SPARK+AI SUMMIT 2019

WIFI SSID:SparkAISummit | Password: UnifiedAnalytics

ORGANIZED BY databricks

Spark Streaming

Headaches and Breakthroughs in Building Continuous Applications

Landon Robinson & Jack Chapa

SPOTX

#UnifiedAnalytics #SparkAISummit

Who We Are

Landon Robinson
Data Engineer

Jack Chapa
Data Engineer

Big Data Team @ SpotX

But first… why are we here?

Because Spark Streaming...

- is very **powerful**
- can **supercharge your infrastructure**
- … and *can* be very complex**!**

Lots of headaches and breakthroughs!

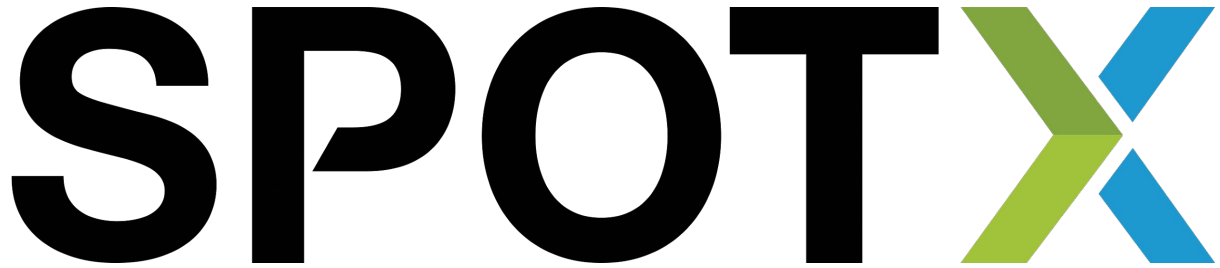
Takeaways

Leave with a few actionable items that we wish we knew when we started with Spark Streaming.

Focus Areas

- Streaming Basics
- Testing
- Monitoring & Alerts
- Batch Intervals & Resources
- Helpful Configurations
- Backpressure
- Data Enrichment & Transformations

Our Company



The Trusted Platform For Premium Publishers and Broadcasters

We Process a Lot of Data

Data:

- 220 MM+ Total Files/Blocks
- 8 PB+ HDFS Space
- 20 TB+ new data daily
- 100MM+ records/minute
- 300+ Data Nodes

Apps:

- Thousands of daily Spark apps
- Hundreds of daily user queries
- Multiple 24/7 Streaming apps

Spark Streaming is Key for Us

Our uses include:

- **Rapid ingestion of data** into warehouse for querying
- **Machine learning on near-live data** streams
- **Ability to react to and impact** live situations
- **Accelerated processing / updating** of metadata
- **Real-time visualization** of data streams and processing

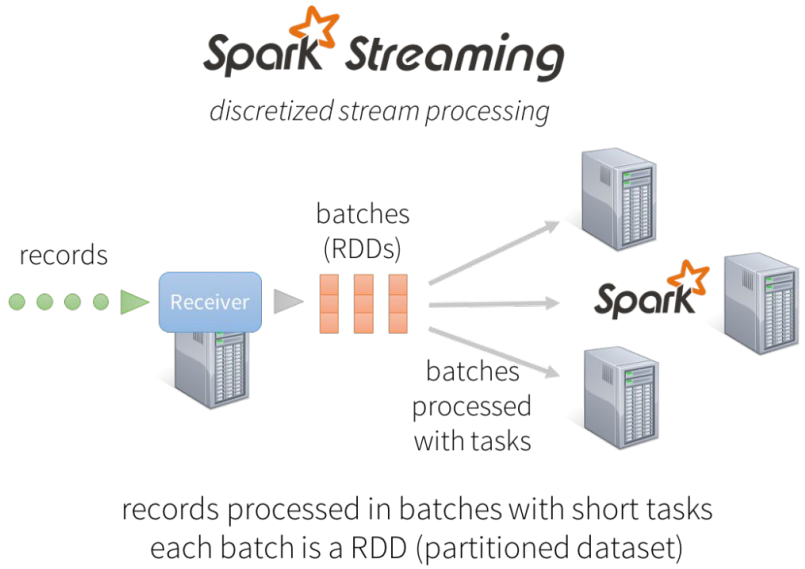
Spark Streaming Basics

a brief overview

Spark Streaming Basics

Spark Streaming is an extension of Spark that enables scalable, high-throughput, fault-tolerant processing of live data streams.

- **Stream** == live data stream
  - **Topic** == Kafka's name for a stream
- **DStream** == sequence of RDDs formed from reading a data stream
- **Batch** == a self-contained job within your Streaming app that processes a segment of the stream.



Testing

Rapid development and testing of Spark apps

Use Spark in Local Mode

You can start building Spark Streaming apps in minutes, **using Spark locally!**

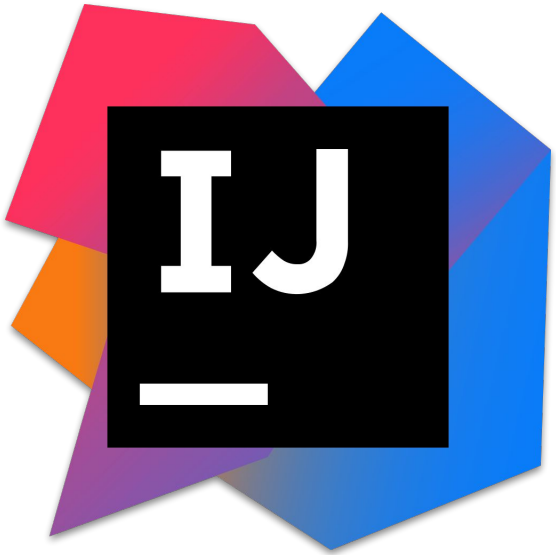
On your local machine

- No cluster needed!
- Great for rough testing

We Recommend:

IntelliJ Community Edition

- with SBT: For dependency management



Use Spark in Local Mode

The Scala Build Tool is your friend!

Simply:

- Import Spark libraries
- Invoke a Context and/or Session
- Set master to local[*] or local[n]

In your build.sbt:

- src/**test**/scala => “provided”
- src/**main**/scala => “compiled”

```
// Dependency Chain
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % spark_version % "provided",
  "org.apache.spark" %% "spark-sql"  % spark_version  % "provided",
  "org.apache.spark" %% "spark-hive" % spark_version % "provided",
```

Example Unit Test using just a SparkContext

```scala
// Setup Spark
val spark = SparkSession
  .builder()
  .master("local[1]")
  .appName("CassandraTests")
  .enableHiveSupport()
  .getOrCreate()

// Import Implicits (case class casting)
import spark.implicits._

// Unit Test #1: Retrieve audience_data sample and cast it
"Cassandra Connector" should "retrieve a 10 row audience_data sample from Cassandra (dev env)" in {

  val df = spark
    .read
    .format("org.apache.spark.sql.cassandra")
    .options(table_audience)
    .load()
    .as[Audience]
    .limit(10)

  assert(df.count() === 10L)
}
```

Invoke a local session:

- In your unit test classes
- Test logic on small datasets

Add to your deployment pipeline for a nice pre-release gut check!

Unit Testing

Spark Streaming Apps can easily be unit tested

- Using *.queueStream()*
- Using a spark testing library

Libraries

- spark-testing-base
- sscheck
- spark-tests

Use Cases

- DStream actions
- Business Logic
- Integration

Example Library: spark-testing-base

- Easy to Use
- Helpful wrappers
- Integrates w/ *scalatest*
- Minimal code required
- Clock management
- Runs alongside other tests

```
class FileSparkStreamingTests extends FunSuite
  with StreamingSuiteBase with SharedSparkContext {

  test( testName = "Convert CSV data to Session") {
    val testData: List[List[String]] = List(FileSparkStreamingTests.getTestData)
    val expected = List(List(FileSparkStreamingTests.getTestSession))

    def validateResults(ds: DStream[String]): DStream[Session] = {
      FileSparkStreaming.processSessionData(ds)
    }

    testOperation[String, Session](testData, validateResults _, expected, ordered = true)

  }
}
```

```
val spark_version = "2.4.0"
val spark_testing_version = spark_version + "_0.11.0"

libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % spark_version % "provided",
  "org.apache.spark" %% "spark-sql" % spark_version % "provided",
  "org.apache.spark" %% "spark-streaming" % spark_version % "provided",
  "com.typesafe.slick" %% "slick" % "3.3.0",
  "org.slf4j" % "slf4j-nop" % "1.7.10",
  "com.holdenkarau" %% "spark-testing-base" % spark_testing_version % "test"
)

fork in Test := true
javaOptions ++= Seq("-Xms512M", "-Xmx2048M", "-XX:MaxPermSize=2048M", "-XX:+CMSClassUnloadingEnabled")
```

GitHub: https://github.com/holdenk/spark-testing-base

Monitoring

Tracking and visualizing performance of your app

Monitoring is Awesome

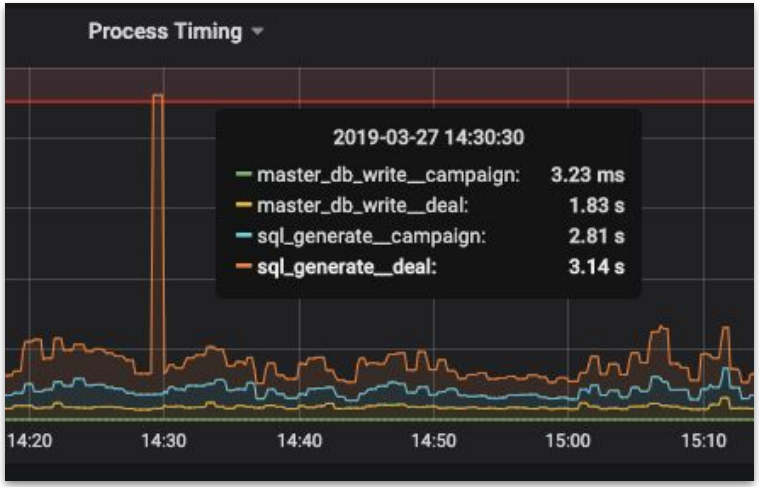
It can reveal:

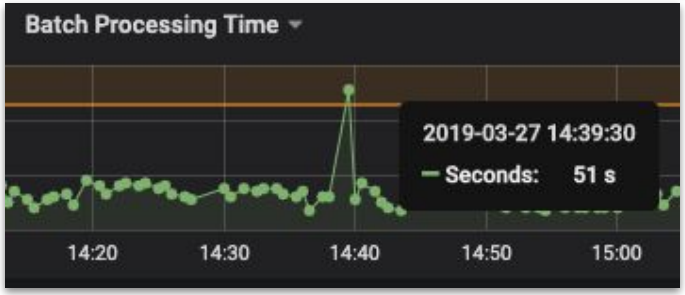
- **How your app is performing**
- **Problems + Bugs!**

And provide opportunities to:

- **See and address issues**
- **Observe behavior visually**

But monitoring can be *tough* to implement!





Monitoring (a less than ideal approach)

You *could* do it all in the app...

Example: Looping over RDDs to:

- **Count records**
- **Track Kafka offsets**
- **Processing time / delays**

But it's less than ideal...

- Calculating performance significantly impacts performance… not great.
- All these metrics are **calculated by Spark!**

```
myStreamingData.count().print()
```

```
myStreamingData.foreachRDD(rdd => {
  rdd.map(x => {
    println(x.topic())
    println(x.offset())
    println(x.partition())
  })
})
```

```
val t0 = System.nanoTime()
// execute a task
val t1 = System.nanoTime()
val processing_time = (t1 - t0) / 1000
println("I took this long: " + processing_time)
```

Monitoring and Visualization (using Listeners)

Use **Spark Listeners** to access metrics in the background!

Let Spark do the hard work:

- Batch duration, delays
- Record throughput
- Stream position recovery

Come to our talk: Spark Listeners: A Crash Course in Fast, Easy Monitoring!

- Room 3016 | Today @ 5:30 PM

```
package org.apache.spark.streaming.scheduler
*/
@DeveloperApi
trait StreamingListener {

  /** Called when a receiver has been started */
  def onReceiverStarted(receiverStarted: StreamingListenerReceiverStarted) { }

  /** Called when a receiver has reported an error */
  def onReceiverError(receiverError: StreamingListenerReceiverError) { }

  /** Called when a receiver has been stopped */
  def onReceiverStopped(receiverStopped: StreamingListenerReceiverStopped) { }

  /** Called when a batch of jobs has been submitted for processing. */
  def onBatchSubmitted(batchSubmitted: StreamingListenerBatchSubmitted) { }

  /** Called when processing of a batch of jobs has started.  */
  def onBatchStarted(batchStarted: StreamingListenerBatchStarted) { }

  /** Called when processing of a batch of jobs has completed. */
  def onBatchCompleted(batchCompleted: StreamingListenerBatchCompleted) { }

  /** Called when processing of a job of a batch has started. */
  def onOutputOperationStarted(
      outputOperationStarted: StreamingListenerOutputOperationStarted) { }

  /** Called when processing of a job of a batch has completed. */
  def onOutputOperationCompleted(
```

Kafka Offset Recovery

Saving your place elsewhere

Writing Offsets to MySQL

Inside the Spark Listener class, after a batch completes...

You can access an object generated by Spark containing your offsets processed.

Take those offsets and back them up to a DB...

```
/**
  * Takes a topic object and writes the max offset for each partition it contains this batch to MySQL.
  *
  * @param topic A topic object within a Batch's StreamIdToInputInfo
  */
def writeTopicOffsetsToMySQL(topic: Tuple2[Int, StreamInputInfo]): Unit = {

  // map offset info to OffsetRange objects
  val partitionOffsets = topic._2.metadata("offsets").asInstanceOf[List[OffsetRange]]

  // for every partition's range of offsets
  partitionOffsets.map(offsetRange => {

    // write the new starting offset for each partition in the topic to the state db
    var maxOffset = offsetRange.untilOffset - 1

    // create a now() timestamp
    val now = new DateTime().toString("YYYY-MM-dd HH:mm:ss")

    // form the sql
    val sql =
      s"""INSERT INTO $mySQLDB.$mySQLTable (consumer, topic, partition_id, offset, offset_ts, batch_size)
         VALUES
         ('$mySQLConsumer', "${offsetRange.topic}", ${offsetRange.partition}, '$maxOffset', '$now', ${offsetRange.count})
         ON DUPLICATE KEY UPDATE offset_ts = VALUES(offset_ts), offset = VALUES(offset),
           batch_size = VALUES(batch_size)
      """

    // execute the sql to offload offsets to the table
    val st = mySQLConnectionPool.createStatement
    st.execute(sql)
    st.close()
  })
}
```

Reading Offsets from MySQL

Your offsets are now stored in a DB after each batch completes.

Whenever your app restarts, it reads those offsets from the DB...

And starts processing where it last left off!

```
def subscribeToTopicWithState(
                                streamingContext: StreamingContext,
                                topic: String,
                                kafkaParams: Map[String, Object],
                                dbConnection: MySQLConnection,
                                stateConsumer: String,
                                resetOffsets: Boolean = false
                    ): InputDStream[ConsumerRecord[String, Option[GenericRecord]]] = {

  // Store list of topics in an array
  val topics = Array(topic)

  // Either Get Offsets from a State Database or Reset to Latest
  val offsets = getOffsetsFromState(dbConnection, stateConsumer, topic, resetOffsets)

  // Subscribe to the topic based on the offsets provided
  val strategy = getStrategy(offsets, kafkaParams, topics)

  // Create a direct stream and pull data
  val subscribed = KafkaUtils.createDirectStream[String, Option[GenericRecord]](
    streamingContext,
    LocationStrategies.PreferConsistent,
    strategy
  )

  // Filter data coming out of Kafka to remove bad records
  subscribed
}
```

Getting Offsets from the Database

```
def getOffsetsFromState(dbConnection: MySQLConnection,
                        consumer: String,
                        topic: String,
                        resetOffsets: Boolean
                       ): Map[TopicPartition, Long] = {

  // Establish a MySQL Connection given a URL
  val stateDbConn = ConnectionPool(dbConnection.toString).getConnection

  // If You want to reset offsets, just return an empty map
  val offsets =
    if (resetOffsets) {
      new HashMap[TopicPartition, Long]()
    }

    // If you want to use your existing offsets, get them from MySQL
    else {
      val stmt = stateDbConn.createStatement
      val query = "SELECT * FROM " + dbConnection.database + "." + dbConnection.table +
        " WHERE consumer = '" + consumer + "' AND topic = '" + topic + "'"
      logger.info(query)
      val result = stmt.executeQuery(query)
      new RsIterator(result).map(
        x =>
          new TopicPartition(
            x.getString("topic"),
            x.getInt("partition_id")
          ) ->
            (x.getLong("offset") + 1L) // Add one because we record last known offset
      ).toMap
    }

  offsets

}
```

Example: Reading Offsets from MySQL

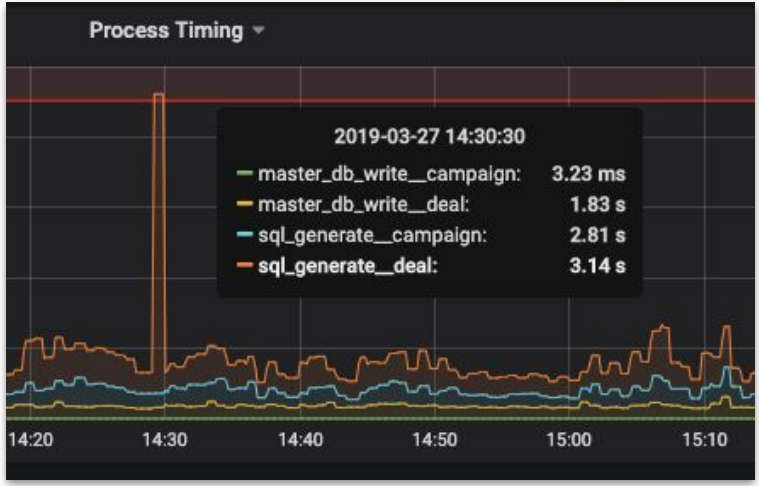
```scala
def subscribeToTopicWithState(
                              streamingContext: StreamingContext,
                              topic: String,
                              kafkaParams: Map[String, Object],
                              dbConnection: MySQLConnection,
                              stateConsumer: String,
                              resetOffsets: Boolean = false
                    ): InputDStream[ConsumerRecord[String, Option[GenericRecord]]] = {

  // Store list of topics in an array
  val topics = Array(topic)

  // Either Get Offsets from a State Database or Reset to Latest
  val offsets = getOffsetsFromState(dbConnection, stateConsumer, topic, resetOffsets)

  // Subscribe to the topic based on the offsets provided
  val strategy = getStrategy(offsets, kafkaParams, topics)

  // Create a direct stream and pull data
  val subscribed = KafkaUtils.createDirectStream[String, Option[GenericRecord]](
    streamingContext,
    LocationStrategies.PreferConsistent,
    strategy
  )

  // Filter data coming out of Kafka to remove bad records
  subscribed
}
```

Example: Reading Offsets from MySQL

```
def getStrategy(
                offsets: Map[TopicPartition, Long],
                kafkaParams: Map[String, Object],
                topics: Array[String]
              ): ConsumerStrategy[String, Option[GenericRecord]] ={

  // if offsets were pulled from MySQL, assign the stream to start there
  val strategy = if (offsets.size > 0) {
    ConsumerStrategies.Assign[String, Option[GenericRecord]](offsets.keys.toList, kafkaParams, offsets)
  }

  // otherwise, start at latest data
  else {
    ConsumerStrategies.Subscribe[String, Option[GenericRecord]](topics, kafkaParams)
  }

  strategy
}
```

Timing Logging (around actions)

- **Record timing info** for fast troubleshooting
  - **Escalate alarms** to the appropriate team
  - **Quickly resolve** while app continues running

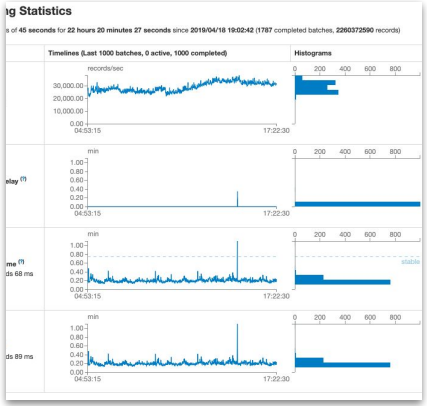


React

How do I react to this monitoring?

- Heartbeats
- Scheduled Monitor Jobs
  - Version Updates
  - Ensure Running
  - Act on failure/problem
- Monitoring Alarms
- Look at them!

pagerduty



Batch Intervals

Optimizing for speed
and resource efficiency

Setting Appropriate Batch Intervals

An **appropriate batch interval is key** to an app that is quick and efficient.

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

  val sparkConf = new SparkConf()
    .setAppName("My Streaming App")
    .setMaster("local[*]")

  val ssc = new StreamingContext(sparkConf,
                                 batchDuration = Seconds(30))
```

Batch interval

Effectiveness of interval is affected by:

- Resources alloc (cpu + ram)
- Quantity of work
- Quantity of data

You want batches that **process faster than the interval**, but **not so fast that resources are idling and therefore wasted**!

Setting Appropriate Batch Intervals

Consider these questions:

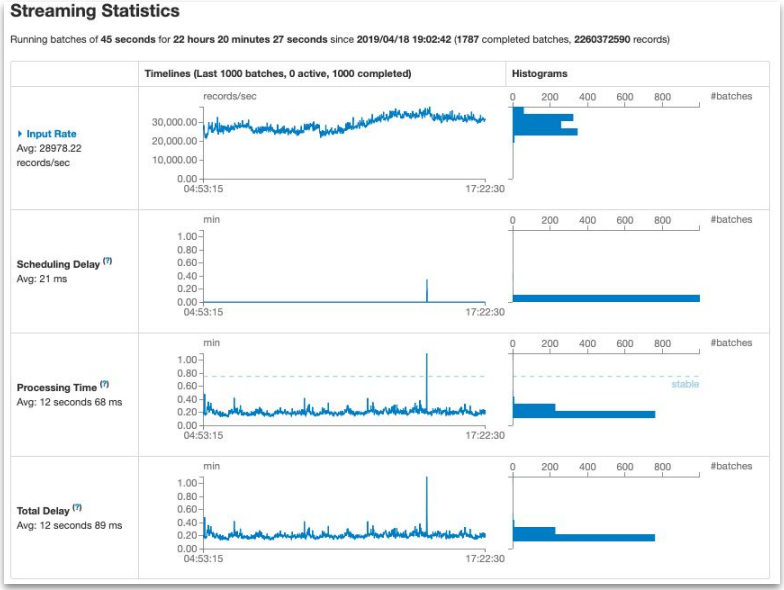
How quickly do I need to process data?

- Can I slow it down to save resources?

What is my resource budget / allocation?

- Can I increase? Can I cut back?

- **<u>Bigger</u> interval = more time to process**
- … but also more data to process
- **<u>Smaller</u> interval = the opposite**



Setting Appropriate Batch Intervals

Tips for finding an optimal combination:

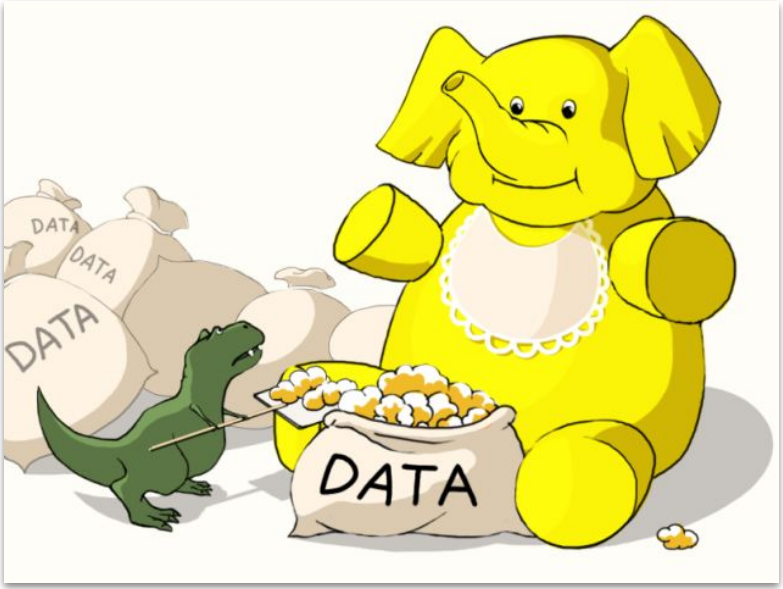
Start small!

a. **Short batch interval (seconds)**
b. **Modest resources**

Whichever you have in more flexible supply (a or b), increasing accordingly.

Again: processing time < interval = good

Comfortably less, not significantly less.



Additional Resource Notes

- **Scale down** when possible
  - Free up resources or save on cloud utilization spend
- **Avoid preemption**
  - Use resource pools with prioritization
  - With preemption disabled if you can
- **Set appropriate # of partitions** for Kafka topics
  - Higher volume == higher partition count
  - Higher partition count == greater parallelization

Helpful Configuration Settings

Configuring your app to be performant and efficient

Helpful Configuration Settings

Spark

- *spark.memory.useLegacyMode* = true
  - *spark.storage.memoryFraction*=0.03
- *spark.submit.deployMode* = cluster
- *spark.serializer* = org.apache.spark.serializer.KryoSerializer
- *spark.rdd.compress* = true
  - *spark.io.compression.codec*=org.apache.spark.io.SnappyCompressionCodec
- *spark.shuffle.service.enabled* = true
- *spark.streaming.blockInterval* = 300

Kafka

- *enable.auto.commit* = ‘false’

Backpressure

Use Case:

You have irregular spikes in message throughput from Kafka topics

- **Backpressure dynamically alters rate data is received per batch from Kafka.**
- **Prevents overwhelming of app at startup and peak load.**

Settings:

- *spark.streaming.backpressure.enabled* = true
- *spark.streaming.kafka.maxRatePerPartition* = 20000
  - max rate (messages/second) at which each Kafka partition will be read
- **PID Rate Estimator**: can be used to tweak the rate based on batch performance
  - spark.streaming.backpressure.pid.*

Source: https://www.linkedin.com/pulse/enable-back-pressure-make-your-spark-streaming-production-lan-jiang/

Transformations

Bringing streaming and static data together

Transformations (Streaming + Static)

transform()

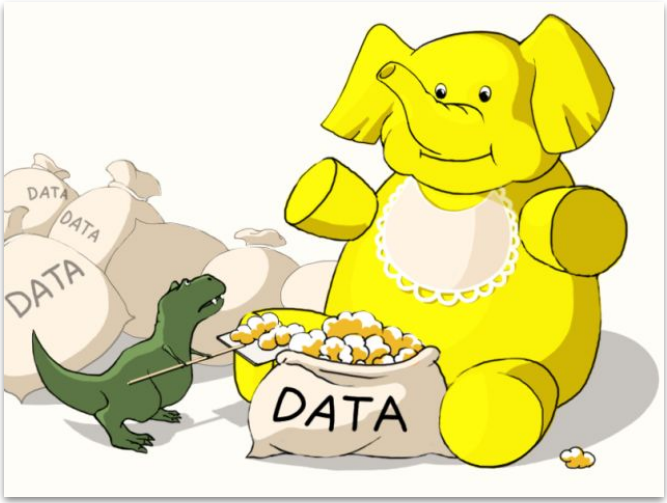
- Allows RDD-level access to data.
- Use case: joining with another RDD

updateStateByKey() / mapWithState()

- Apply function to each key - useful for keeping track of state
- Use case: maintaining state between batches (e.g. rolling join w/ two streams)

reduceByKey()

- Reduce a keyed RDD with appropriate function.
- Use case: deduping, aggregations



Joining Streaming and Static Data

Using the transform() method on DStream:

Apply an RDD-to-RDD function to every RDD of the DStream. Used for arbitrary RDD operations on the DStream.

- **Useful for applying arbitrary RDD operations on a DStream.**
- **Great for enriching streaming data with supplemental static data**

```
transactions = … // streaming dataset (dstream)

transaction_details = … // static dataset (rdd)


val complete_transaction_data  = transactions.transform(live_transaction  =>

  live_transaction.join(transaction_details))
```

Source: https://hadoopsters.net/2017/11/26/how-to-join-static-data-with-streaming-data-dstream-in-spark/

Effective Static Joining

How do we handle static and persistent data?

Driver:

- Broadcast if small enough
- Read on driver every batch, then join

Worker:

- Connect on worker - lazy val connection object
- Useful for persisting data

Review

Streaming isn't always easy… but **here are some great takeaways!**

- **Testing**: Use Spark Locally w/ Unit Tests
- **Monitoring**: Use Listeners & React
- **Batch Intervals & Resources**: Be thoughtful!
- **Configuration**: Lots of awesome ones!
- **Transformations**: Do more with your streaming data!
- **Offset Recovery**: Stop worrying and love the offset management!

Contact Us

Landon Robinson

- lrobinson@spotx.tv

Jack Chapa

- jchapa@spotx.tv

hadoopsters.dev

https://gist.github.com/hadoopsters


SPARK+AI
SUMMIT 2019
Q & A
#UnifiedAnalytics #SparkAISummit

SPARK+AI SUMMIT 2019

DON'T FORGET TO RATE AND REVIEW THE SESSIONS

SEARCH SPARK + AI SUMMIT

Download on the App Store

GET IT ON Google Play