SPARK SUMMIT

Apache Spark Streaming + Kafka 0.10: An Integration Story

Joan Viladrosa, Billy Mobile

#EUstr5

About me

Joan Viladrosa Riera

@joanvr

joanviladrosa

joan.viladrosa@billymob.com

Degree In Computer Science

Advanced Programming Techniques &
System Interfaces and Integration

Co-Founder, Educabits

Educational Big data solutions
using AWS cloud

Big Data Developer, Trovit

Hadoop and MapReduce Framework
SEM keywords optimization

Big Data Architect & Tech Lead
BillyMobile

Full architecture with Hadoop:
Kafka, Storm, Hive, HBase, Spark, Druid, …

Apache Kafka

What is Apache Kafka?

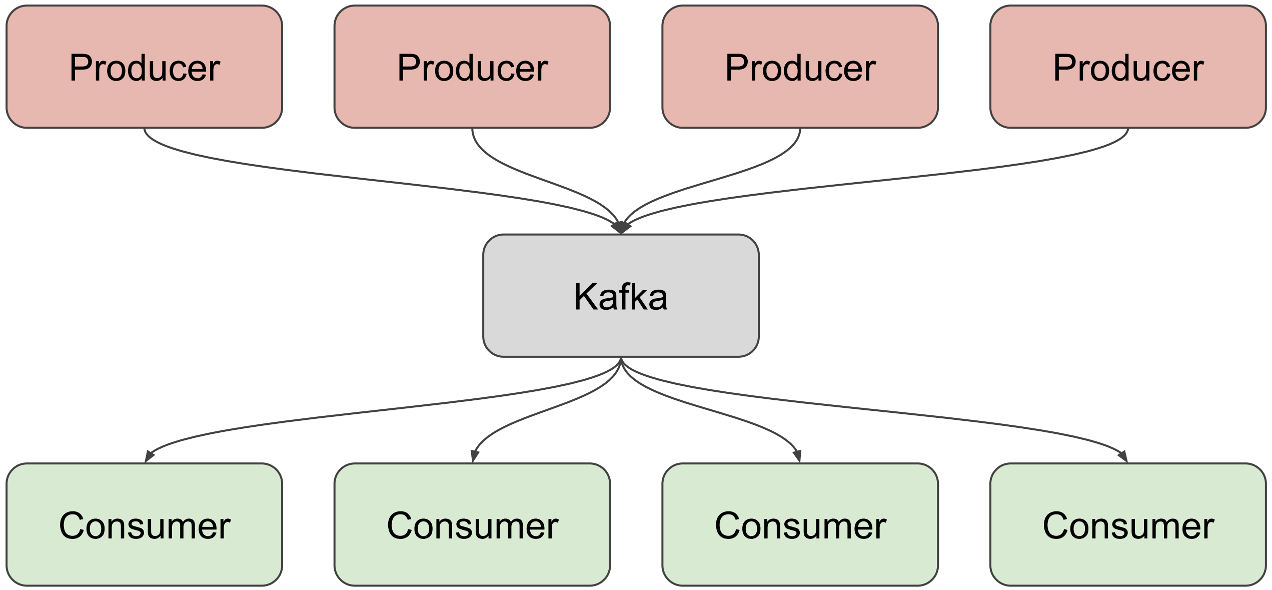
- Publish - Subscribe Message System

What is Apache Kafka?

- Publish - Subscribe Message System

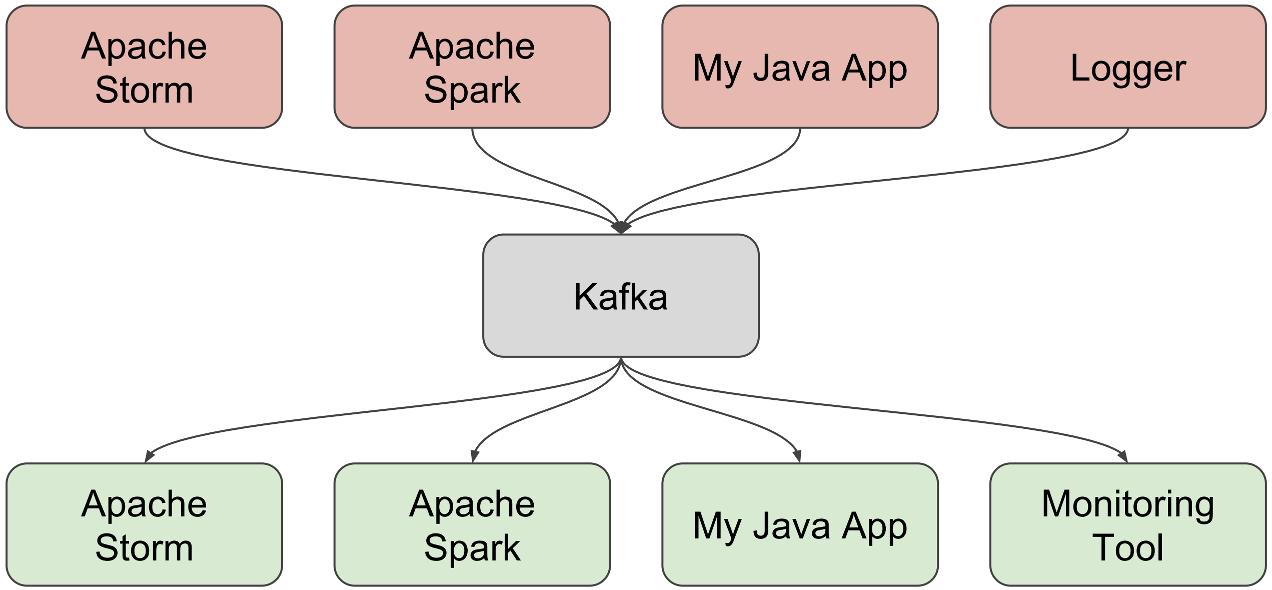
What makes it great?

- Fast
- Scalable
- Durable
- Fault-tolerant



What is Apache Kafka?

As a central point



What is Apache Kafka?

A lot of different connectors

Kafka Terminology

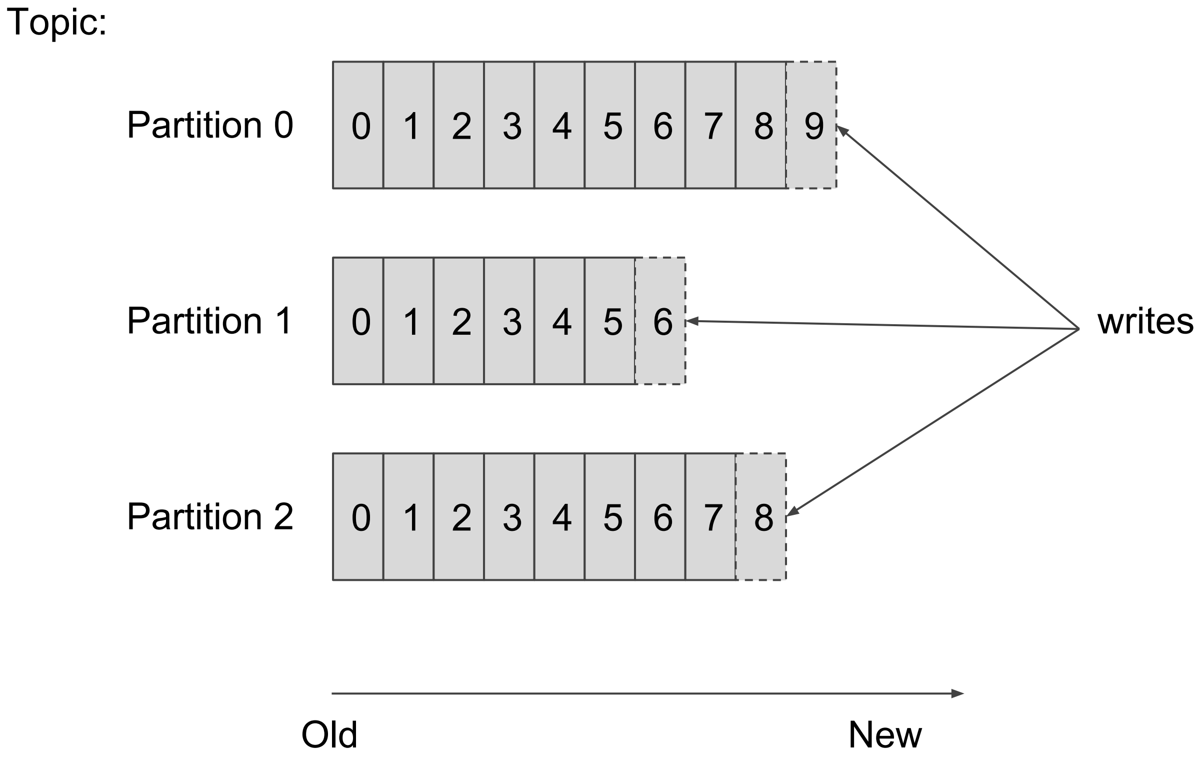
Topic: A feed of messages

Producer: Processes that publish messages to a topic

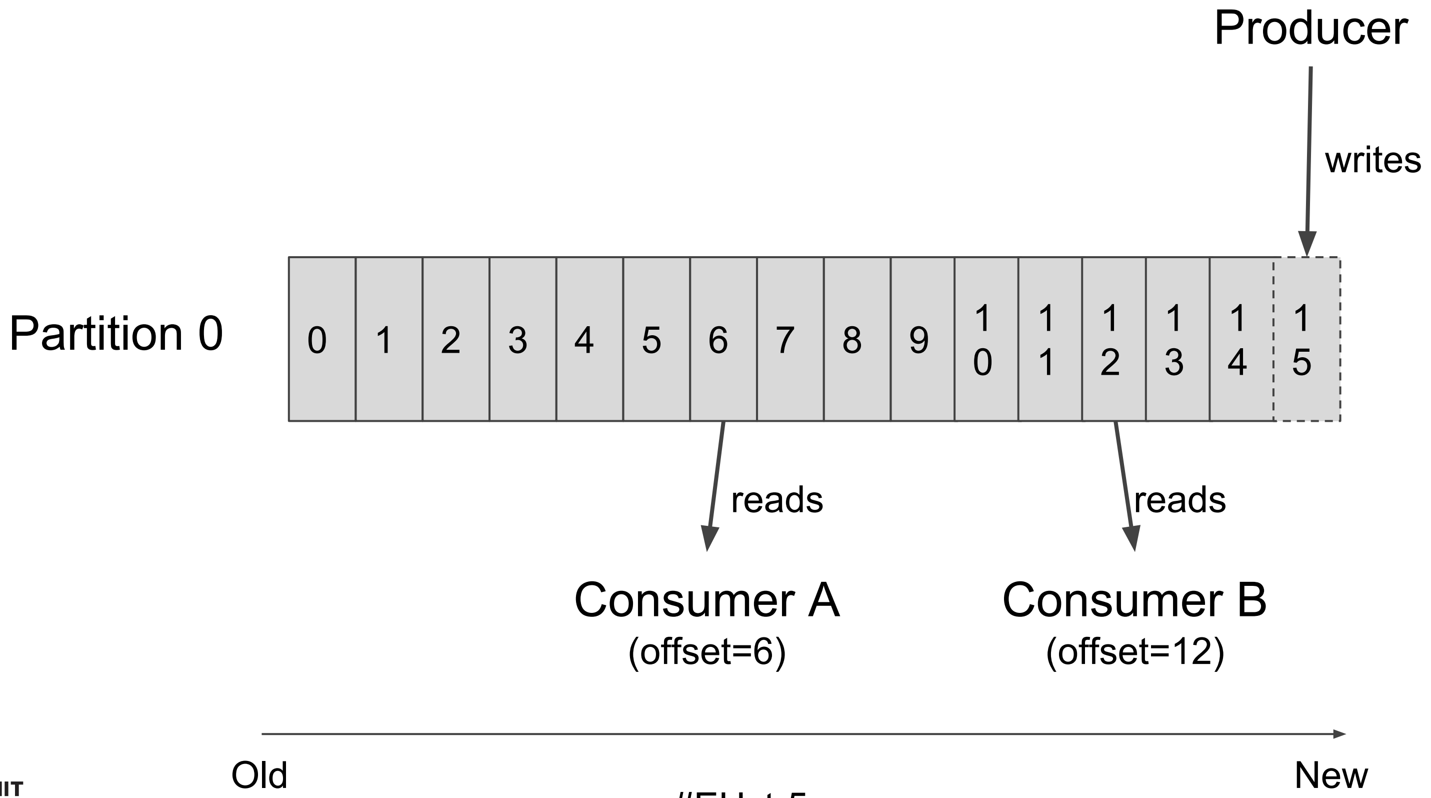
Consumer: Processes that subscribe to topics and process the feed of published messages

Broker: Each server of a kafka cluster that holds, receives and sends the actual data

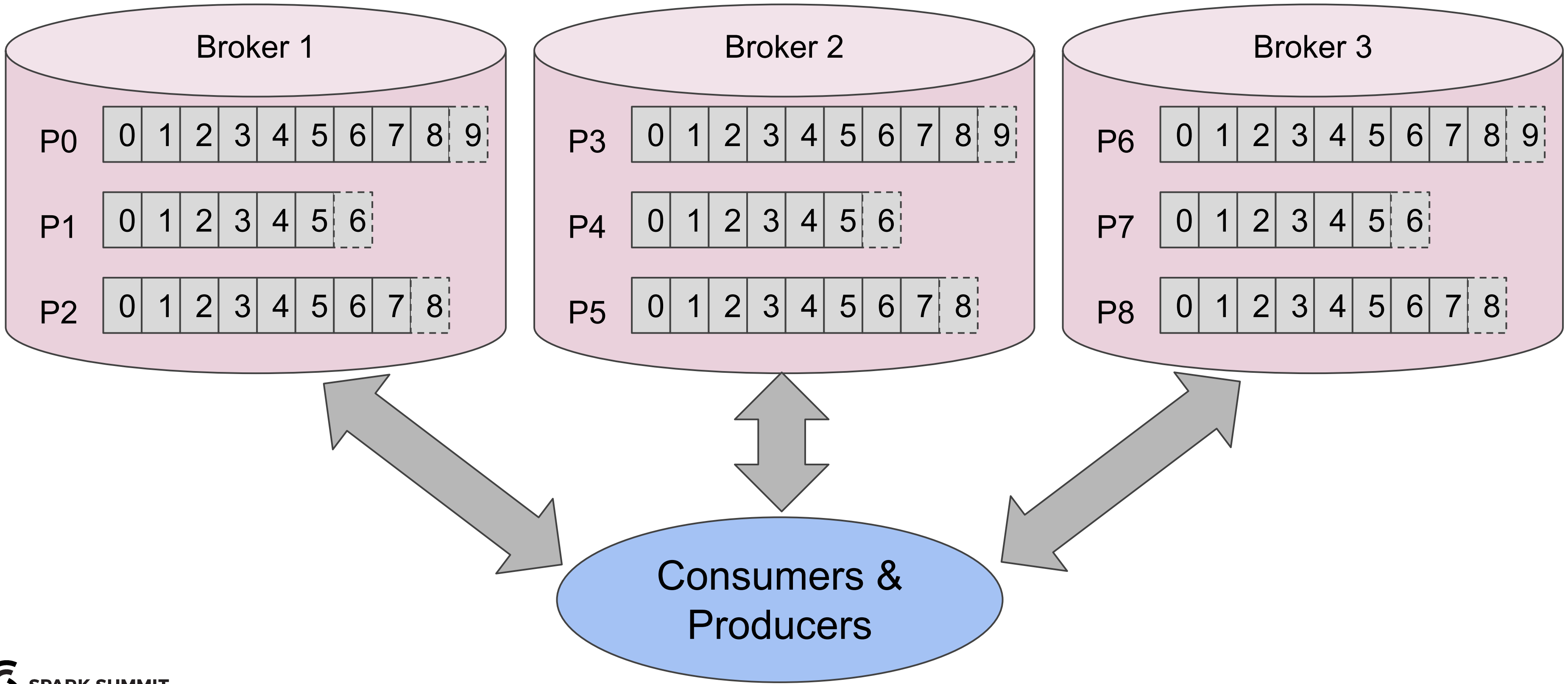
Kafka Topic Partitions



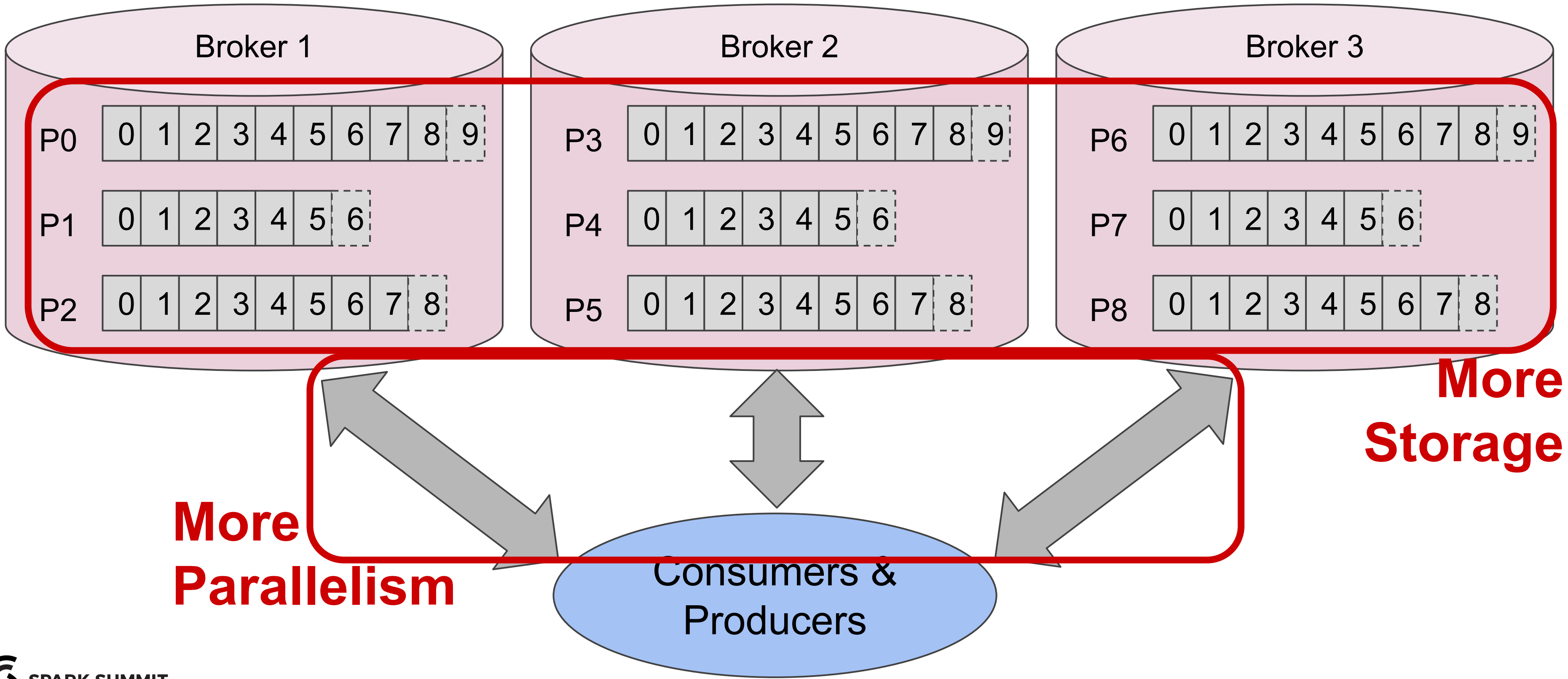
Kafka Topic Partitions



Kafka Topic Partitions



Kafka Topic Partitions

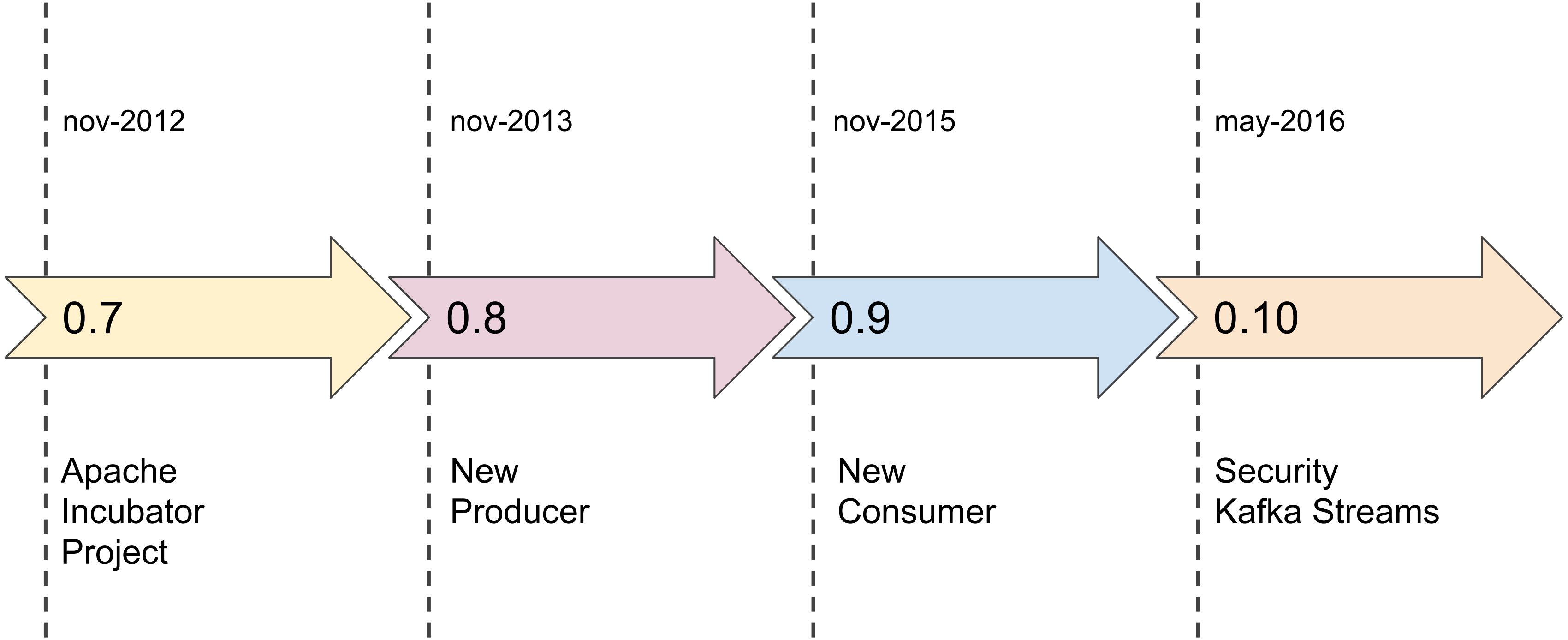


Kafka Semantics

In short: consumer delivery semantics are up to you, not Kafka

- Kafka **doesn’t** store the state of the consumers*
- It just sends you what you ask for (topic, partition, offset, length)
- You have to take care of your state

Apache Kafka Timeline



Apache Spark Streaming

What is Apache Spark Streaming?

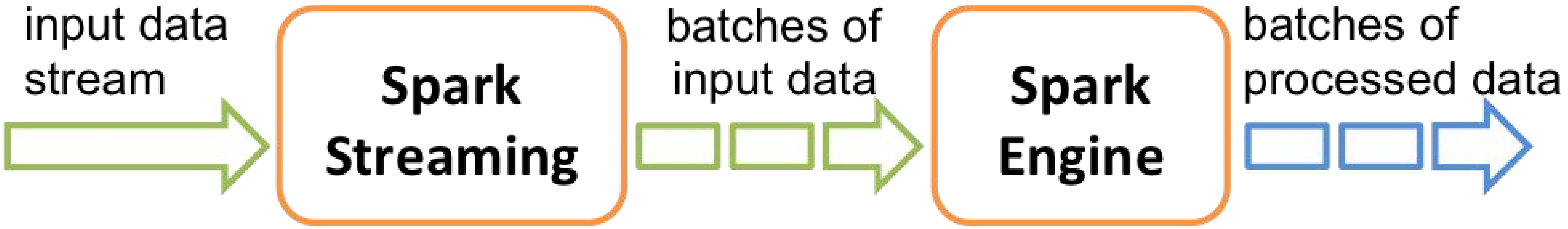
- Process streams of data
- Micro-batching approach

What is Apache Spark Streaming?

What makes it great?

- Process streams of data
- Micro-batching approach

- Same API as Spark
- Same integrations as Spark
- Same guarantees & semantics as Spark



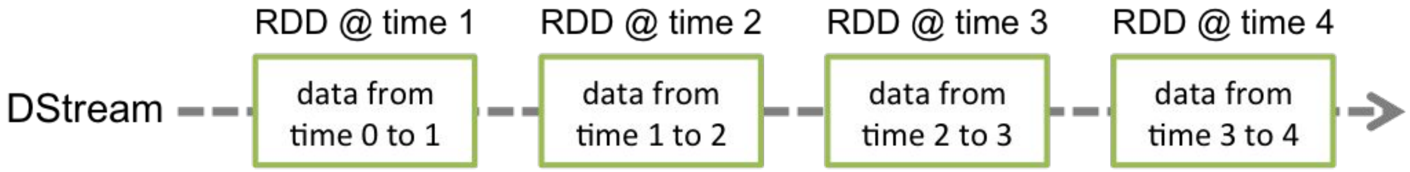
What is Apache Spark Streaming?

Relying on the same Spark Engine: “same syntax” as batch jobs

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

How does it work?

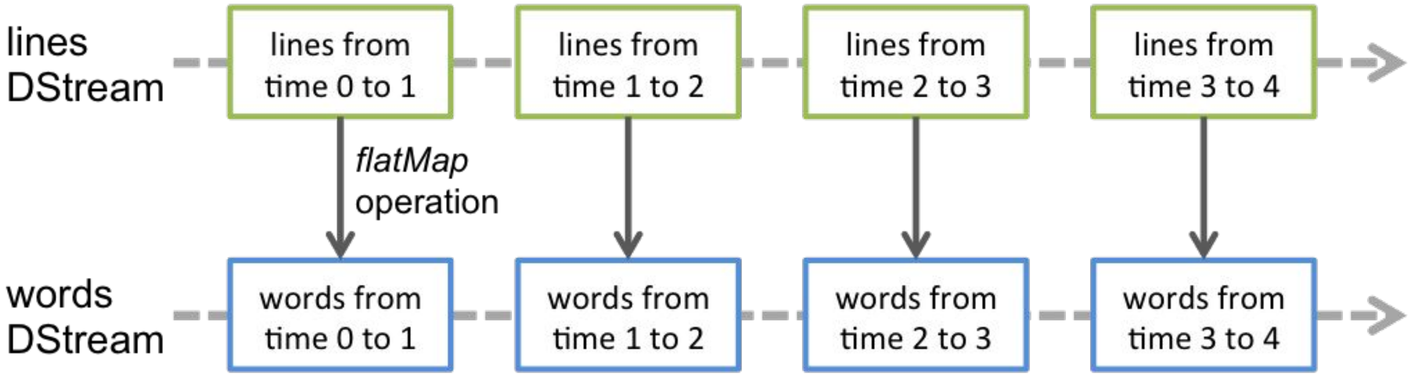
- Discretized Streams



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

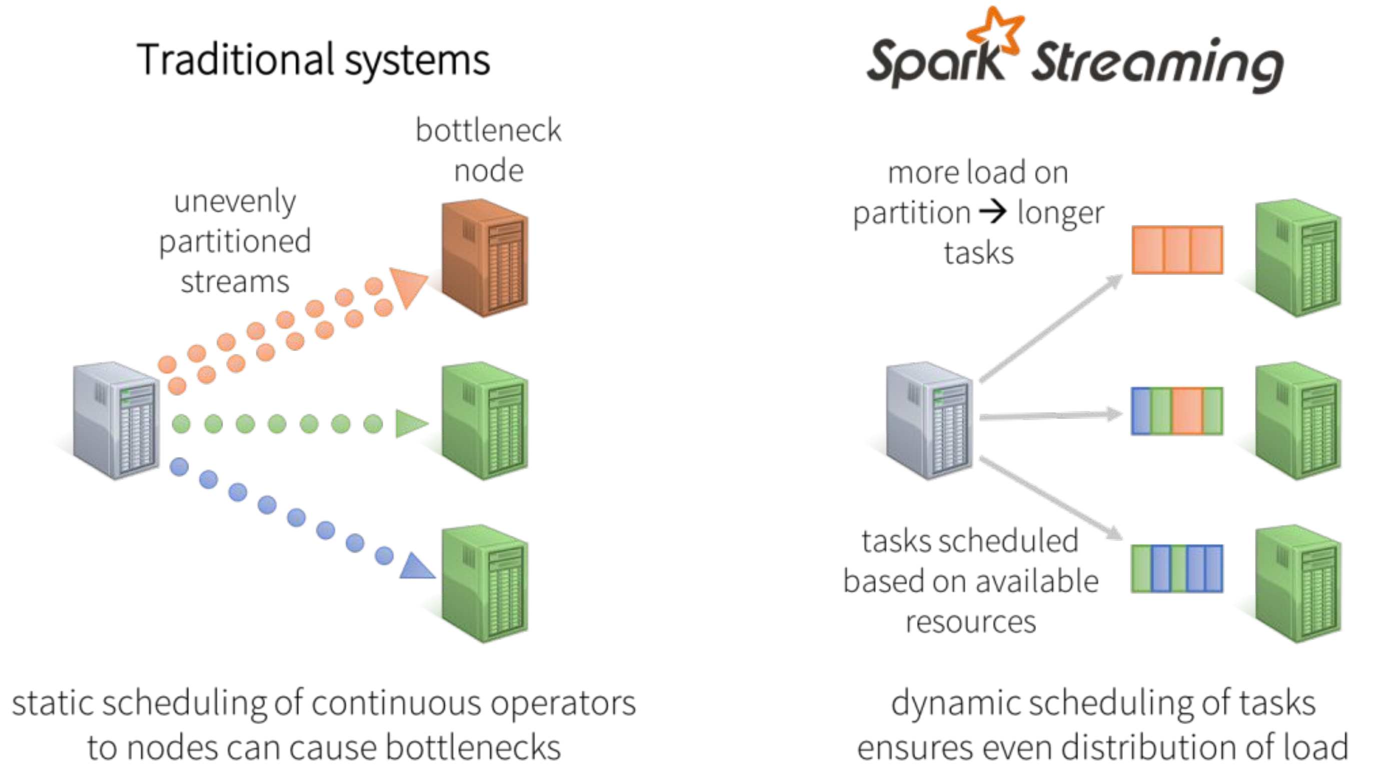
How does it work?

- Discretized Streams



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

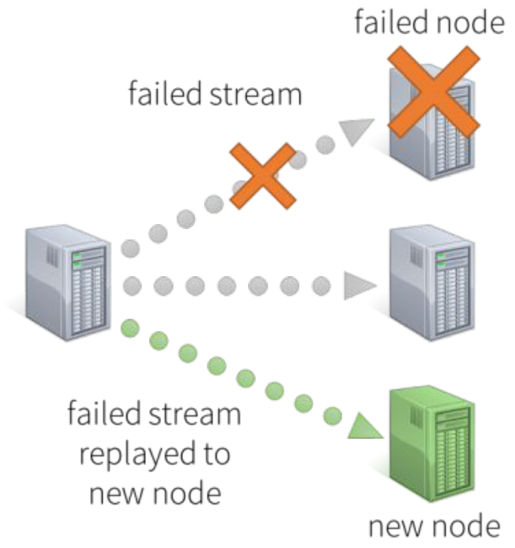
How does it work?


Traditional systems
bottleneck node
unevenly partitioned streams
static scheduling of continuous operators to nodes can cause bottlenecks
Spark Streaming
more load on partition → longer tasks
tasks scheduled based on available resources
dynamic scheduling of tasks ensures even distribution of load

https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html

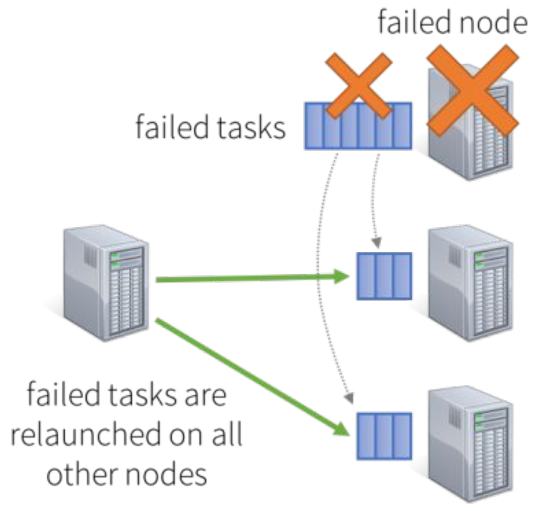
How does it work?

Traditional systems



slower recovery by using single node for recomputations

Spark Streaming



faster recovery by using multiple nodes for recomputations

https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html

Spark Streaming Semantics

Side effects

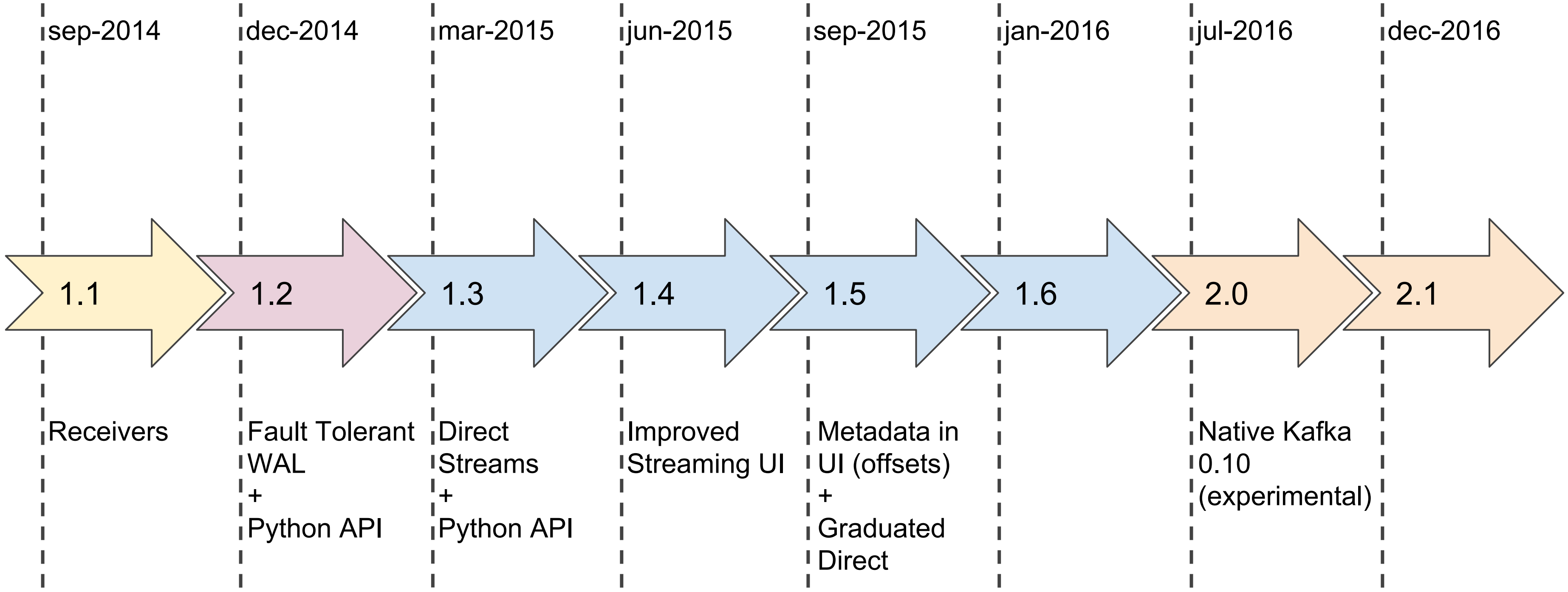
As in Spark:

- **Not guarantee** exactly-once semantics for output actions
- Any side-effecting output operations **may be repeated**
- Because of node failure, process failure, etc.

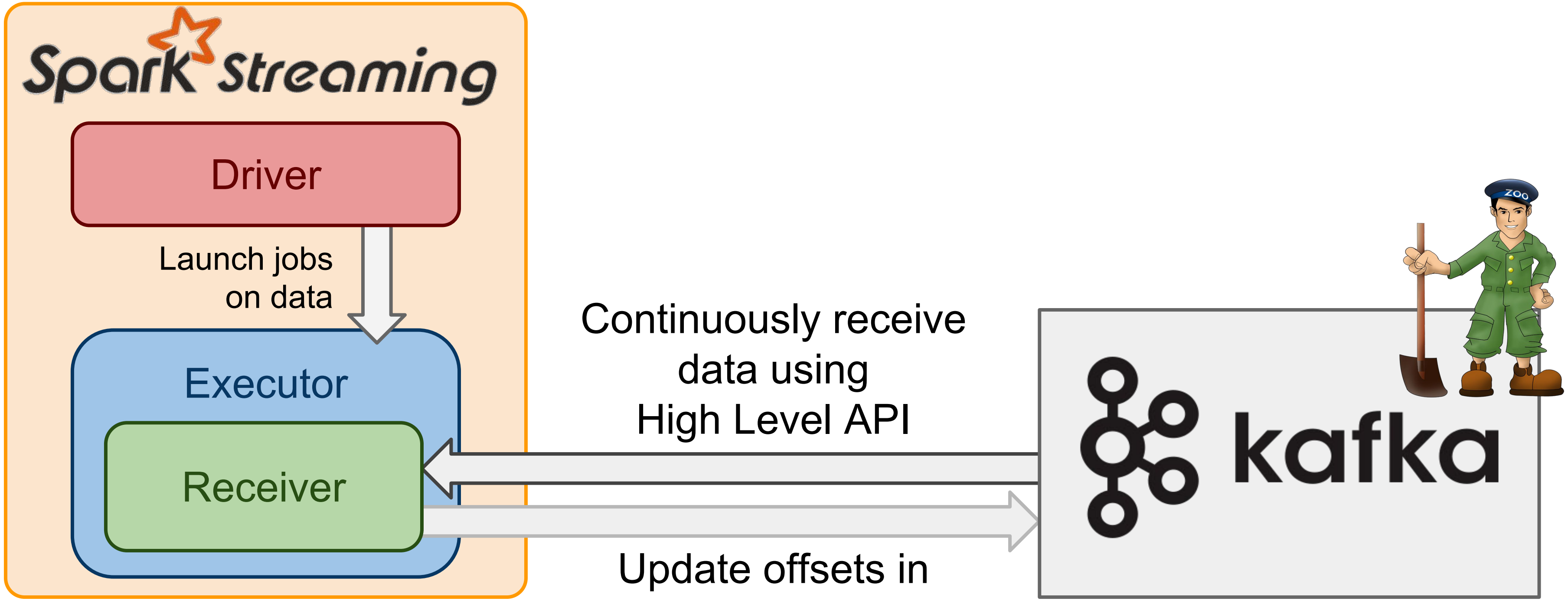
So, be careful when outputting to external sources

Spark Streaming Kafka Integration

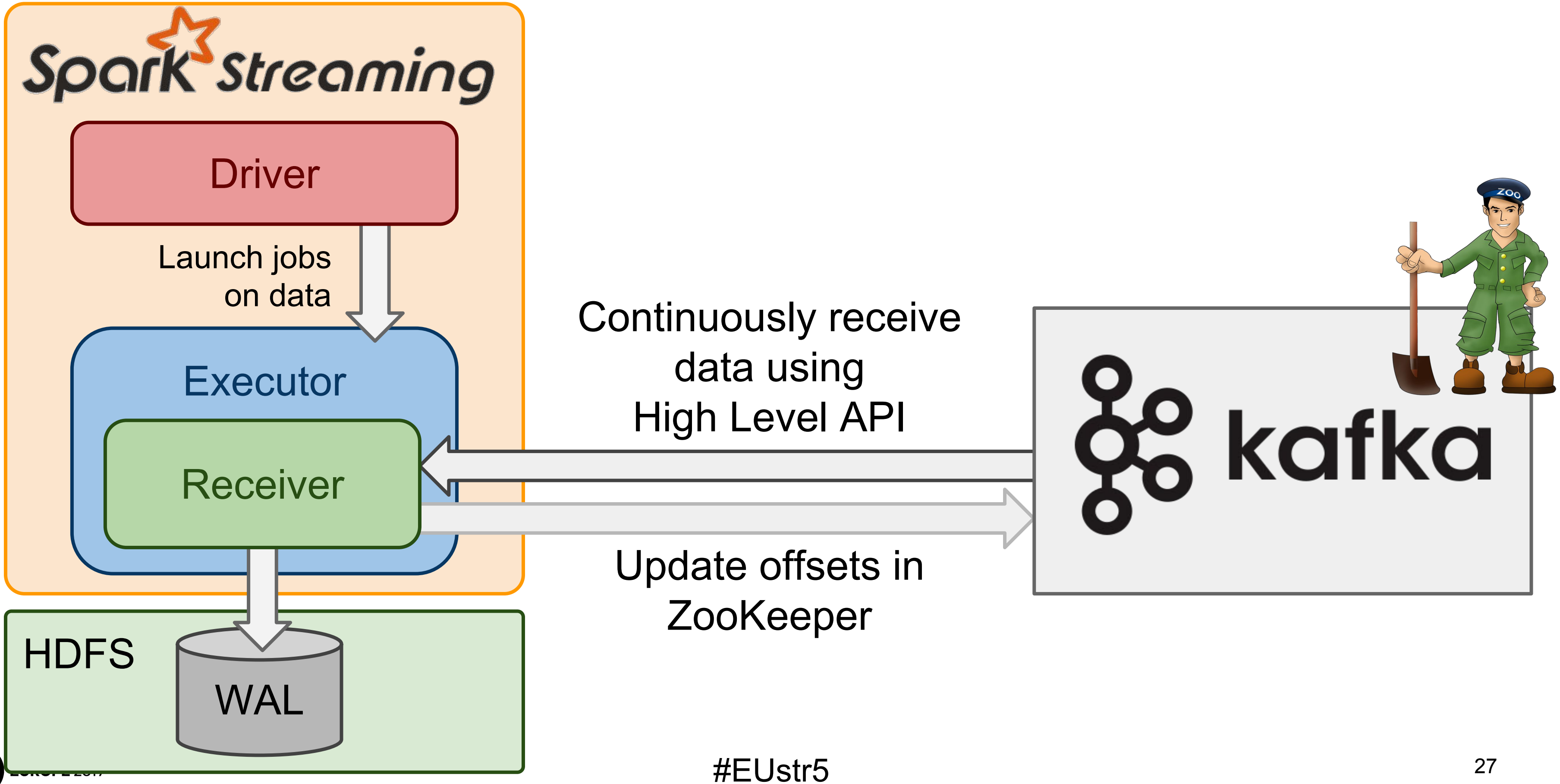
Spark Streaming Kafka Integration Timeline


sep-2014
1.1
Receivers
dec-2014
1.2
Fault Tolerant WAL + Python API
mar-2015
1.3
Direct Streams + Python API
jun-2015
1.4
Improved Streaming UI
sep-2015
1.5
Metadata in UI (offsets) + Graduated Direct
jan-2016
1.6
jul-2016
2.0
Native Kafka 0.10 (experimental)
dec-2016
2.1

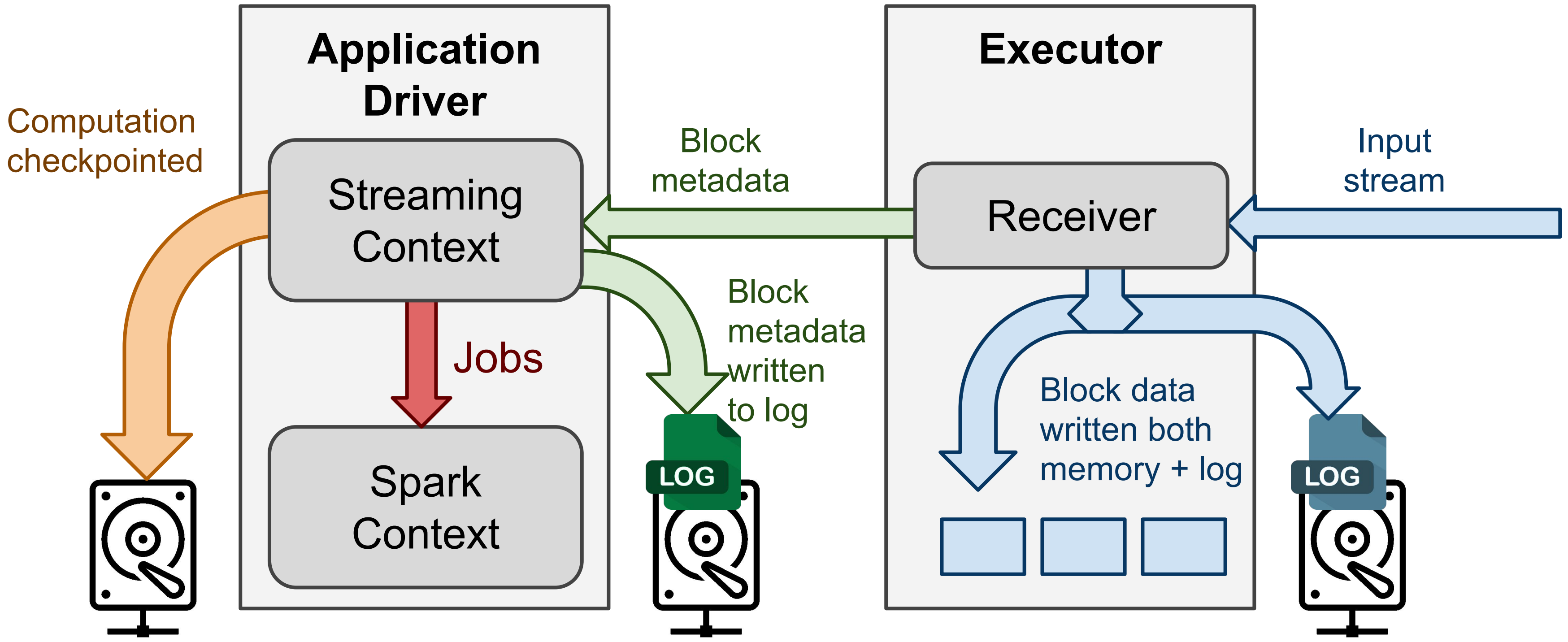
Kafka Receiver (≤ Spark 1.1)


Spark Streaming
Driver
Launch jobs on data
Executor
Receiver
Continuously receive data using High Level API
Update offsets in ZooKeeper
kafka
zoo

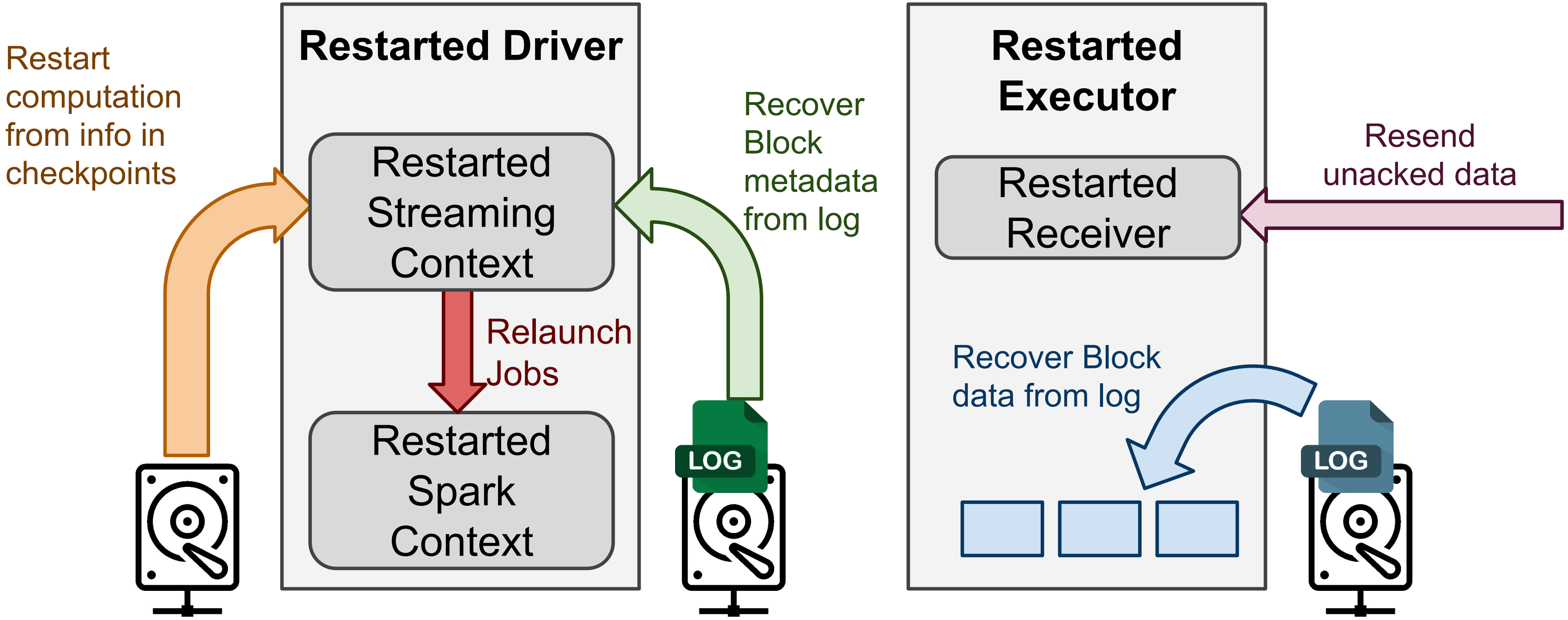
Kafka Receiver with WAL (Spark 1.2)


Spark Streaming
Driver
Launch jobs on data
Executor
Receiver
Continuously receive data using High Level API
ZOO
kafka
Update offsets in ZooKeeper
HDFS
WAL

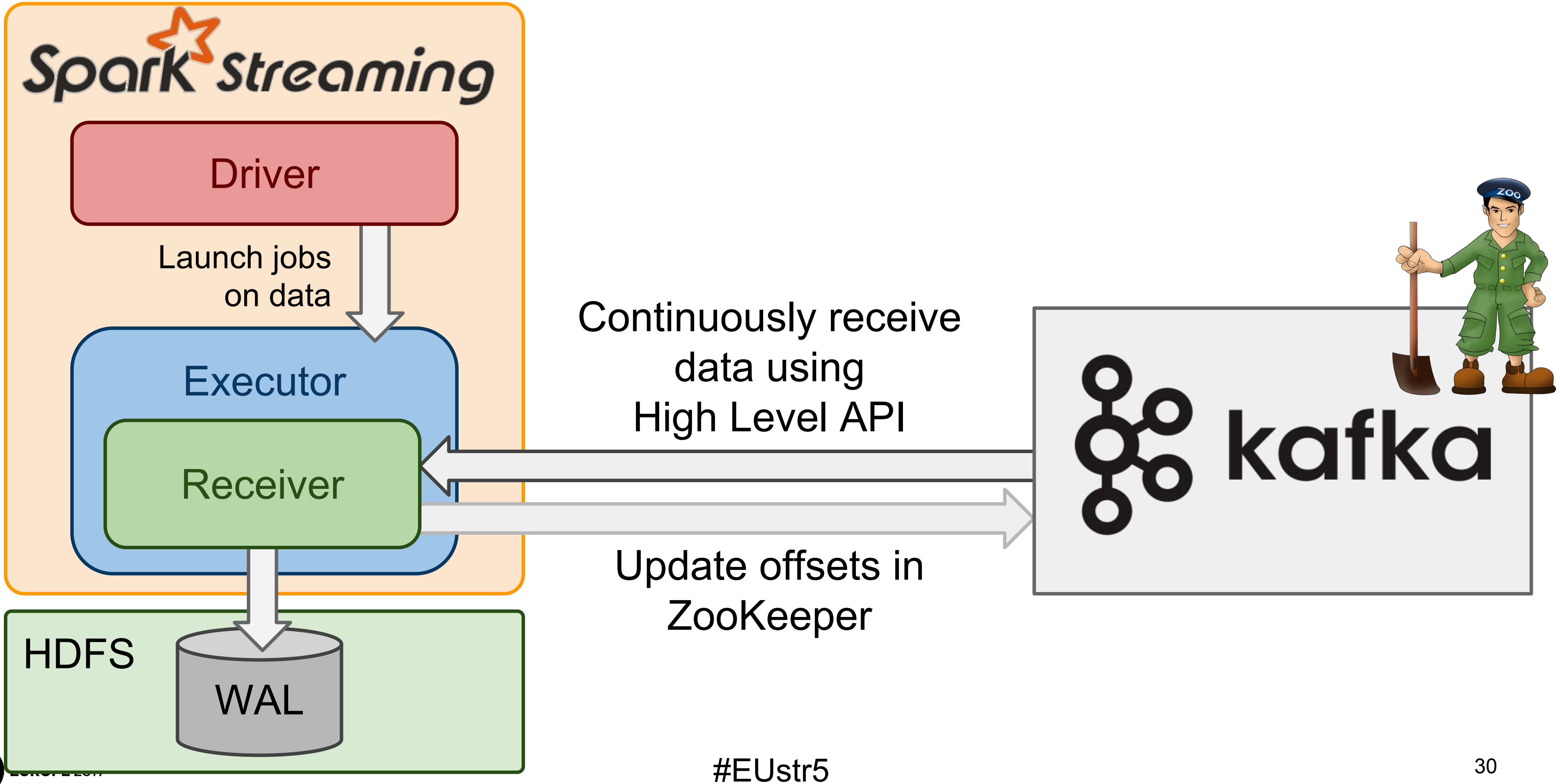
Kafka Receiver with WAL (Spark 1.2)



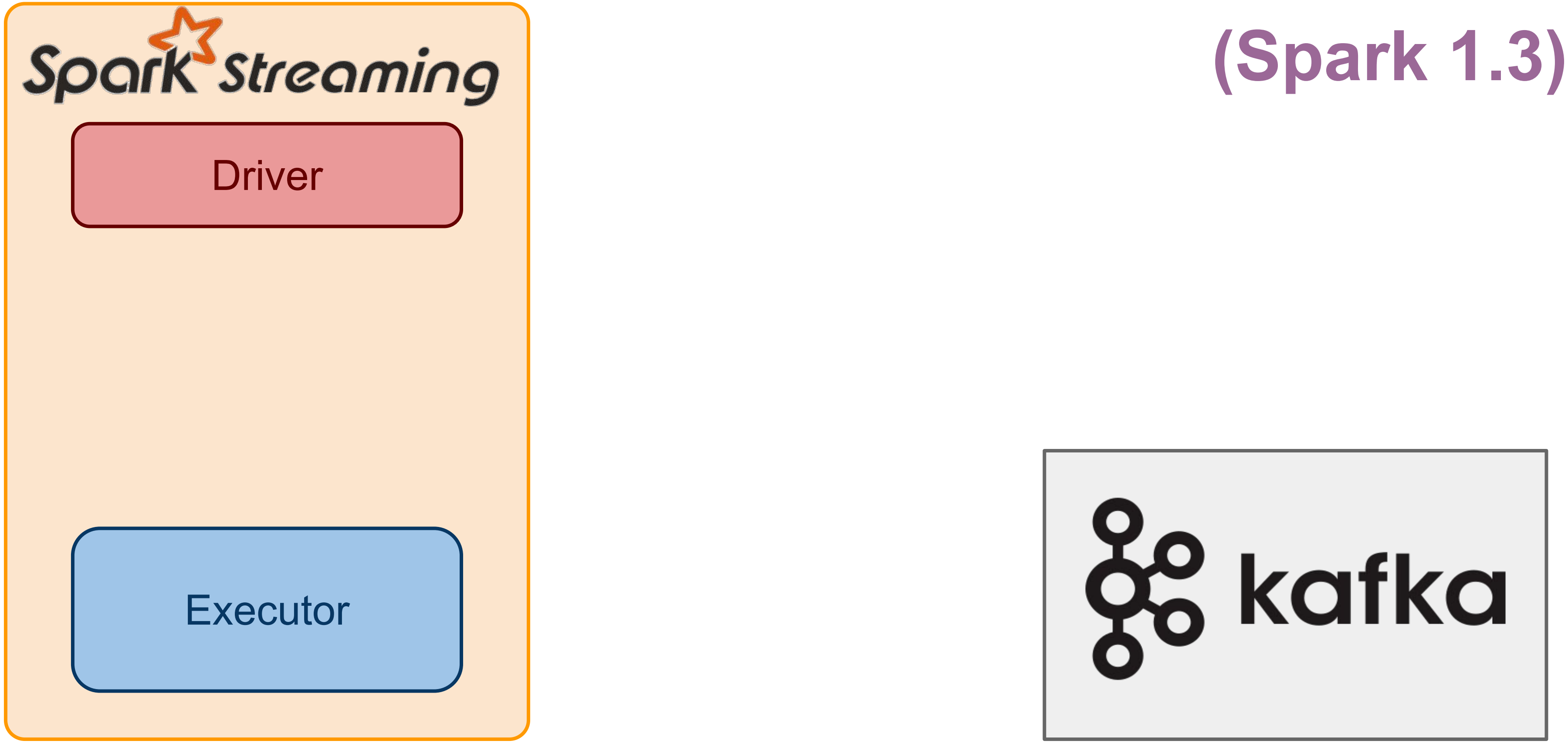
Kafka Receiver with WAL (Spark 1.2)


Restart computation from info in checkpoints
Restarted Driver
Restarted Streaming Context
Relaunch Jobs
Restarted Spark Context
Recover Block metadata from log
LOG
Restarted Executor
Restarted Receiver
Resend unacked data
Recover Block data from log
LOG

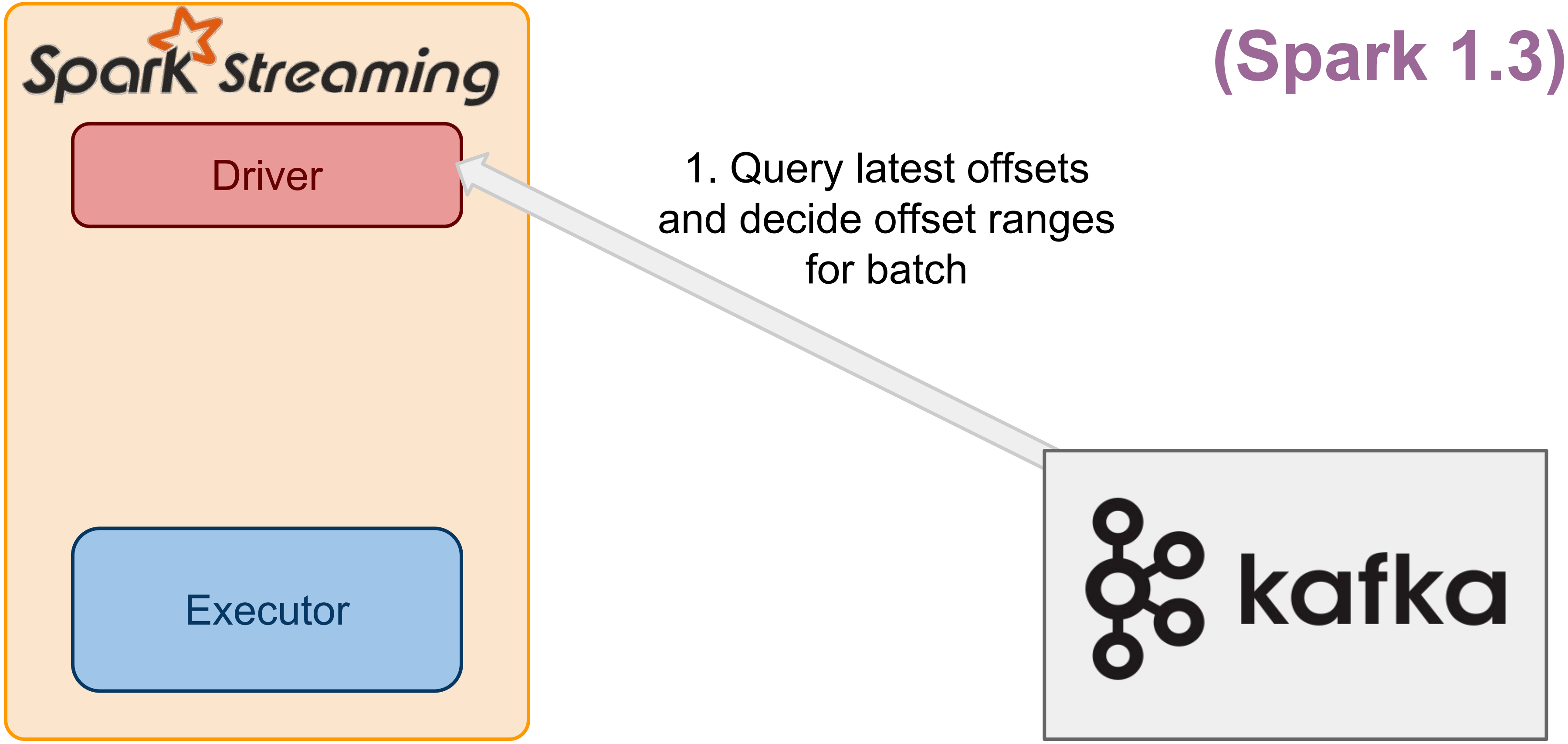
Kafka Receiver with WAL (Spark 1.2)


Spark Streaming
Driver
Launch jobs on data
Executor
Receiver
HDFS
WAL
Continuously receive data using High Level API
Update offsets in ZooKeeper
kafka
ZOO

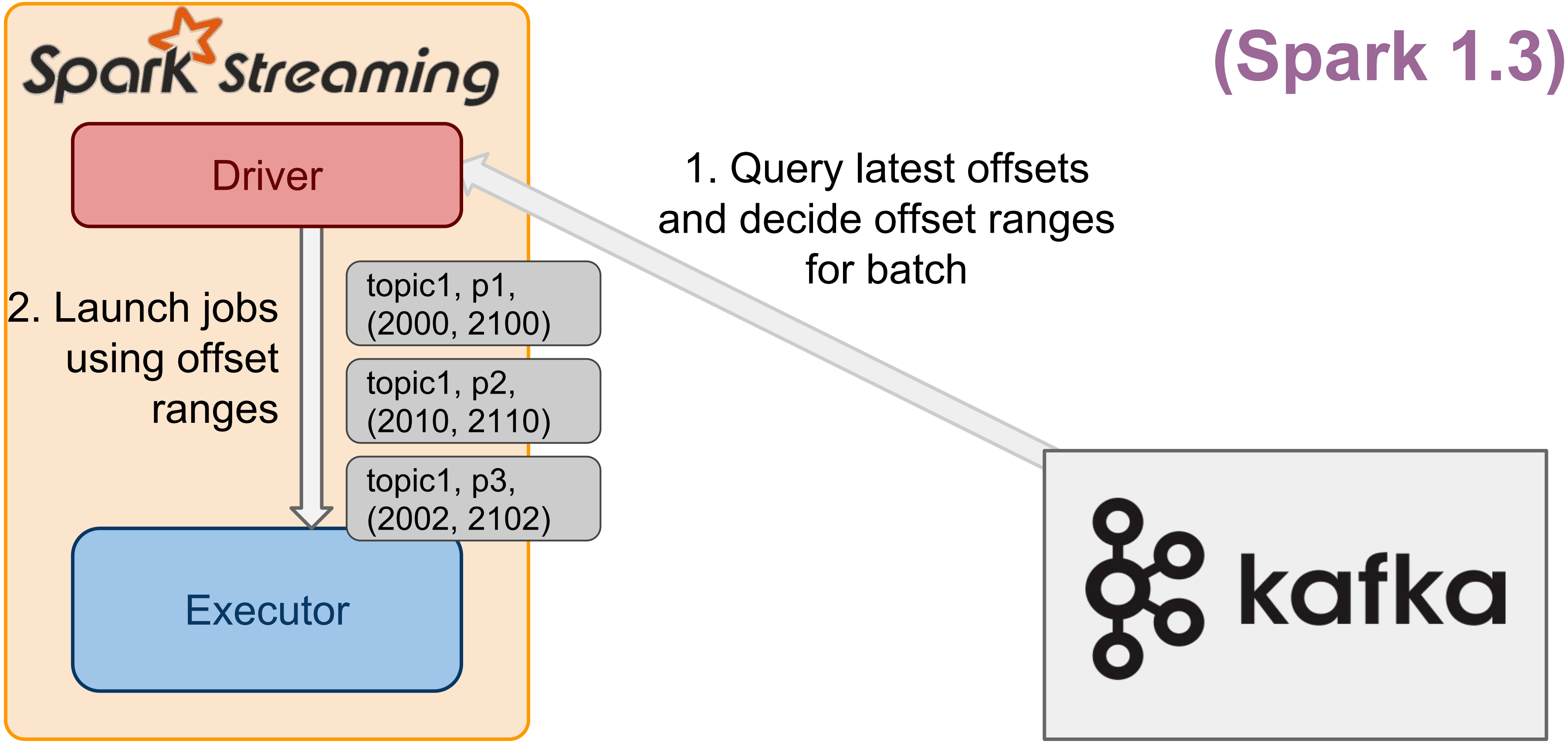
Direct Kafka Integration w/o Receivers or WALs (Spark 1.3)



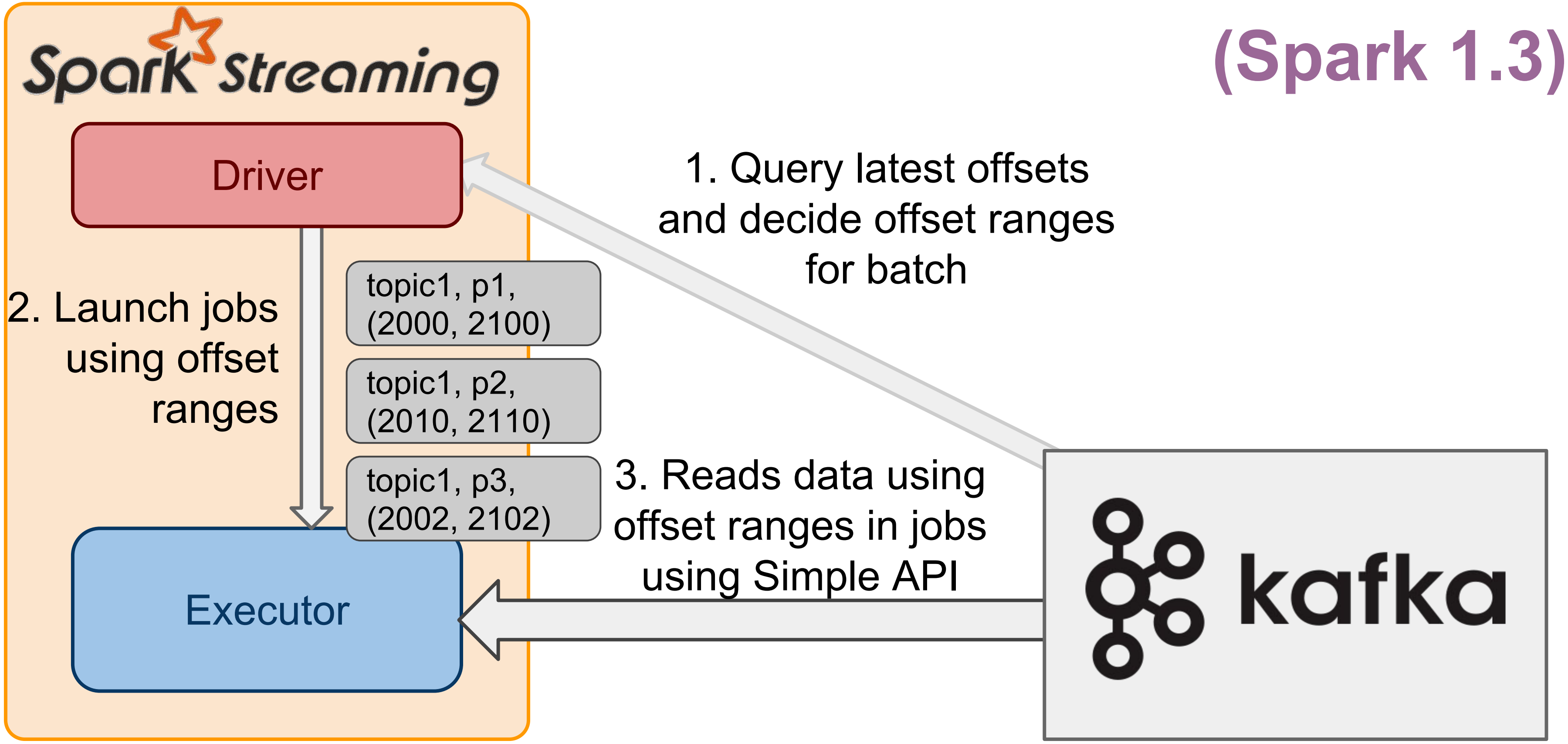
Direct Kafka Integration w/o Receivers or WALs (Spark 1.3)



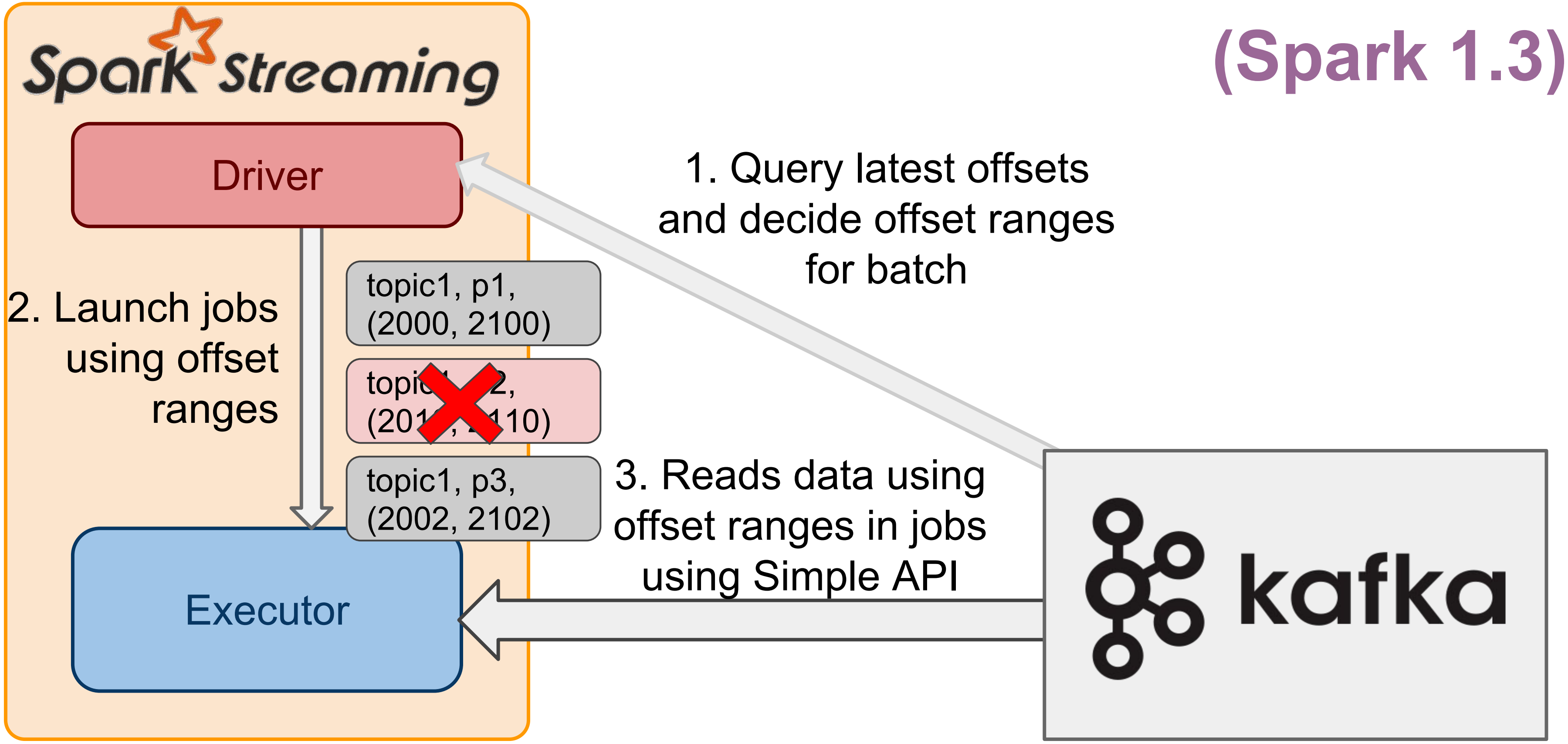
Direct Kafka Integration w/o Receivers or WALs (Spark 1.3)



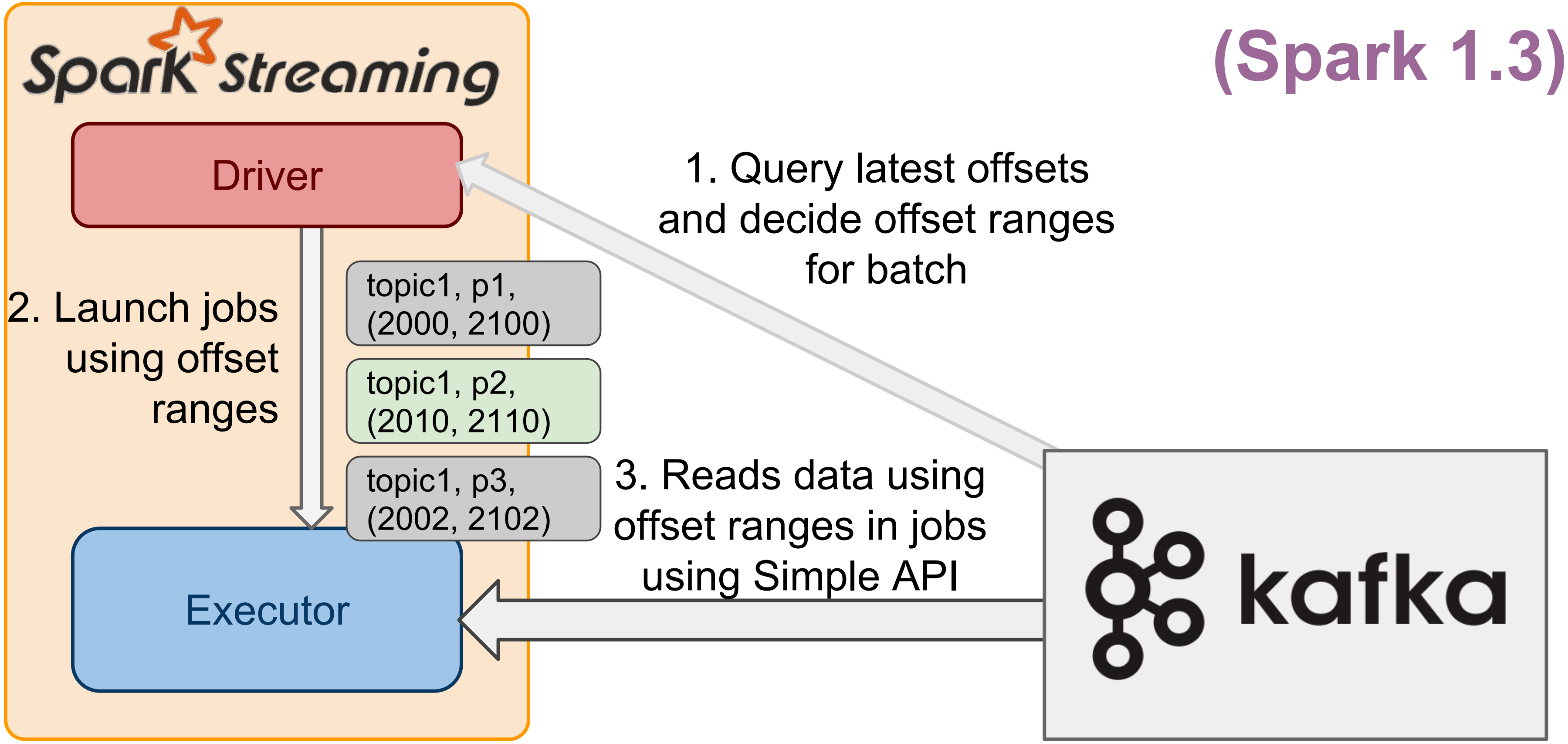
Direct Kafka Integration w/o Receivers or WALs (Spark 1.3)


Spark Streaming
Driver
2. Launch jobs using offset ranges
topic1, p1, (2000, 2100)
topic1, p2, (2010, 2110)
topic1, p3, (2002, 2102)
Executor
1. Query latest offsets and decide offset ranges for batch
3. Reads data using offset ranges in jobs using Simple API
kafka

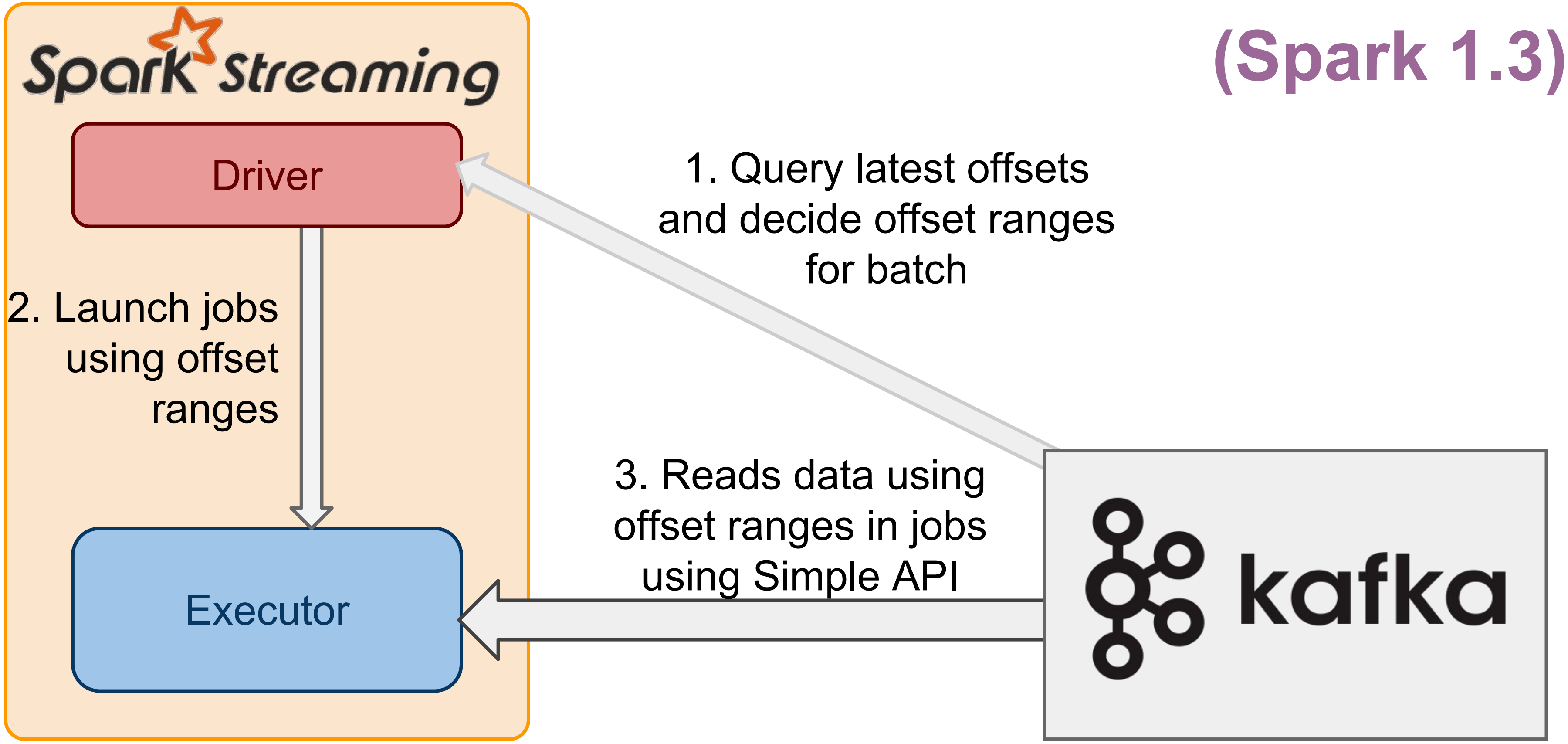
Direct Kafka Integration w/o Receivers or WALs (Spark 1.3)


Spark Streaming
Driver
2. Launch jobs using offset ranges
topic1, p1, (2000, 2100)
topic1, p2, (2010, 2110)
topic1, p3, (2002, 2102)
Executor
1. Query latest offsets and decide offset ranges for batch
3. Reads data using offset ranges in jobs using Simple API
kafka

Direct Kafka Integration w/o Receivers or WALs (Spark 1.3)


Spark Streaming
Driver
2. Launch jobs using offset ranges
topic1, p1, (2000, 2100)
topic1, p2, (2010, 2110)
topic1, p3, (2002, 2102)
Executor
1. Query latest offsets and decide offset ranges for batch
3. Reads data using offset ranges in jobs using Simple API
kafka

Direct Kafka Integration w/o Receivers or WALs (Spark 1.3)

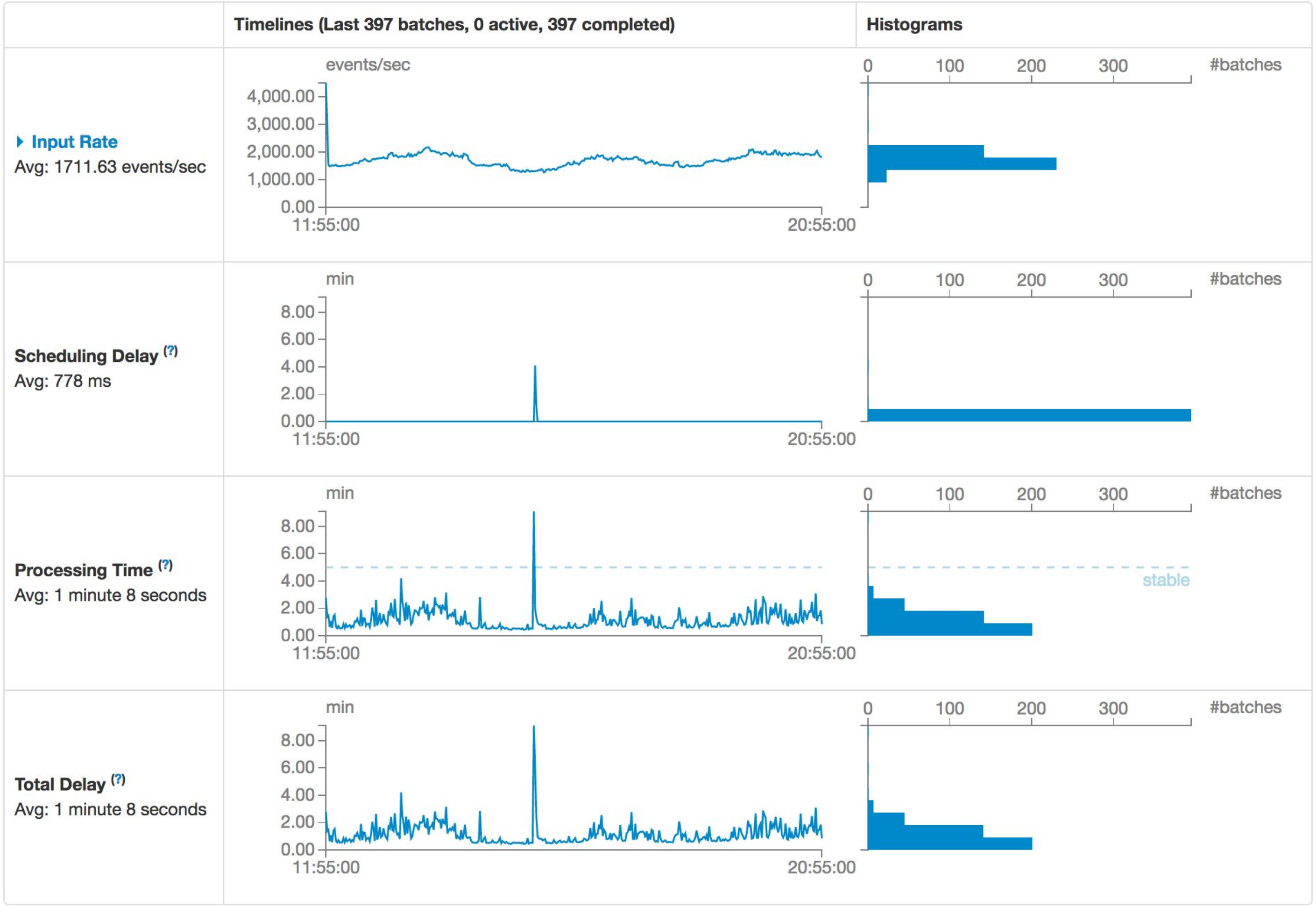

Spark Streaming
Driver
1. Query latest offsets and decide offset ranges for batch
2. Launch jobs using offset ranges
Executor
3. Reads data using offset ranges in jobs using Simple API
kafka

Direct Kafka API benefits

- No WALs or Receivers
- Allows end-to-end exactly-once semantics pipelines *

  * updates to downstream systems should be idempotent or transactional

- More fault-tolerant
- More efficient
- Easier to use.



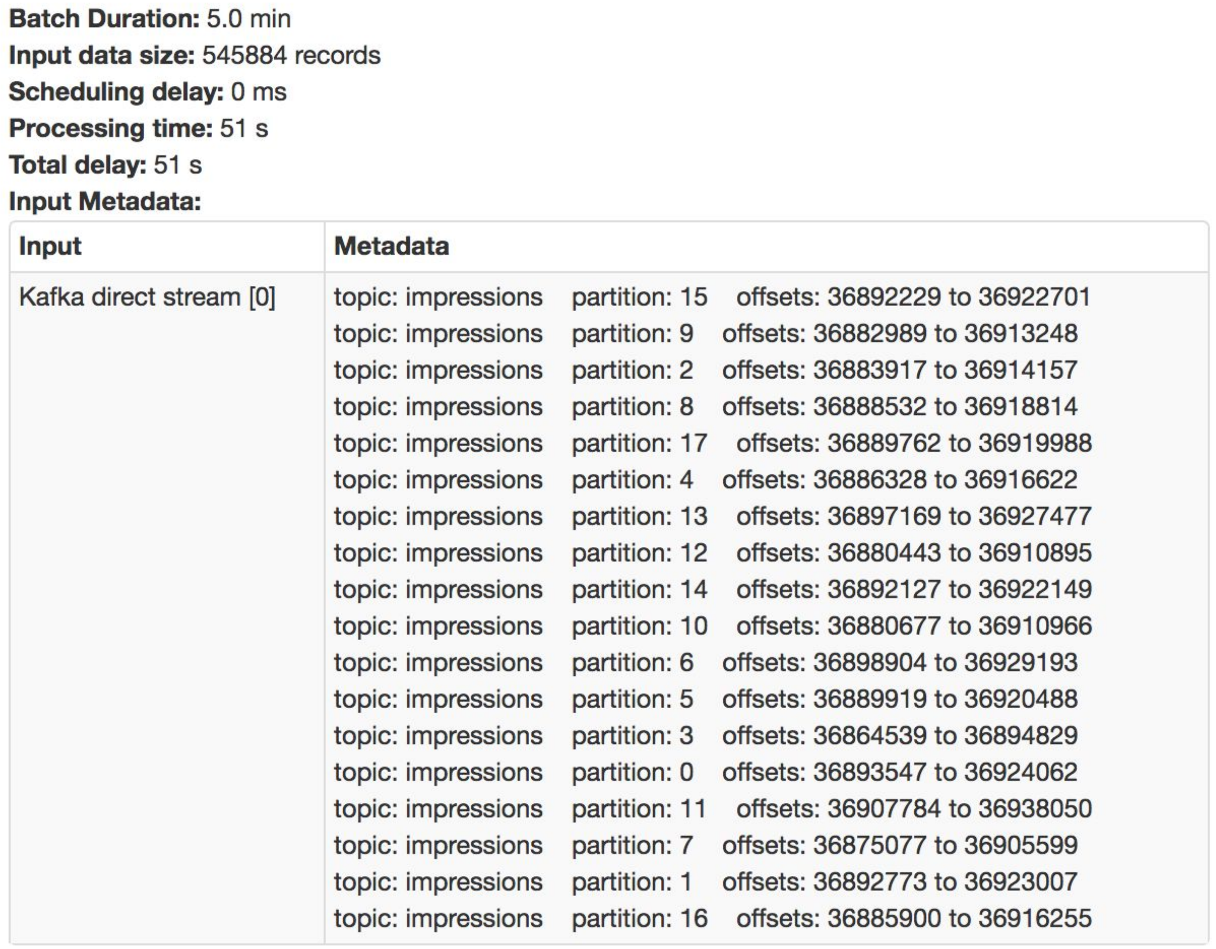
Timelines (Last 397 batches, 0 active, 397 completed)

Histograms

Input Rate Avg: 1711.63 events/sec		
Scheduling Delay (?) Avg: 778 ms		
Processing Time (?) Avg: 1 minute 8 seconds		
Total Delay (?) Avg: 1 minute 8 seconds		

SPARK SUMMIT EUROPE 2017

Spark Streaming UI improvements (Spark 1.4)



Batch Duration: 5.0 min
Input data size: 545884 records
Scheduling delay: 0 ms
Processing time: 51 s
Total delay: 51 s
Input Metadata:

Input	Metadata
Kafka direct stream [0]	topic: impressions partition: 15 offsets: 36892229 to 36922701
	topic: impressions partition: 9 offsets: 36882989 to 36913248
	topic: impressions partition: 2 offsets: 36883917 to 36914157
	topic: impressions partition: 8 offsets: 36888532 to 36918814
	topic: impressions partition: 17 offsets: 36889762 to 36919988
	topic: impressions partition: 4 offsets: 36886328 to 36916622
	topic: impressions partition: 13 offsets: 36897169 to 36927477
	topic: impressions partition: 12 offsets: 36880443 to 36910895
	topic: impressions partition: 14 offsets: 36892127 to 36922149
	topic: impressions partition: 10 offsets: 36880677 to 36910966
	topic: impressions partition: 6 offsets: 36898904 to 36929193
	topic: impressions partition: 5 offsets: 36889919 to 36920488
	topic: impressions partition: 3 offsets: 36864539 to 36894829
	topic: impressions partition: 0 offsets: 36893547 to 36924062
	topic: impressions partition: 11 offsets: 36907784 to 36938050
	topic: impressions partition: 7 offsets: 36875077 to 36905599
	topic: impressions partition: 1 offsets: 36892773 to 36923007
	topic: impressions partition: 16 offsets: 36885900 to 36916255

Kafka Metadata (offsets) in UI (Spark 1.5)

What about Spark 2.0+ and new Kafka Integration?

This is why we are here, right?

Spark 2.0+ new Kafka Integration

	spark-streaming-kafka-0-8	spark-streaming-kafka-0-10
Broker Version	0.8.2.1 or higher	0.10.0 or higher
Api Stability	Stable	Experimental
Language Support	Scala, Java, Python	Scala, Java
Receiver DStream	Yes	No
Direct DStream	Yes	Yes
SSL / TLS Support	No	Yes
Offset Commit Api	No	Yes
Dynamic Topic Subscription	No	Yes

What’s really New with this New Kafka Integration?

- New Consumer API
  * Instead of Simple API
- Location Strategies
- Consumer Strategies
- SSL / TLS

- No Python API :(

Location Strategies

- New consumer API will pre-fetch messages into buffers
- So, keep cached consumers into executors
- It’s better to schedule partitions on the host with appropriate consumers

Location Strategies

- `PreferConsistent`

  Distribute partitions evenly across available executors

- `PreferBrokers`

  If your executors are on the same hosts as your Kafka brokers

- `PreferFixed`

  Specify an explicit mapping of partitions to hosts

Consumer Strategies

- New consumer API has a number of different ways to specify topics, some of which require considerable post-object-instantiation setup.

- `ConsumerStrategies` provides an abstraction that allows Spark to obtain properly configured consumers even after restart from checkpoint.

Consumer Strategies

- `Subscribe` subscribe to a fixed collection of topics
- `SubscribePattern` use a regex to specify topics of interest
- `Assign` specify a fixed collection of partitions

- Overloaded constructors to specify the starting offset for a particular partition.
- `ConsumerStrategy` is a public class that you can extend.

SSL/TTL encryption

- New consumer API supports SSL
- **Only** applies to communication between Spark and Kafka brokers
- **Still** responsible for separately securing Spark inter-node communication

How to use New Kafka Integration on Spark 2.0+

Scala Example Code

Basic usage

```
val kafkaParams = Map[String, Object](
  "bootstrap.servers" -> "broker01:9092,broker02:9092",
  "key.deserializer" -> classOf[StringDeserializer],
  "value.deserializer" -> classOf[StringDeserializer],
  "group.id" -> "stream_group_id",
  "auto.offset.reset" -> "latest",
  "enable.auto.commit" -> (false: java.lang.Boolean)
)

val topics = Array("topicA", "topicB")

val stream = KafkaUtils.createDirectStream[String, String](
  streamingContext,
  PreferConsistent,
  Subscribe[String, String](topics, kafkaParams)
)

stream.map(record => (record.key, record.value))
```

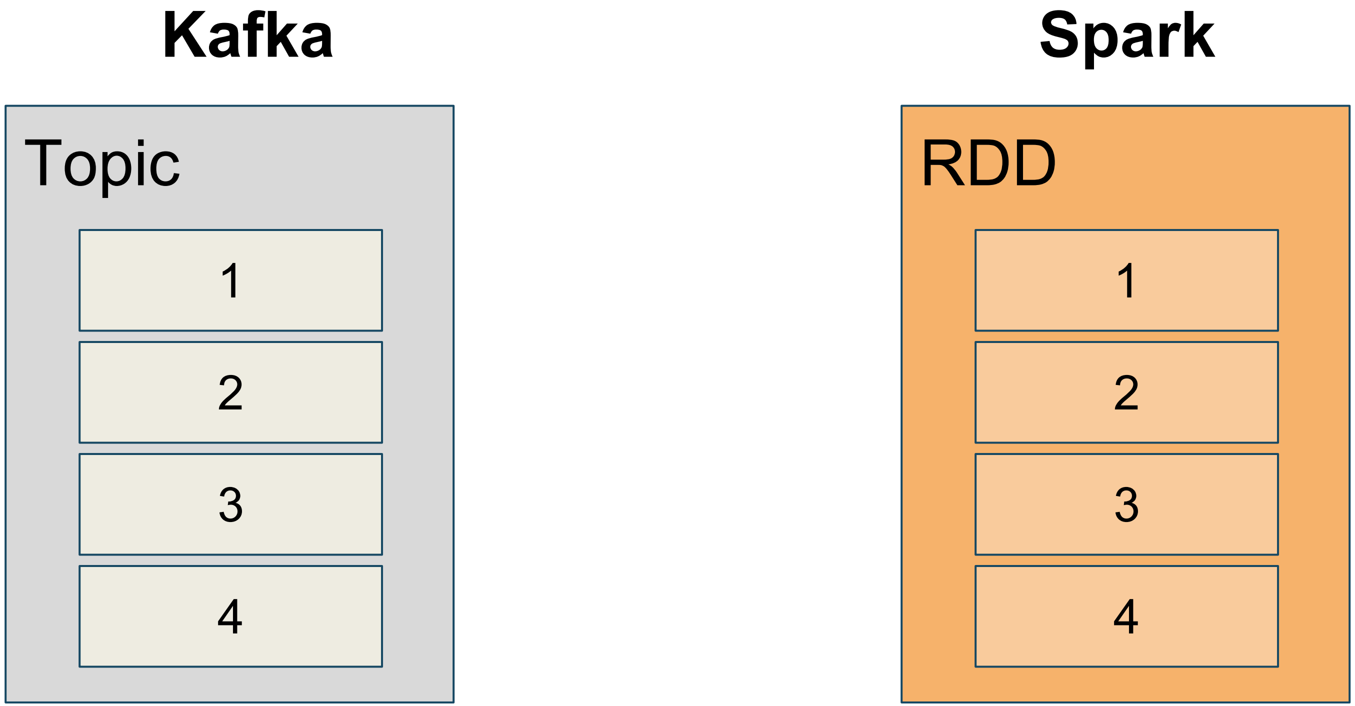
How to use New Kafka Integration on Spark 2.0+

Java Example Code

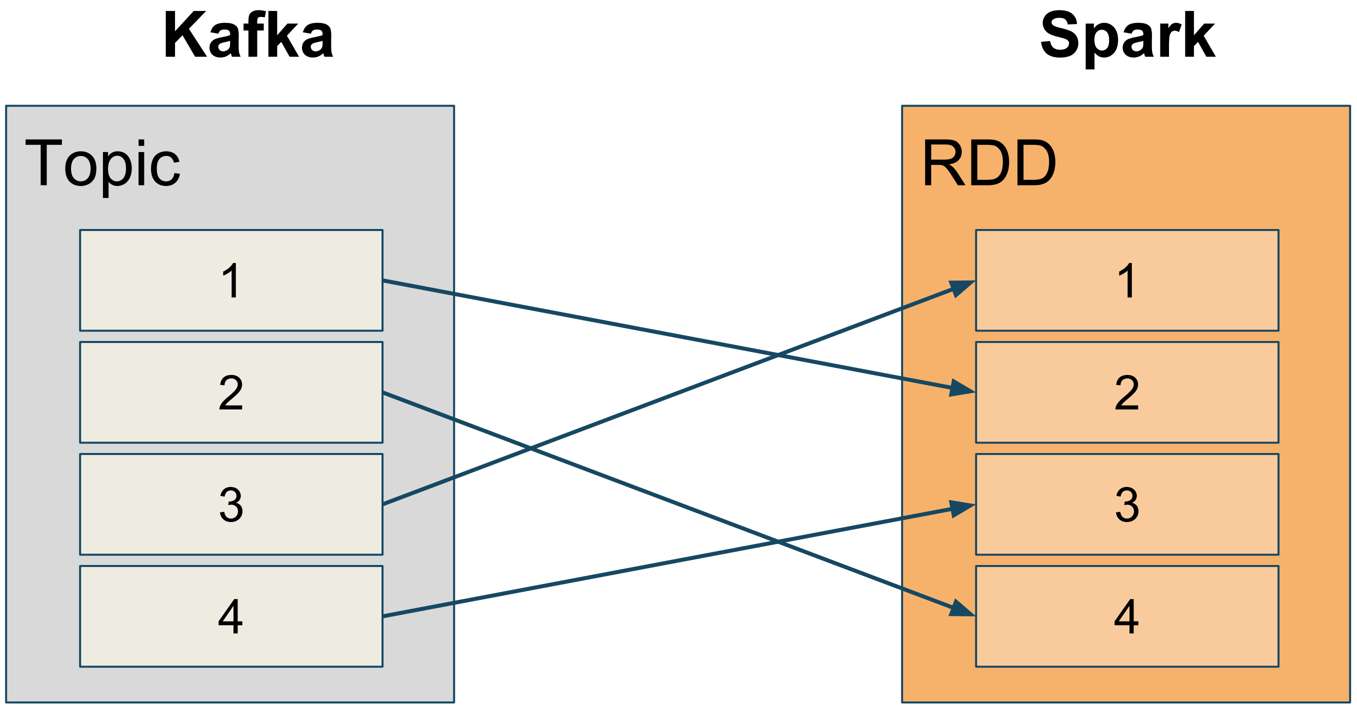
Getting metadata

```
stream.foreachRDD { rdd =>
  val offsetRanges = rdd.asInstanceOf[HasOffsetRanges]
                                                  .offsetRanges
  rdd.foreachPartition { iter =>
      val osr: OffsetRange = offsetRanges(
                             TaskContext.get.partitionId)
      // get any needed data from the offset range
      val topic = osr.topic
      val kafkaPartitionId = osr.partition
      val begin = osr.fromOffset
      val end = osr.untilOffset
  }
}
```

Kafka or Spark RDD Partitions?



Kafka or Spark RDD Partitions?



How to use New Kafka Integration on Spark 2.0+

Java Example Code

Getting metadata

```
stream.foreachRDD { rdd =>
  val offsetRanges = rdd.asInstanceOf[HasOffsetRanges]
                                                      .offsetRanges
  rdd.foreachPartition { iter =>
      val osr: OffsetRange = offsetRanges(
                                 TaskContext.get.partitionId)
      // get any needed data from the offset range
      val topic = osr.topic
      val kafkaPartitionId = osr.partition
      val begin = osr.fromOffset
      val end = osr.untilOffset
  }
}
```

How to use New Kafka Integration on Spark 2.0+

Java Example Code

Store offsets in Kafka itself: Commit API

```
stream.foreachRDD { rdd =>
  val offsetRanges = rdd.asInstanceOf[HasOffsetRanges]
                                                     .offsetRanges

  // DO YOUR STUFF with DATA

  stream.asInstanceOf[CanCommitOffsets]
                                             .commitAsync(offsetRanges)
  }
}
```

- At most once
- At least once
- Exactly once

Kafka + Spark Semantics

Kafka + Spark Semantics

At most once

- We don’t want duplicates
- Not worth the hassle of ensuring that messages don’t get lost
- Example: Sending statistics over UDP

1. Set `spark.task.maxFailures` to `1`
2. Make sure spark.speculation is `false` (the default)
3. Set Kafka param `auto.offset.reset` to `“largest”`
4. Set Kafka param `enable.auto.commit` to `true`

Kafka + Spark Semantics

At most once

- This will mean you lose messages on restart
- At least they shouldn’t get replayed.

- Test this carefully if it’s actually important to you that a message ***never*** gets repeated, because it’s not a common use case.

Kafka + Spark Semantics

At least once

- We don’t want to loose any record
- We don’t care about duplicates
- Example: Sending internal alerts on relative rare occurrences on the stream

1. Set `spark.task.maxFailures` > `1000`
2. Set Kafka param `auto.offset.reset` to `“smallest”`
3. Set Kafka param `enable.auto.commit` to `false`

Kafka + Spark Semantics

At least once

- Don’t be silly! Do **NOT** replay your whole log on every restart…
- Manually commit the offsets when you are 100% sure records are processed

- If this is “too hard” you’d better have a relative short retention log
- Or be **REALLY** ok with duplicates. For example, you are outputting to an external system that handles duplicates for you (HBase)

Kafka + Spark Semantics

Exactly once

- We don’t want to loose any record
- We don’t want duplicates either
- Example: Storing stream in data warehouse

1. We need some kind of idempotent writes, or whole-or-nothing writes (transactions)
2. Only store offsets EXACTLY after writing data
3. Same parameters as at least once

Kafka + Spark Semantics

Exactly once

- Probably the hardest to achieve right
- Still *some small* chance of failure if your app fails just between writing data and committing offsets… (but ***REALLY*** small)

Apache Kafka
Apacke Spark

at Billy Mobile

billy

15B
records monthly

35TB
weekly retention log

6K
events/second

x4
growth/year

Our use cases

ETL to Data Warehouse

- Input events from Kafka
- Enrich events with some external data sources
- Finally store it to Hive

We do **NOT** want duplicates

We do **NOT** want to lose events

Our use cases

ETL to Data Warehouse

- Hive is not transactional
- Neither idempotent writes
- Writing files to HDFS is “atomic” (whole or nothing)

- A relation 1:1 from each partition-batch to file in HDFS
- Store to ZK the current state of the batch
- Store to ZK offsets of last finished batch

Our use cases

Anomalies detector

billy

- Input events from Kafka
- Periodically load batch-computed model
- Detect when an offer stops converting (or too much)

- We do not care about losing some events (on restart)
- We always need to process the ***“real-time”*** stream

Our use cases

Anomalies detector

billy

- It’s useless to detect anomalies on a lagged stream!
- Actually it could be very bad

- Always restart stream on latest offsets
- Restart with “fresh” state

Our use cases

Store to
Entity Cache

billy

- Input events from Kafka
- Almost no processing
- Store it to HBase
    - (has idempotent writes)

- We do not care about duplicates
- We can **NOT** lose a single event

Our use cases

Store to Entity Cache

- Since HBase has idempotent writes, we can write events multiple times without hassle
- But, we do NOT start with earliest offsets…
    - That would be 7 days of redundant writes…!!!

- We store offsets of last finished batch
- But obviously we might re-write some events on restart or failure

SPARK SUMMIT EUROPE 2017

Lessons Learned

- Do NOT use checkpointing
  - Not recoverable across code upgrades
  - Do your own checkpointing
- Track offsets yourself
  - In general, more reliable:
    HDFS, ZK, RMDBS...
- Memory usually is an issue
  - You don’t want to waste it
  - Adjust `batchDuration`
  - Adjust `maxRatePerPartition`

Further Improvements

- Dynamic Allocation

  `spark.dynamicAllocation.enabled` vs
  `spark.streaming.dynamicAllocation.enabled`
  https://issues.apache.org/jira/browse/SPARK-12133
  But no reference in docs...

- Graceful shutdown

- Structured Streaming

Thank you very much!

Questions?

@joanvr

joanviladrosa

joan.viladrosa@billymob.com