CASSANDRA SUMMIT 2016

Rocco Varela
Software Engineer in Test, DataStax Analytics Team

Spark Streaming Fault Tolerance with DataStax Enterprise 5.0

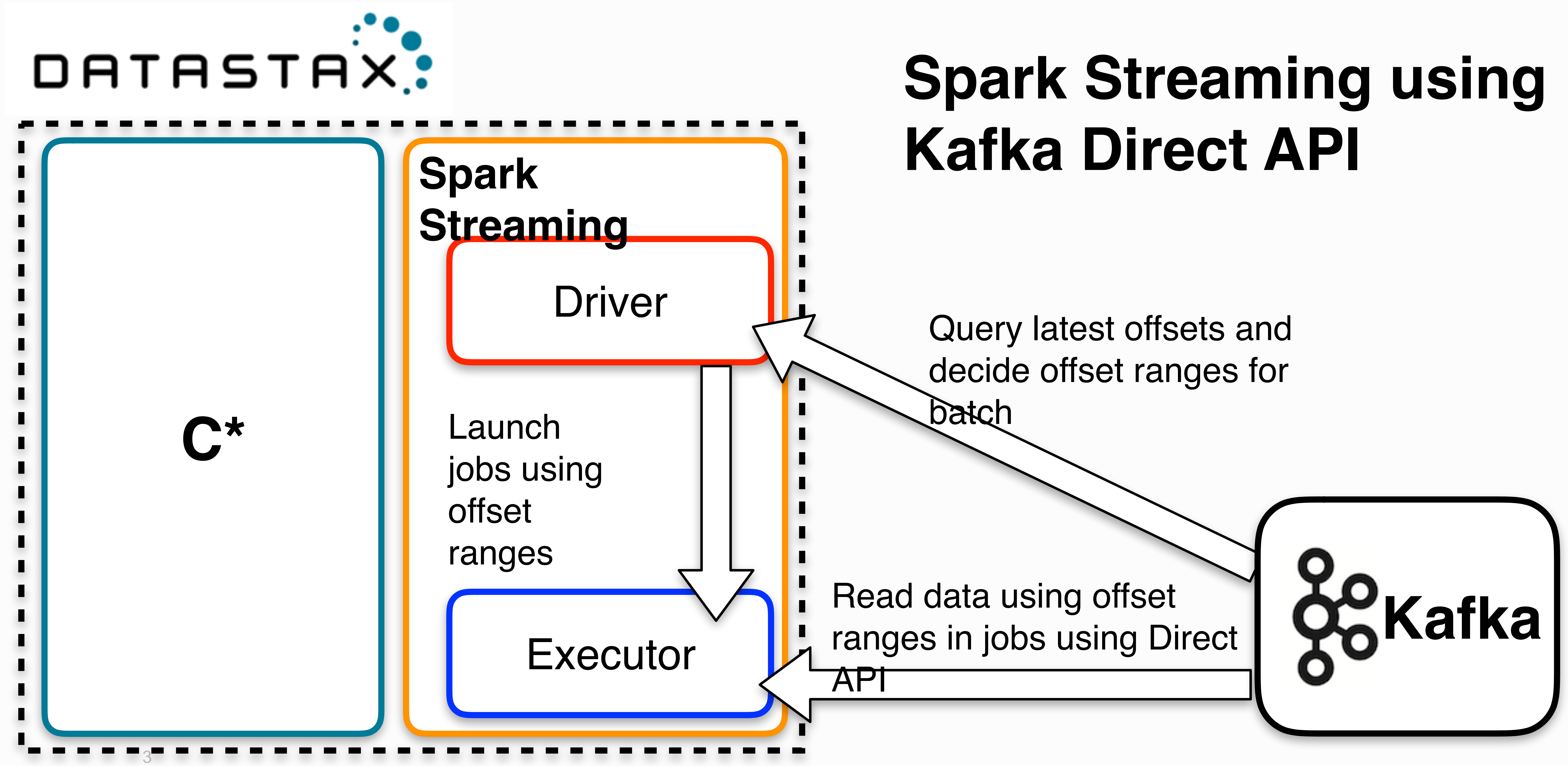
© DataStax, All Rights Reserved.

CASSANDRA SUMMIT2016

DSE supports Spark Streaming fault-tolerance

DataStax is a great place to run Spark Streaming, but applications can die and may need special care.

The Spark-Cassandra stack on DataStax Enterprise (DSE) provides all the fault-tolerant requirements needed for Spark Streaming.

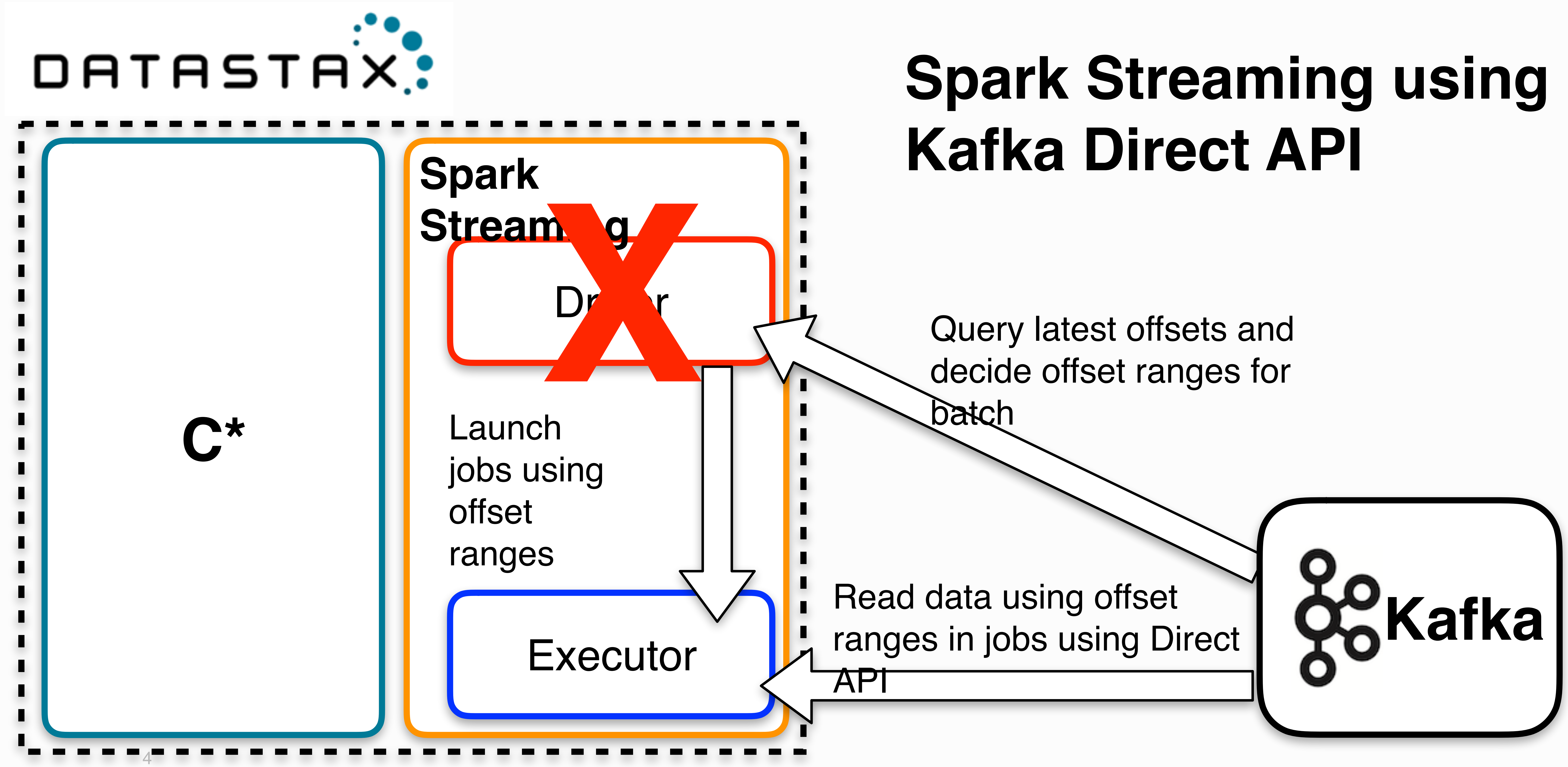

DATASTAX
Spark Streaming using Kafka Direct API
C*
Spark Streaming
Driver
Launch jobs using offset ranges
Executor
Query latest offsets and decide offset ranges for batch
Read data using offset ranges in jobs using Direct API
Kafka

CASSANDRA SUMMIT2016

We should be prepared for driver failures

If the driver node running the Spark Streaming application fails, then the Spark Context is lost, and all executors with their in-memory data are lost as well.

How can we protect ourselves from this scenario?


DATASTAX
Spark Streaming using Kafka Direct API
C*
Spark Streaming
Driver
Launch jobs using offset ranges
Executor
Query latest offsets and decide offset ranges for batch
Read data using offset ranges in jobs using Direct API
Kafka

Checkpoints are used for recovery automatically

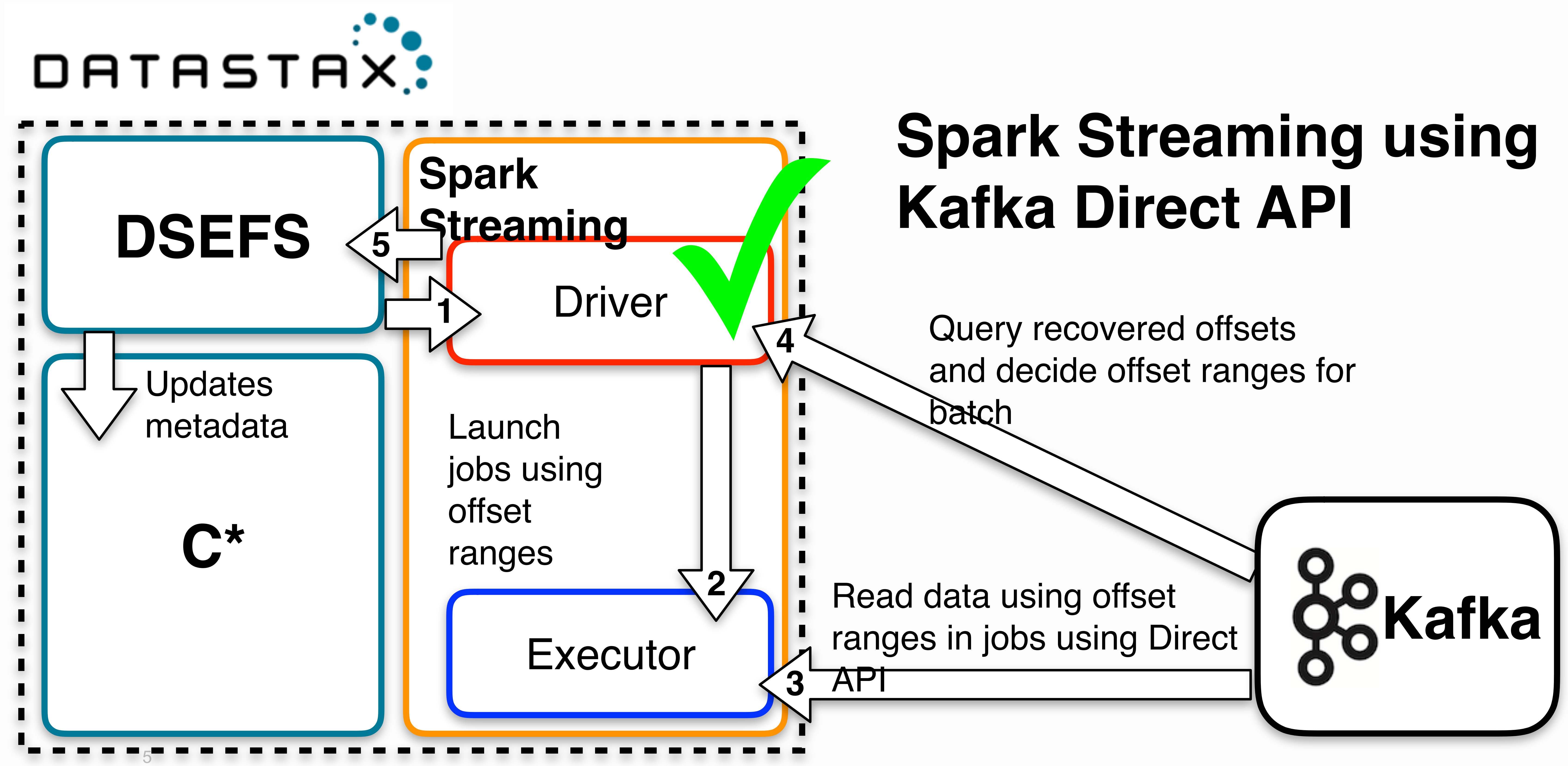
Checkpointing is basically saving a snapshot of an application's state, which can be used for recovery in case of failure.

DataStax Enterprise Filesystem (DSEFS) is our reliable storage mechanism, **new in DSE 5.0**.

We can run Spark Streaming with fault-tolerance without HDFS, or any other filesystem.

1 Retrieves checkpoint

5 Write checkpoint



DATASTAX
DSEFS
Spark Streaming
Driver
5
1
Updates metadata
C*
Launch jobs using offset ranges
2
Executor
4
Query recovered offsets and decide offset ranges for batch
Read data using offset ranges in jobs using Direct API
3
Kafka

Spark Streaming using Kafka Direct API

CASSANDRA SUMMIT2016

DataStax Enterprise Filesystem is our reliable storage

DataStax Enterprise Filesystem (DSEFS) is our reliable storage mechanism, new in DSE 5.0.

At the right shows a snapshot of the DSEFS shell with some utilities we can use to examine our data.

Improvements over Cassandra Filesystem (CFS) include:

- Broader support for the HDFS API:
  - append, hflush, getFileBlockLocations
- More fine-grained file replication
- Designed for optimal performance

DSEFS Shell

```
$ dse fs

dsefs / > help
append          Append a local file to a remote file
cat             Concatenate files and print on the
                standard output
cd              Change remote working directory
df              List file system status and disk space
usage
exit            Exit shell client
fsck            Check filesystem and repair errors
get             Copy a DSE file to the local filesystem
help            Display this list
ls              List directory contents
mkdir           Make directories
mv              Move a file or directory
put             Copy a local file to the DSE filesystem
rename          Rename a file or directory without moving
                it to a different directory
rm              Remove files or directories
rmdir           Remove empty directories
stat            Display file status
truncate        Truncate files
umount          Unmount file system storage locations
```

We can benchmark DSEFS with cfs-stress

Usage:

```
$ cfs-stress [options] filesystem_directory
```

Options (truncated):

Short form	Long form	Description
-s	--size	Size of each file in KB. Default 1024.
-n	--count	Total number of files read/written. Default 100.
-t	--threads	Number of threads.
-r	--streams	Maximum number of streams kept open per thread. Default 2.
-o	--operation	(R)ead, (W)rite, (WR) write & read, (WRD) write, read, delete

Example output:

```
$ cfs-stress [options] filesystem_directory

$ cfs-stress -s 64000 -n 1000 -t 8 -r 2 -o WR --shared-cfs dsefs:///stressdir
Warming up...
Writing and reading 65536 MB to/from dsefs:/cfs-stress-test in 1000 files.
progress        bytes      curr rate       avg rate     max latency
   0.7%      895.9 MB     0.0 MB/s     983.4 MB/s       35.354 ms
   0.8%     1048.6 MB   984.2 MB/s     546.4 MB/s       23.390 ms
...
  99.4%   130252.9 MB   652.3 MB/s     764.3 MB/s      162.944 ms
  99.8%   130761.1 MB   668.5 MB/s     762.8 MB/s       23.540 ms
 100.0%   131072.0 MB   491.2 MB/s     762.7 MB/s        5.776 ms
```

Data Output	Description
progress	Total progress of the stress operation
bytes	Total bytes written/read
curr rate	Current rate of bytes being written/read per second
avg rate	Average rate of bytes being written/read per second
max latency	Maximum latency in milliseconds during the current reporting window

http://docs.datastax.com/en/latest-dse/datastax_enterprise/tools/CFSstress.html?hl=cfs,stress

We can deploy multiple DSE Filesystems across multiple data centers

Each datacenter should maintain a **separate** instance of DSEFS.

1. In the dse.yaml files, specify a separate DSEFS keyspace for each data center.

On each DC1 node:

```
dsefs_options:
    ...
    keyspace_name: dsefs1
```

On each DC2 node:

```
dsefs_options:
    ...
    keyspace_name: dsefs2
```

2. Set the keyspace replication appropriately for each datacenter.

On each DC1 node:

```
ALTER KEYSPACE dsefs1 WITH replication = {
  'class': 'NetworkTopologyStrategy',
  'DC1': '3'
};
```

On each DC2 node:

```
ALTER KEYSPACE dsefs2 WITH replication = {
  'class': 'NetworkTopologyStrategy',
  'DC2': '3'
};
```

https://docs.datastax.com/en/latest-dse/datastax_enterprise/ana/configUsingDsefs.html#configUsingDsefs__configMultiDC

DSE Filesystem supports JBOD

- Data directories can be setup with JBOD (Just a bunch of disks)
- **`storage_weight`** controls how much data is stored in each directory relative to other directories
- Directories should be mounted on different physical devices
- If a server fails, we can move the storage to another node
- Designed for efficiency:
  - Does not use JVM heap for FS data
  - Uses async I/O and Cassandra queries
  - Uses thread-per-core threading model

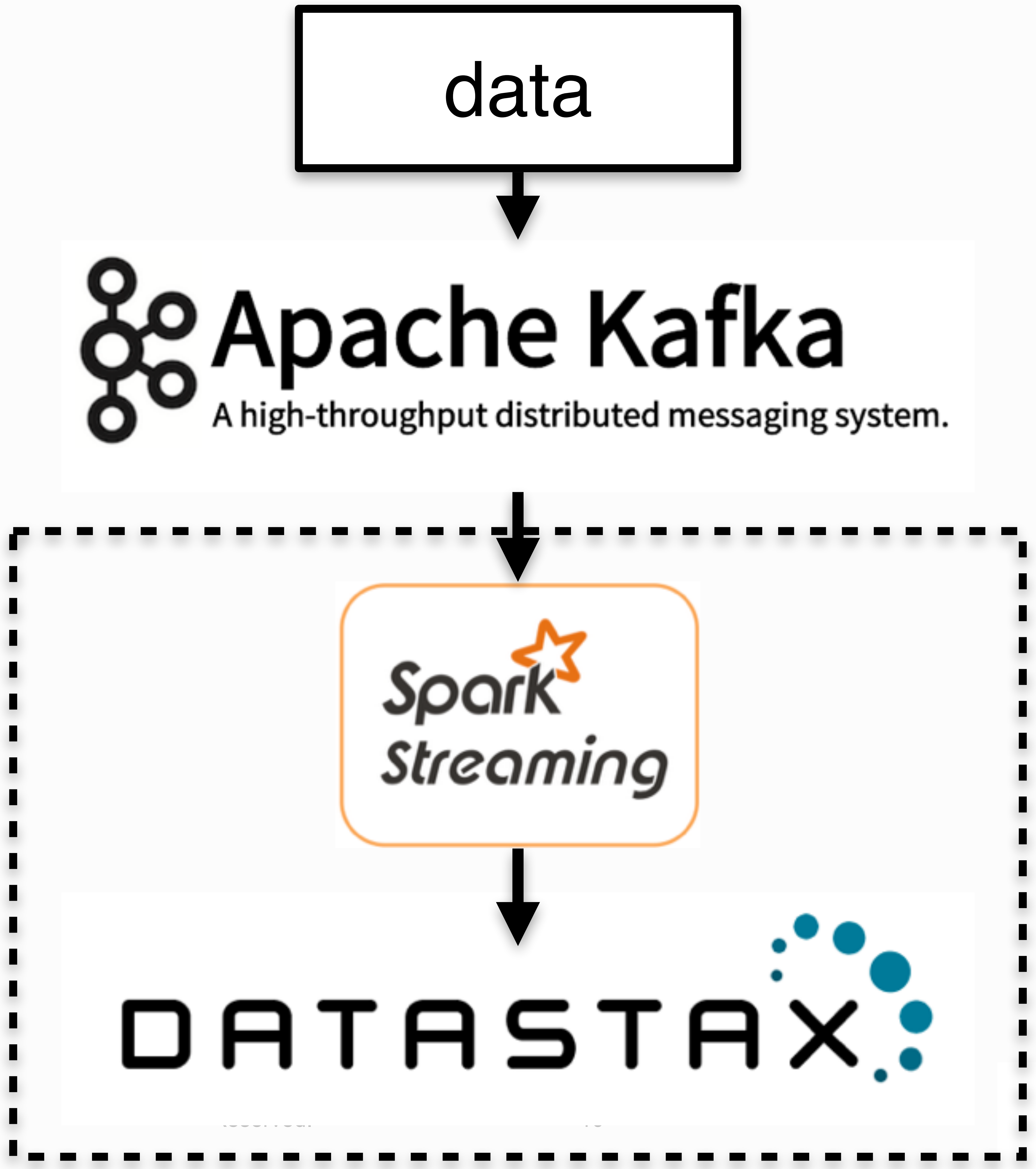
DSE File System Options (dse.yaml)

```
dsefs_options:
  enabled: true

  ...

 data_directories:
    - dir: /mnt/cass_data_disks/dsefs_data1
      # How much data should be placed in this directory relative to
      # other directories in the cluster
      storage_weight: 1.0
      ...
    - dir: /mnt/cass_data_disks/dsefs_data2
      # How much data should be placed in this directory relative to
      # other directories in the cluster
      storage_weight: 5.0
      ...
```

CASSANDRA SUMMIT2016

We use an email messaging app to test fault-tolerance

Application Setup



Data Model

```
CREATE TABLE email_msg_tracker (
    msg_id text,
    tenant_id uuid,
    mailbox_id uuid,
    time_delivered timestamp,
    time_forwarded timestamp,
    time_read timestamp,
    time_replied timestamp,
    PRIMARY KEY ((msg_id, tenant_id), mailbox_id)
)
```

https://github.com/datastax/code-samples/tree/master/

Enabling fault-tolerance in Spark is simple

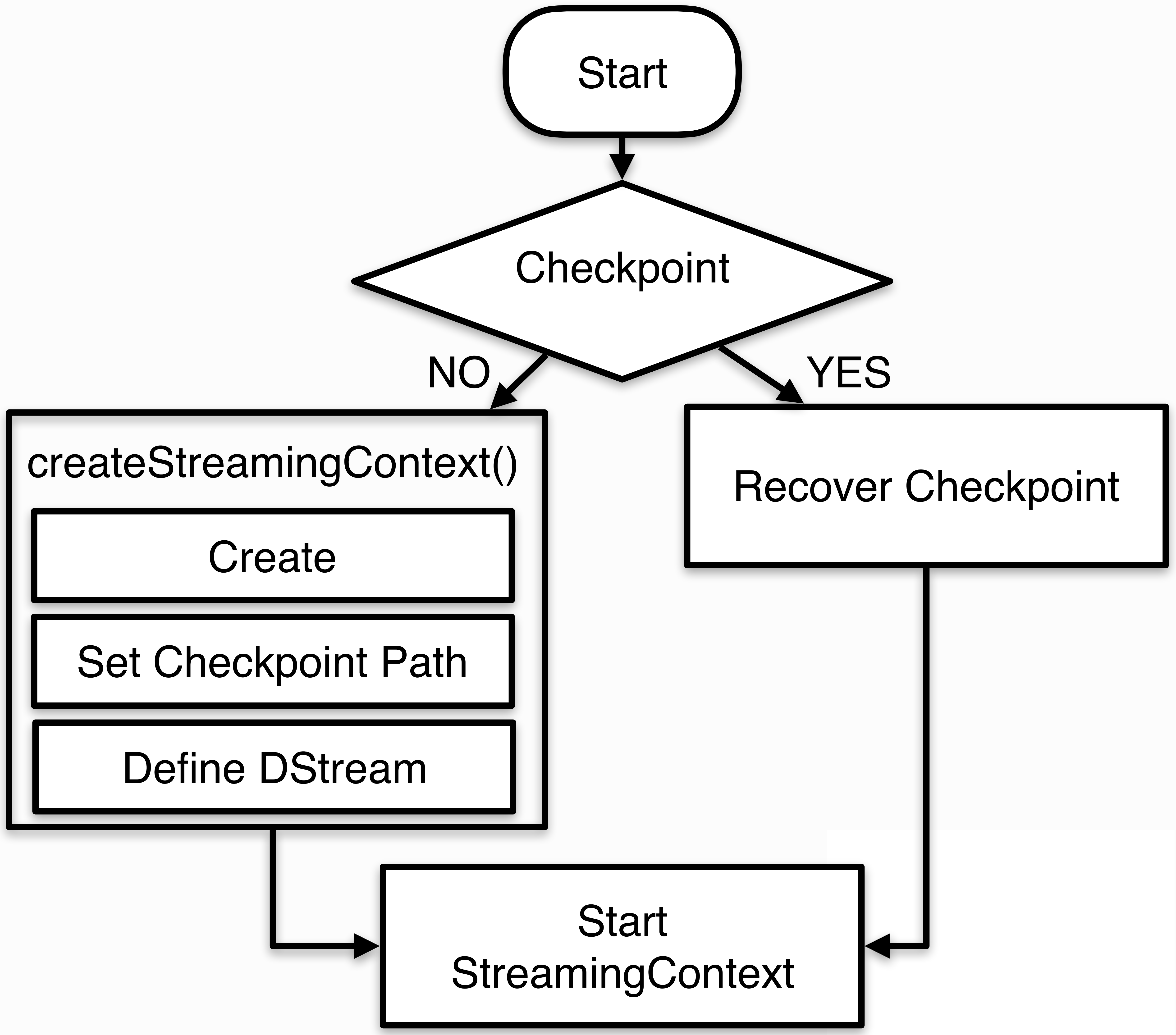
1. On startup for the first time, it will create a new StreamingContext.

2. Within our context creation method we configure checkpointing with **ssc.checkpoint(path)**.

3. We will also define our DStream in this function.

4. When the program is restarted after failure, it will re-create a StreamingContext from the checkpoint.

Spark Streaming Application Setup



Enabling the DSE Filesystem is even easier

Configurations:

1. Enable DSEFS

2. Keyspace name for storing metadata
   - metadata is stored in Cassandra
   - no SPOF, always-on, scalable, durable, etc.

3. Working directory
   - data is tiny, no special throughput, latency or capacity requirements

4. Data directory
   - JBOD support
   - Designed for efficiency

DSE File System Options (dse.yaml)

```
  dsefs_options:
1   enabled: true

    # Keyspace for storing the DSE FS metadata
2   keyspace_name: dsefs

    # The local directory for storing node-local metadata
    # The work directory must not be shared by DSE FS nodes.
3   work_dir: /mnt/data_disks/data1/dse/dsefs

    . . .

4   data_directories:
      - dir: /var/lib/dsefs/data
        # How much data should be placed in this directory relative to
        # other directories in the cluster
        storage_weight: 1.0
        # Reserved space (in bytes) that is not going to be used for
        # storing blocks
        min_free_space: 5368709120
```

CASSANDRA SUMMIT2016

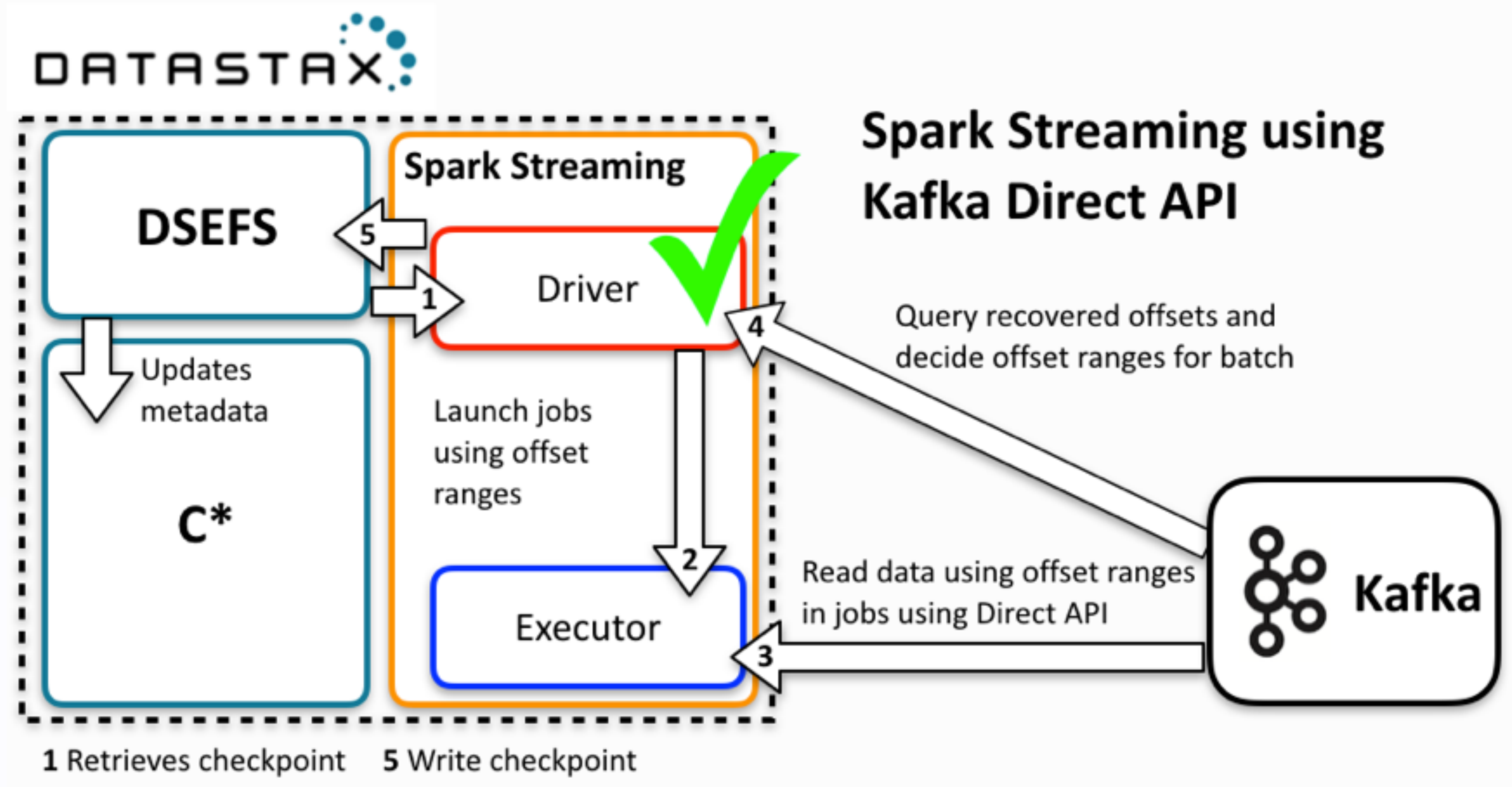
Checkpoint files are managed automatically

These files are **automatically managed** by our application and DataStax Filesystem (DSEFS).

Under the hood, these are some files we expect to see:

- **checkpoint-1471492965000**: serialized checkpoint file containing data used for recovery
- a2f2452d-3608-43b9-ba60-92725b54399a: directory used for storing Write-Ahead-Log data
- receivedBlockMetadata: used for Write-Ahead-Logging to store received block metadata

```
# Start streaming application with DSEFS URI and checkpoint dir
$ dse spark-submit \
    -class sparkAtScale.StreamingDirectEmails \
    path/to/myjar/streaming-assembly-0.1.jar  \
    dsefs:///emails_checkpoint


$ dse fs
dsefs / > ls -l emails_checkpoint
Type  ...  Modified                  Name
dir   ...  2016-08-17 21:02:40-0700  a2f2452d-3608-43b9-ba60-92725b54399a
dir   ...  2016-08-17 21:02:50-0700  receivedBlockMetadata
file  ...  2016-08-17 21:02:49-0700  checkpoint-1471492965000
```


DATASTAX
Spark Streaming using Kafka Direct API
DSEFS
Spark Streaming
Driver
Updates metadata
C*
Launch jobs using offset ranges
Executor
Query recovered offsets and decide offset ranges for batch
Read data using offset ranges in jobs using Direct API
Kafka
1
2
3
4
5
1 Retrieves checkpoint
5 Write checkpoint

DSE Filesystem stands up real world challenges

DSE Filesystem demonstrated stable perfomance.

In this experiment we tested a 3-node cluster streaming data from Kafka.

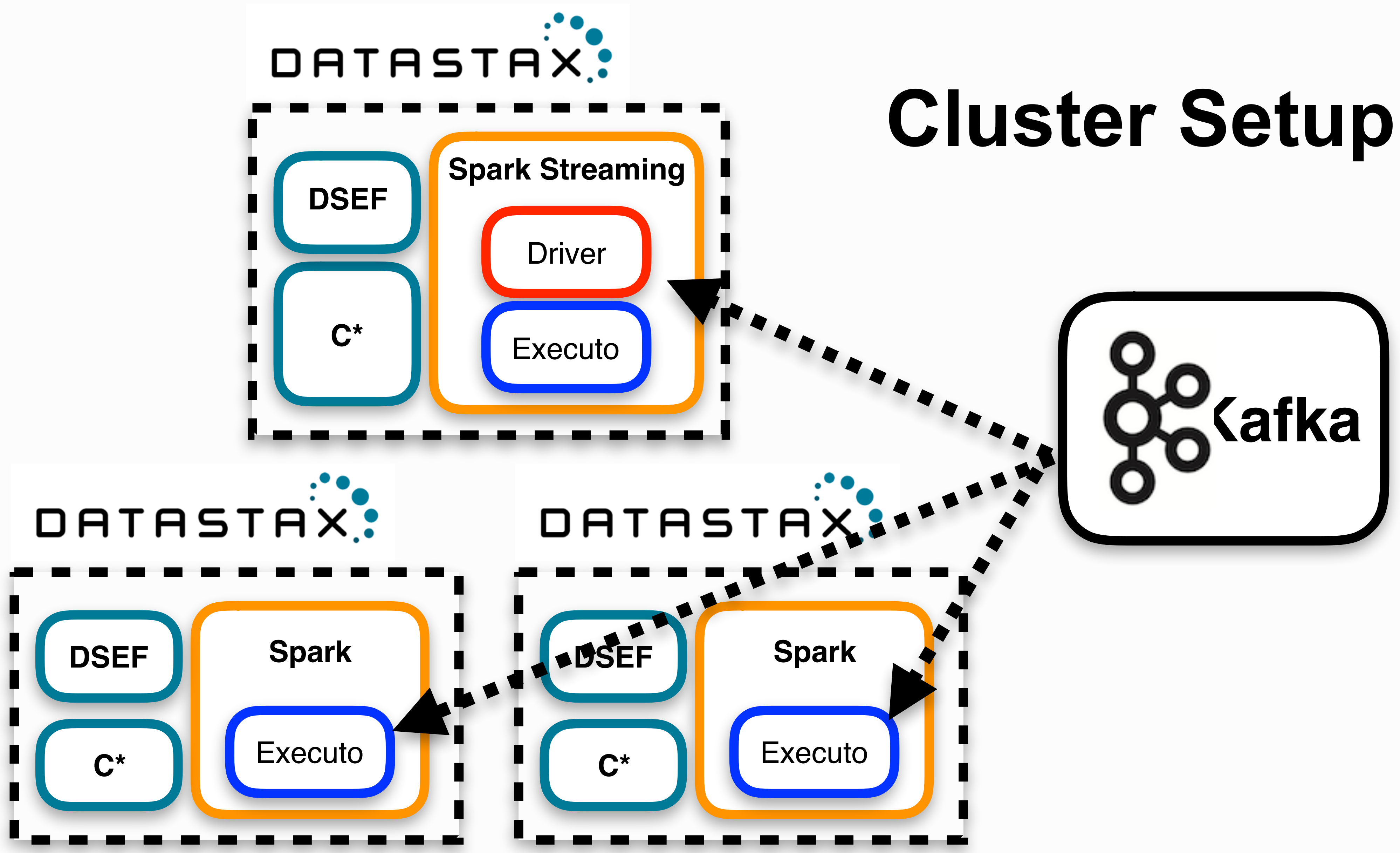
The summary on the right shows stable input rate and processing time.

- DSEFS Rep. Factor: 3
- Kafka/Spark partitions: 306
- Spark cores/node: 34
- Spark Executor Mem: 2G

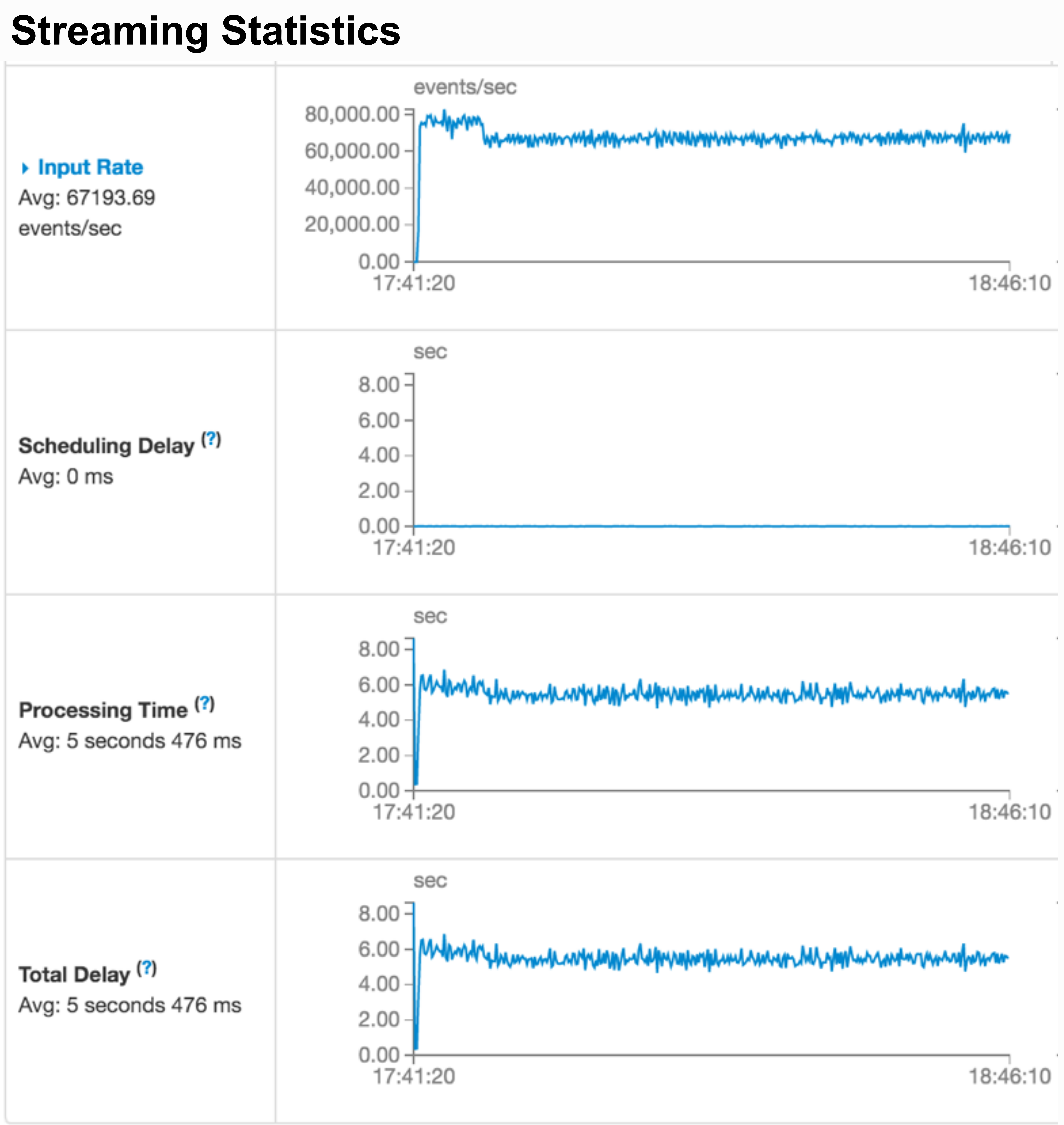
Streaming

- Rate: ~67K events/sec
- Batch Interval: 10 sec
- Total Delay: 5.5 sec

Cluster Setup



Metadata checkpointing (Kafka Direct API)



Streaming Statistics

Input Rate
Avg: 67193.69 events/sec

Scheduling Delay (?)
Avg: 0 ms

Processing Time (?)
Avg: 5 seconds 476 ms

Total Delay (?)
Avg: 5 seconds 476 ms

Tuning Spark Streaming in a nutshell

Tuning Spark Streaming boils down to balancing the input rate and processing throughput.

At a high level, we need to consider two things:

1. Reducing the **processing time** of each batch
2. Setting the **right batch size** so batches can be processed as fast as they are received

What are some ways we can accomplish these goals?

Reduce processing time by increasing parallelism

Exaggerated example obviously, main point:
Proper parallelism can achieve better performance and stability.

Processing Time: The time to process each batch of data.

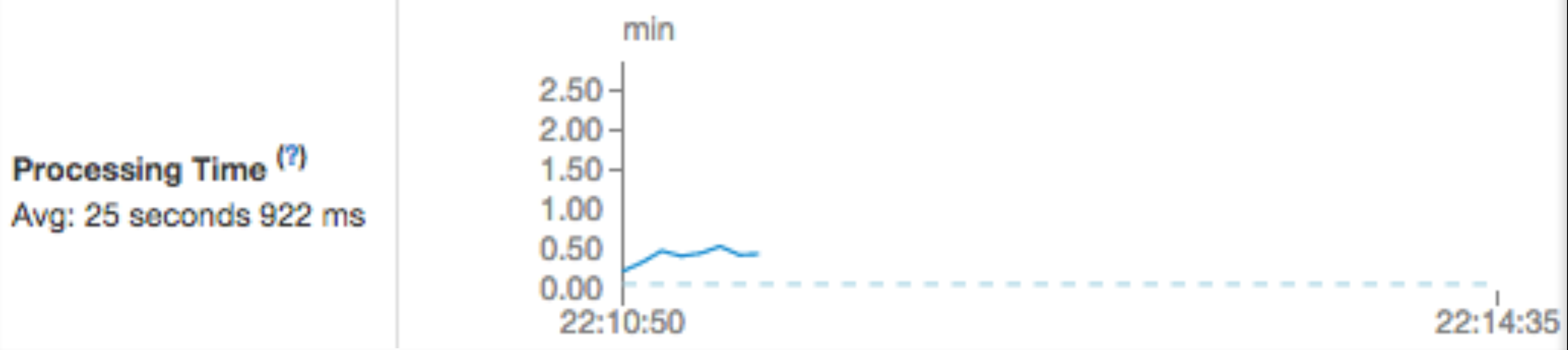
Both experiments had the same configs:
- input rate: ~85K events/sec
- batch interval: 5 sec
- maxRatePerPartition: 100K

Experiment #2 has a better:
- workload distribution
- lower processing time and delays
- processing time remains below the batch interval
- overall better stability

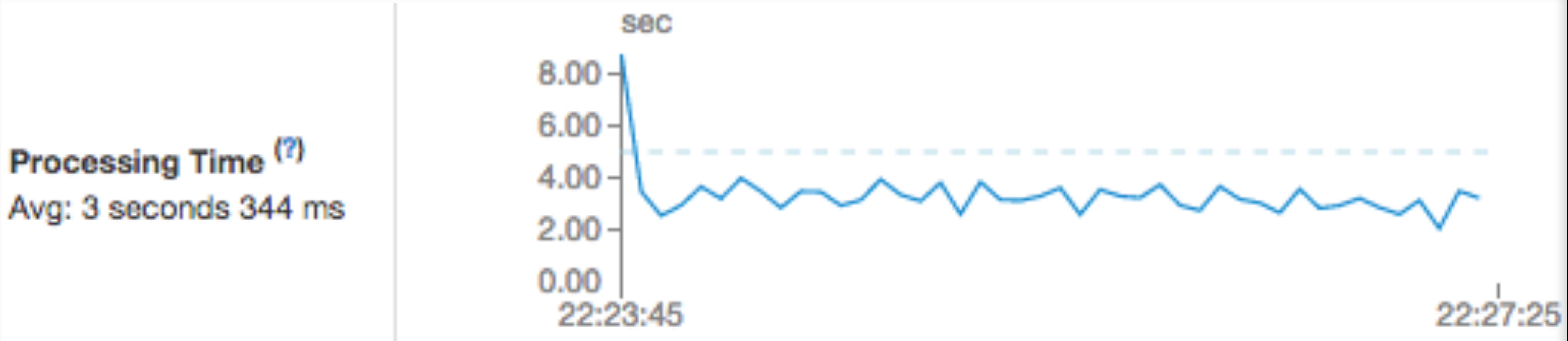
If we see Exp. #1 behavior even with parallelism set properly, we need to exam CPU utilization more closely.

Spark Streaming Statistics

Experiment #1: 1 Spark partition



Experiment #2: 100 Spark partitions



Reduce processing time by increasing parallelism

Exaggerated example obviously, main point:
Proper parallelism can achieve better performance and stability.

Scheduling Delay: the time a batch waits in a queue for the processing of previous batches to finish.

Both experiments had the same configs:
- input rate: ~85K events/sec
- batch interval: 5 sec
- maxRatePerPartition: 100K

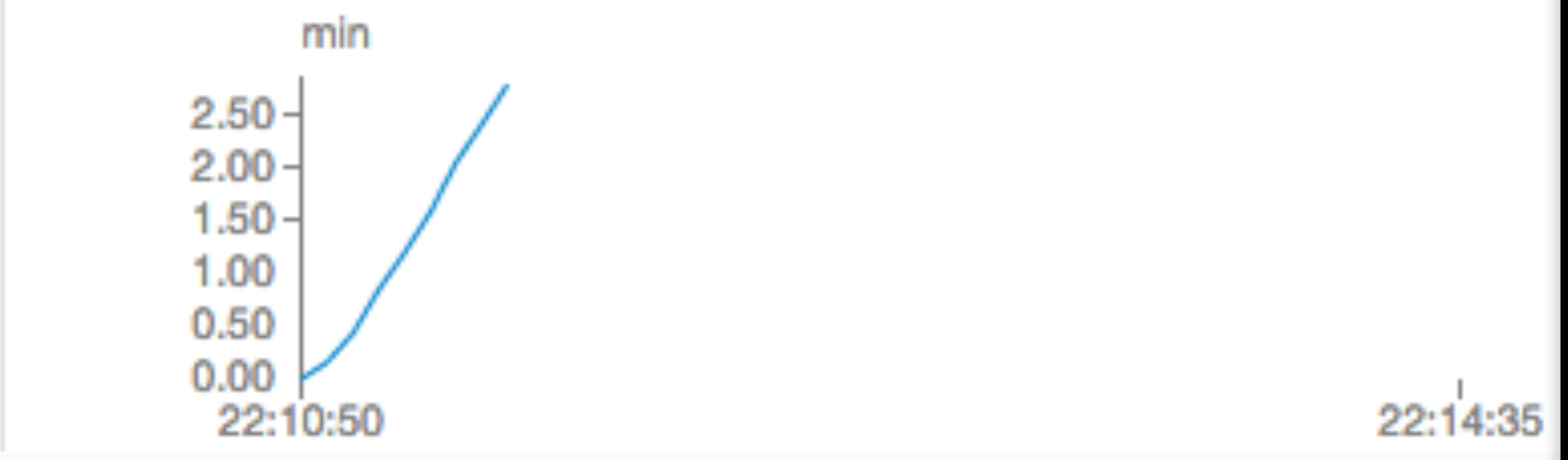
Experiment #2 has a better:
- workload distribution
- lower processing time and delays
- processing time remains below the batch interval
- overall better stability

If we see Exp. #1 behavior even with parallelism set properly, we need to exam CPU utilization more closely.

Spark Streaming Statistics

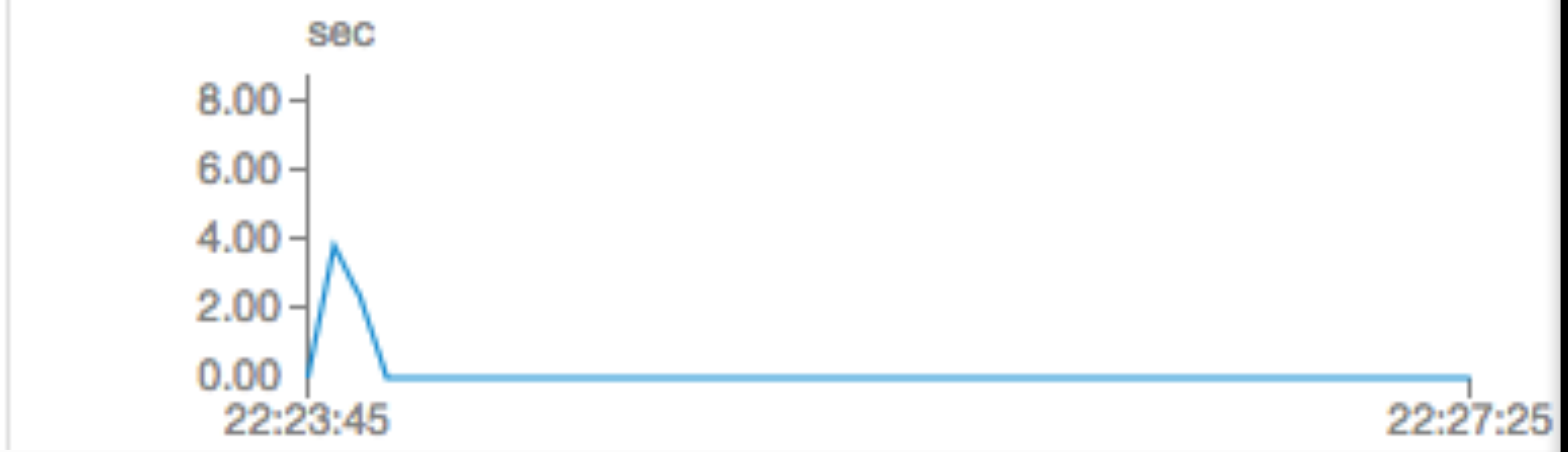
Experiment #1: 1 Spark partition

Scheduling Delay (?)
Avg: 1 minute 16 seconds



Experiment #2: 100 Spark partitions

Scheduling Delay (?)
Avg: 135 ms



Reduce processing time by increasing parallelism

Exaggerated example obviously, main point:
Proper parallelism can achieve better performance and stability.

Total Delay: Scheduling Delay + Processing Time

Both experiments had the same configs:
- input rate: ~85K events/sec
- batch interval: 5 sec
- maxRatePerPartition: 100K

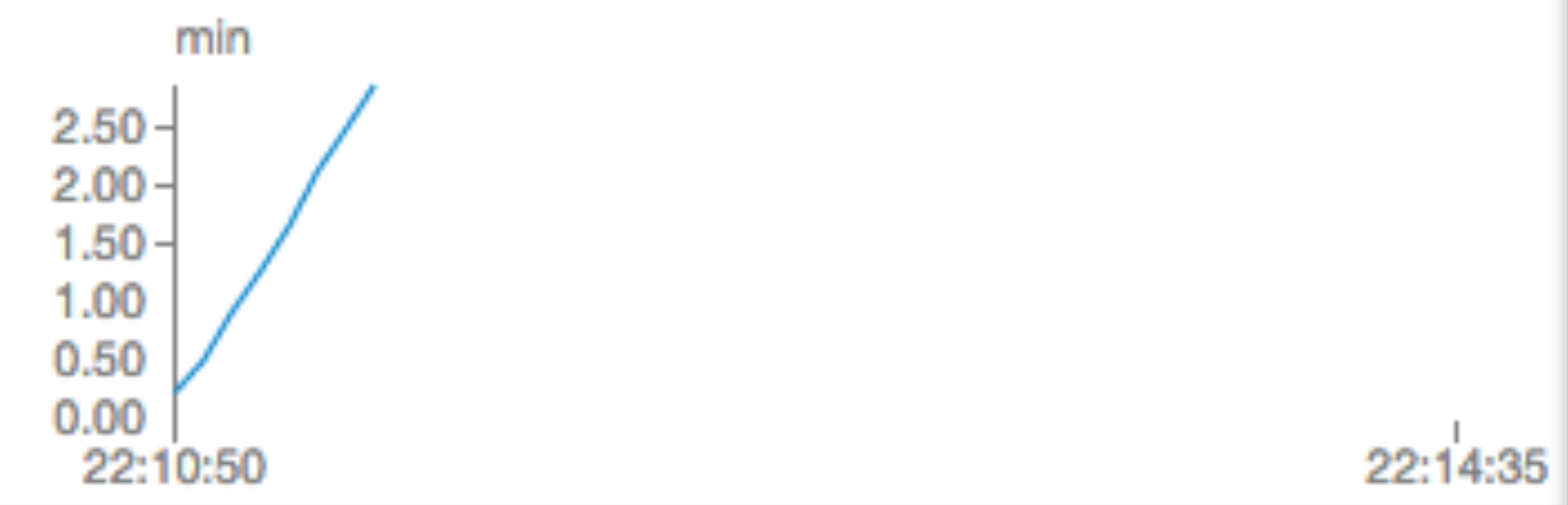
Experiment #2 has a better:
- workload distribution
- lower processing time and delays
- processing time remains below the batch interval
- overall better stability

If we see Exp. #1 behavior even with parallelism set properly, we need to exam CPU utilization more closely.

Spark Streaming Statistics

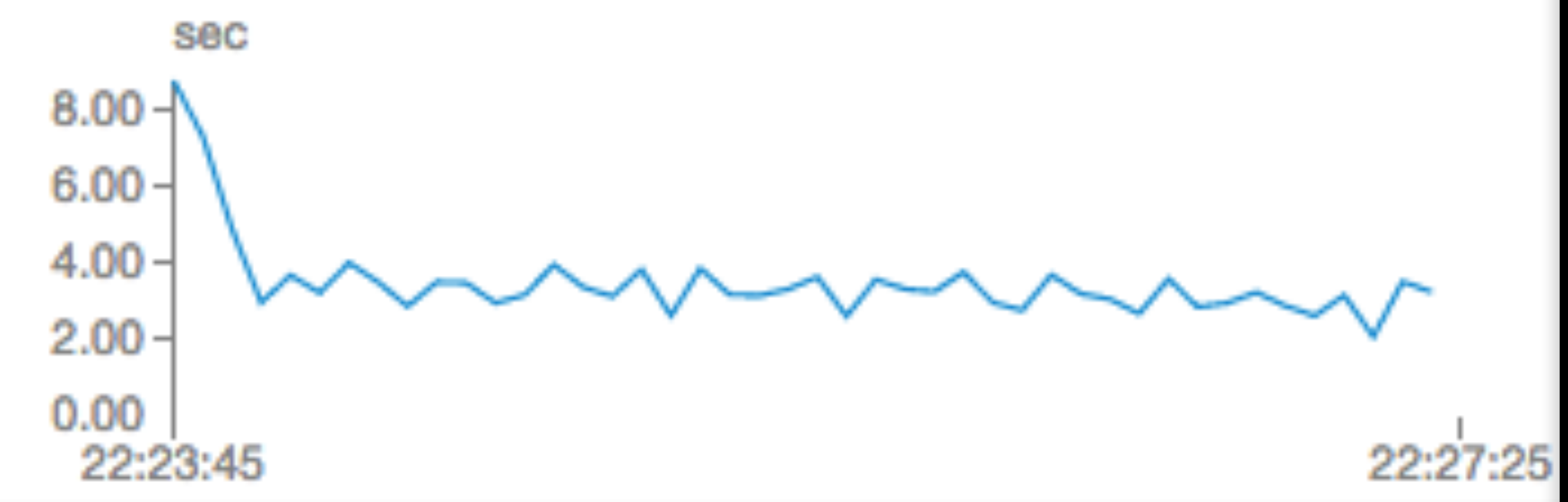
Experiment #1: 1 Spark partition

Total Delay (?)
Avg: 1 minute 31 seconds



Experiment #2: 100 Spark partitions

Total Delay (?)
Avg: 3 seconds 483 ms



Setting the right batch interval depends on the setup

Experiment #2 is a good example of stability:

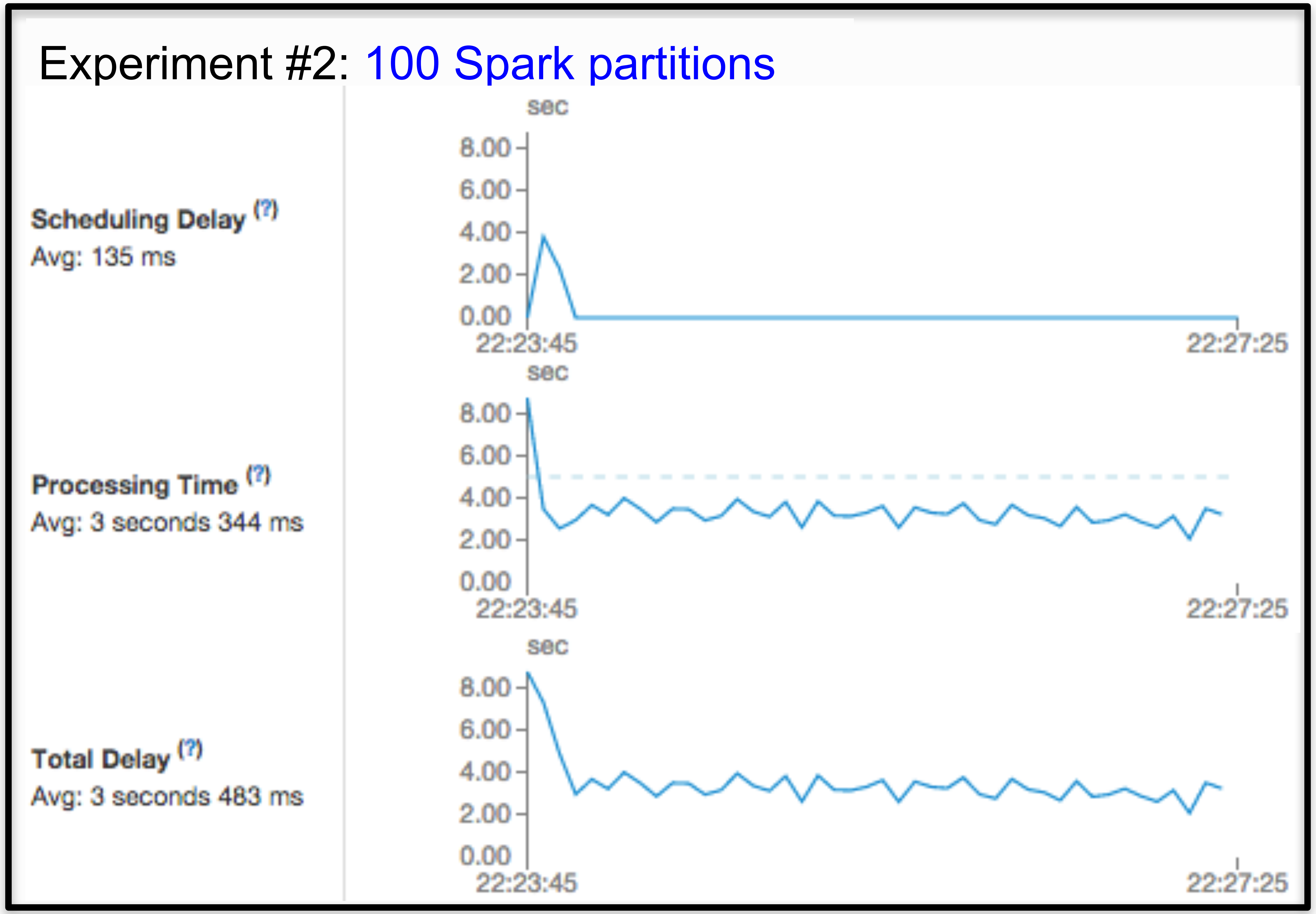
- Batch Interval: 5 sec
- Avg. Processing Time: 3.3 sec
- Avg. Scheduling Delay: 135 ms
- Avg. Total Delay: 3.4 sec
- Minimal scheduling delay
- Consistent processing time below batch interval

A good approach is to test with a conservative batch interval (say, 5-10s) and a low data input rate.

If the **processing time < batch interval**, then the system is stable.

If the **processing time > batch interval** OR **scheduling delay increases**, then reduce the processing time.

Spark Streaming Statistics



Experiment #2: 100 Spark partitions

Scheduling Delay (?)
Avg: 135 ms

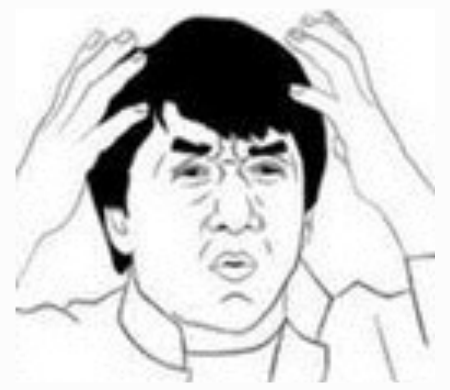
Processing Time (?)
Avg: 3 seconds 344 ms

Total Delay (?)
Avg: 3 seconds 483 ms

How to implement checkpointing correctly

Consider this scenario: The streaming app is ingesting data and checkpointing without issues, but when attempting to recover from a saved checkpoint the following error occurs.

What does this mean? Let’s look at our setup.



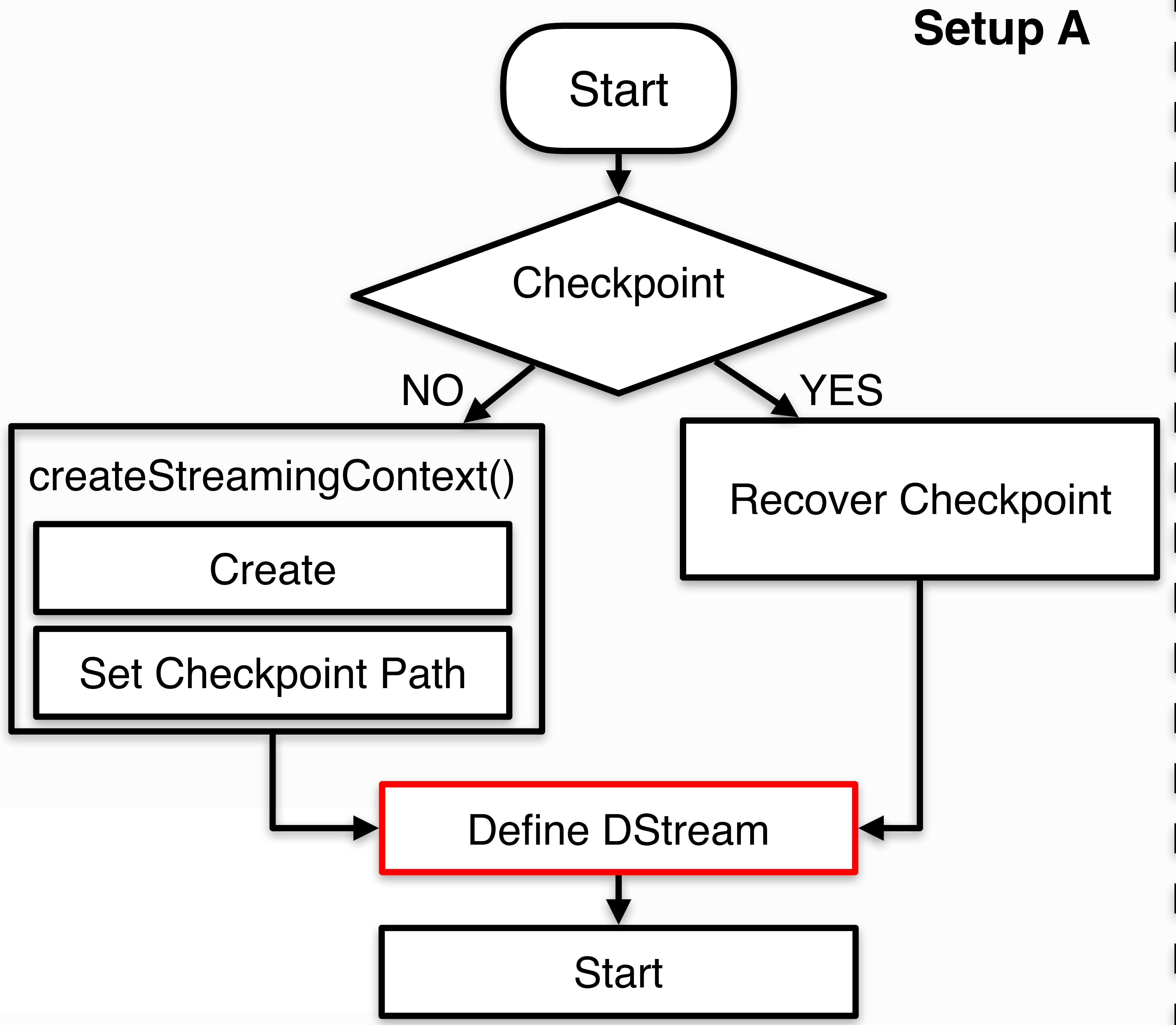
```
ERROR … org.apache.spark.streaming.StreamingContext: Error starting the context, marking it as stopped
org.apache.spark.SparkException: org.apache.spark.streaming.kafka.DirectKafkaInputDStream@45984654 has
not been initialized
  at scala.Option.orElse(Option.scala:257) ~[scala-library-2.10.5.jar:na]
  at org.apache.spark.streaming.dstream.DStream.getOrCompute(DStream.scala:339)
…
Exception in thread "main" org.apache.spark.SparkException:
org.apache.spark.streaming.kafka.DirectKafkaInputDStream@45984654 has not been initialized
  at org.apache.spark.streaming.dstream.DStream.isTimeValid(DStream.scala:321)
…
```

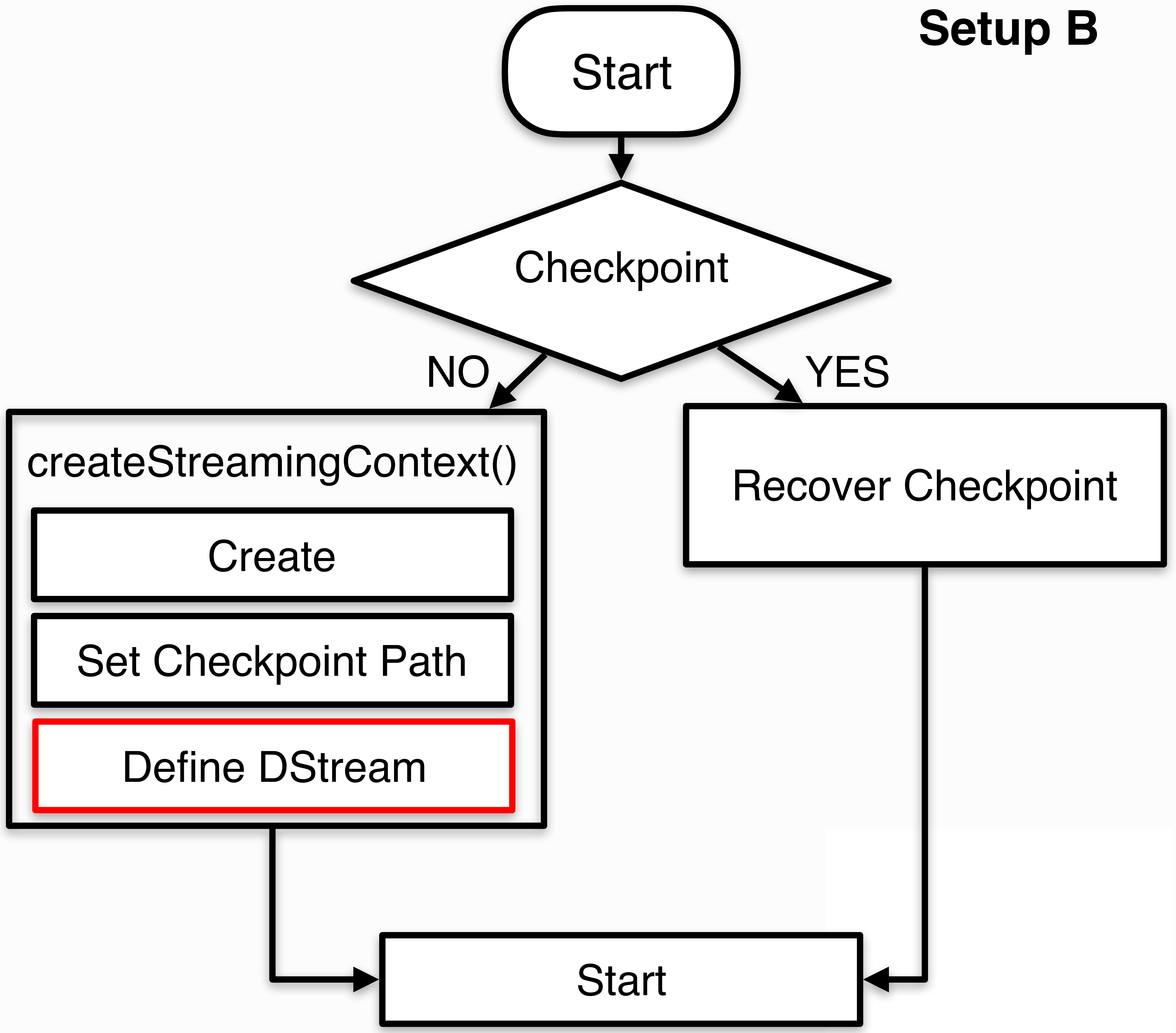
© DataStax, All Rights Reserved.

How to implement checkpointing correctly

Error starting the context… o.a.s.SparkException: o.a.s.s.kafka.DirectKafkaInputDStream@45984654 has not been initialized

Hint: In **red** shows where the input stream is created, notice the different location between examples.





How to implement checkpointing correctly

Error starting the context… o.a.s.SparkException: o.a.s.s.kafka.DirectKafkaInputDStream@45984654 has not been initialized

Hint: In **red** shows where the input stream is created, notice the different location between examples.

Setup A
Start
Checkpoint
NO
YES
createStreamingContext()
Create
Set Checkpoint Path
Recover Checkpoint
Define DStream
Start
Setup B
Start
Checkpoint
NO
YES
crea
Create
S
Define DStream
Start

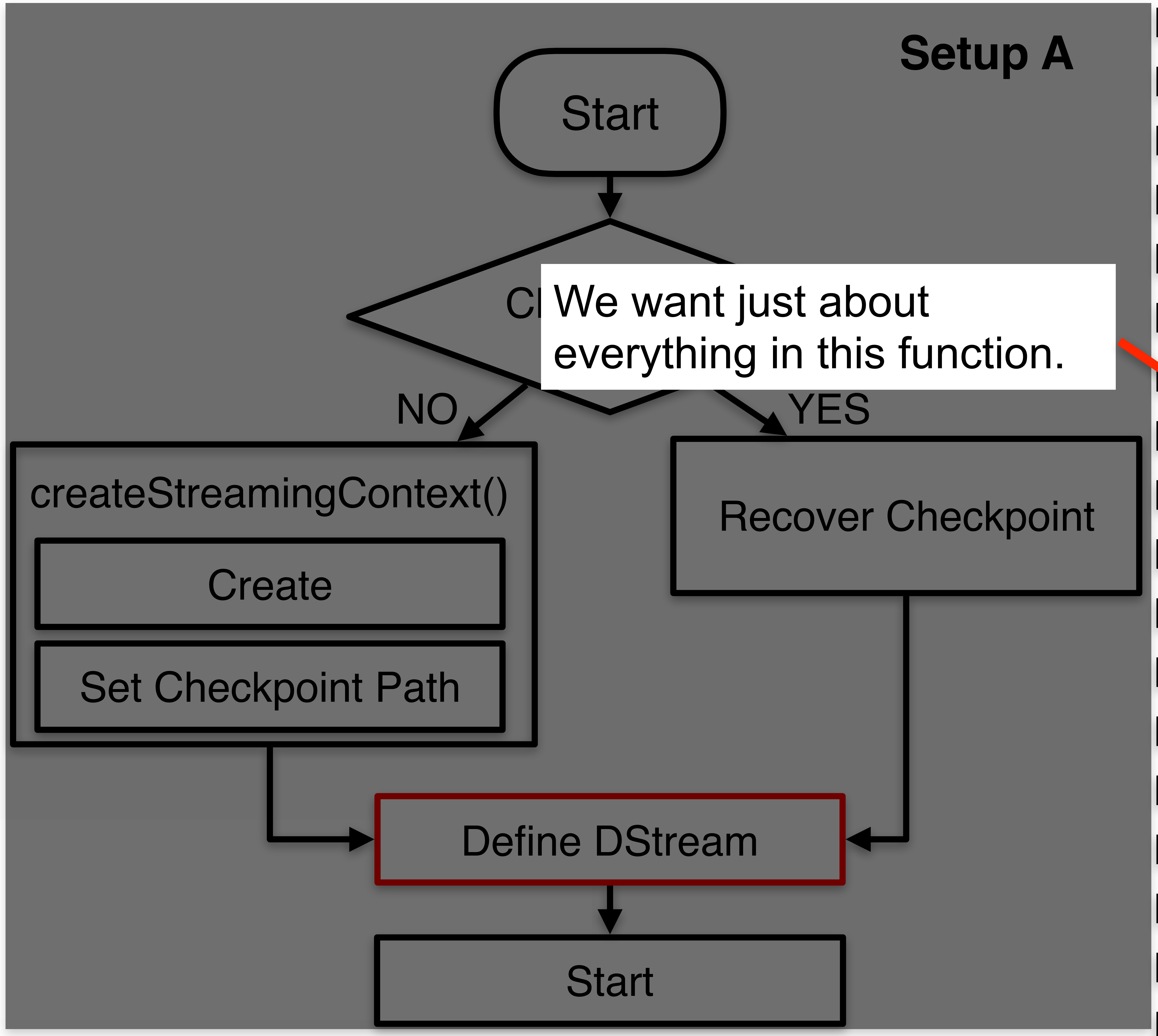
The problem here is that we are creating the stream in the wrong place.

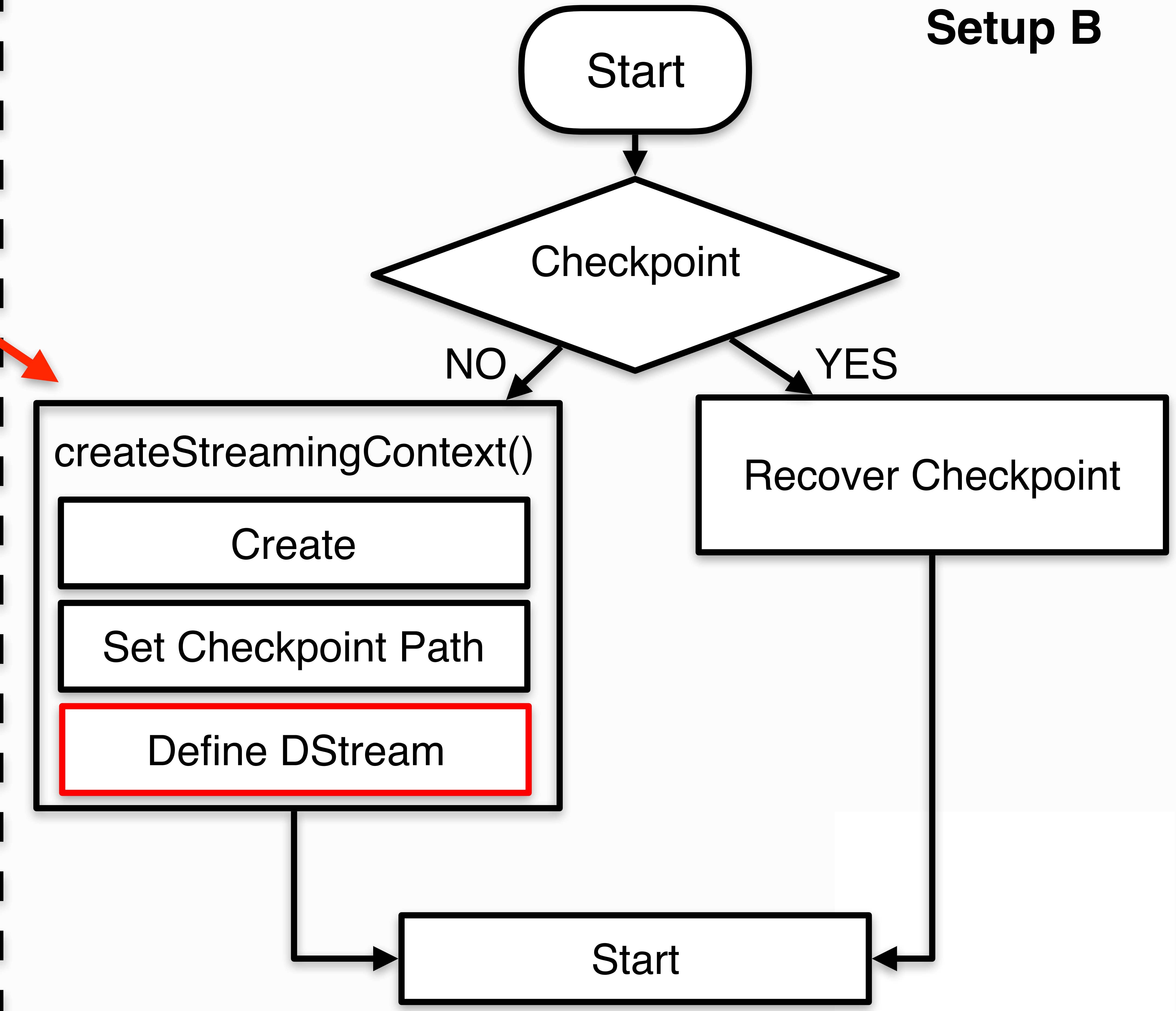
We want the stream to be recoverable, so we need to move this into createStreamingContext().

How to implement checkpointing correctly

Error starting the context… o.a.s.SparkException: o.a.s.s.kafka.DirectKafkaInputDStream@45984654 has not been initialized

Hint: In **red** shows where the input stream is created, notice the different location between examples.





How to implement checkpointing correctly

Error starting the context… o.a.s.SparkException: o.a.s.s.kafka.DirectKafkaInputDStream@45984654 has not been initialized

Hint: In **red** shows where the input stream is created, notice the different location between examples.

■: Highlights the method for creating the streaming context.

Code Sample A

```
def main(args: Array[String]) {

  val conf = new SparkConf() // with addition settings as needed
  val sc = SparkContext.getOrCreate(conf)
  val sqlContext = SQLContext.getOrCreate(sc)
  import sqlContext.implicits._

  def createStreamingContext(): StreamingContext = {
    val newSsc = new StreamingContext(sc, Seconds(batchInterval))
    newSsc.checkpoint(checkpoint_path)
    newSsc
  }

  val ssc = StreamingContext.getActiveOrCreate(checkpoint_path,
createStreamingContext)
  . . .

  val emailsStream = KafkaUtils.createDirectStream[String, String,
StringDecoder, StringDecoder](ssc, kafkaParams, topics)

  . . .
  ssc.start()
  ssc.awaitTermination()
}
```

Code Sample B

```
def main(args: Array[String]) {

  val conf = new SparkConf() // with addition se
  val sc = SparkContext.getOrCreate(conf)
  val sqlContext = SQLContext.getOrCreate(sc)
  import sqlContext.implicits._

  def createStreamingContext(): StreamingContext = {
    val newSsc = new StreamingContext(sc, Seconds(batchInterval))
    newSsc.checkpoint(checkpoint_path)
    . . .

    val emailsStream = KafkaUtils.createDirectStream[String,
String, StringDecoder, StringDecoder](ssc, kafkaParams, topics)

    . . .
    newSsc
  }

  val ssc = StreamingContext.getActiveOrCreate(checkpoint_path,
createStreamingContext)

  ssc.start()
  ssc.awaitTermination()
}
```

© DataStax, All Rights Reserved.

https://github.com/datastax/code-samples/tree/master/

General tips to keep things running smoothly

- One DSE node can run at most one DSEFS server.
- In production, always set RF >= 3 for the DSEFS keyspace.
- For optimal performance, organize local DSEFS data on **different physical drives** than the Cassandra database.
- Currently DSEFS is **intended for Spark streaming use cases**, support for broader use cases is underway.

http://docs.datastax.com/en/latest-dse/datastax_enterprise/ana/aboutDsefs.html?hl=dsefs

DSE Filesystem supports all Spark fault-tolerant goals

1. DataStax Enterprise Filesystem (DSEFS) can support all our needs for Spark Streaming fault-tolerance (Metadata checkpointing, Write-ahead-logging, RDD checkpointing).
2. Setting up the DSEFS is very straight forward, and provides common utilities for managing data.
3. We covered some details to consider when build a Spark Streaming App.
4. We can use existing tools such as `cfs-stress` to benchmark I/O before deployment.
5. Many more features are currently underway for upcoming releases of DSEFS:
   - Enhanced security
   - Hardening for long-term storage: fsck improvements, checksumming
   - Performance improvements: Compression, data locality improvements and more
   - Access to other types of filesystems: HDFS, local file system from the DSEFS shell
   - User-experience improvements in the DSEFS shell: tab-completion, wildcards, etc.

CASSANDRA SUMMIT2016

Thank You

- Piotr Kołaczkowski (DSEFS Architect and Lead Developer)
- Brian Hess (Product Management)
- Jarosław Grabowski (DSEFS Developer)
- Russell Spitzer (DSE Analytics Developer)
- Ryan Knight (Solutions Engineer)
- Giovanny Castro (DSE Test Engineer)
- DSE Analytics Team

https://github.com/datastax/code-samples/tree/master/