kixer

Exactly-Once Streaming from Kafka

cody@koeninger.org

https://github.com/koeninger/kafka-exactly-once

Kafka is a ~~message queue~~ **circular buffer**

- Fixed size, based on disk space or time
- Oldest messages deleted to maintain size
- Messages are otherwise immutable
- Split into topic/partition
- Indexed only by offset
- Client tracks read offset, not server

Delivery semantics are *your responsibility*

At-most-once

1. Save offsets
2. !! Possible failure !!
3. Save results

On failure, restart at saved offset, messages are lost

At-least-once

1. Save results
2. !! Possible failure !!
3. Save offsets

On failure, messages are repeated

No possible magic config option to do better than this

Idempotent exactly-once

1. Save results with a natural unique key
2. !! Possible failure !!
3. Save offsets

On failure, messages are repeated, but we don't care

Immutable messages, pure transformation, same results

Idempotent pros / cons

Pro:

- Simple
- Works well for shape-preserving transformations (map)

Con:

- May be hard to identify natural unique key
- Especially hard for aggregate transformations (fold)
- Won't work for destructive updates

Note:

Results and offsets may be in different data stores

Transactional exactly-once

1. Begin transaction
2. Save results
3. Save offsets
4. Ensure offsets are ok (increasing without gaps)
5. Commit transaction

On failure, rollback, results and offsets remain in sync

Transactional pros / cons

Pro:

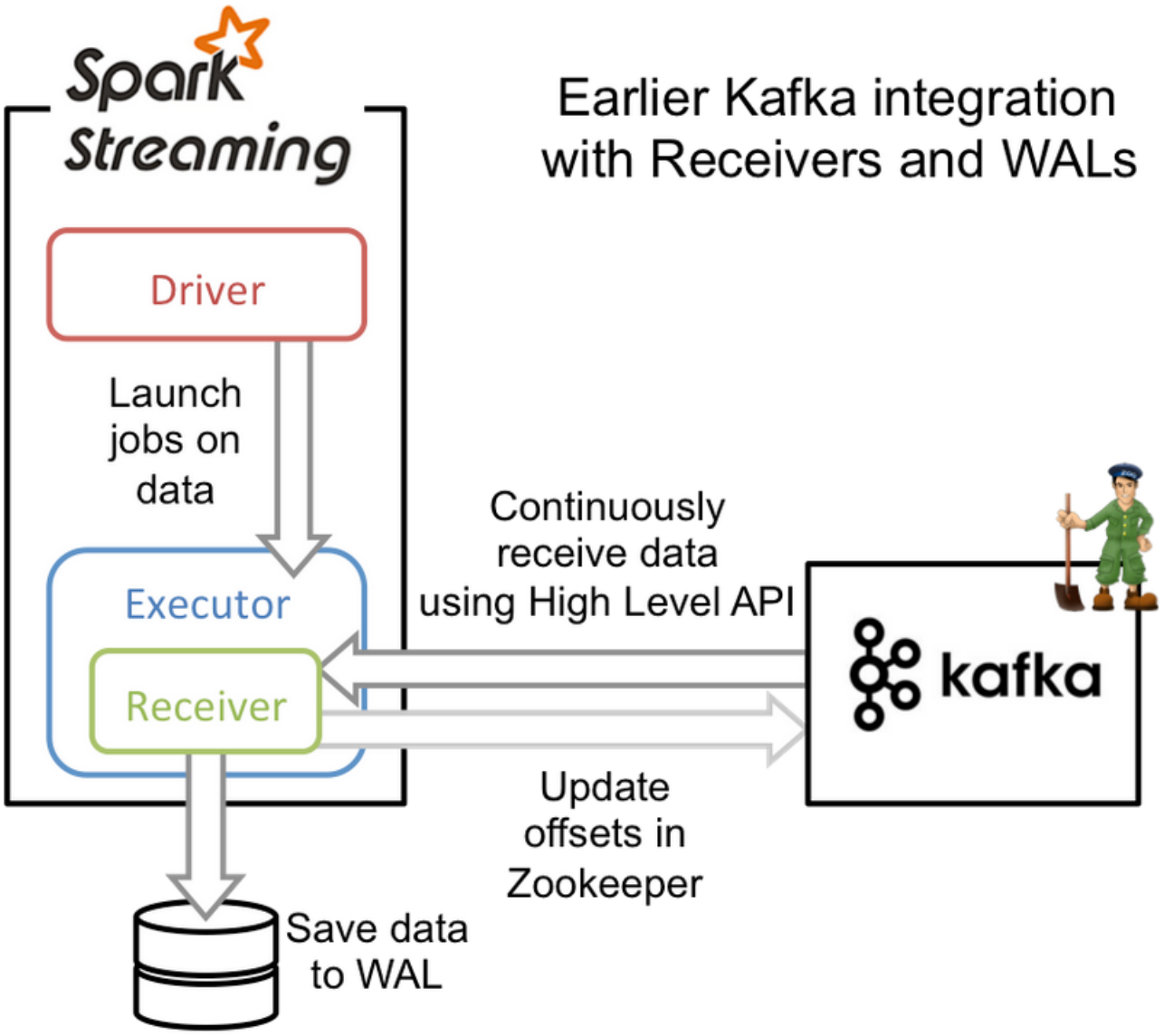
- Works easily for any transformation
- Destructive updates ok

Con:

- More complex
- Requires a transactional data store

Note:

- Results and offsets must be in same data store


Spark Streaming
Earlier Kafka integration with Receivers and WALs
Driver
Launch jobs on data
Executor
Receiver
Continuously receive data using High Level API
kafka
Update offsets in Zookeeper
Save data to WAL

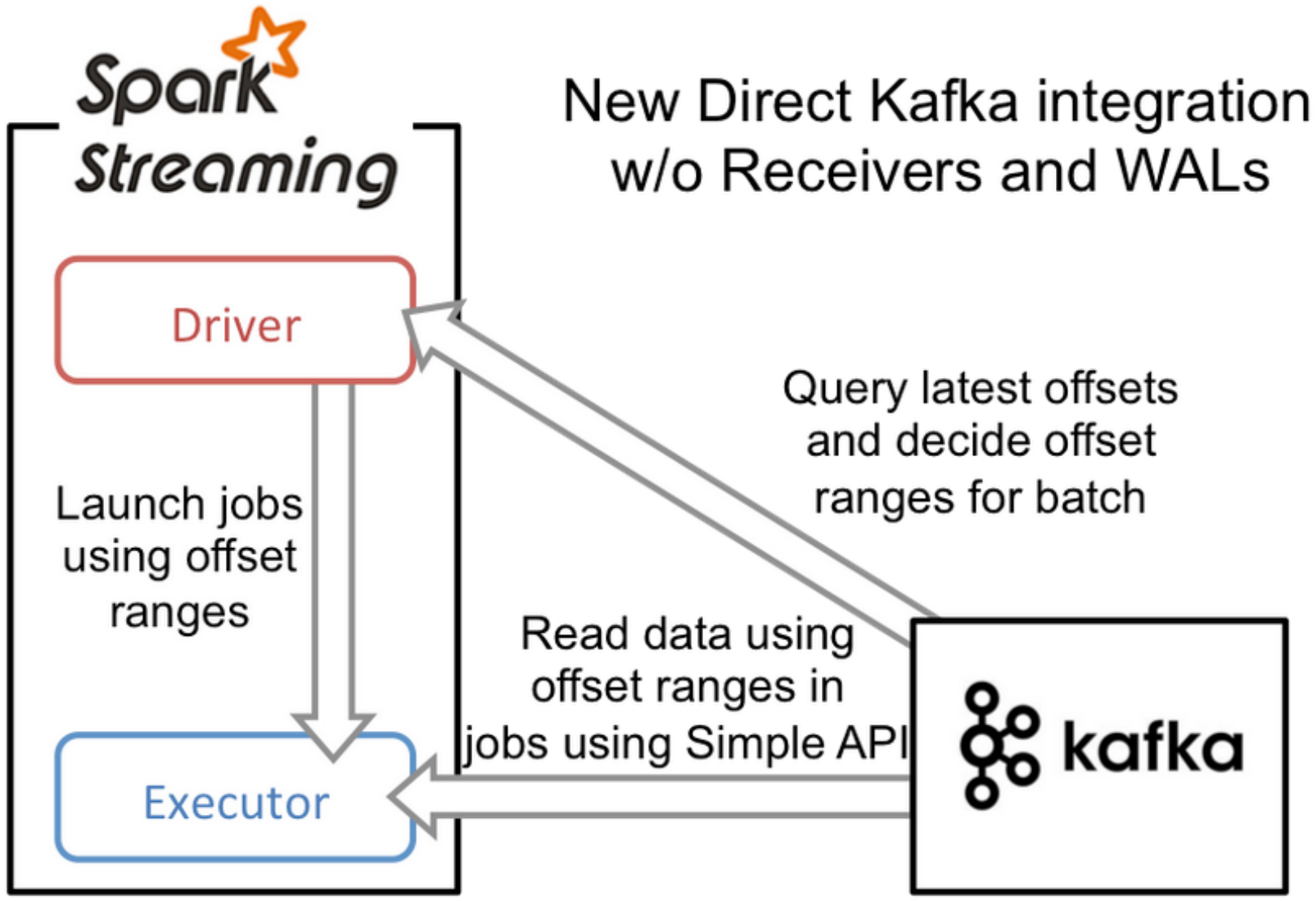
Receiver-based stream pros / cons

Pro:

- WAL design could work with non-Kafka data sources

Con:

- Long running receivers = parallelism awkward and costly
- Duplication of write operations
- Dependent on HDFS
- Must use idempotence for exactly-once
- No access to offsets, can't use transactional approach


Spark Streaming
New Direct Kafka integration w/o Receivers and WALs
Driver
Launch jobs using offset ranges
Executor
Query latest offsets and decide offset ranges for batch
Read data using offset ranges in jobs using Simple API
kafka

Direct stream pros / cons

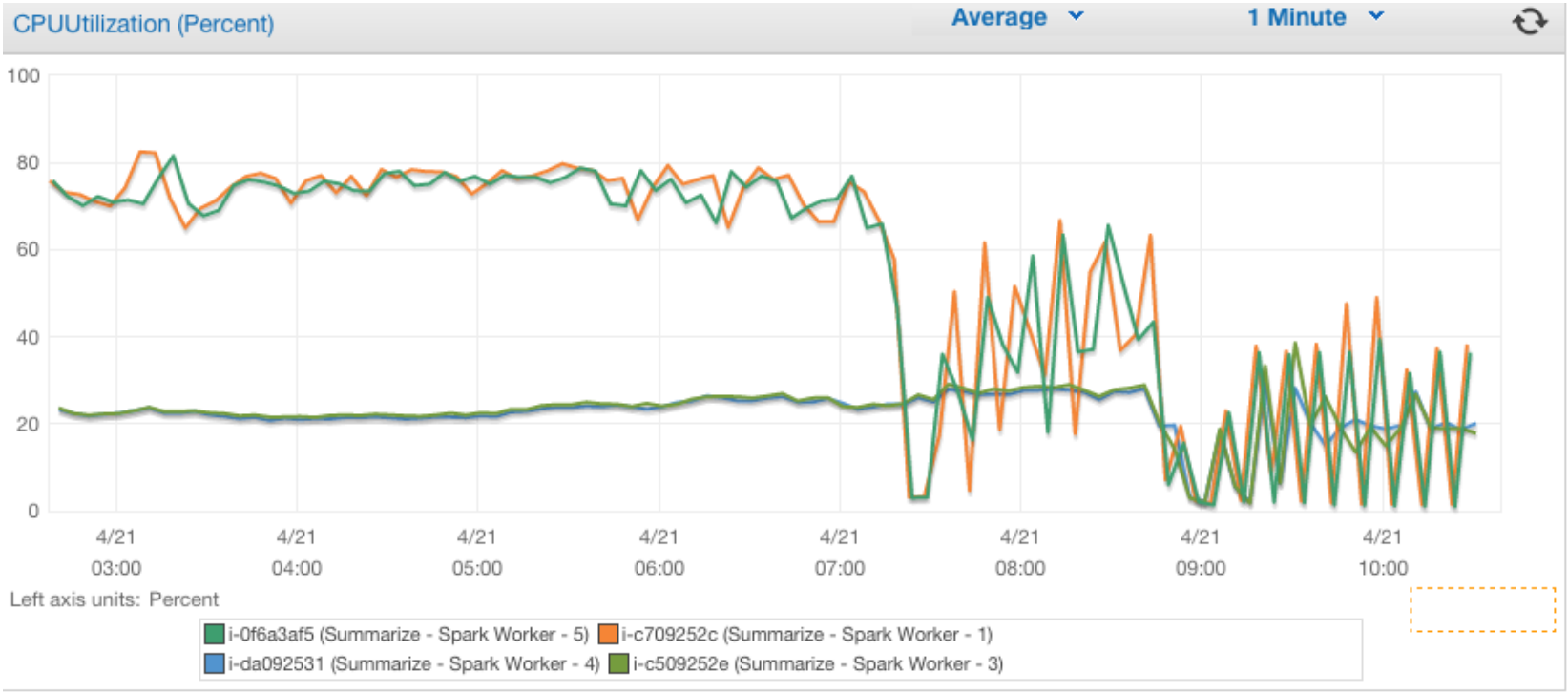
Pro:

- Spark partition 1:1 Kafka topic/partition, cheap parallelism
- No duplicate writes
- No dependency on HDFS
- Access to offsets, can use idempotent or transactional

Con:

- Specific to Kafka
- Need enough Kafka disk (OffsetOutOfRange is *your fault*)

Don’t care about semantics?



How about server cost?

Basic direct stream API

```
val stream: InputDStream[(String, String)] =
 KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder](
    streamingContext,
    // kafka config parameters
    Map(
      "metadata.broker.list" -> "localhost:9092,anotherhost:9092"),
      "auto.offset.reset" -> "largest"
    ),
    // set of topics to consume
    Set("sometopic", "anothertopic")
  )
```

Basic direct stream API semantics

auto.offset.reset -> largest:

- Starts at latest offset, thus losing data
- Not at-most-once (need to set maxFailures as well)

auto.offset.reset -> smallest:

- Starts at earliest offset
- At-least-once, but replays whole log

If you want finer grained control, must store offsets

Where to store offsets

Easy - Spark Checkpoint:

- No need to access offsets, automatically used on restart
- Must use idempotent, not transactional
- Checkpoints may not be recoverable

Complex - Your own data store:

- Must access offsets, save them, and provide on (re)start
- Idempotent or transactional
- Offsets are just as recoverable as your results

Spark checkpoint

```
// Direct copy-paste from the docs, same as any other spark checkpoint
def functionToCreateContext(): StreamingContext = {
  val ssc = new StreamingContext(...)   // new context
  val stream = KafkaUtils.createDirectStream(...) // setup DStream
...
  ssc.checkpoint(checkpointDirectory)   // set checkpoint directory
  ssc
}
// Get StreamingContext from checkpoint data or create a new one
val context = StreamingContext.getOrCreate(
  checkpointDirectory,functionToCreateContext _)
```

Your own data store

```
val stream: InputDStream[Int] =
  KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder,
Int](
    streamingContext,
    Map("metadata.broker.list" -> “localhost:9092,anotherhost:9092"),
    // map of starting offsets, would be read from your storage in real code
    Map(TopicAndPartition(“someTopic”, 0) -> someStartingOffset,
       TopicAndPartition(“anotherTopic”, 0) -> anotherStartingOffset),
    // message handler to get desired value from each message and metadata
    (mmd: MessageAndMetadata[String, String]) => mmd.message.length
  )
```

Accessing offsets, per message

```
val messageHandler =
  (mmd: MessageAndMetadata[String, String]) =>
    (mmd.topic, mmd.partition, mmd.offset, mmd.key, mmd.message)
```

Your message handler has full access to all of the metadata.

Saving offsets per message may not be efficient, though.

Accessing offsets, per batch

```
stream.foreachRDD { rdd =>
  // Cast the rdd to an interface that lets us get an array of OffsetRange
  val offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges
  val results = rdd.someTransformationUsingSparkMethods
  ...
  // Your save method. Note that this runs on the driver
  mySaveBatch(offsetRanges, results)
}
```

Accessing offsets, per partition

```
stream.foreachRDD { rdd =>
  // Cast the rdd to an interface that lets us get an array of OffsetRange
  val offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges
  rdd.foreachPartition { iter =>
    // index to get the correct offset range for the rdd partition we're working on
    val offsetRange: OffsetRange = offsetRanges(TaskContext.get.partitionId)
    val perPartitionResult = iter.someTransformationUsingScalaMethods
    // Your save method.  Note this runs on the executors.
    mySavePartition(offsetRange, perPartitionResult)
  }
}
```

Be aware of partitioning

Safe because KafkaRDD partition is 1:1 with Kafka partition

```
rdd.foreachPartition { iter =>
  val offsetRange: OffsetRange = offsetRanges(TaskContext.get.partitionId)
```

Not safe because there is a shuffle, so no longer 1:1

```
rdd.reduceByKey.foreachPartition { iter =>
  val offsetRange: OffsetRange = offsetRanges(TaskContext.get.partitionId)
```

kixer

Questions?

cody@koeninger.org

https://github.com/koeninger/kafka-exactly-once