Spark streaming

~~Exactly once~~
At least once + idempotence

Quentin Ambard
@qambard

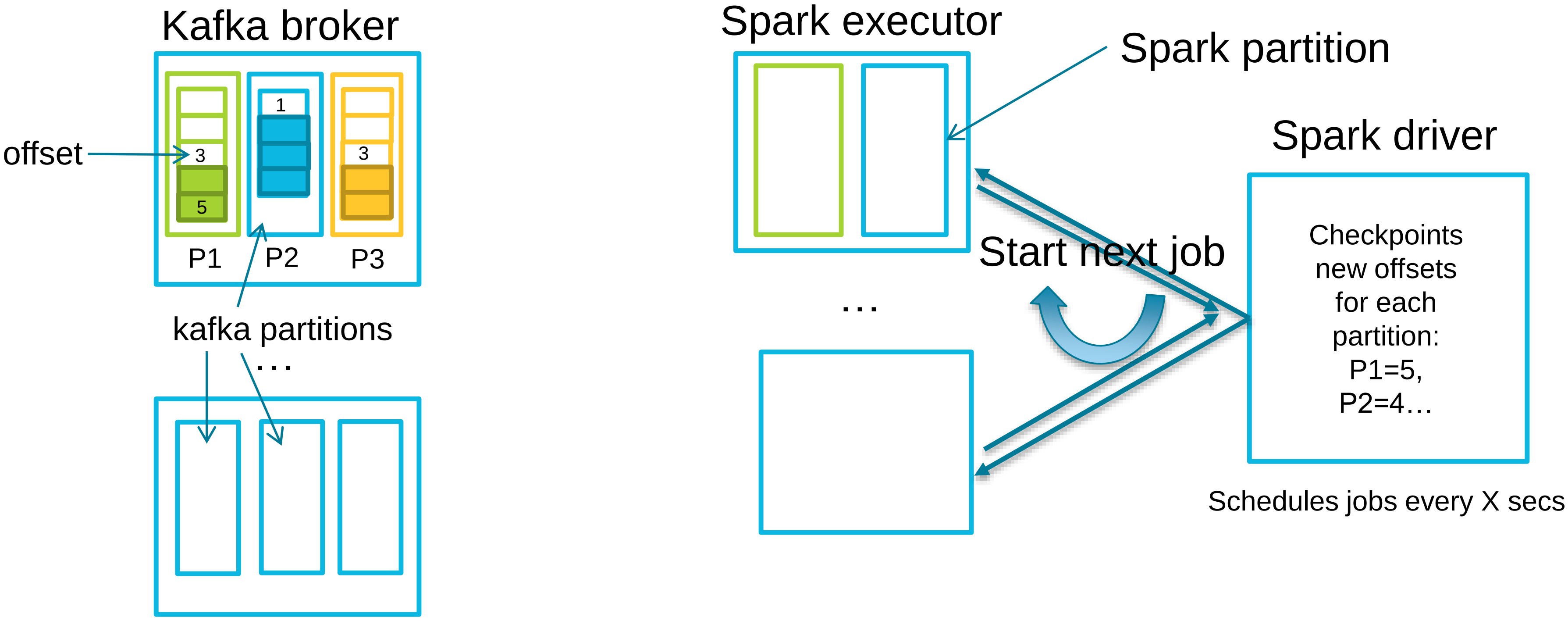
DATASTAX
The power behind the moment.

Agenda

- Native checkpointing
    - Driver & executor failure
- Custom checkpointing
- Performance tips
- Structured streaming (spark 2.2)
- Logs
- Kafka
- Bonus: Kafka 1 & “exactly-once”

Spark & kafka for (near)realtime

Once up a time, Spark met Kafka and…



Spark native checkpointing

Native checkpointing

Spark saves for you the job state to a resilient storage (dsefs) :

- Save metadata, serialize the streaming context with:
    - Spark configuration
    - DStream operations (what your job is doing)
    - Incomplete batches (scheduled but not yet executed)
    - Last kafka offsets

  Checkpoints twice: before and after each micro-batch (blocking)
  Enabled with `ssc.checkpoint(dir)`

- Save the actual RDD data.
  Should be enabled for stateful streams only (sliding window etc)
  Expensive, enabled with `kafkaStream.checkpoint(Seconds(10))`

Checkpointing: example

```
val dir = "dsefs://35.188.108.191:5598/exactlyonce“

def createStreamingCtx() = {
   val ssc = new StreamingContext(conf, Seconds(5))
   val stream = KafkaUtils.createDirectStream(ssc, kafkaParams, topics)
   //Apply transformations on each microbatch
   stream.foreachRDD(rdd => println(rdd.count()))
   //Set checkpointing directory
   ssc.checkpoint(dir)
   ssc
 }
 val ssc = StreamingContext.getOrCreate(dir, createStreamingCtx)
```

Checkpointing: restart behavior

- Read data saved in the checkpointing directory

- Schedule all missing job:
  1 hour outage with a 1 seconds window will generate 60*60 micro-batches (!).

- First micro-batch starts from the last saved offsets

- First micro-batch will read all available kafka messages
  Can be limited with
  - **spark.streaming.kafka.maxRatePerPartition = XXX**
  - **spark.streaming.backpressure.enabled = true**

Checkpointing: serialization

Serialize Dstream to disk

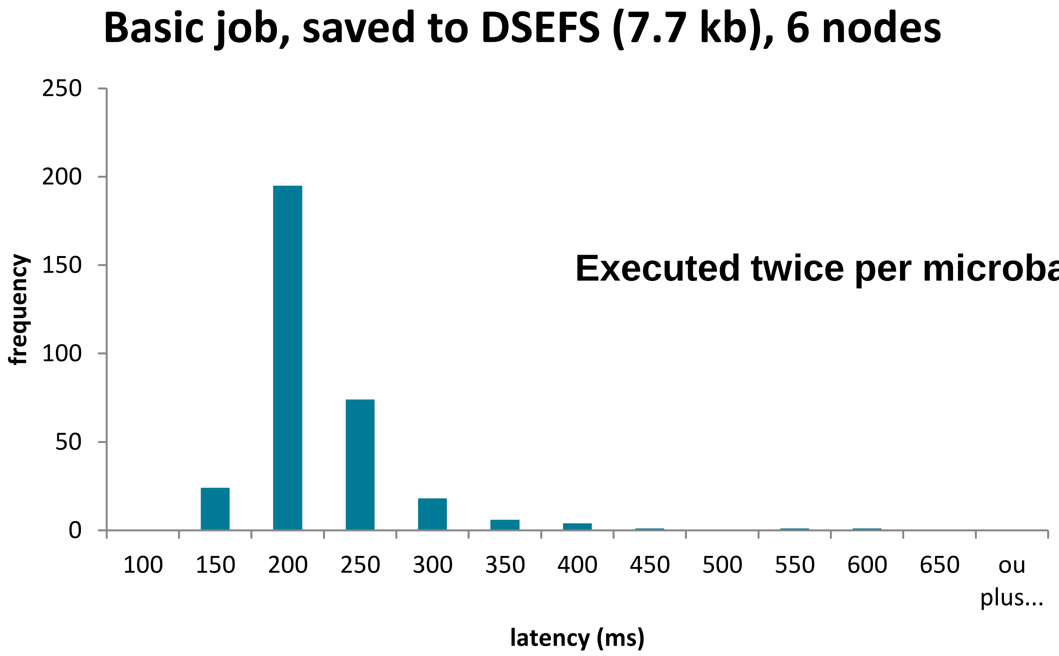
- Need to be serializable
- Won’t be able to restart a job if the Dstream operations have changed (classes are differents)

This problem alone discard native checkpointing for a lot of customers

Workaround:

- Run the job twice in parallel during upgrade (can be an issue if the order matters)
- “Clean stop”, restart from the beginning of the kafka topic

Checkpointing: speed

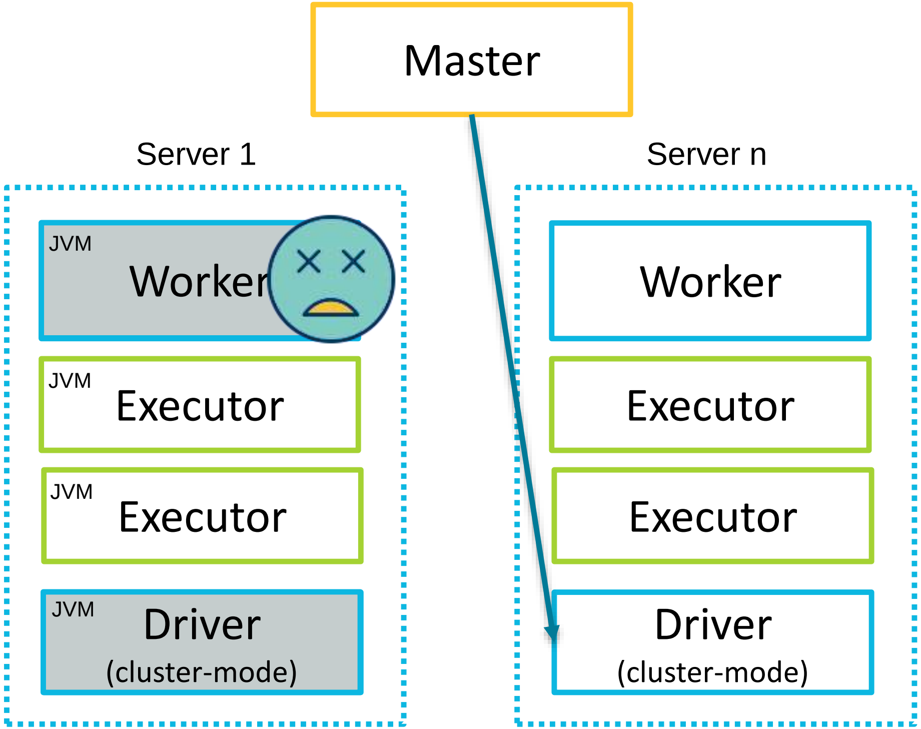


Executed twice per microbatch!

Failure handling

Worker failure

Scenario: the worker running the driver get killed



- Master will restart the driver on another worker
- Driver will stop… Sometime… (watches its worker)

- 2 spark apps running at the same time for a few sec
- Can break order
- Same checkpointing dir

Worker failure

Scenario: worker running the driver get killed

Worker failure

Workaround:

Make sure it’s the only instance running before starting the stream

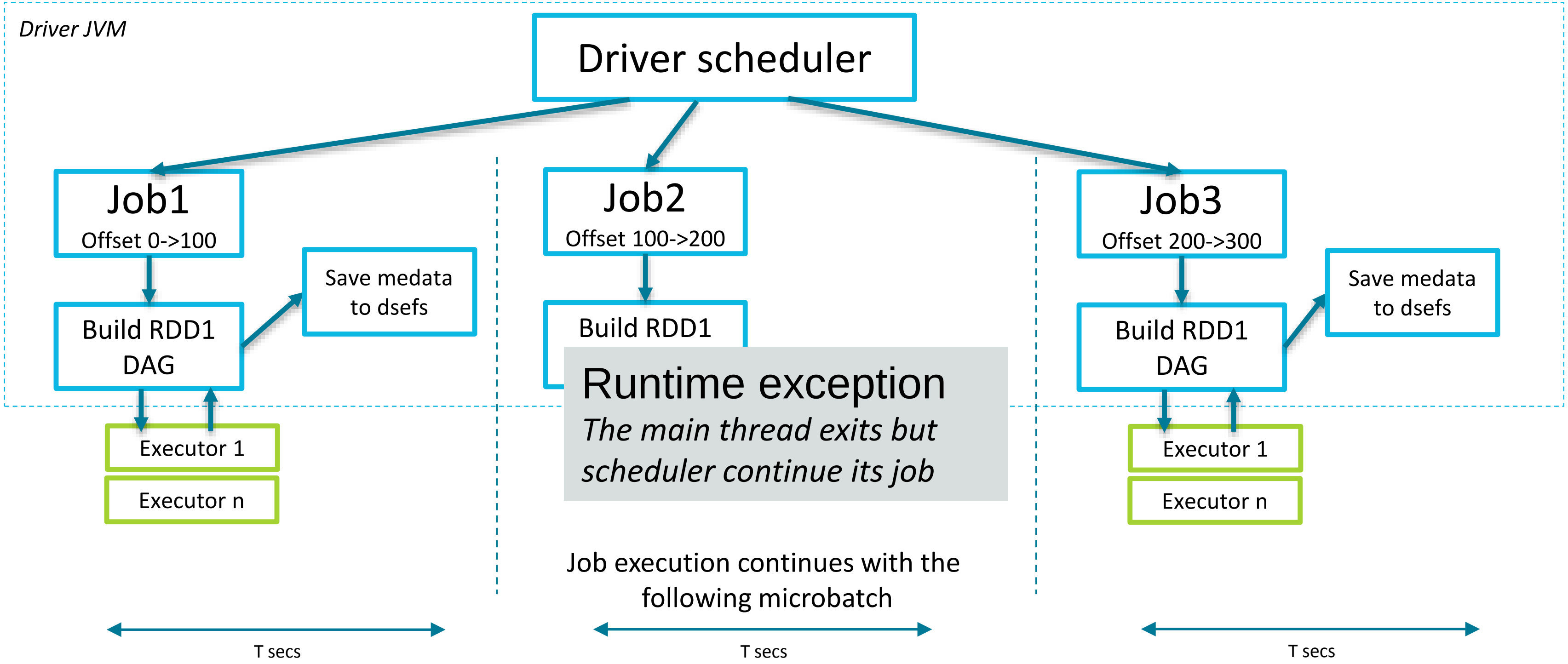
curl http://35.188.90.197:7080/api/v1/applications

or cql secret api ?

Checkpointing: failure in the driver

```
kafkaStream.foreachRDD(rdd => {
   i += 1
   println(s"Processing RDD $i")
   if (i == 3) {
       throw new RuntimeException(s"Exception for RDD $i")
     }
    val count = rdd.map(_._2+ "computed ").count()
    println(s"RDD $i completed : $count")
   })
```

Checkpointing: failure in the driver



Runtime exception

The main thread exits but scheduler continue its job

Job execution continues with the following microbatch

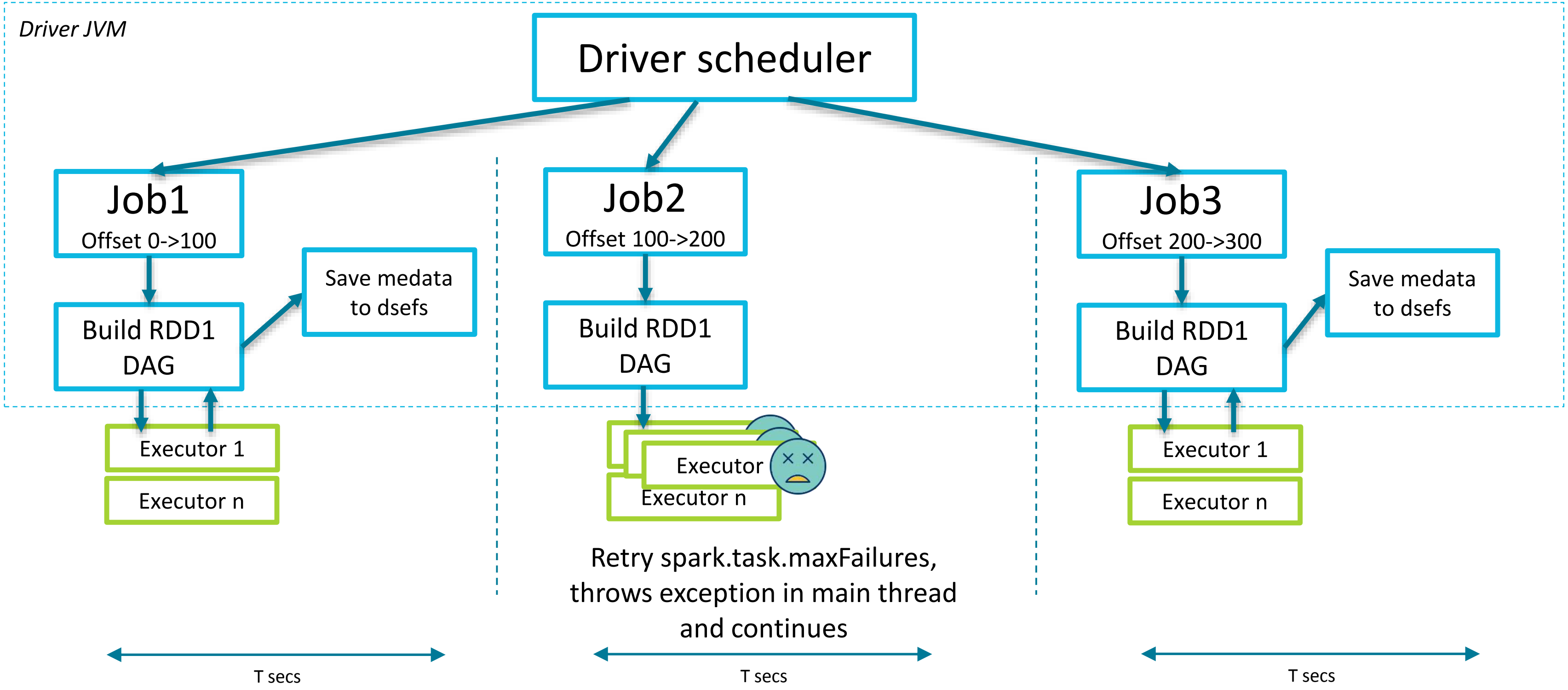
Data from Job #2 (offset 100->200) is lost and won’t be recovered (erased by the next checkpointing)

Checkpointing: failure in an executor

Inside
an executor

```
kafkaStream.foreachRDD(rdd => {
   i += 1
   println(s"Processing RDD $i")
   val count = rdd.map(row => {
       row._2+ "computed "
       if (i == 3) {
          throw new RuntimeException(s"Exception for RDD $i")
     }}).count()
    println(s"RDD $i completed : $count")
   })
```

Checkpointing: failure in the driver



Data from Job #2 (offset 100->200) is lost and won't be recovered (erased by the next checkpointing)

Checkpointing: fail-fast

Need a solution to stop job execution as soon as a micro-batch fail

https://issues.apache.org/jira/browse/SPARK-6415

Workaround:

- Catch the error and kill the driver asap ?

Checkpointing: fail-fast

```
ssc.start()
Try {
  ssc.awaitTerminati
 } match {
  case Success(result)
    println("StreamingCo                een stopped")
  case Failure(excep
    println("Stopping     an        river...", exception)
    ssc.stop(stopSpark    ntext = t     stopGracefully = false)
    System.exit(50) //sparkExitCode.UNCAUGHT_EXCEPTION
 }
```

Checkpointing: fail-fast

- Streaming context waits 2 seconds before stopping (with graceful = false).
- Scheduler continues its job and launch the next batch
- Shutdown hook to stop context on System.exit()
- Managed to reproduce with a Runtime.getRuntime.halt()

- Not guarantee that the next microbatch won't be checkpointed

Conclusion

Checkpointing: conclusion

- ✓ Easy to setup (yeee)

- ✗ Reschedule all missed jobs during startup
  Will likely kill your driver if it was stopped for a few days :/

- ✗ Code/spark upgrade non-trivial (serialization issue)
- ✗ Can’t easily monitor Kafka offset consumption
- ✗ Slow
- ✗ Lot of corner-cases
- ✗ **Couldn’t find a way to make it 100% reliable**

Custom checkpointing

Read kafka offset from rdd

Stored with RDD metadata

```
val offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges

offsetRanges.foreach{range =>
    println(s"OFFSET RANGES for partition :${range.partition} =
[${range.fromOffset}-${range.untilOffset}]")
}
//OFFSET RANGES for partition :1 = [0-10]
//OFFSET RANGES for partition :2 = [0-13]
…
```

Store kafka offset

Store data in C* yourself instead of hdfs

```
def saveOffsetRanges (topic, ranges) = {
    val rangesJson = ranges.sortBy(_.partition).toJson()
    session.execute("insert into ks.checkpoint (topic, offsets) values (
${topic}, ${rangesJson} ")
}
```

Doesn’t handle failures better than native checkpointing

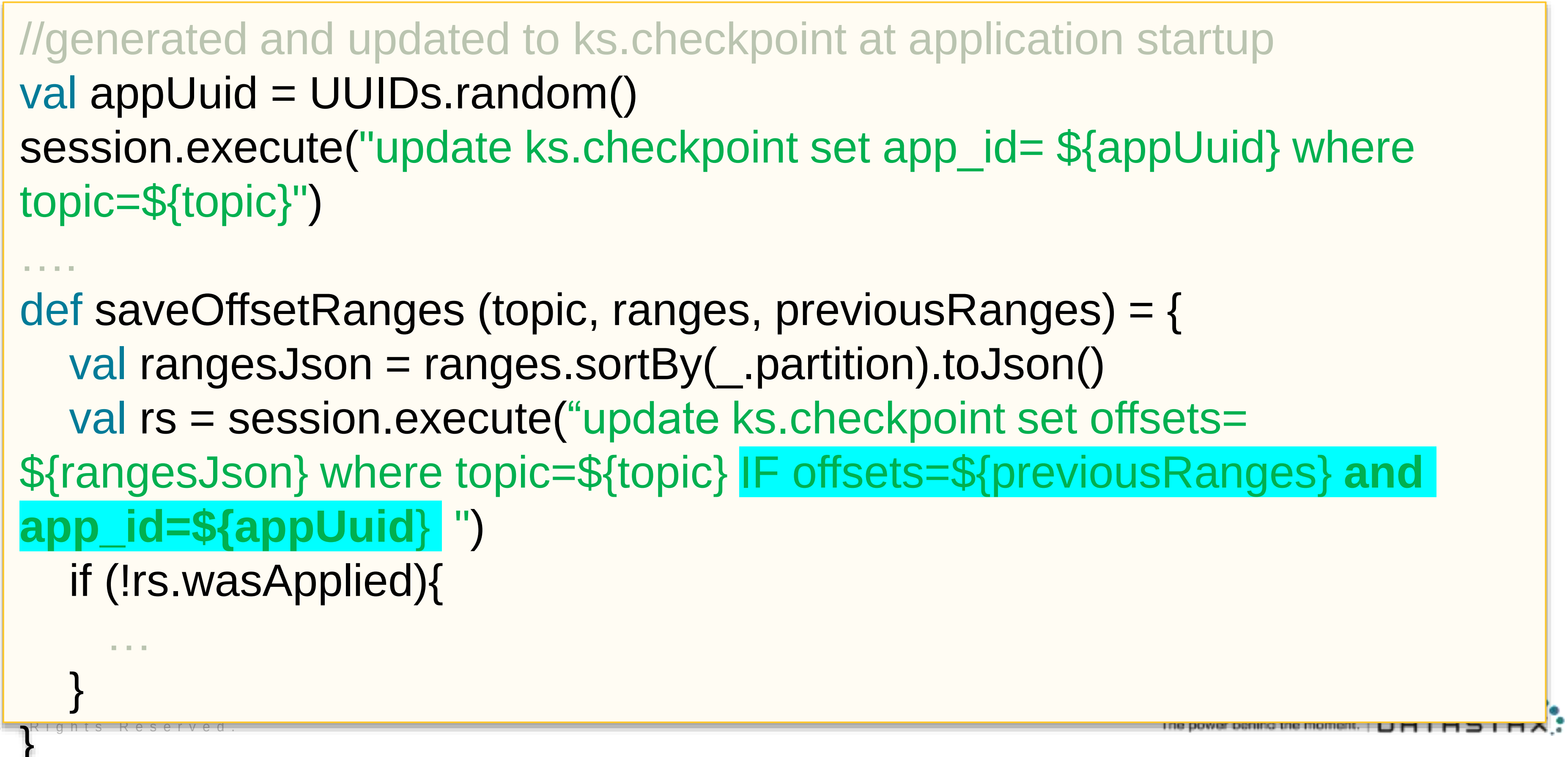
Need to make sure the next batch won’t checkpoint until the previous is actually saved

Store kafka offset

Use a LWT to guarantee exactly-once

```
def saveOffsetRanges (topic, ranges, previousRanges) = {
    val rangesJson = ranges.sortBy(_.partition).toJson()
    val rs = session.execute(" update ks.checkpoint set offsets=
${rangesJson} where topic=${topic} IF offsets=${previousRanges} ")
    if (!rs.wasApplied){
        log.error("last offsets doesn’t match. Killing driver. ")
        System.exit(50)
    }
    rangesJson
}
```

Ensure application uniqueness



```
//generated and updated to ks.checkpoint at application startup
val appUuid = UUIDs.random()
session.execute("update ks.checkpoint set app_id= ${appUuid} where 
topic=${topic}")
….
def saveOffsetRanges (topic, ranges, previousRanges) = {
    val rangesJson = ranges.sortBy(_.partition).toJson()
    val rs = session.execute("update ks.checkpoint set offsets= 
${rangesJson} where topic=${topic} IF offsets=${previousRanges} and 
app_id=${appUuid}  ")
    if (!rs.wasApplied){
        …
    }
}
```

When should we save offsets ?

Always after the executor work

```
kafkaStream.transform(rdd => {
    rdd.doSomething…
}).foreachRDD(rdd => {
    rdd.doSomething(…)
    val offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges
    previousRanges = saveOffsetRanges(offsetRanges)
})
```

If you need to save the state at the beginning

```
kafkaStream.transform(rdd => {
    //executed by job scheduler (LWT will fail with the first lag)
    saveInitialState(rdd, topic, previousRange)
}).foreachRDD(rdd => {
     rdd.doSomething…
     previousRanges = saveOffsetRanges (rdd, topic, previousRange)
})
```

If you need to save the state at the beginning

✓

```
kafkaStream.foreachRDD(rdd => {
    //Must be protected by LWT
    saveInitialState(rdd, topic, previousRange)
    rdd.doSomething…
    previousRanges = saveOffsetRanges (rdd, topic, previousRange)
})
```

Custom checkpointing: conclusion

- ✓ Reliable
- ✓ Kafka offset consumption easy to monitor
- ✓ Easy upgrade
- ✓ Fast

- ✗ Harder to setup (Need to use LWT)

Performances tips

Markus Stöber

Performances

Very easy to get dead-slow throughput with complex jobs
(15 messages/sec for a solid 3 nodes cluster!)

The usual recommendations apply:

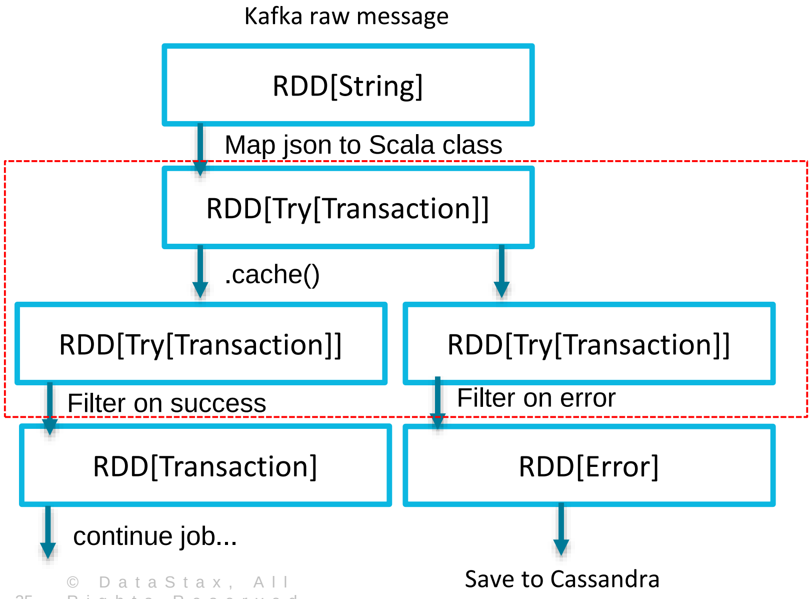
- Data distribution
- Avoid shuffle
- Prefer DataSet
- Use reducer instead of groupBy when possible…

Many kafka partitions with small process time?

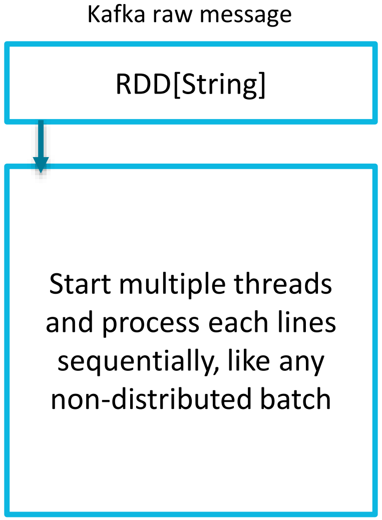
- rdd.coalesce(numberOfExecutor)
- Disable spark.locality.wait

Performances

- Doing a lot of .cache() with multiple actions will hurt
- It’s often easier and faster to parallelize synchronous computation in your executors



200ms: fast for long batch, slow for a 3 seconds micro-batch


Kafka raw message
RDD[String]
Start multiple threads and process each lines sequentially, like any non-distributed batch

http://www.russellspitzer.com/2017/02/27/Concurrency-In-Spark/

Performances

Monitor the driver and executor JVM !!

Typical scenario on driver:

- microbatches get slow for some reason
- Scheduler schedules many batches, add pressure on the heap
- Many GC, even slower, more lag, more GC, more lag, more GC…

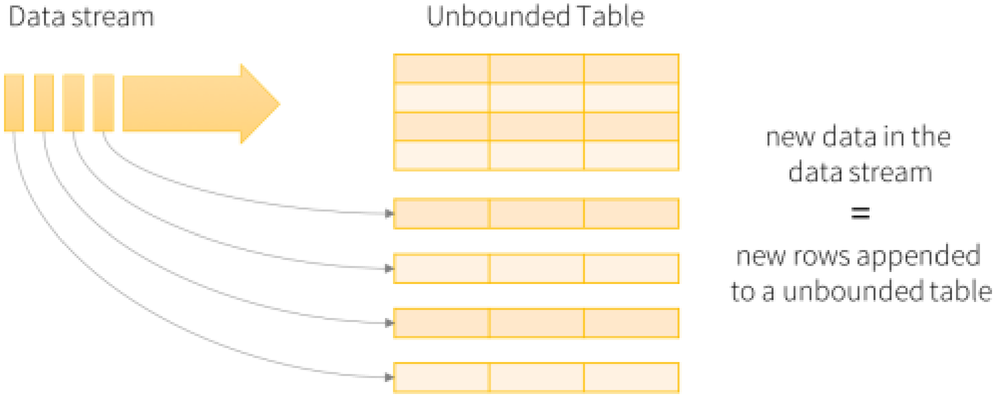
On executors:

- By default, heap at 1GB
- “Only” (1024MB-300MB)*0.25 = 180MB for your objects!
  (the rest is for spark storage and execution)

Structured streaming (2.2)

Unified API

- Uses DataSet instead of RDD
- Better support for aggregations
- Based on message timestamp
- Support “late data” (watermark)
- Stop/start queries
- …


Data stream
Unbounded Table
new data in the data stream
=
new rows appended to a unbounded table

Data stream as an unbounded table

Example

```
val ds = sparkSession.readStream
    .format("kafka")
    .option("kafka.bootstrap.servers", "localhost:9092")
    .option("subscribe", “myTopic")
    .load()
    .select(from_json($"value", schema).as("json"))
    .select("json.*").as[Transaction]

val query = ds.writeStream
    .cassandraFormat(“transaction“, ks)
    .outputMode(OutputMode.Update)
    .start()
```

How checkpointing works

- Checkpoint data to resilient storage (DSEFS, HDFS etc.)
- Once after each microbatch
- Conceptually similar to custom checkpointing
- Data saved as JSON

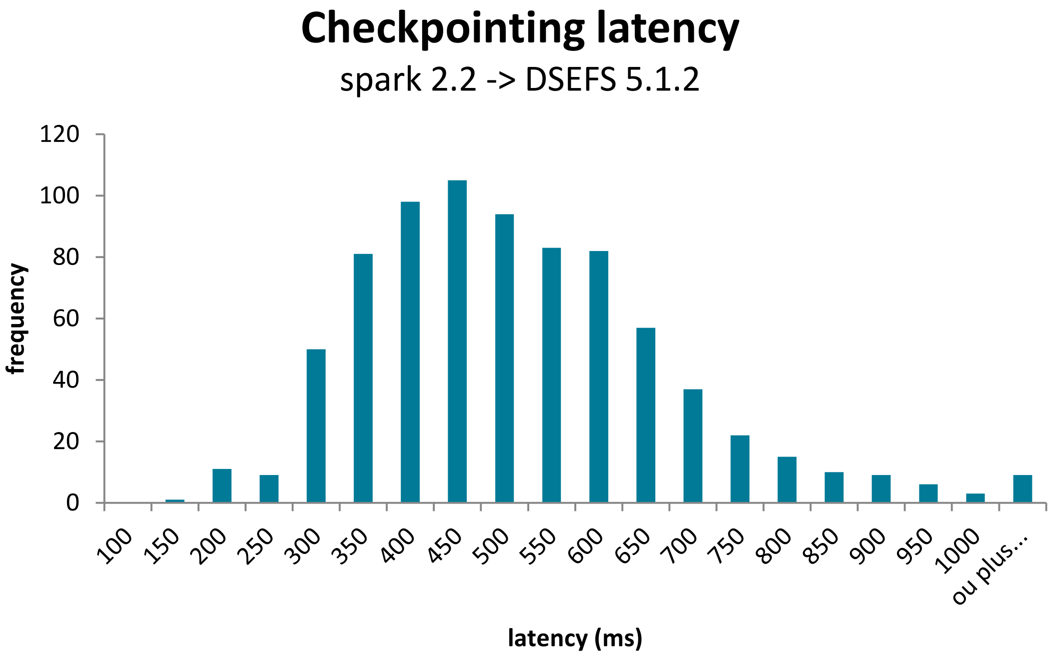
{"batchWatermarkMs":0,"batchTimestampMs":1502995346109,"conf":{"spark.sql.shuffle.partitions":"200"}}
{"exactlyonce":{"0":32625}}

Failures

- Fails fast and safely!
- Catch exception and safely restart the query without killing the driver

- Still need to ensure the same query isn't running twice

Checkpointing speed



DSEFS operations per microbatch: 9 read, 2 creation, 2 rename, 2 deletes

Speed & Triggers

- Equivalent to window size
- 0 by default (= proceed asap)

```
ds.writeStream
 .trigger(Trigger.ProcessingTime(“1 seconds"))
 .queryName("exactlyOnce").foreach(writer).start
```

- Increase throughput with a higher maxOffsetsPerTrigger

timeToProcess(maxOffsetsPerTrigger) >> checkpointing time
(otherwise spend too much time checkpointing)

Faster checkpointing?

Implement a custom FileSystem to store on C* only (instead of DSEFS)

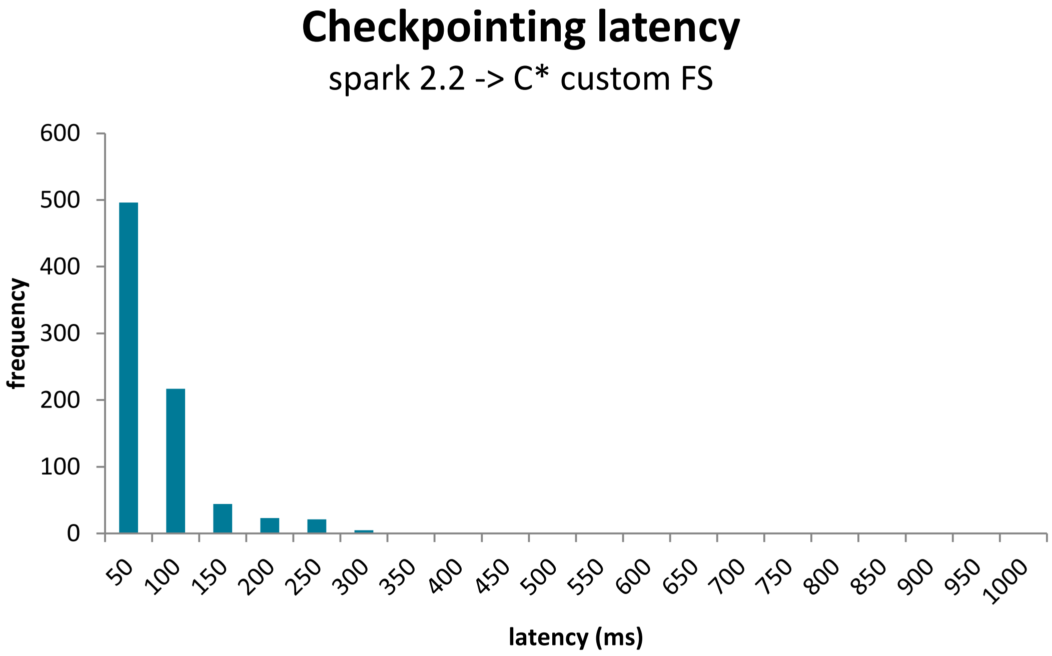
```
class CassandraSimpleFileSystem extends hadoop.fs.FileSystem {
  override def mkdirs(f: Path, permission: FsPermission): Boolean = ???
  …
```

```
SparkSession.builder.config("spark.hadoop.fs.ckfs.impl",
"exactlyonce.CassandraSimpleFileSystem")

ds.writeStream.option("checkpointLocation",
"ckfs://127.0.0.1/checkpointing/exactlyOnce")
.queryName("exactlyOnce").foreach(writer).start
```

https://github.com/QuentinAmbard/cassandracheckointingfs/

Custom checkpointing



Structured streaming: conclusion

- ✓ Reliable
- ✓ Fast
- ✓ DataSet & powerful time-based aggregation (support for late data)
- ✓ Easy upgrade (as long as aggregation type doesn’t change)
- ✓ It’s the future!

- ✗ Currently slow with DSEFS
- ✗ Old (blocked) data could be discarded if using producer time aggregation.
  Use custom sink to reproduce previous streaming behavior

Logs

Logs

spark-env.sh

```
export SPARK_WORKER_OPTS="$SPARK_WORKER_OPTS -
Dspark.worker.cleanup.enabled=true -
Dspark.worker.cleanup.interval=300 -
Dspark.worker.cleanup.appDataTtl=300"
```

Cleanup non-running application only

Won’t help for long-living streaming job

Logs

Limit the executor log size:

(SPARK_WORKER_DIR/worker-n/application_id/executor_id/…)

spark-defaults.conf

```
spark.executor.logs.rolling.maxRetainedFiles 3
spark.executor.logs.rolling.strategy size
spark.executor.logs.rolling.maxSize 50000
```

Logs

- If an executor jvm crashes, a new executor is started. Creates new folder with a copy of the application .jar

- Crashed executors folder won’t be cleanup.

- Good practice to monitor the size of spark worker directory

(but your executors shouldn’t OOM too often right?)

Kafka

Kafka durability

- Disable unclean leader election (now default)
- Define in-synch / RF (ex: RF=3, insynch=2)
- Publish at ALL (ack from all insynch)
- If order matters, allow only 1 inflight request (or 0 retries)

```
unclean.leader.election.enable = false
default.replication.factor = 3
min.insync.replicas=2
#producer
max.in.flight.requests.per.connection = 1
retries=3
acks = all
#consumer (disable autocommit for direct stream)
enable.auto.commit = false
```

How many kafka partitions ?

- Lot of partition will slow things down
- ~3x your number of spark cores is a good start
- Must allow to scale if needed
  - If order matters, changing changing kafka partitioning can be hard

Kafka 11: idempotent producer

- Idempotent producer:
    - Generate a unique id **per connection/producer**

      Your executors might restart with different producers id!
    - Retry an unlimited number of time
    - Broker de-duplicate based on the message id

Kafka 11: Transactions

```
props.put("transactional.id", “myId")
…
producer.initTransactions()

producer.beginTransaction()
kafka.producer.send(myProducerRecord1)
kafka.producer.send(myProducerRecord2)
kafka.producer.commitTransaction()
```

```
//Consumer configuration
props.put("isolation.level", "read_committed")
```

Kafka 11: Transactions

- Consumer blocked to the last stable offset (LSO)
  - 1 transaction stalled => LSO blocked => consumer blocked

- Transaction.max.timeout.ms set to 900000 (15min) by default

- “Exactly-once streams” based on transactions

Thank you