LARGE-SCALE DATA ANALYSIS WITH APACHE SPARK

ALEXEY SVYATKOVSKIY

PRINCETON UNIVERSITY

OUTLINE

This talk is intended to give a quick intro to the Spark programming model, give an overview of using Apache Spark on Princeton clusters, as well as explore it's possible applications in the HEP

- INTRO TO DATA ANALYSIS WITH APACHE SPARK
  - SPARK SOFTWARE STACK
  - PROGRAMMING WITH RDDs
  - SPARK JOB ANATOMY: RUNNING LOCALLY AND ON A CLUSTER
  - PRINCETON BIG DATA EXPERIENCE
- REAL-TIME ANALYSIS PIPELINES USING SPARK STREAMING
- MACHINE LEARNING LIBRARIES
  - ANALYSIS EXAMPLES
- SUMMARY AND POSSIBLE USE CASES IN HEP

WHAT IS APACHE SPARK

- APACHE SPARK IS A FAST AND GENERAL PURPOSE CLUSTER COMPUTING FRAMEWORK FOR LARGE-SCALE DATA PROCESSING
  - IT BECAME A DE-FACTO INDUSTRY STANDARD FOR DATA ANALYSIS, REPLACING MAPREDUCE COMPUTING ENGINE
  - MAPREDUCE ENGINE IS GOING TO BE RETIRED BY CLOUDERA - A MAJOR HADOOP DISTRIBUTION PROVIDER – STARTING THE VERSION CDH5.5
- SPARK DOES NOT USE THE MAPREDUCE AS AN EXECUTION ENGINE, HOWEVER, IT IS CLOSELY INTEGRATED WITH HADOOP ECOSYSTEM AND CAN BE RUN VIA YARN, USE THE HADOOP FILE FORMATS, AND HDFS STORAGE
- ON THE OTHER HAND, IT CAN BE USED IN A STANDALONE MODE ON ANY HPC CLUSTERS
  - E.G. VIA SLURM RESOURCE MANAGER AS IT IS DONE AT PRINCETON
- SPARK IS BEST KNOWN FOR ITS ABILITY TO PERSIST LARGE DATASETS IN MEMORY BETWEEN JOBS
- SPARK IS WRITTEN IN SCALA, BUT THERE ARE LANGUAGE BINDINGS FOR PYTHON, SCALA, AND JAVA

Red color: Specific to Princeton clusters

Spark R

Available starting Spark 1.5

SPARK SOFTWARE STACK

Spark SQL
structured data
Read: Distributed Pandas Dataframe

Spark Streaming
real-time
Supports Kafka, Flume sinks

MLib
machine
learning

GraphX
graph
processing

HEP use case: possibility to run ROOT in a distributed way using Spark RDDs?

Storage:

GPFS

NFS

HDFS

Cassandra

Interconnect:

Infiniband

10g ethernet

Spark Core

Standalone Scheduler
Use SLURM on general purpose HPC clusters

YARN
Use on Hadoop cluster

Mesos

https://www.princeton.edu/researchcomputing/computational-hardware/hadoop/
https://www.princeton.edu/researchcomputing/computational-hardware/machine-2/

PROGRAMMING WITH RDDS (I)

- RDDs (RESILIENT DISTRIBUTED DATASETS) ARE READ-ONLY PARTITIONED COLLECTIONS OF OBJECTS
- EACH RDD IS SPLIT INTO PARTITIONS WHICH CAN BE COMPUTED ON DIFFERENT NODES OF A CLUSTER
- PARTITIONS DEFINE THE LEVEL OF PARALLELISM IN A SPARK APP: IMPORTANT PARAMETER TO TUNE!
- CREATE AN RDD BY LOADING DATA INTO IT, OR PARALLELIZING EXISTING COLLECTION OF OBJECTS (LIST, SET…)

```
npartitions = 10
sc = SparkContext(master, "TestApp")
lines = sc.parallelize(["pandas", "i like pandas"],npartitions)
```

When data has no parent RDD/input file the # of partitions is set according to the total number of cores on nodes which run executors on them.

```
npartitions = 10
sc = SparkContext(master, "TestApp")
lines = sc.textFile("/user/alexeys/test.txt",npartitions)
```

Spark automatically sets the # of partitions according to the number of file system block the file spans over. For reduce tasks, it is set according to the parent RDD

See “Learning Spark” to get started: https://github.com/holdenk/learning-spark-examples

PROGRAMMING WITH RDDS (II)

- TRANSFORMATIONS: OPERATIONS ON RDD THAT RETURN A NEW RDD. THEY DO NOT MUTATE THE OLD RDD, BUT RATHER RETURN A POINTER TO IT

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.

- ACTIONS: ACTIONS FORCE PROGRAM TO PRODUCE SOME OUTPOUT (NOTE: RDDs ARE LAZILY EVALUATED)

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.

Similar to Pythonic map, filter and reduce syntax, except operator chaining is allowed in Spark.
I use lambda-functions most of the time

- LAZY EVALUATION: RDD TRANSFORMATIONS ARE NOT EVALUATED UNTIL AN ACTION IS CALLED ON IT
- PERSISTANCE/CACHING: UNLIKE MAPREDUCE, WHERE MAKING AN INTERMEDIATE RESULT OBTAINED ON THE ENTIRE DATASET AVAILABLE TO ALL NODES WOULD ONLY BE POSSIBLE BY SPLITTING THE CALCULATION INTO MULTIPLE MAP-REDUCE STAGES (CHAINING) AND PERFORMING AN INTERMEDIATE SHUFFLE
  - CRUCIAL FEATURE FOR ITERATIVE ALGORITHMS LIKE, FOR INSTANCE, K-MEANS

PROGRAMMING WITH RDDS (III)

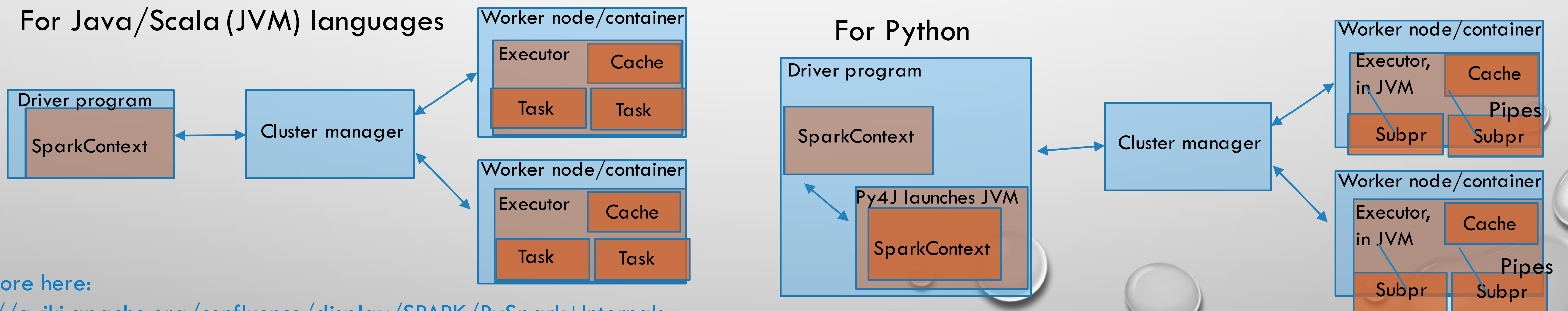
- SPARK ALLOWS TO PERSIST DATASETS IN MEMORY AS WELL AS MEMORY/DISK (SPLIT IN A SPECIFIED PROPORTION CONTROLLED IN CONFIG)
- PYSPARK USES CPICKLE FOR SERIALIZING DATA

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Store RDD in serialized format in Tachyon. Compared to MEMORY_ONLY_SER, OFF_HEAP reduces garbage collection overhead and allows executors to be smaller and to share a pool of memory, making it attractive in environments with large heaps or multiple concurrent applications. Furthermore, as the RDDs reside in Tachyon, the crash of an executor does not lead to losing the in-memory cache. In this mode, the memory in Tachyon is discardable. Thus, Tachyon does not attempt to reconstruct a block that it evicts from memory. If you plan to use Tachyon as the off heap store, Spark is compatible with Tachyon out-of-the-box. Please refer to this page for the suggested version pairings.

From Spark Manual

ANATOMY OF A SPARK APP: RUNING ON A CLUSTER (I)

- SPARK USES MASTER/SLAVE ARCHITECTURE WITH ONE CENTRAL COORDINATOR (DRIVER) AND MANY DISTRIBTUED WORKERS (EXECUTORS)
- DRIVER RUNS ITS OWN JAVA PROCESS, EXECUTORS EACH RUN THEIR OWN JAVA PROCESSES
- FOR PYSPARK, SPARKCONTEXT USES PY4J TO LAUNCH A JVM AND CREATE A JAVASPARKCONTEXT
    - EXECUTORS ALL RAN IN JVMS, AND PYTHON SUBPROCESSES (TASKS) WHICH ARE LAUNCHED COMMUNICATE WITH THEM USING PIPES



See more here:
https://cwiki.apache.org/confluence/display/SPARK/PySpark+Internals

SPARK USER EXPERIENCE AT PRINCETON

- MOST OF THE CURRENT SPARK USERS COME FROM CS AND POLITICS DEPARTMENTS
- WE STARTED OUT BY USING SPARK VIA YARN INSTALLED AS A PART OF THE CLOUDERA HADOOP DISTRIBUTION (STILL AVAILABLE ON THE BIGDATA CLUSTER)
- SWITCHED TO SPARK IN A STANDALONE MODE VIA SLURM ON GENERAL HPC CLUSTERS
  - YARN USES CONTAINERS (SLURM WILL SOON TOO…), ALLOWS DYNAMIC ALLOCATION
  - SLURM IS A BETTER CHOICE FOR US BECAUSE OUR CLUSTERS ARE NOT PURE SPARK OR HADOOP CLUSTERS, AND RESOURCES NEED TO BE SHARED
  - SLURM SOLUTION IS RATHER MATURE: ALLOWS ALLOCATION OF MULTIPLE EXECUTORS PER NODE
  - MOST DIFFICULTIES FOR A USER IS IN MEMORY ALLOCATION
- SEE THE FOLLOWING RESOURCES FOR MORE DETAILS ON SPARK+SLURM:
  - HTTPS://WWW.PRINCETON.EDU/RESEARCHCOMPUTING/FAQ/SPARK-VIA-SLURM/
  - HTTPS://WWW.PRINCETON.EDU/RESEARCHCOMPUTING/COMPUTATIONAL-HARDWARE/HADOOP/SPARK-MEMORY/

ANALYSIS EXAMPLE WITH SPARK, SPARK SQL AND MLLIB

```
def main(argv):
    #STEP1: data ingestion
    sc = SparkContext(appName="KaggleDato_Step2")
    sqlContext = SQLContext(sc)

    #read data into RDD
    input_schema_rdd = sqlContext.read.json("file:///scratch/network/alexeys/KaggleDato/Preprocessed/0_1/part-00000")

    train_label_rdd = sqlContext.read.json(PATH_TO_TRAIN_LABELS)
    sub_label_rdd = sqlContext.read.json(PATH_TO_SUB_LABELS)

    input_schema_rdd.registerTempTable("input")
    train_label_rdd.registerTempTable("train_label")
    sub_label_rdd.registerTempTable("sub_label")

    #Split into 2 subsamples with different label for classification
    train_wlabels_0 = sqlContext.sql("SELECT title,text,images,links,label FROM input JOIN train_label WHERE input.id = train_label.id AND label = 0")
    train_wlabels_1 = sqlContext.sql("SELECT title,text,images,links,label FROM input JOIN train_label WHERE input.id = train_label.id AND label = 1")
    sub_wlabels = sqlContext.sql("SELECT title,text,images,links,label FROM input JOIN sub_label WHERE input.id = sub_label.id")

    text_only_0 = train_wlabels_0.map(lambda p: p.text)
    text_only_1 = train_wlabels_1.map(lambda p: p.text)
    image_only_0 = train_wlabels_0.map(lambda p: p.images)
    image_only_1 = train_wlabels_1.map(lambda p: p.images)
    links_only_0 = train_wlabels_0.map(lambda p: p.links)
    links_only_1 = train_wlabels_1.map(lambda p: p.links)
    title_only_0 = train_wlabels_0.map(lambda p: p.title)
    title_only_1 = train_wlabels_1.map(lambda p: p.title)

    tf = HashingTF(numFeatures=10)
    #preprocess text features
    text_documents_0 = text_only_0.map(lambda line: tokenize(line)).map(lambda word: tf.transform(word))
    text_documents_1 = text_only_1.map(lambda line: tokenize(line)).map(lambda word: tf.transform(word))

    #add the adhoc non-text features
    documents_0 = text_documents_0.zip(image_only_0).zip(links_only_0).zip(title_only_0)
    documents_1 = text_documents_1.zip(image_only_1).zip(links_only_1).zip(title_only_1)

    #turn into a format expected by MLlib classifiers
    labeled_tfidf_0 = documents_0.map(lambda row: parsePoint(0,row))
    labeled_tfidf_1 = documents_1.map(lambda row: parsePoint(1,row))

    labeled_tfidf = labeled_tfidf_0.union(labeled_tfidf_1)
    labeled_tfidf.cache()

    #CV split
    (trainData, cvData) = labeled_tfidf.randomSplit([0.7, 0.3])
    trainData.cache()
    cvData.cache()

    #Try various classifiers
    model = RandomForest.trainClassifier(trainData, numClasses=2, categoricalFeaturesInfo={},
                                         numTrees=3, featureSubsetStrategy="auto",
                                         impurity='gini', maxDepth=4, maxBins=32)

    # Evaluate model on test instances and compute test error
    predictions = model.predict(cvData.map(lambda x: x.features))
    labelsAndPredictions = cvData.map(lambda lp: lp.label).zip(predictions)
    testErr = labelsAndPredictions.filter(lambda (v, p): v != p).count() / float(cvData.count())
    print('Test Error = ' + str(testErr))
    print('Learned classification forest model:')
    print(model.toDebugString())
```

Load scraped data

Select/Join as in Pandas DataFrame are avail starting Spark 1.5

Feature engineering

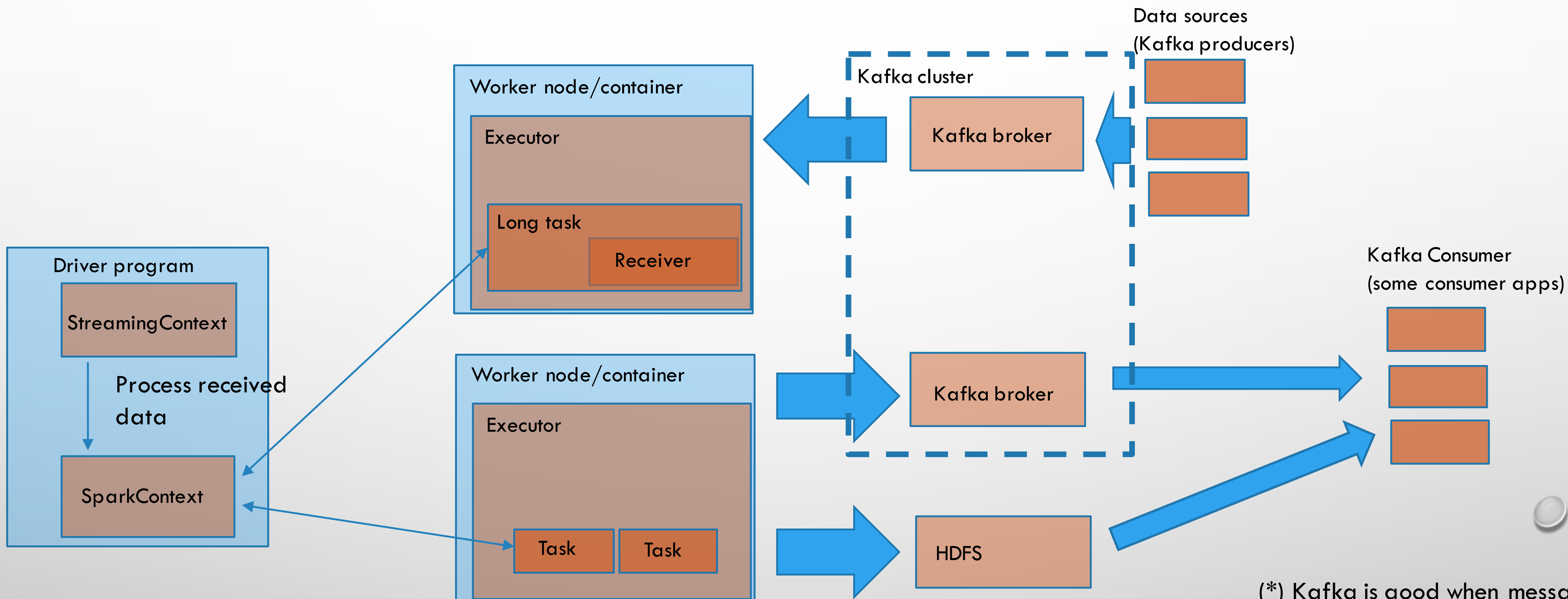
Train/predict

- MLLIB IS THE MAIN SPARK'S MACHINE LEARNING LIBRARY
- IT CONTAINS MOST OF THE ML CLASSIFIERS:
  - LOGISTIC REGRESSION, TREE/FOREST CLASSIFIERS, SVM, CLUSTERING ALGORITHMS…
  - INTRODUCES NEW DATA FORMAT: LABELEDPOINT (FOR SUPERVISED LEARNING)
- AS AN EXAMPLE, LETS US TAKE A KAGGLE COMPETITION:
  - HTTPS://WWW.KAGGLE.COM/C/DATO-NATIVE
- DATASET FOR THAT COMPETITION CONSISTED OF OVER 300K RAW HTML FILES CONTAINING TEXT, LINKS, AND DOWNLOADABLE IMAGES
  - THE CHALLENGE WAS TO IDENTIFY THE PAID CONTENT DISGUISED AS JUST ANOTHER INTERNET GEM (I.E. "NON-NATIVE" ADVERTISEMENT)
- ANALYSIS FLOW:
  - SCRAPE THE DATA FROM THE WEB-PAGES: TEXT, IMAGES, LINKS…
  - FEATURE ENGINEERING: EXTRACT FEATURES FOR CLASSIFICATION
  - TRAIN/CROSS VALIDATE A MACHINE LEARNING MODEL
  - PREDICT

REAL-TIME ANALYSES WITH APACHE SPARK

- MANY APPLICATIONS BENEFIT FROM ACTING ON DATA AS SOON AS IT ARRIVES
  - NOT A TYPICAL CASE FOR PHYSICS ANALYSES AT CMS…
  - HOWEVER, IT COULD BE A PERFECT FIT FOR EXOTICA HOTLINE OR ANY OTHER “HOTLINE” TYPE SYSTEMS OR ANOMALY DETECTION DURING DATA COLLECTION
- SPARK STREAMING USES A CONCEPT OF DSTREAMS (SEQ. OF DATA ARRIVING OVER TIME)
  - INGEST AND ANALYZE DATA COLLECTED OVER A BATCH INTERVAL
- SUPPORTS VARIOUS INPUT SOURCES: AKKA, FLUME, KAFKA, HDFS
- CAN OPERATE 24/7, BUT IS NOT TRULY REAL-TIME (LIKE E.G. APACHE STORM) – IT IS A MICRO-BATCH SYSTEM WITH A FIXED (CONTROLLED) BATCH INTERVAL
- FULLY FAULT TOLERANT, OFFERING “EXACTLY ONCE” SEMANTICS, SO THAT THE DATA WILL BE ANALYSED FOR SURE EVEN IF A NODE FAILS
  - SUPPORTS CHECKPOINTING – ALLOWING TO RESTORE DATA FROM A GIVEN POINT IN TIME

EXAMPLE OF A REAL-TIME PIPELINE USING APACHE KAFKA AND SPARK STREAMING



(*) Kafka is good when message ordering matters

(*) Spark+Spark Streaming – use same programming model for online and offline analysis

SUMMARY AND POSSIBLE USE CASES IN HEP

- PRINCETON BIG DATA EXPERIENCE:
  - SET UP AND DEPLOYED A HADOOP CLUSTER STARTING WITH THE CLOUDERA DISTRIBUTION USING THE HIGH AVAILABILITY CONFIGURATION (A TOPIC FOR A SEPARATE TALK…)
  - PROVIDED A SOLUTION TO RUN SPARK ON NON-HADOOP CLUSTERS VIA SLURM:
    - HTTPS://WWW.PRINCETON.EDU/RESEARCHCOMPUTING/FAQ/SPARK-VIA-SLURM/
    - HTTPS://WWW.PRINCETON.EDU/RESEARCHCOMPUTING/COMPUTATIONAL-HARDWARE/HADOOP/SPARK-MEMORY/
  - GAINED SOME EXPERIENCE IN PYSPARK PROGRAMMING
  - CURRENTLY USED BY: POLITICS, CS AND EXPECTING ENVIRONMENTAL ENGINEERING GROUP
- POSSIBLE USE CASES IN HEP TO EXPLORE:
  - ABILITY TO PERFROM OFFLINE (PYTHON) ROOT BASED ANALYSES USING APACHE SPARK
  - UNDERSTAND IF ROOT AND SPARK ARE INTEROPERABLE
  - REAL-TIME SYSTEMS WITH SPARK STREAMING/KAFKA: E.G. EXOTICA HOTLINE, HLT

BACKUP

SAMPLE SLURM SUBMISSION SCRIPT FOR A SPARK APP

```
#!/bin/bash
#SBATCH -N 4
#SBATCH -t 06:00:00
#SBATCH --ntasks-per-node 2
#SBATCH --cpus-per-task 4
#SBATCH --mem=15000
module load spark/hadoop2.6/1.4.1
export PYSPARK_PYTHON=/usr/bin/python2.7
export SPARK_LOG_DIR=/tmp/logs
export SPARK_WORKER_DIR=/tmp/work
export SPARK_LOCAL_DIRS=$SPARK_WORKER_DIR
mkdir -p $SPARK_LOG_DIR $SPARK_WORKER_DIR
start-master.sh
sleep 15s
export MASTER=spark://`hostname`:7077
echo $MASTER
srun spark-class org.apache.spark.deploy.worker.Worker $MASTER -d $SPARK_WORKER_DIR &
# sleep a bit to let workers start up fully
sleep 15s
spark-submit --executor-memory 5G --total-executor-cores 32 --py-files SparkyBillAnalysisTools.py,scam_dist.py findLikeBills_spark.py 10 10 --prefix_from /scratch/network/alexeys/RandomSparkTests/t
ext_3states_partitioned/ --prefix_to /scratch/network/alexeys/matches/ --minMatchThr 30 --similarMeasure default2
```