SPARK SUMMIT

SPARK STREAMING PROGRAMMING TECHNIQUES YOU SHOULD KNOW

Gerard Maas Lightbend

#EUstr2

Gerard Maas

Sr SW Engineer

Lightbend

@maasg

https://github.com/maasg

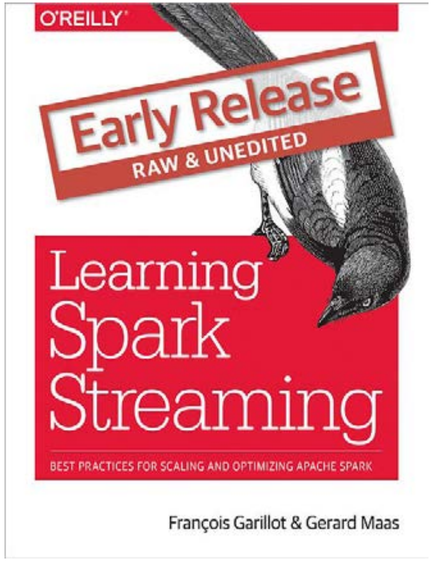
https://www.linkedin.com/in/gerardmaas/

https://stackoverflow.com/users/764040/maasg

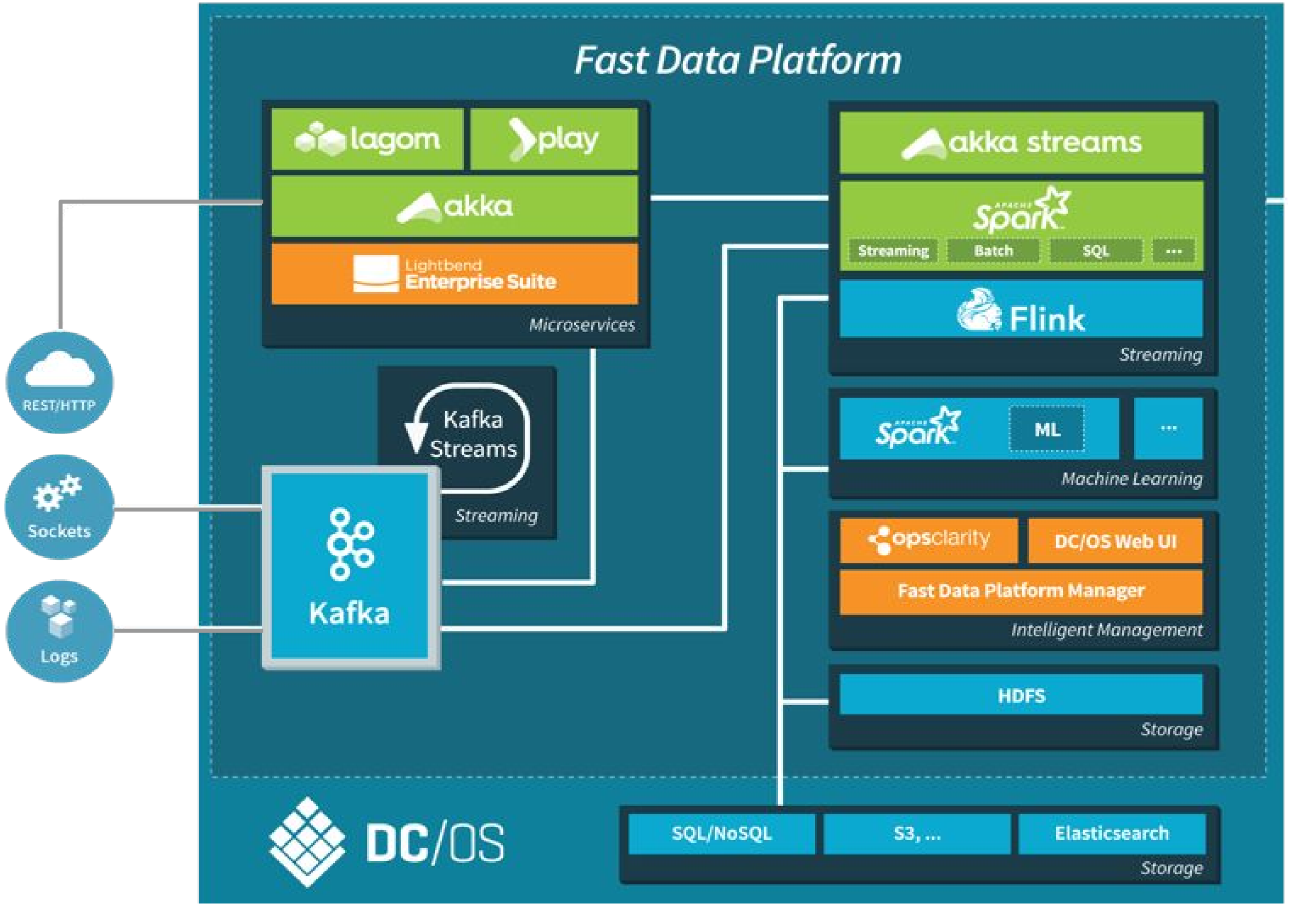
Computer Engineer
Scala Programmer
Early Spark Adopter (v0.9)
Spark Notebook Contributor

Cassandra MVP (2015, 2016)
Stack Overflow Top Contributor
(Spark, Spark Streaming, Scala)

Wannabe {
  IoT Maker
  Drone crasher/tinkerer
}


O'REILLY
Early Release
RAW & UNEDITED
Learning Spark Streaming
BEST PRACTICES FOR SCALING AND OPTIMIZING APACHE SPARK
François Garillot & Gerard Maas

Lightbend Fast Data Platform V1.0



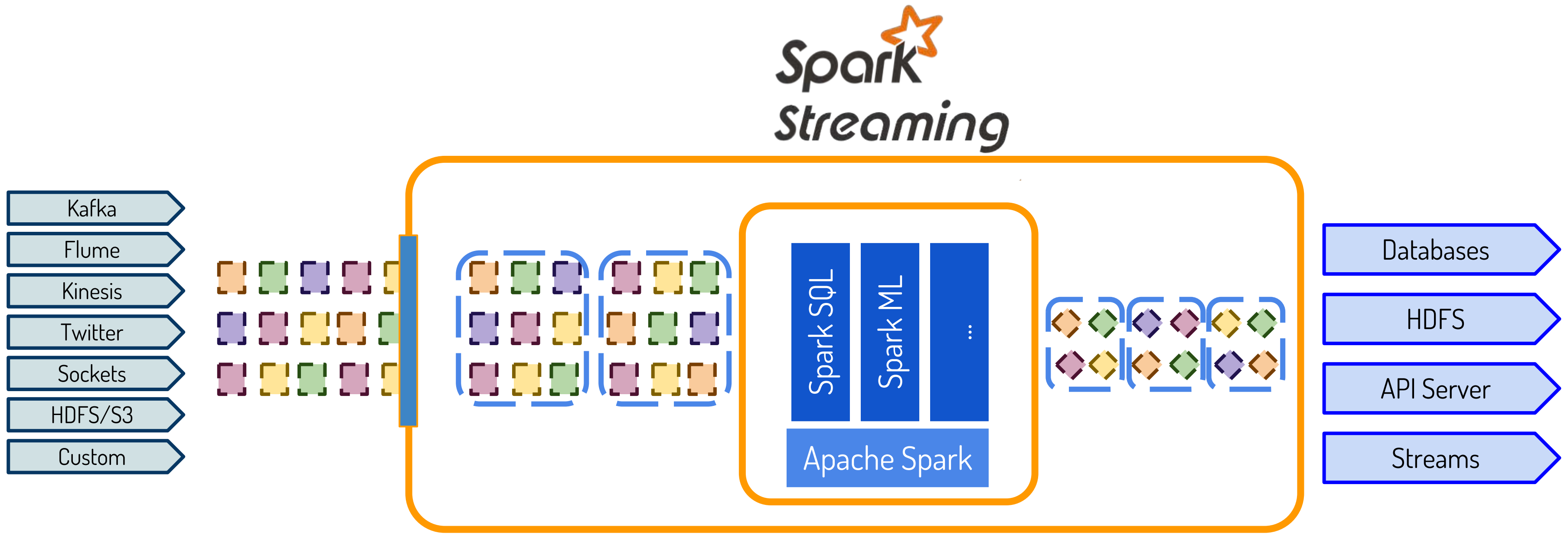
lightbend.com/fast-data-platform

Lightbend

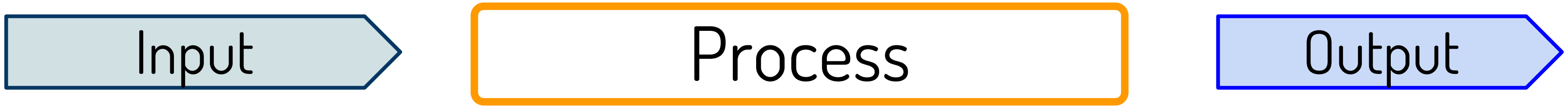
Agenda

- Spark Streaming Refresher
  - Model
  - Operations
- Techniques
  - Self-contained stream generation
  - Refreshing external data
  - Structured Streaming compatibility
  - Keeping arbitrary state
  - Probabilistic accumulators

Spark Streaming Refresher


Spark
Streaming
Kafka
Flume
Kinesis
Twitter
Sockets
HDFS/S3
Custom
Spark SQL
Spark ML
...
Apache Spark
Databases
HDFS
API Server
Streams

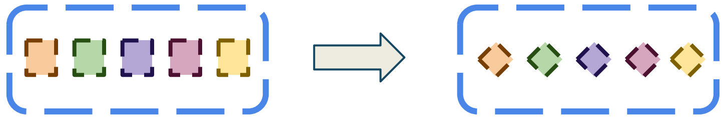
API



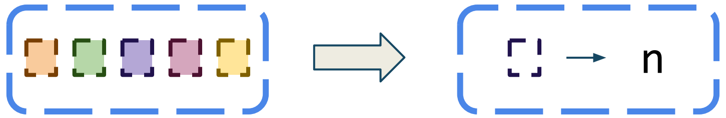
Input	Process	Output
DStream	Transformations	Output Operations

Transformations

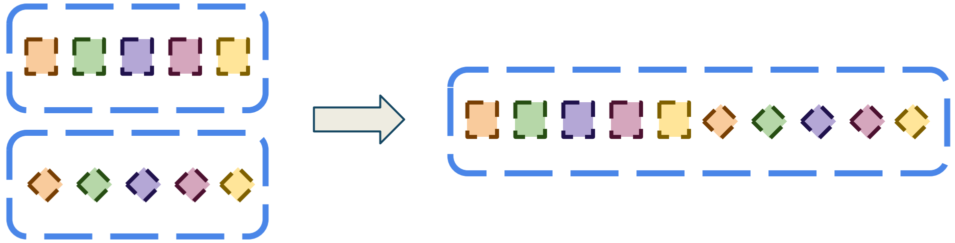
map,
flatmap,
filter



count,
reduce,
countByValue,
reduceByKey

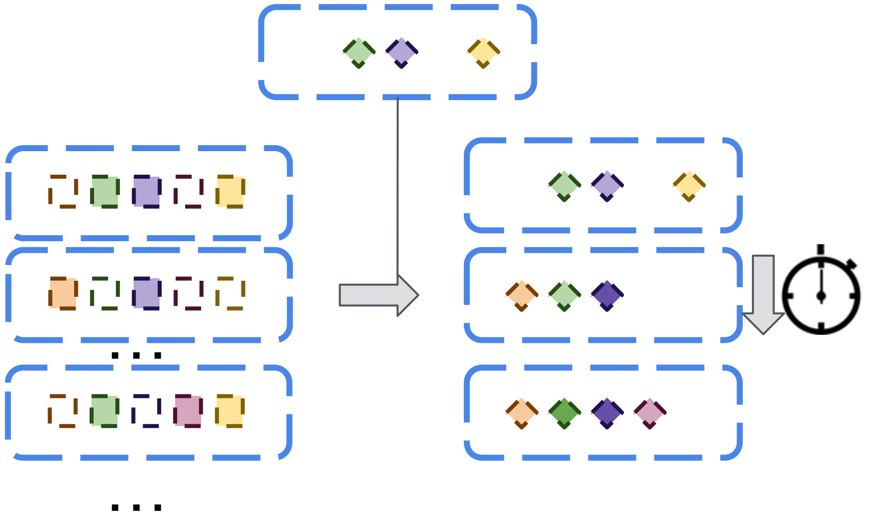


union,
join
cogroup



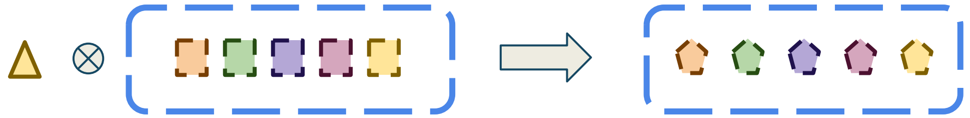
Transformations

mapWithState



Transformations

transform



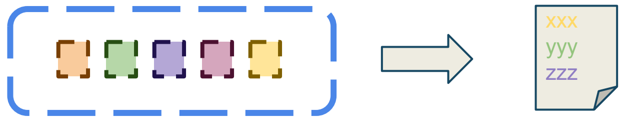
```
val iotDstream = MQTTUtils.createStream(...)
val devicePriority = sparkContext.cassandraTable(...)
val prioritizedDStream = iotDstream.transform{rdd =>
     rdd.map(d => (d.id, d)).join(devicePriority)
}
```

Actions

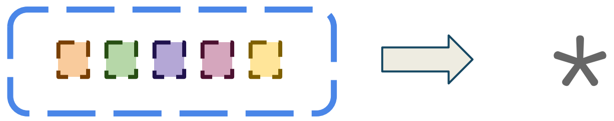
print

```
-------------------------------------------
Time: 1459875469000 ms
-------------------------------------------

data1

data2
```

saveAsTextFiles,
saveAsObjectFiles,
saveAsHadoopFiles



foreachRDD



Actions

```
def print(num: Int): Unit = ssc.withScope {
  def foreachFunc: (RDD[T], Time) => Unit = {
    (rdd: RDD[T], time: Time) => {
      val firstNum = rdd.take(num + 1)
      // scalastyle:off println
      println("-------------------------------------------")
      println(s"Time: $time")
      println("-------------------------------------------")
      firstNum.take(num).foreach(println)
      if (firstNum.length > num) println("...")
      println()
      // scalastyle:on println
    }
  }
  foreachRDD(context.sparkContext.clean(foreachFunc), displayInnerRDDOps = false)
}
```

Actions - foreachRDD

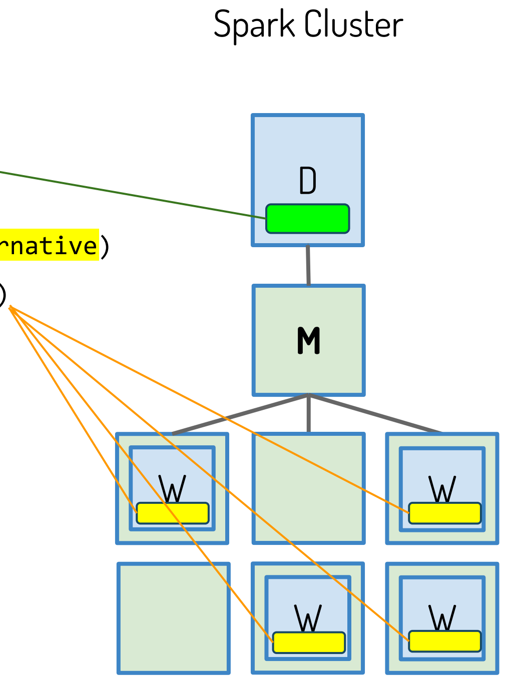
```
dstream.foreachRDD{rdd =>
      rdd.cache()
      val alternatives = restServer.get("/v1/alternatives").toSet
      alternatives.foreach{alternative =>
          val byAlternative = rdd.filter(element => element.kind == alternative)
          val asRecords = byAlternative.map(element => asRecord(element))
          asRecords.foreachPartition{partition =>
              val conn = DB.connect(server)
              partition.foreach(element => conn.insert(element))
          }
      }
      rdd.unpersist(true)
}
```

Executes **local** on the Driver

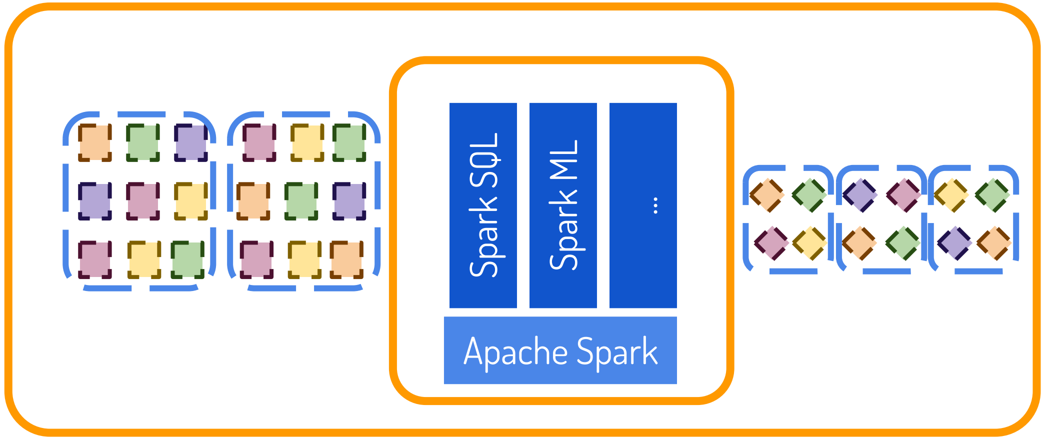
Executes **distributed** on the Workers

Actions – foreachRDD

```
dstream.foreachRDD{rdd =>
      rdd.cache()
      val alternatives = restServer.get(“/v1/alternatives”).toSet
      alternatives.foreach{alternative =>
          val byAlternative = rdd.filter(element => element.kind == alternative)
          val asRecords = byAlternative.map(element => asRecord(element))
          asRecords.foreachPartition{partition =>
              val conn = DB.connect(server)
              partition.foreach(element => conn.insert(element)
          }
      }
      rdd.unpersist(true)

}
```



Ready to Dive in?



Self Contained Stream Generation

ConstantInputDStream

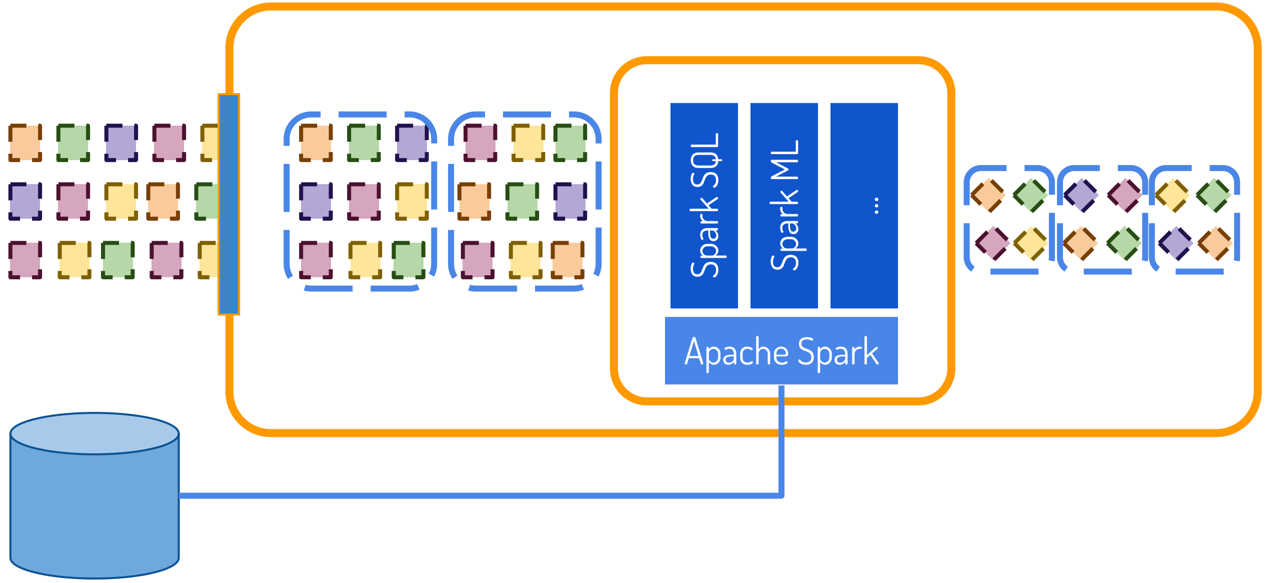
```
/**
* An input stream that always returns the same RDD on each time step. Useful for testing.
*/
class ConstantInputDStream[T: ClassTag](_ssc: StreamingContext, rdd: RDD[T])


// Usage
val constantDStream  = new ConstantInputDStream(streamingContext, rdd)
```

ConstantInputDStream: Generate Data

```scala
import scala.util.Random
val sensorId: () => Int = () =>  Random.nextInt(sensorCount)
val data: () => Double = () => Random.nextDouble
val timestamp: () => Long = () => System.currentTimeMillis
// Generates records with Random data
val recordFunction: () => String = { () =>
   if (Random.nextDouble < 0.9) {
     Seq(sensorId().toString, timestamp(), data()).mkString(",")
   } else {
     // simulate 10% crap data as well… real world streams are seldom clean
     "!!~corrupt~^&##$"
   }
}
val sensorDataGenerator = sparkContext.parallelize(1 to n).map(_ => recordFunction)
```

$\}$ `RDD[() => Record]`

```scala
val sensorData = sensorDataGenerator.map(recordFun => recordFun())

val rawDStream  = new ConstantInputDStream(streamingContext, sensorData)
```



Stream Enrichment with External Data

ConstantInputDStream + foreachRDD= Reload External Data Periodically

```
var sensorReference = sparkSession.read.parquet(s"$referenceFile")
sensorRef.cache()

val refreshDStream  = new ConstantInputDStream(streamingContext, sparkContext.emptyRDD[Int])
// Refresh data every 5 minutes
val refreshIntervalDStream = refreshDStream.window(Seconds(300), Seconds(300))
refreshIntervalDStream.foreachRDD{ _ =>
 sensorRef.unpersist(false)
 sensorRef = sparkSession.read.parquet(s"$referenceFile")
 sensorRef.cache()
}
```

DStream + foreachRDD= Reload External Data with a Trigger

```
var sensorReference = sparkSession.read.parquet(s"$referenceFile")
sensorRef.cache()

val triggerRefreshDStream: DStream  = // create a DStream from a source. e.g. Kafka

val referenceStream = triggerRefreshDStream.transform { rdd =>
   if (rdd.take(1) == “refreshNow”) {
     sensorRef.unpersist(false)
     sensorRef = sparkSession.read.parquet(s"$referenceFile")
     sensorRef.cache()
  }
  sensorRef.rdd
}
incomingStream.join(referenceStream) ...
```

Spark Streaming

Spark Structured Streaming

ForeachRDD + Datasets + Functional = Structured Streaming Portability

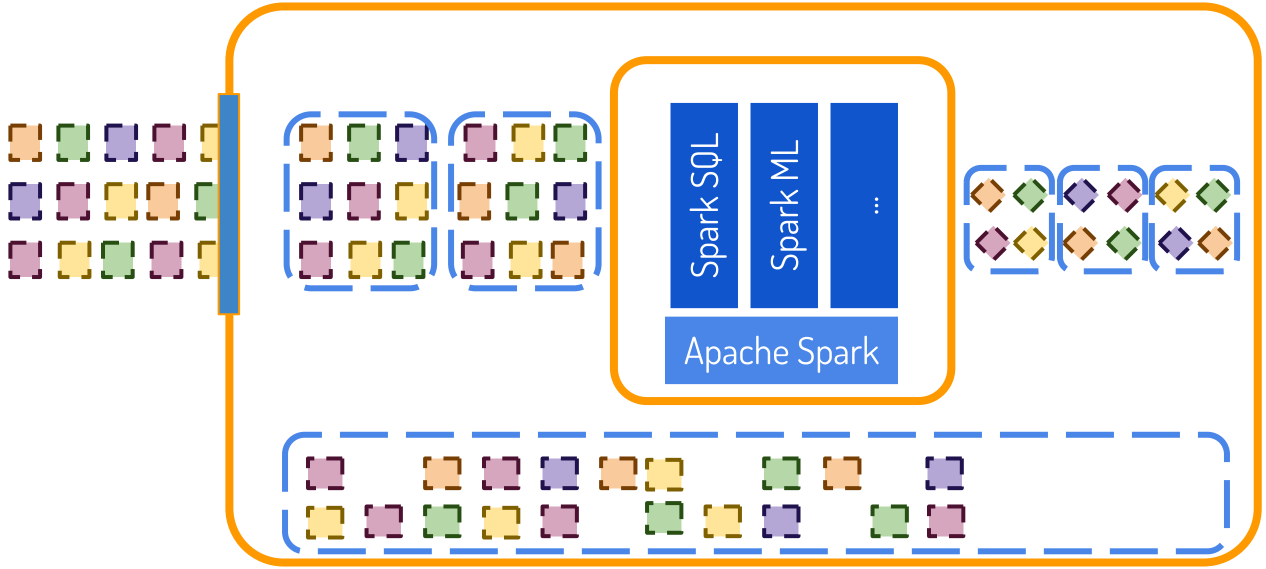
```
val parse: Dataset[String] => Dataset[Record] = ???
val process: Dataset[Record] => Dataset[Result] = ???
val serialize: Dataset[Result] => Dataset[String] = ???
```

Spark Streaming

```
val dstream = KafkaUtils.createDirectStream(...)
dstream.map{rdd =>
   val ds = sparkSession.createDataset(rdd)
   val f = parse andThen process andThen serialize
   val result = f(ds)
   result.write.format("kafka")
          .option("kafka.bootstrap.servers", bootstrapServers)
          .option("topic", writeTopic)
          .option("checkpointLocation", checkpointLocation)
          .save()
}
```

Structured Streaming

```
val kafkaStream = spark.readStream…
val f = parse andThen process andThen serialize
val result = f(kafkaStream)
result.writeStream
     .format("kafka")
     .option("kafka.bootstrap.servers",bootstrapServers)
     .option("topic", writeTopic)
     .option("checkpointLocation", checkpointLocation)
     .start()
```



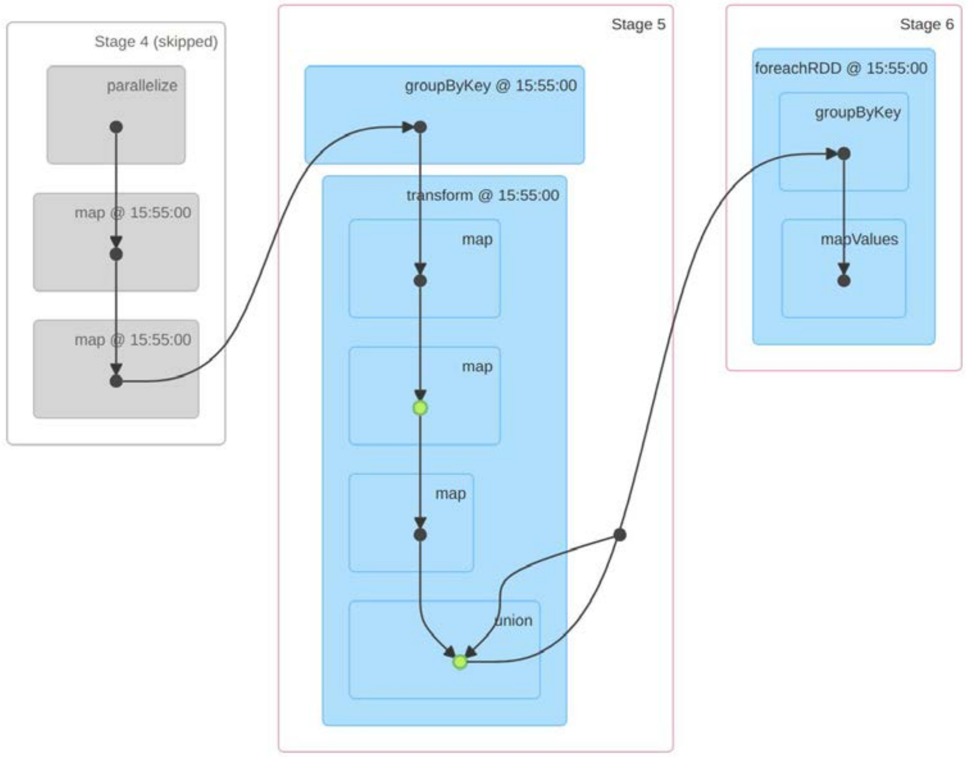
Keep Arbitrary State

Keeping Arbitrary State

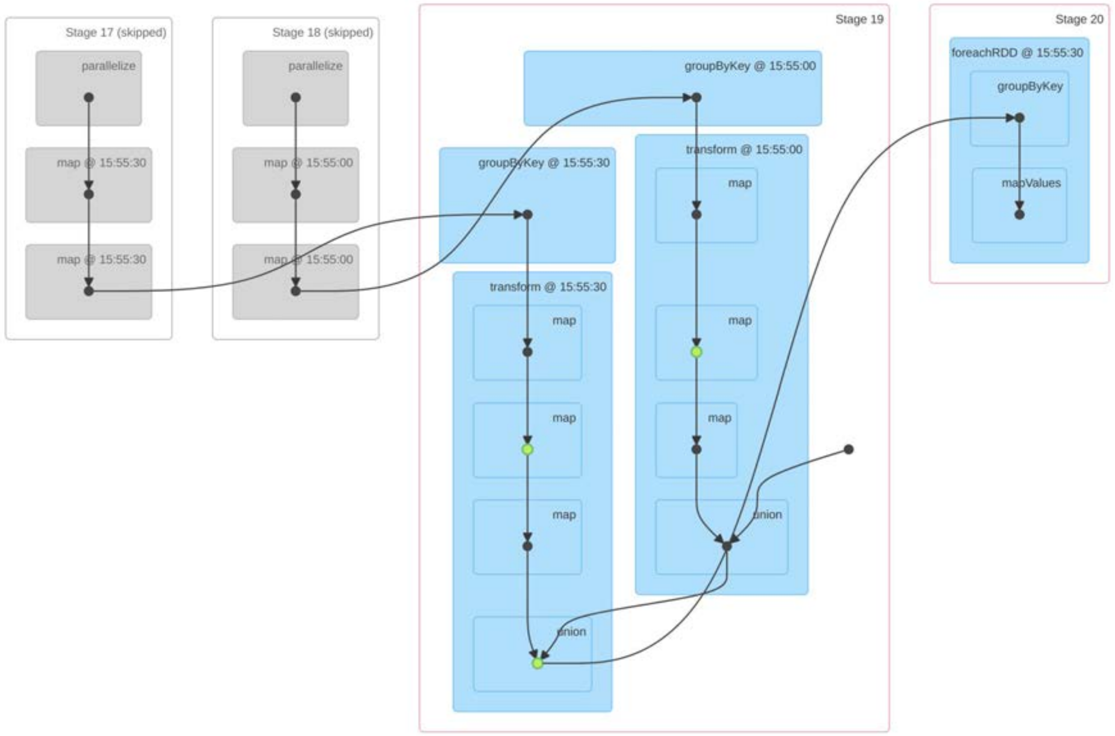
```
var baseline: Dataset[Features] = sparkSession.read.parquet(targetFile).as[Features]
...
stream.foreachRDD{ rdd =>
 val incomingData = sparkSession.createDataset(rdd)
 val incomingFeatures = rawToFeatures(incomingData)
 val analyzed = compare(incomingFeatures, baseline)
 // store analyzed data
 baseline = (baseline union incomingFeatures).filter(isExpired)
}
```

https://gist.github.com/maasg/9d51a2a42fc831e385cf744b84e80479

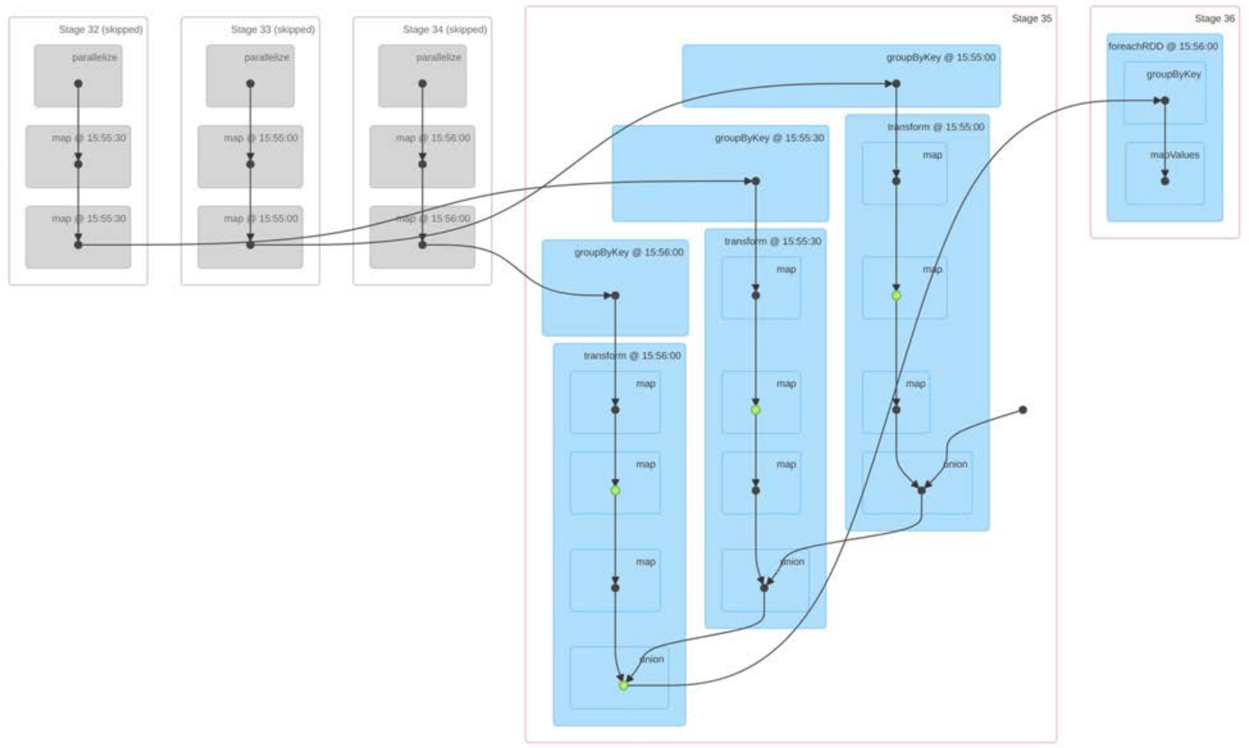
Keeping Arbitrary State



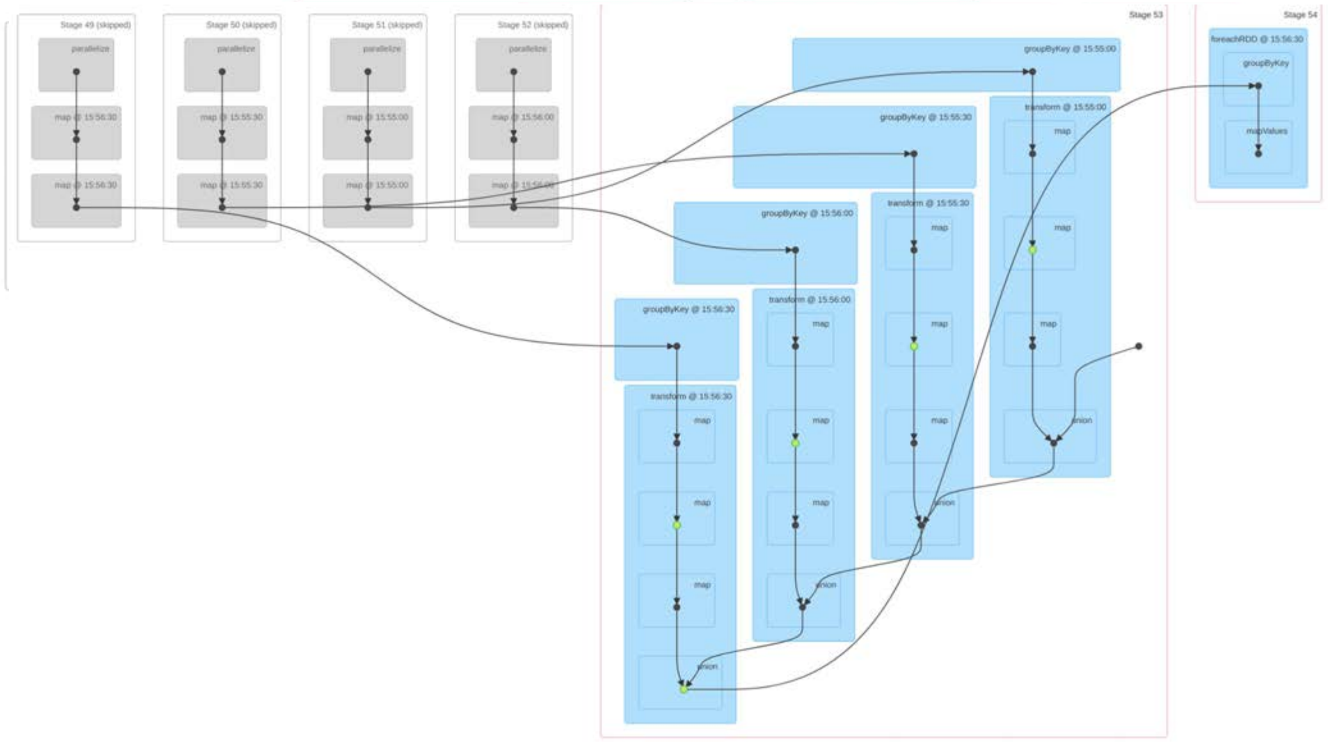
Keeping Arbitrary State



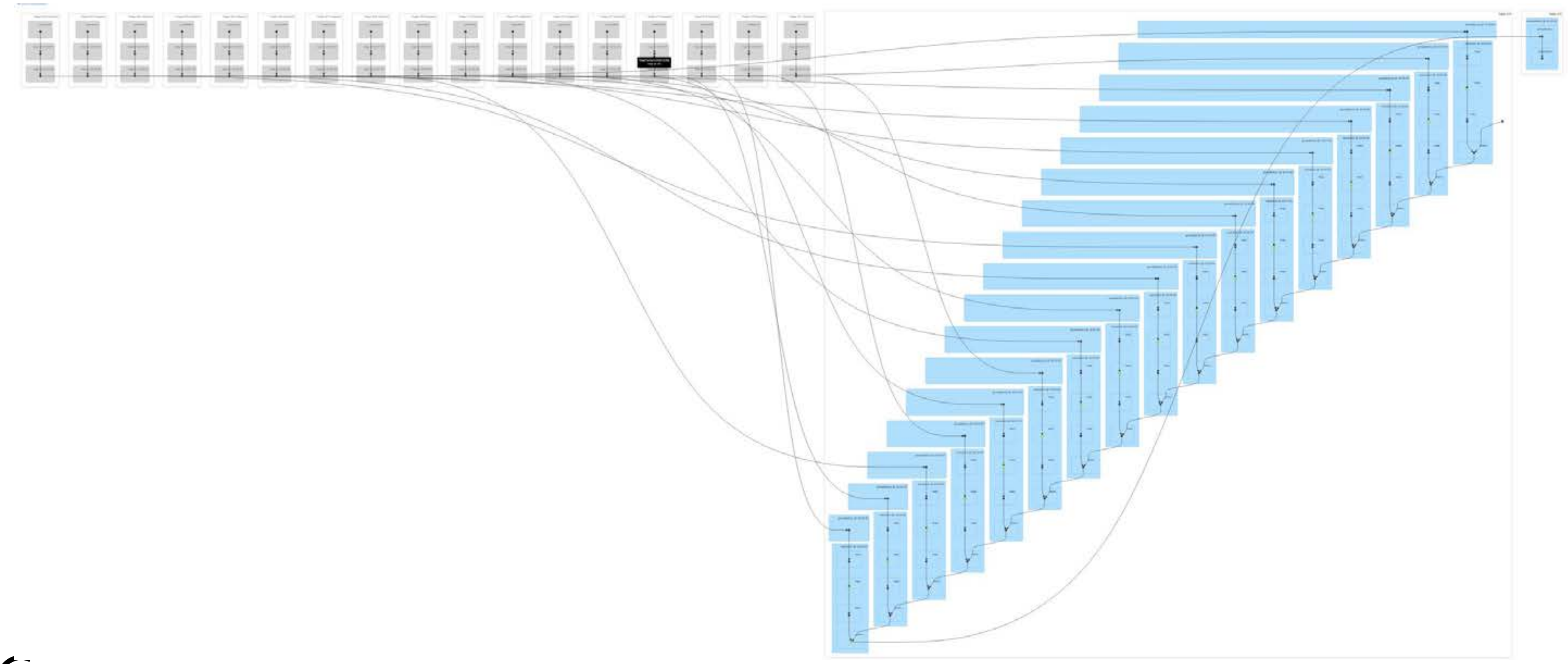
Keeping Arbitrary State



Keeping Arbitrary State

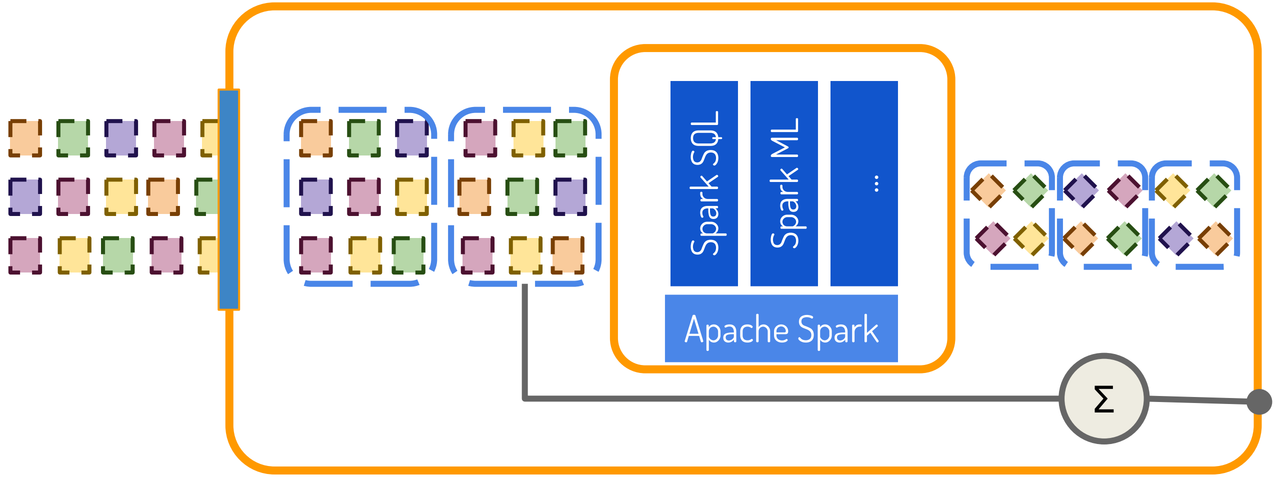


Keeping Arbitrary State

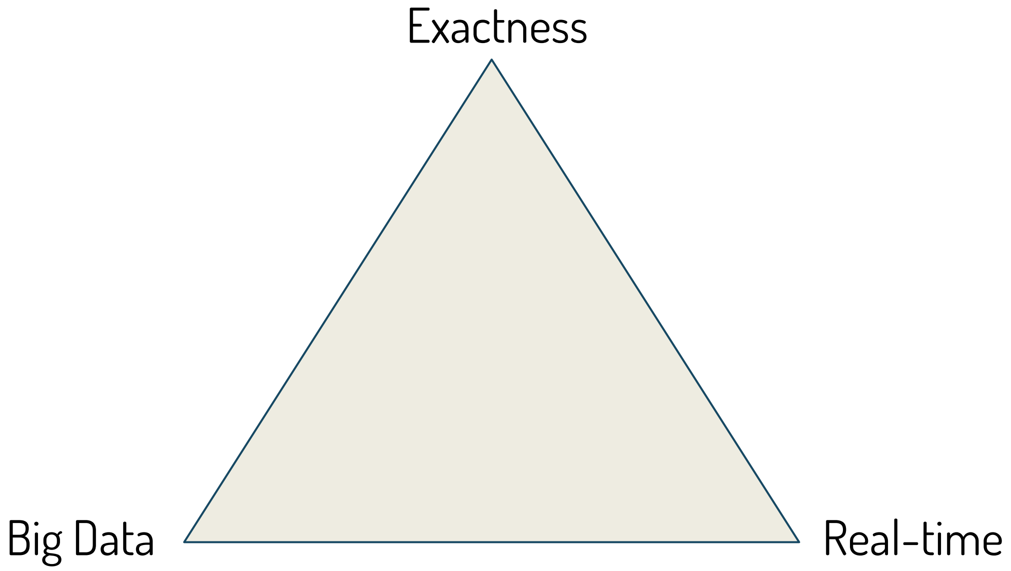


Keeping Arbitrary State: Roll **your own** checkpoints !

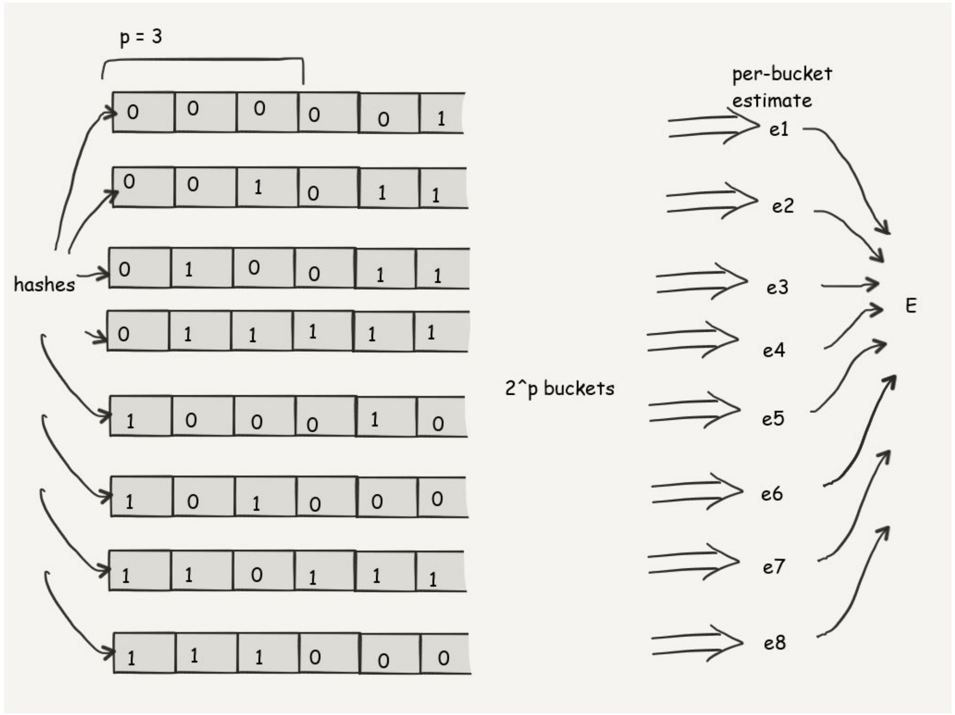
```
var baseline: Dataset[Features] = sparkSession.read.parquet(targetFile).as[Features]
var cycle = 1
var checkpointFile = 0
stream.foreachRDD{ rdd =>
 val incomingData = sparkSession.createDataset(rdd)
 val incomingFeatures = rawToFeatures(incomingData)
 val analyzed = compare(incomingFeatures, baseline)
 // store analyzed data
 baseline = (baseline union incomingFeatures).filter(isOldFeature)
 cycle = (cycle + 1) % checkpointInterval
 if (cycle == 0) {
   checkpointFile = (checkpointFile + 1) % 2
   baseline.write.mode("overwrite").parquet(s"$targetFile_$checkpointFile")
   baseline = baseline.read(s"$targetFile_$checkpointFile")
 }
}
```



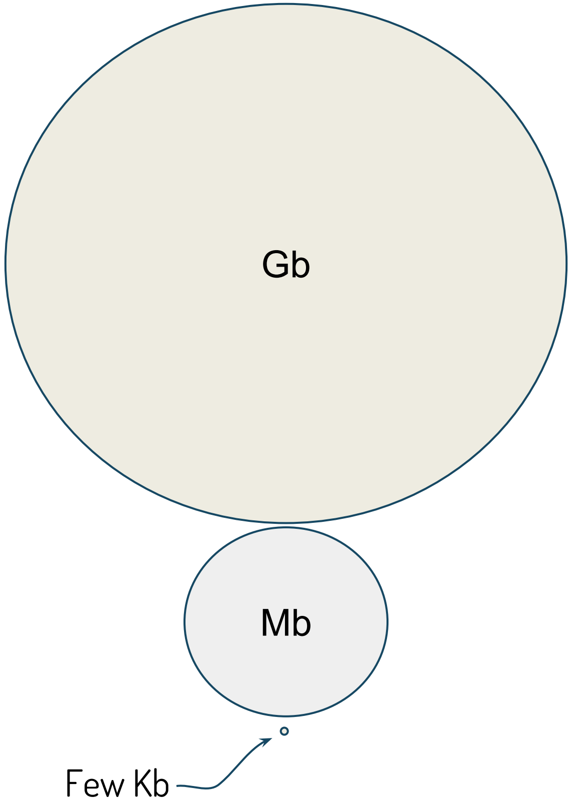
Probabilistic Accumulators


Exactness
Big Data
Real-time

HyperLogLog: Cardinality Estimation



accuracy = 1.054 / sqrt(2^p)



HLL Accumulator

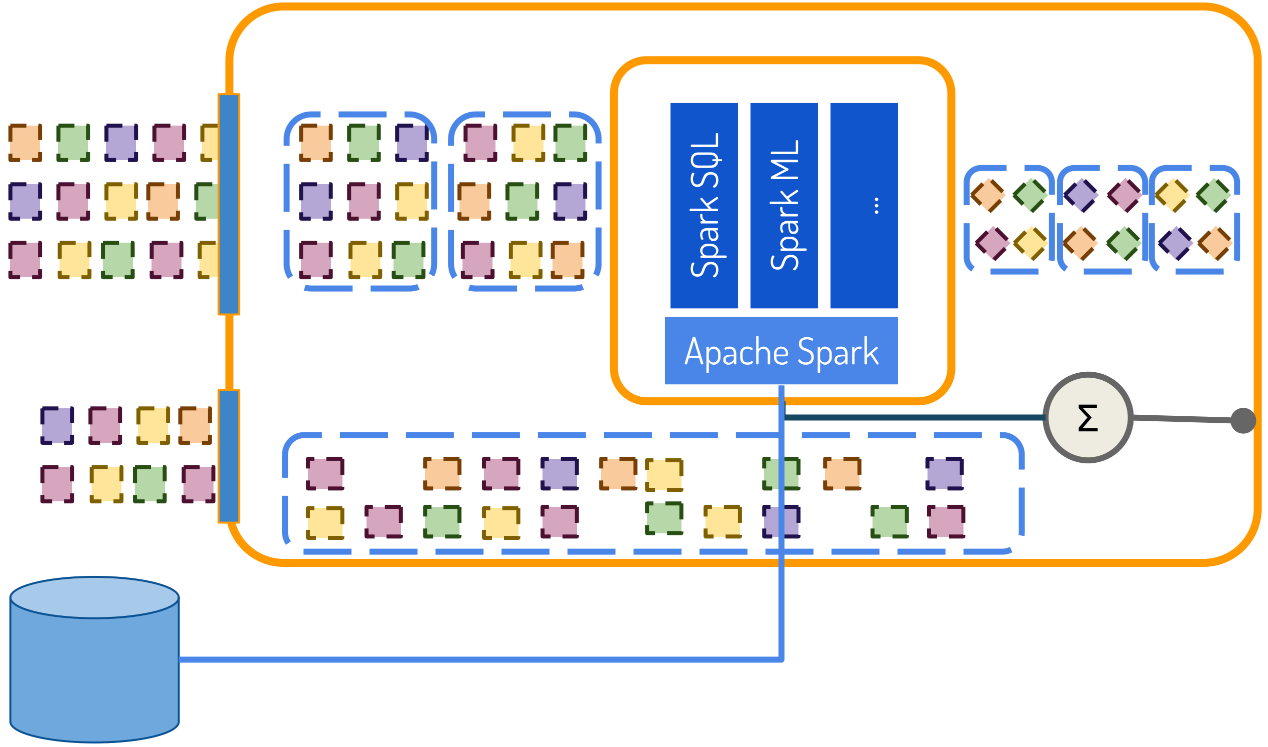
```
class HLLAccumulator[T](precisionValue: Int = 12) extends AccumulatorV2[T, Long] {

  private def instance(): HyperLogLogPlus = new HyperLogLogPlus(precisionValue, 0)

  override def add(v: T): Unit = hll.offer(v)

  override def merge(other: AccumulatorV2[T, Long]): Unit = other match {
     case otherHllAcc: HLLAccumulator[T] => hll.addAll(otherHllAcc.hll)
     case _ => throw new UnsupportedOperationException(
       s"Cannot merge ${this.getClass.getName} with ${other.getClass.getName}")
   }
}
```

https://github.com/LearningSparkStreaming/HLLAccumulator

Using Probabilistic Accumulators

```
import learning.spark.streaming.HLLAccumulator
val uniqueVisitorsAccumulator= new HLLAccumulator[String](precisionValue = 12)
sc.register(uniqueVisitorsAccumulator, "unique-visitors")
...
clickStream.foreachRDD{rdd =>
   rdd.foreach{
       case BlogHit(ts, user, url) => uniqueVisitorsAccumulator.add(user)
   }
   ...
   val currentUniqueVisitors = uniqueVisitorsAccumulator.value
   ...
}
```

Putting it all Together



Questions?

Thank You

@maasg

Lightbend