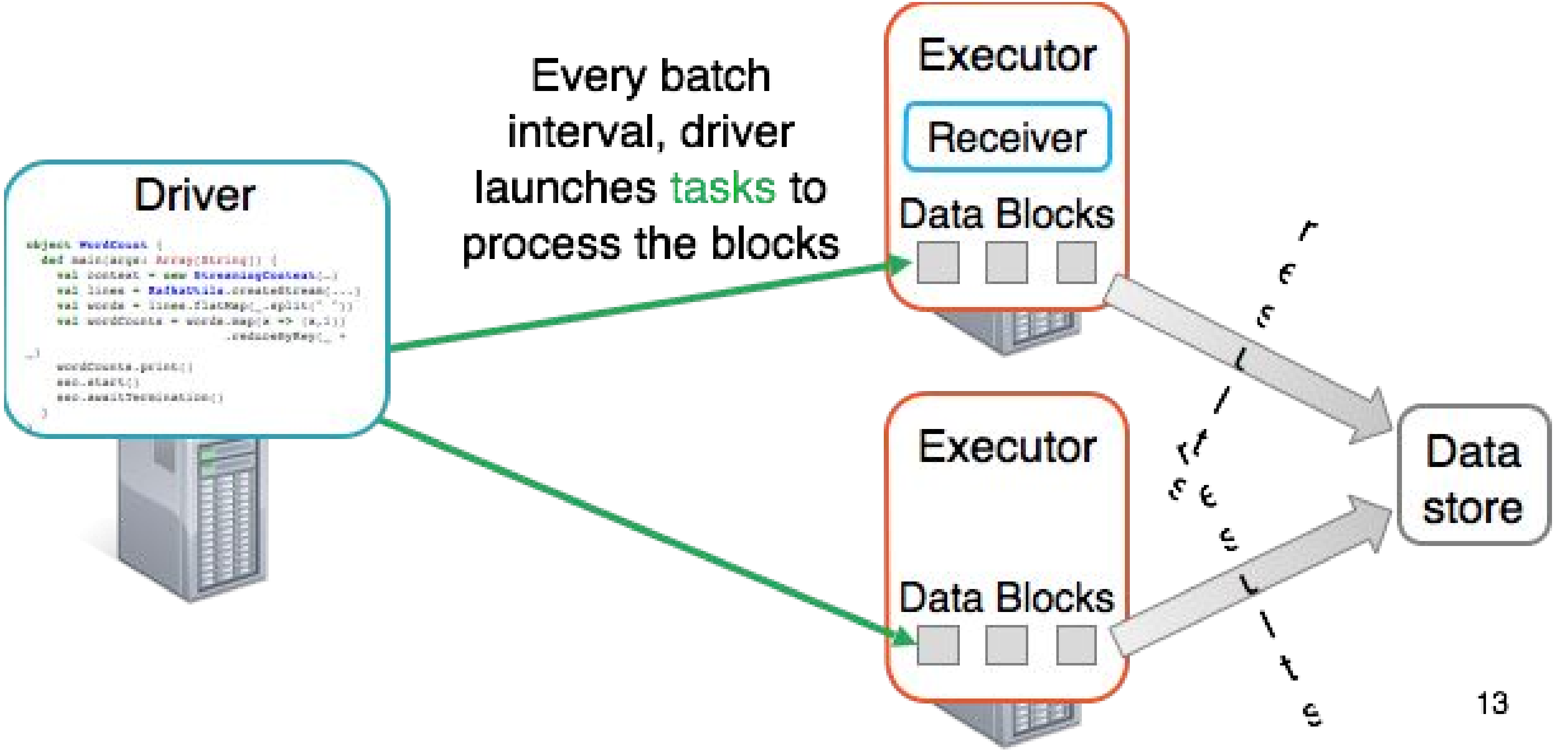


databricks

Execution in Spark Streaming: Receiving data



Execution in Spark Streaming: Processing data



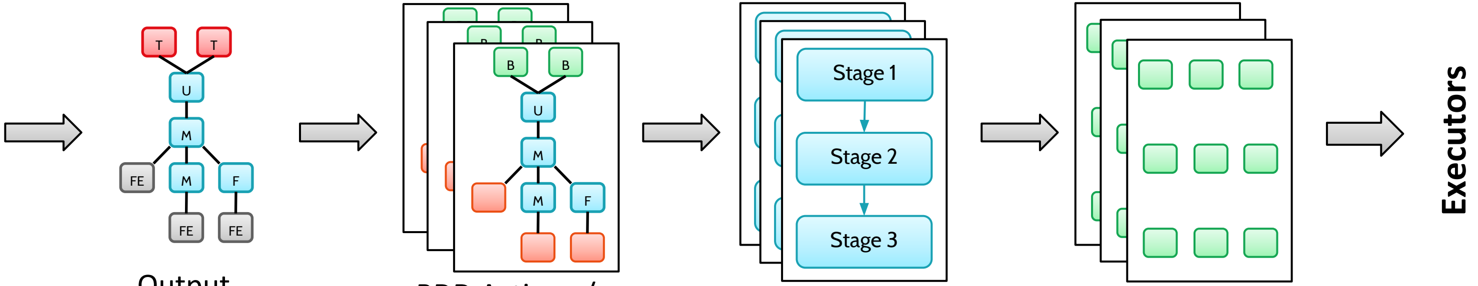
End-to-end view

Streaming app	DStreamGraph	DAG of RDDs every interval	DAG of stages every interval	Tasks every interval

Input DStreams

BlockRDDs

```
t1 = ssc.socketStream("…")
t2 = ssc.socketStream("…")

t = t1.union(t2).map(…)

t.saveAsHadoopFiles(…)
t.map(…).foreach(…)
t.filter(…).foreach(…)
```



Output operations

RDD Actions / Spark Jobs

YOU write this

Spark Streaming JobScheduler + JobGenerator

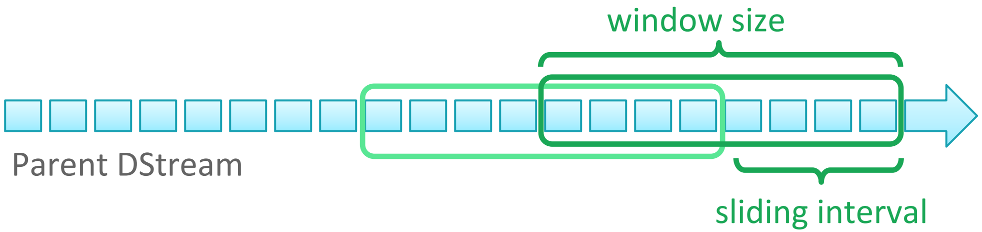
Spark DAGScheduler

Spark TaskScheduler

databricks

Aggregations

Word count over a time window



Reduces over a time window

```
val wordCounts = wordStream.reduceByKeyAndWindow((x:
   Int, y:Int) => x+y, windowSize, slidingInterval)
```

Word count over a time window

Scenario: Word count for the last 30 minutes

How to optimize for good performance?

- Increase batch interval, if possible
- Incremental aggregations with inverse reduce function

```
    val wordCounts = wordStream.reduceByKeyAndWindow((x: Int, y:Int) =>
x+y, (x: Int, y: Int) => x-y, windowSize, slidingInterval)
```

- Checkpointing

```
    wordStream.checkpoint(checkpointInterval)
```

Stateful: Global Aggregations

Scenario: Maintain a global state based on the input events coming in. Ex: Word count from beginning of time.

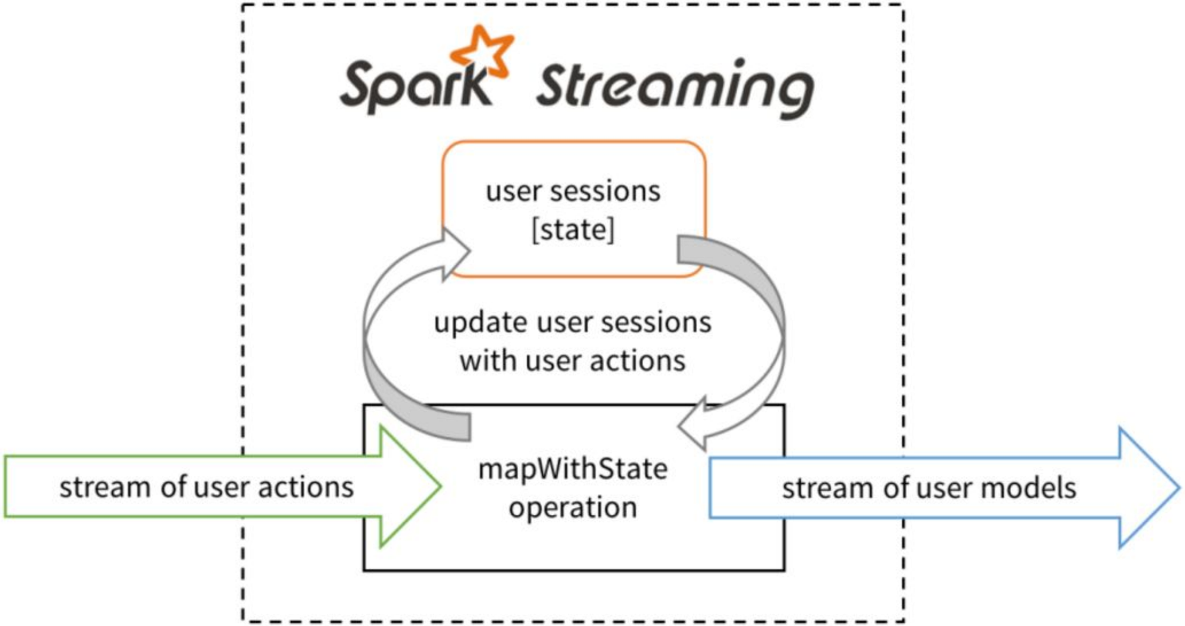
~~updateStateByKey (Spark 1.5 and before)~~

- Performance is proportional to the size of the state.

mapWithState (Spark 1.6+)

- Performance is proportional to the size of the batch.

Stateful: Global Aggregations



Stateful: Global Aggregations

Key features of mapWithState:

- An initial state - Read from somewhere as a RDD
- # of partitions for the state - If you have a good estimate of the size of the state, you can specify the # of partitions.
- Partitioner - Default: Hash partitioner. If you have a good understanding of the key space, then you can provide a custom partitioner
- Timeout - Keys whose values are not updated within the specified timeout period will be removed from the state.

Stateful: Global Aggregations (Word count)

```
val stateSpec = StateSpec.function(updateState _)
                              .initialState(initialRDD)
                              .numPartitions(100)
                              .partitioner(MyPartitioner())
                              .timeout(Minutes(120))
val wordCountState = wordStream.mapWithState(stateSpec)
```

Stateful: Global Aggregations (Word count)

```
def updateState(batchTime: Time,
                key: String,
                value: Option[Int],
                state: State[Long])
            : Option[(String, Long)]
```

Current batch time

A Word in the input stream

Current value (= 1)

Counts so far for the word

The word and its new count

Operationalization

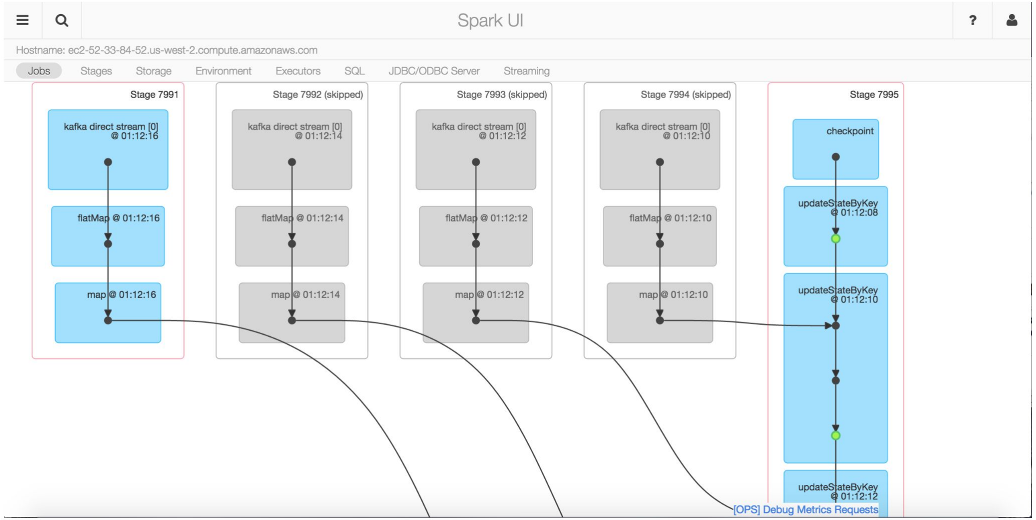
Checkpoint

Two types of checkpointing:

- Checkpointing Data
- Checkpointing Metadata

Checkpoint Data

- Checkpointing DStreams
  - Primarily needed to cut long lineage on past batches (`updateStateByKey/reduceByKeyAndWindow`).
  - Example: `wordStream.checkpoint(checkpointInterval)`



Checkpoint Metadata

- Checkpointing Metadata
  - All the configuration, DStream operations and incomplete batches are checkpointed.
  - Required for failure recovery if the driver process crashes.
  - Example: `streamingContext.checkpoint(directory)`

Active Batches (1)

Batches currently being processed or queued

Batch Time	Input Size	Scheduling Delay (?)	Processing Time (?)	Status
2015/11/09 01:10:08	0 events	1 ms	-	processing

Achieving good throughput

```
context.socketStream(...)
.map(...)
.filter(...)
.saveAsHadoopFile(...)
```

Problem: There will be 1 receiver which receives all the data and stores it in its executor and all the processing happens on that executor. Adding more nodes doesn't help.

Achieving good throughput

Solution: Increase the # of receivers and union them.

- Each receiver is run in 1 executor. Having 5 receivers will ensure that the data gets received in parallel in 5 executors.
- Data gets distributed in 5 executors. So all the subsequent Spark map/filter operations will be distributed

```
val numStreams = 5
val inputStreams = (1 to numStreams).map(i => context.
   socketStream(...))
val fullStream = context.union(inputStreams)
fullStream.map(...).filter(...).saveAsHadoopFile(...)
```

Achieving good throughput

- In the case of direct receivers (like Kafka), set the appropriate # of partitions in Kafka.
- Each kafka paratition gets mapped to a Spark partition.
- More partitions in Kafka = More parallelism in Spark

Achieving good throughput

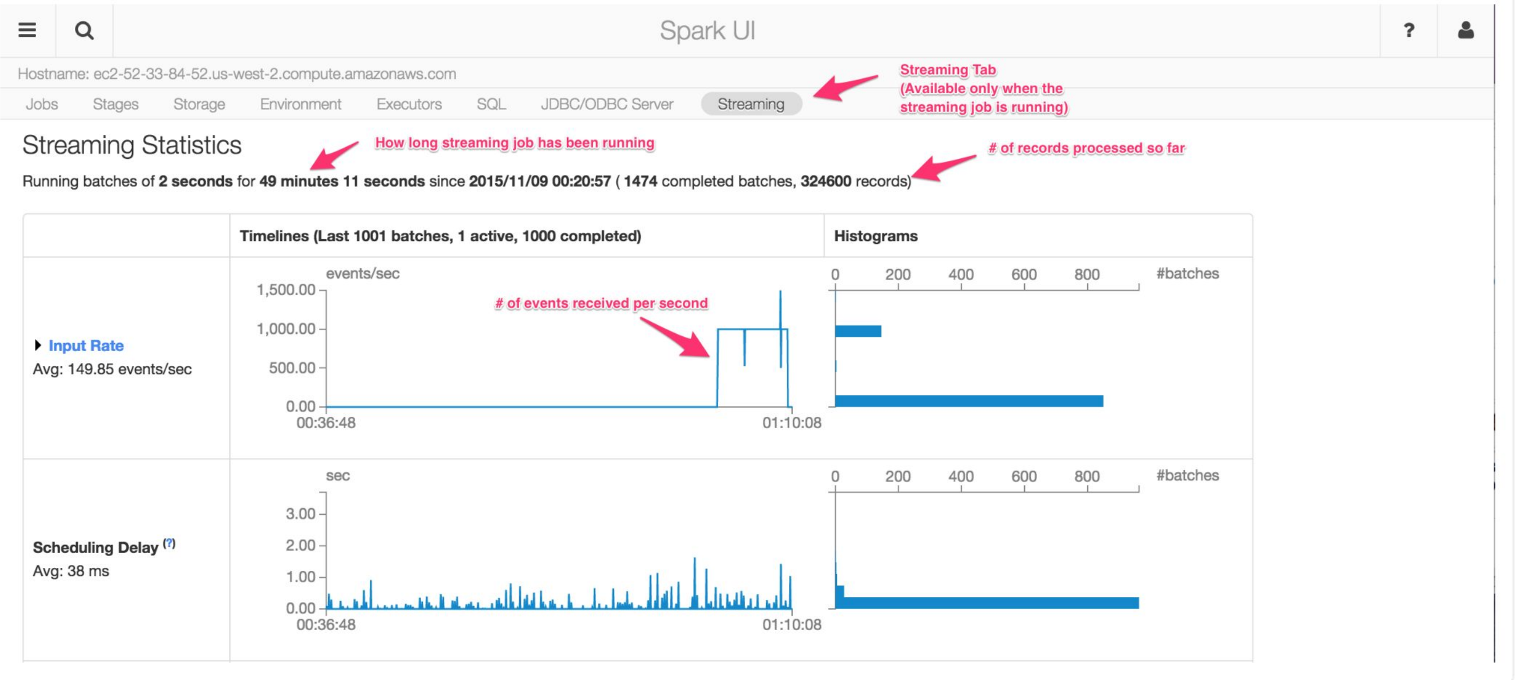
- Provide the right # of partitions based on your cluster size for operations causing shuffles.

```
words.map(x => (x, 1)).reduceByKey(_+_, 100)
```

# of partitions

Debugging a Streaming application

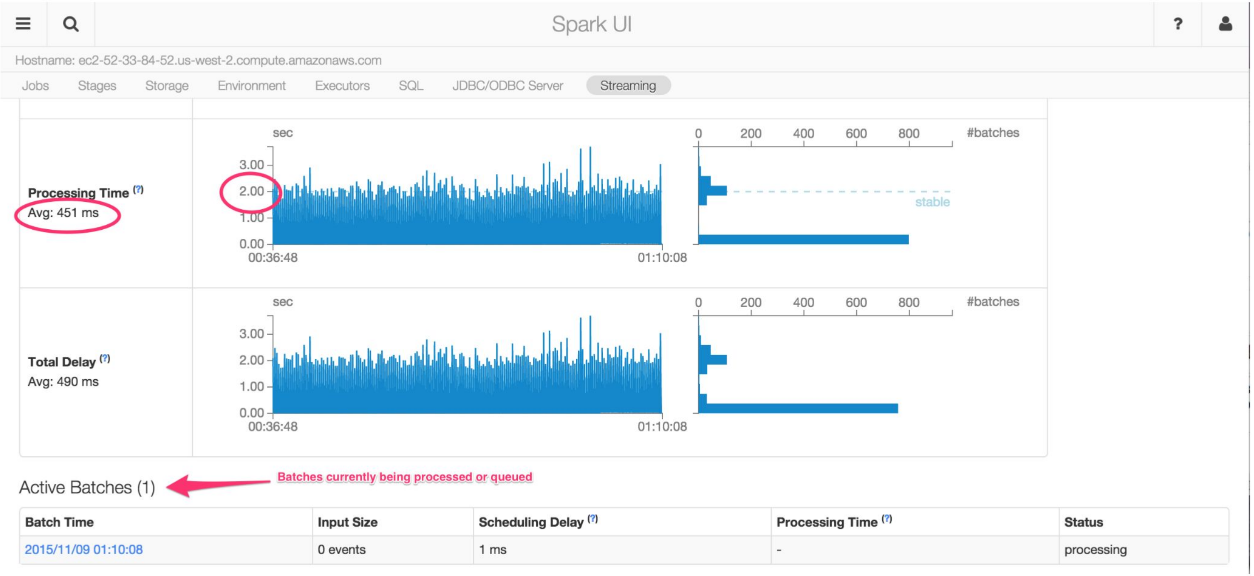
Streaming tab in Spark UI



Debugging a Streaming application

Processing Time

- Make sure that the processing time < batch interval



Debugging a Streaming application

Spark UI

Hostname: ec2-52-33-84-52.us-west-2.compute.amazonaws.com

Jobs Stages Storage Environment Executors SQL JDBC/ODBC Server Streaming

Completed Batches (last 1000 out of 1533)

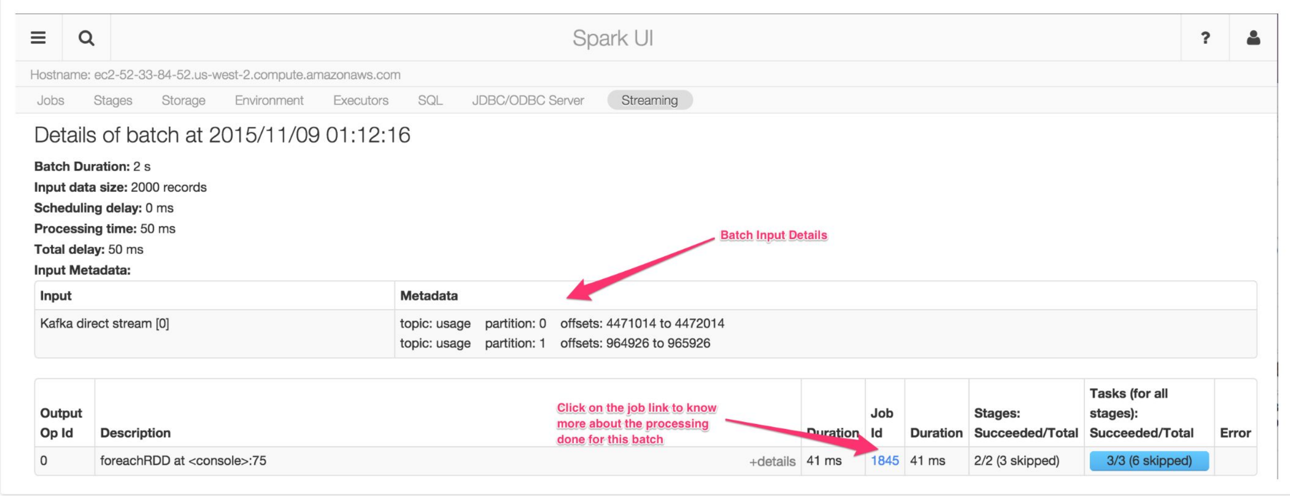
Click on any of the completed batches to know more details

Batch Time	Input Size	Scheduling Delay (?)	Processing Time (?)	Total Delay (?)
2015/11/09 01:12:04	2000 events	1 ms	50 ms	51 ms
2015/11/09 01:12:02	2000 events	0 ms	57 ms	57 ms
2015/11/09 01:12:00	2000 events	1 ms	57 ms	58 ms
2015/11/09 01:11:58	2000 events	0 ms	2 s	2 s
2015/11/09 01:11:56	2000 events	0 ms	50 ms	50 ms
2015/11/09 01:11:54	2000 events	1 ms	50 ms	51 ms
2015/11/09 01:11:52	2000 events	0 ms	51 ms	51 ms
2015/11/09 01:11:50	2000 events	1.0 s	53 ms	1 s
2015/11/09 01:11:48	2000 events	0 ms	3 s	3 s
2015/11/09 01:11:46	2000 events	0 ms	52 ms	52 ms
2015/11/09 01:11:44	2000 events	0 ms	50 ms	50 ms
2015/11/09 01:11:42	2000 events	0 ms	49 ms	49 ms
2015/11/09 01:11:40	2000 events	0.4 s	51 ms	0.4 s
2015/11/09 01:11:38	2000 events	0 ms	2 s	2 s

Debugging a Streaming application

Batch Details Page:

- Input to the batch
- Jobs that were run as part of the processing for the batch



Debugging a Streaming application

Job Details Page

- DAG Visualization
- Stages of a Spark job

Completed Stages (2)

Click on any of the stage links to debug more on the stage

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
7995	762482694511347961	Streaming job from [output operation 0, batch time 01:12:16] foreachRDD at <console>:75 +details	2015/11/09 01:12:16	19 ms	1/1	208.0 B		22.2 KB	
7991	762482694511347961	Streaming job from [output operation 0, batch time 01:12:16] map at <console>:72 +details	2015/11/09 01:12:16	17 ms	2/2				27.6 KB

Skipped Stages (3)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
7994		map at <console>:72 +details	Unknown	Unknown	0/2				
7993		map at <console>:72 +details	Unknown	Unknown	0/2				
7992		map at <console>:72 +details	Unknown	Unknown	0/2				

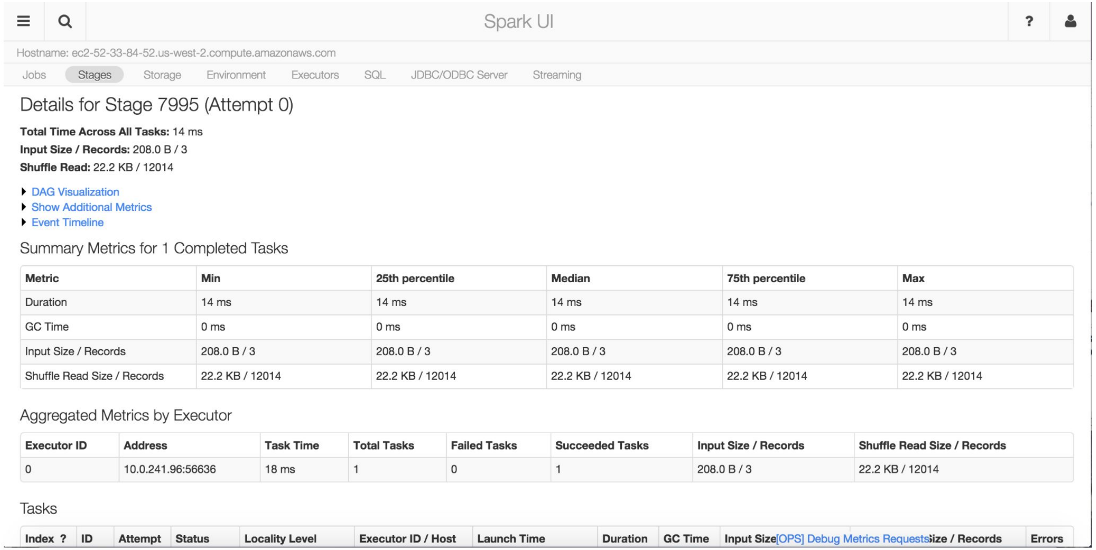
[OPS] Debug Metrics Requests

Debugging a Streaming application

Task Details Page

Ensure that the tasks are executed on multiple executors (nodes) in your cluster to have enough parallelism while processing.

If you have a single receiver, sometimes only one executor might be doing all the work though you have more than one executor in your cluster.



Spark UI

Hostname: ec2-52-33-84-52.us-west-2.compute.amazonaws.com

Jobs Stages Storage Environment Executors SQL JDBC/ODBC Server Streaming

Details for Stage 7995 (Attempt 0)

Total Time Across All Tasks: 14 ms
Input Size / Records: 208.0 B / 3
Shuffle Read: 22.2 KB / 12014

- DAG Visualization
- Show Additional Metrics
- Event Timeline

Summary Metrics for 1 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	14 ms	14 ms	14 ms	14 ms	14 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Input Size / Records	208.0 B / 3	208.0 B / 3	208.0 B / 3	208.0 B / 3	208.0 B / 3
Shuffle Read Size / Records	22.2 KB / 12014	22.2 KB / 12014	22.2 KB / 12014	22.2 KB / 12014	22.2 KB / 12014

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input Size / Records	Shuffle Read Size / Records
0	10.0.241.96:56636	18 ms	1	0	1	208.0 B / 3	22.2 KB / 12014

Tasks

Index ?	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Input Size[OPS] Debug Metrics Requestsize / Records	Errors

Key benefits of Spark streaming

Dynamic Load Balancing

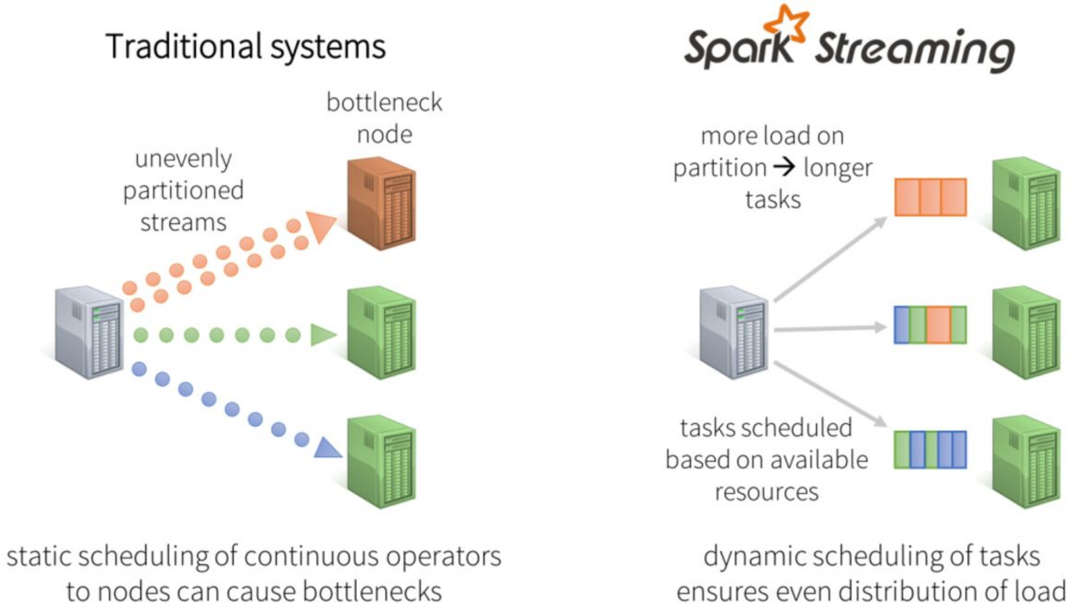


Figure 3: Dynamic load balancing

Fast failure and Straggler recovery

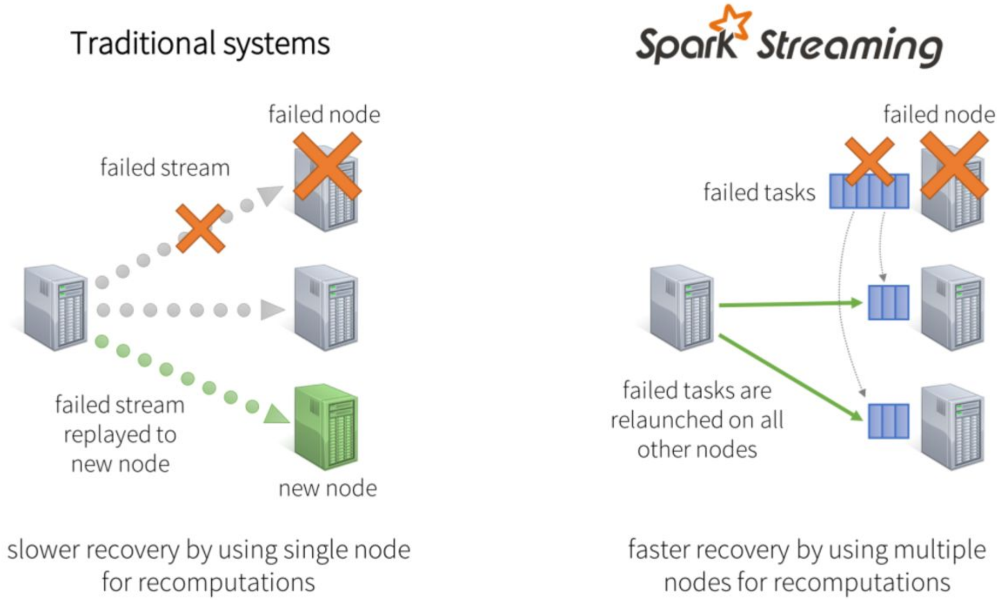


Figure 4: Faster failure recovery with redistribution of computation

databricks™

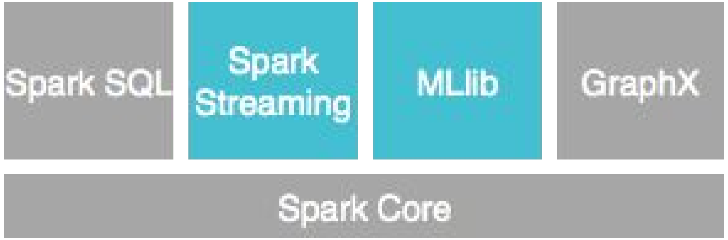
Combine Batch and Stream Processing

Join data streams with static data sets


```
val dataset = sparkContext.hadoopFile("file")
…
kafkaStream.transform{ batchRdd =>
     batchRdd.join(dataset).filter(...)
}
```

Combine ML and Stream Processing

Learn models offline, apply them online




```
val model = KMeans.train(dataset, …)
kakfaStream.map { event =>
    model.predict(event.feature)
}
```

Combine SQL and Stream Processing


```
inputStream.foreachRDD{ rdd =>
    val df = SQLContext.createDataframe(rdd)
    df.select(...).where(...).groupBy(...)
}
```

Thank you.