Spark: Big Data processing framework

Troy Baer[1], Edmon Begoli[2,3], Cristian Capdevila[2], Pragnesh Patel[1], Junqi Yin[1]

1. National Institute for Computational Sciences, University of Tennessee
2. PYA Analytics
3. Joint Institute for Computational Sciences, University of Tennessee

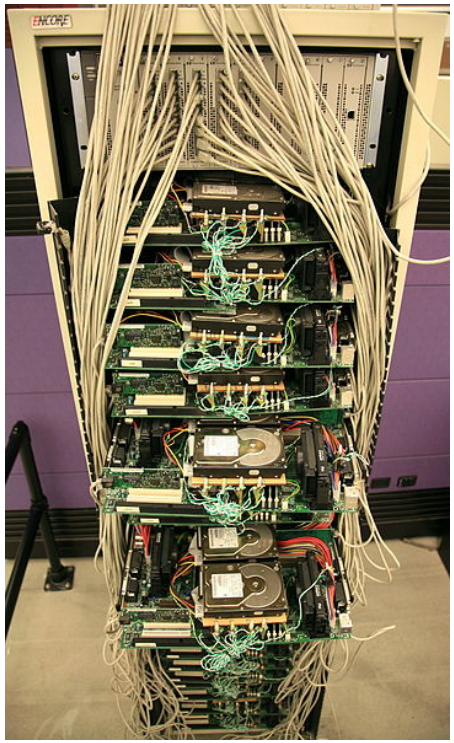
XSEDE Tutorial, July 26, 2015

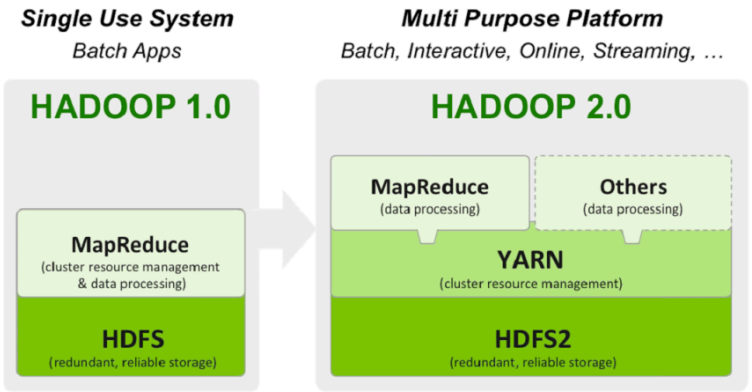
Outline

- Overview of Big Data processing framework
- Introduction to Spark and Spark deployment
- Introduction to Spark SQL and Streaming
- Hands-on
- Spark machine learning and graph libraries
- Hands-on

Overview of Big Data processing framework

A brief history of Hadoop





Why large scale ('Big Data') analytics

- Heterogeneity of the architecture
  - Mixed loads
  - Mixed types
- Comprehensiveness of the analytic tools
  - SQL
  - Machine Learning
  - Data manipulation
  - Programming
  - External libraries
- "Safe-bet" for the future

A general case for ‘Big Data’ in healthcare

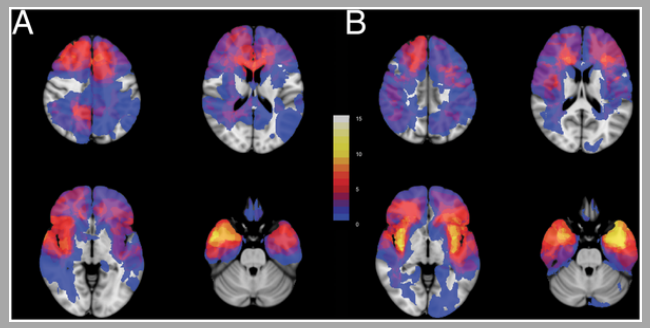
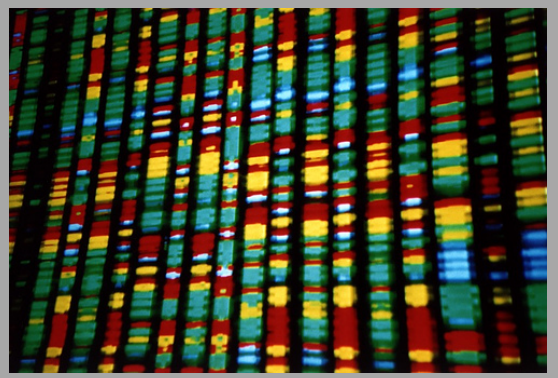


Image and sensor data



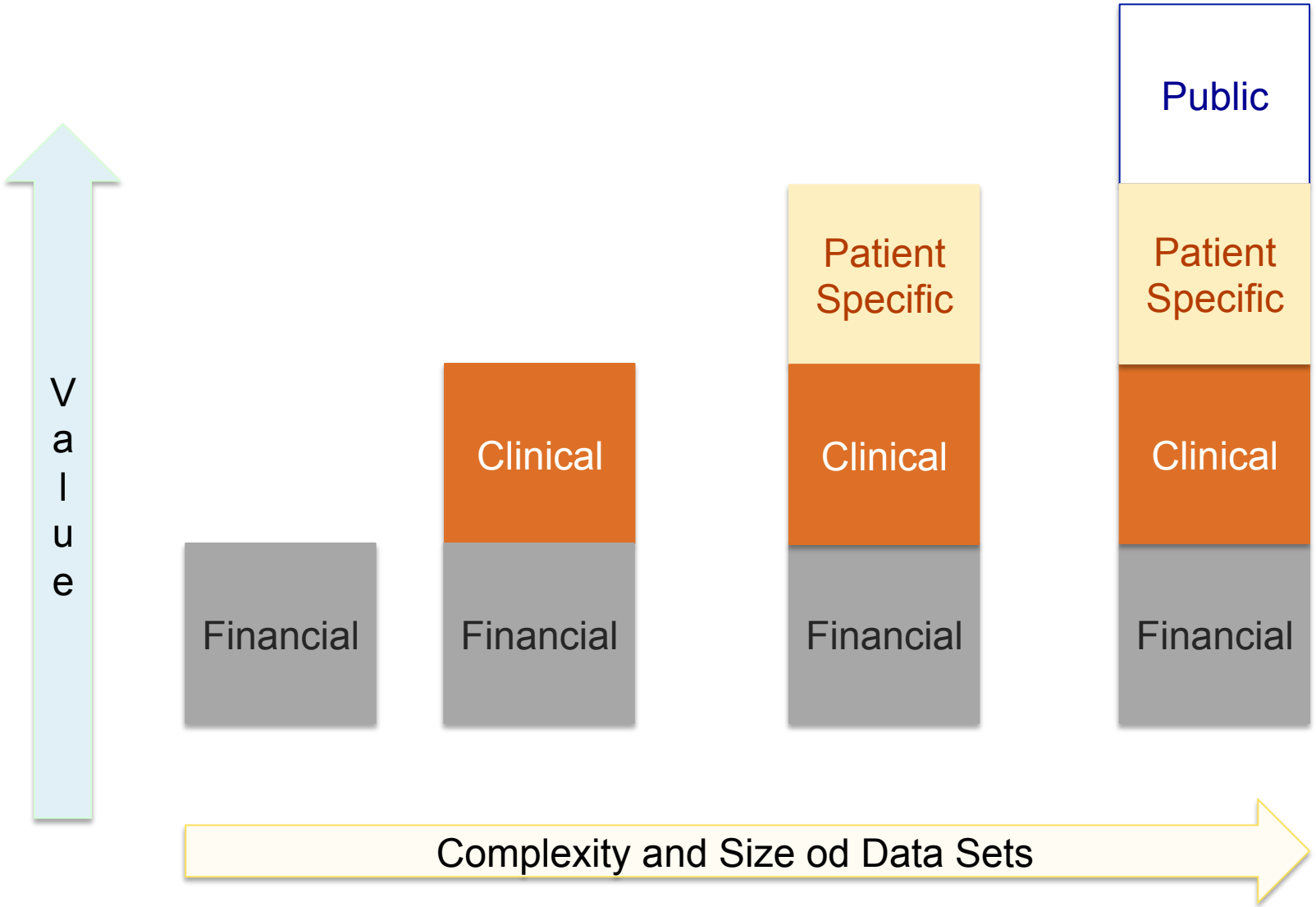
Personal/genomic data

‘Big Data’ Platform

Financial and administrative data

Clinical data

Value of data aggregation

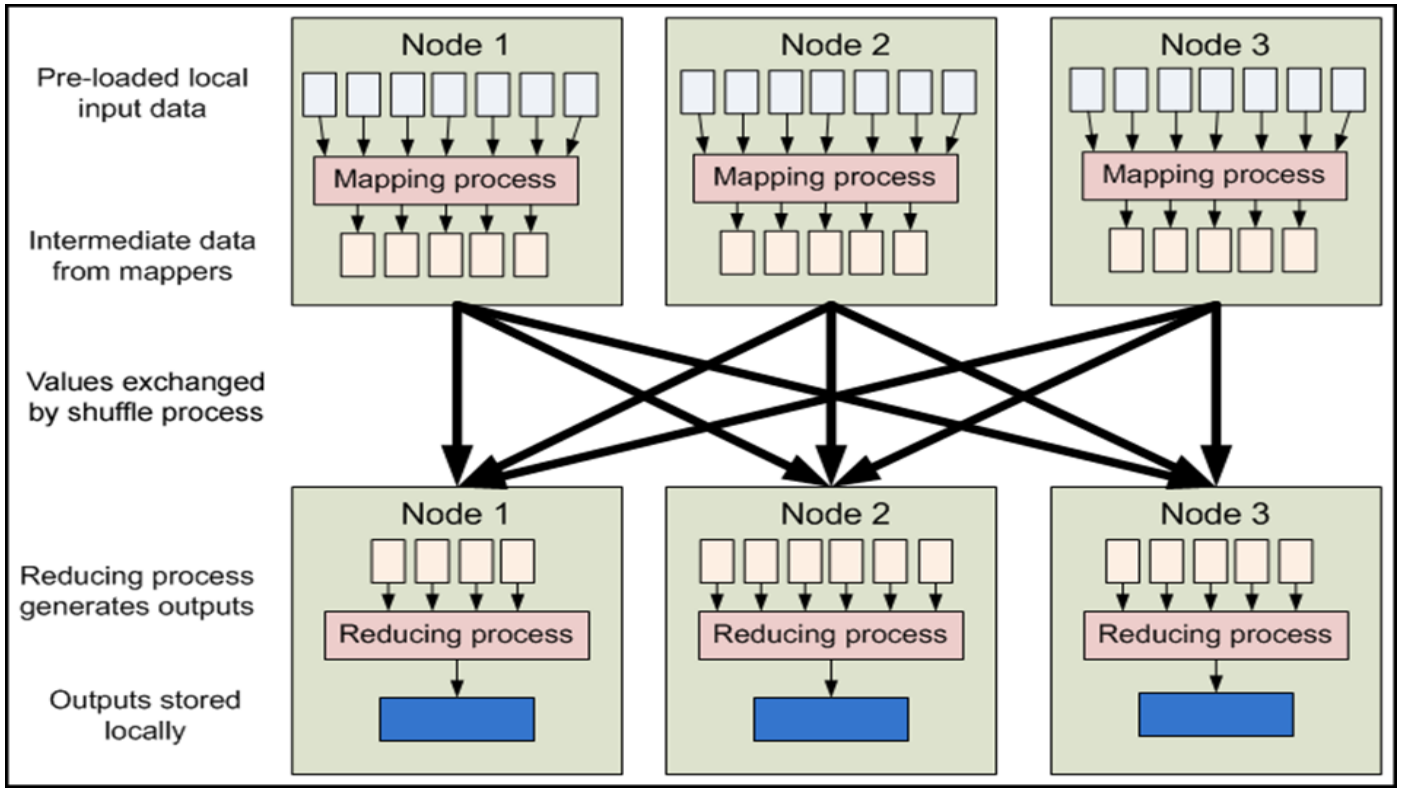


Hadoop: Big Data Platform

- Hadoop has become the de-facto platform for storing and processing large amounts of data.

1. Storage managers : HDFS, HBASE, Kafka, etc.
2. Processing framework: MapReduce, Spark, etc.
3. Resource managers: Yarn, Mesos, etc.

MapReduce programing model (2004)

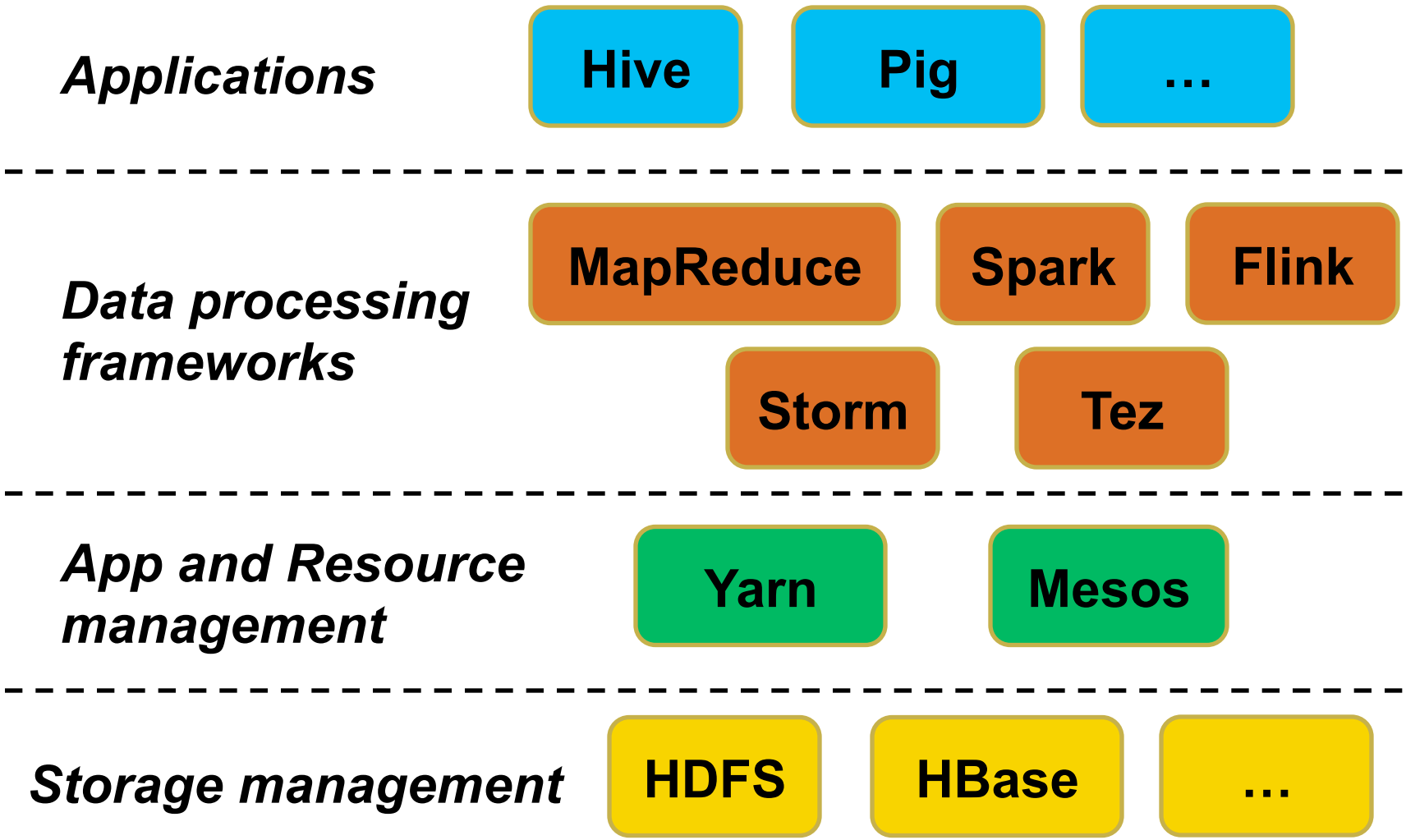


- Mappers perform a transformation
- Reducers perform an aggregation

Beyond MapReduce paradigm

- Extends map/reduce model to Directed Acyclic Graph (DAG) model with backtracking-based recovery (Apache Tez 2007)
- Implementation of above recovery with Resilient Distributed Datasets (RDD) (Apache Spark 2010)
- Extend DAG model to cyclic graphs (Apache Flink 2012)

Hadoop ecosystem



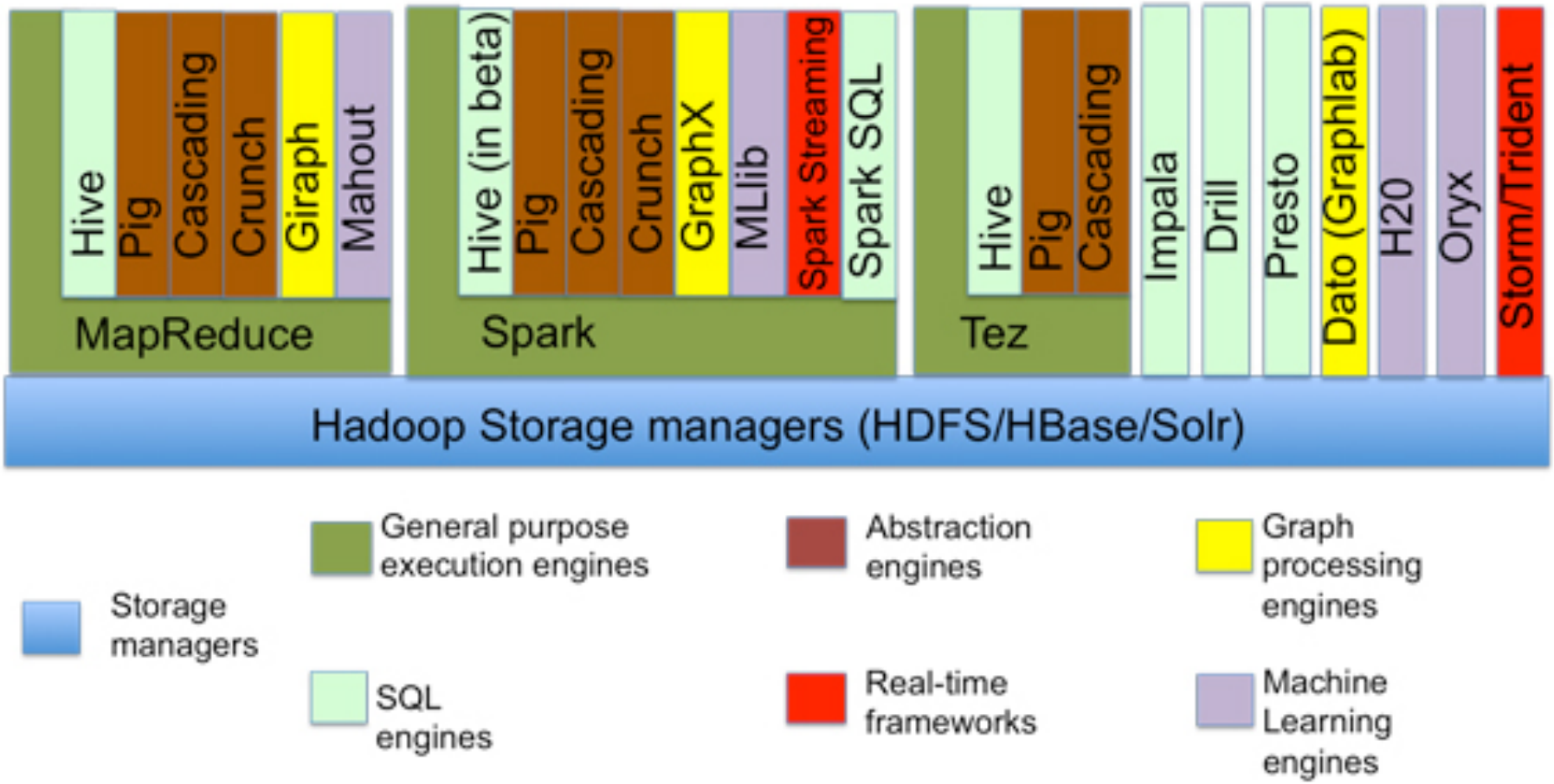
Overview of Big Data processing frameworks

- Categories of processing frameworks
  1. General-purpose processing framework: allow users to process data using a low-level API, e.g. MapReduce, Spark and Flink
  2. Abstraction frameworks: allow users to process data using a higher level abstraction, .e.g. Pig
  3. SQL frameworks: enable querying data, e.g. Hive, Impala

Overview of Big Data processing frameworks

- Categories of processing frameworks
  4. Graph processing frameworks: enable graph processing capabilities, e.g. Giraph
  5. Machine learning frameworks: enable machine learning analysis, e.g. MLlib, Oryx
  6. Real-time/streaming frameworks: provide near real-time processing (several hundred milliseconds to few seconds latency), e.g. Spark Streaming, Storm

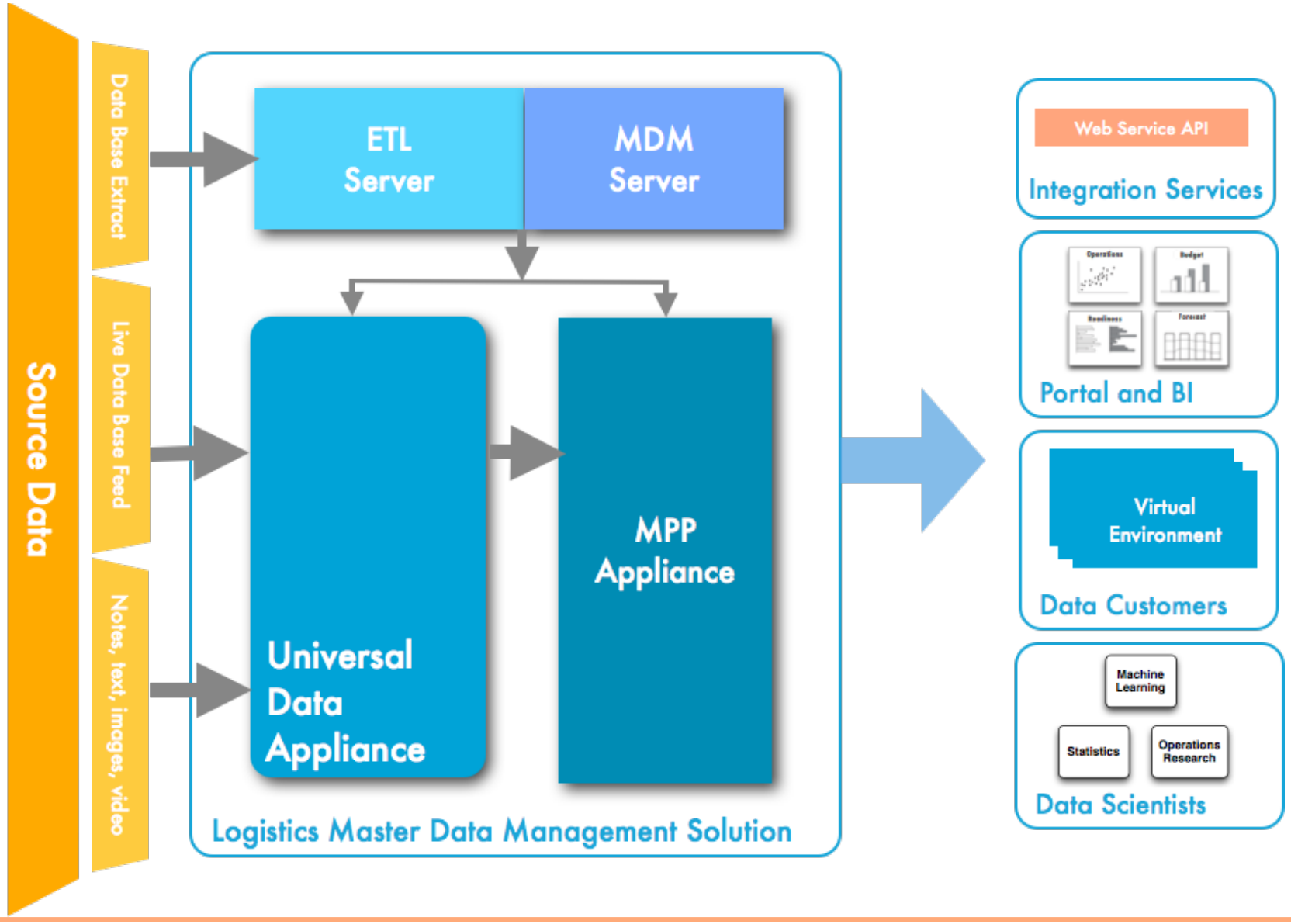
Overview of Big Data processing framework



Uses in the real world – observed Architectures

- Three models in the modern enterprises
  - MPP+Hadoop as NAS/ETL
  - Hadoop/Spark as a first class citizen for analytic discovery
  - *Facebook* architecture – the only citizen

Stereotypical Architecture



Some Interesting Emerging Use Cases

- Near-real time decision making in healthcare regarding critical medical conditions
- Bioinformatics and genomic applications (ADAM, SparkSeq)
- Authorship graph (PYA Analytics, later)

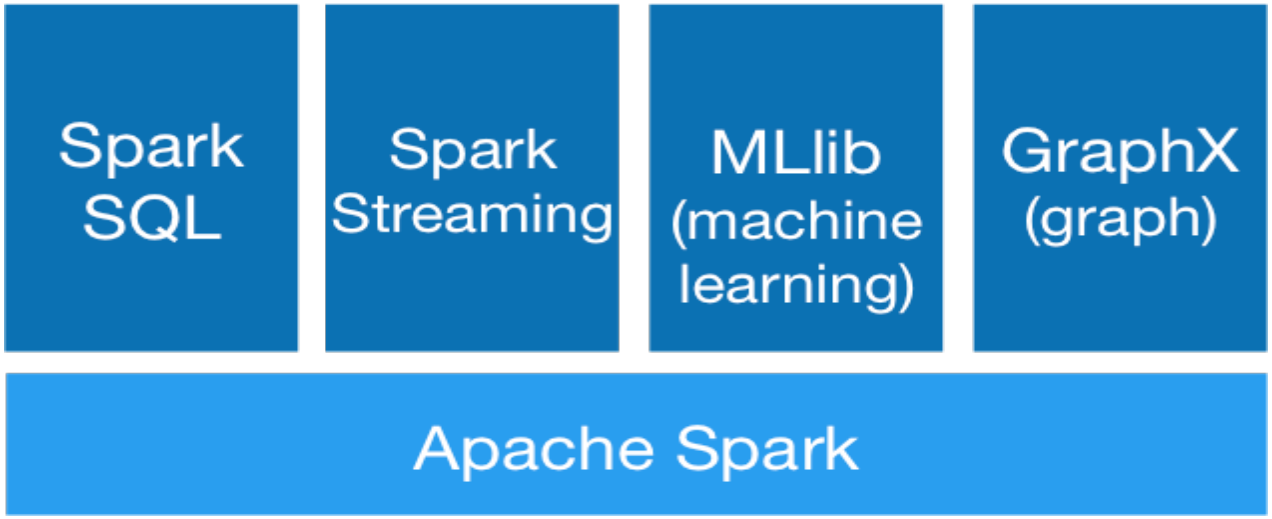
Introduction to Spark and Spark deployment

Spark: Big Data processing framework

- Spark is fast, general-purpose data engine that supports cyclic data flow and in-memory computing.
- Advantages over Hadoop MapReduce

1. Speed : 100x faster in memory; 10x faster on disk
2. Ease of use: supports program in Java, Scala or Python

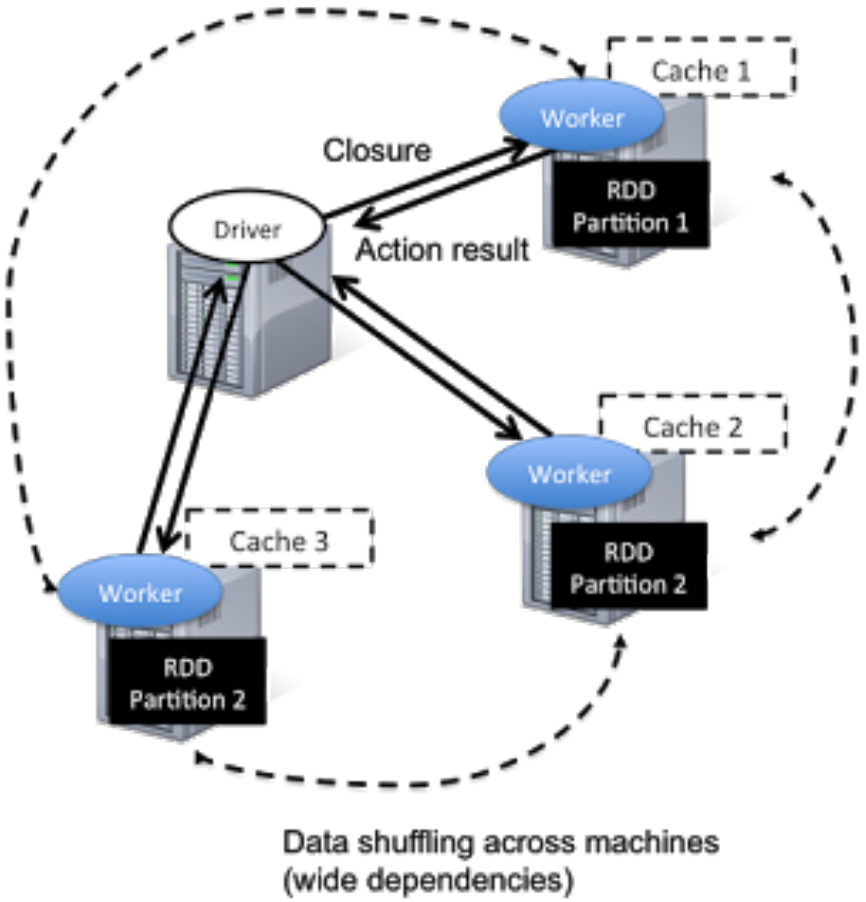
Spark: Big Data processing framework



- **Advantages over Hadoop MapReduce**
  - **Generality: combines SQL, streaming, machine learning and graph processing capabilities**
  - **Speed**
  - **Compatibility: runs on Hadoop, standalone, or cloud**

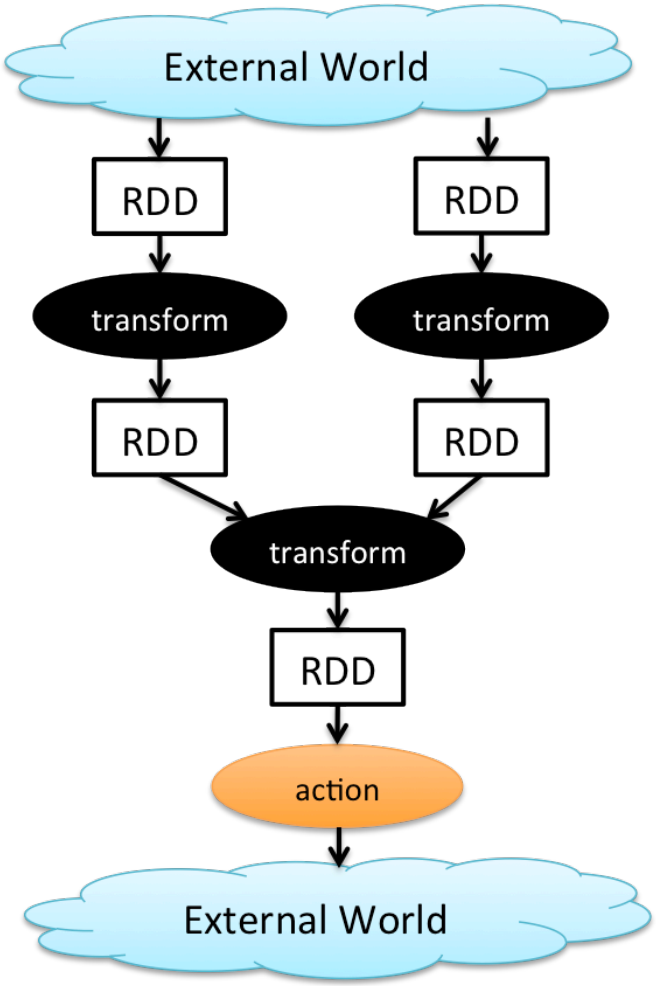
Spark Architecture

- **Resilient Distributed Datasets (RDD) : distributed objects that can be cached in-memory, across a cluster of compute nodes.**
- **Directed Acyclic Graph (DAG) execution engine that eliminates the MapReduce multi-stage execution model**

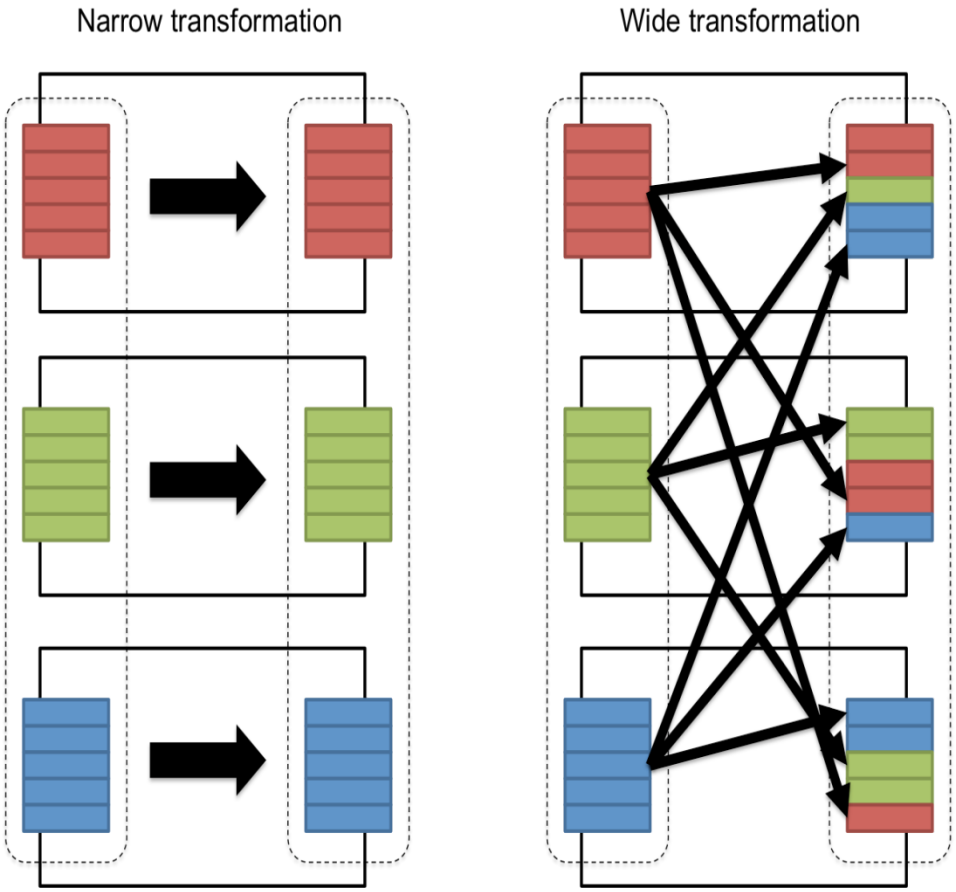


Operations on RDD

- Transformation: specifies the processing dependency DAG among RDDs
- Action: specifies what the output will be
- The scheduler performs a topology sort to determine the execution sequence of the DAG



Operations on RDD



- Narrow transformation (involves no data shuffling): Map, FlatMap, Filter, and Sample
- Wide transformation (involves data shuffling): SortByKey, ReduceByKey, GroupByKey, Join, etc
- Action: Collect , Reduce, ForEach, Count, Save, etc.

Operations on RDD

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Spark examples

- **In-Memory Text Search**

```
val file = spark.textFile("hdfs://...")
val errors = file.filter(line => line.contains("ERROR"))
                 .cache()
// Count all the errors
errors.count()
// Count errors mentioning MySQL
errors.filter(line => line.contains("MySQL")).count()
// Fetch the MySQL errors as an array of strings
errors.filter(line => line.contains("MySQL")).collect()
```

Spark examples

- **Word Count**

```
val file = spark.textFile("hdfs://...")

val counts = file.flatMap(line => line.split(“ "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)

Counts.saveAsTextFile(“hdfs://…”)
```

Spark Deployment Modes

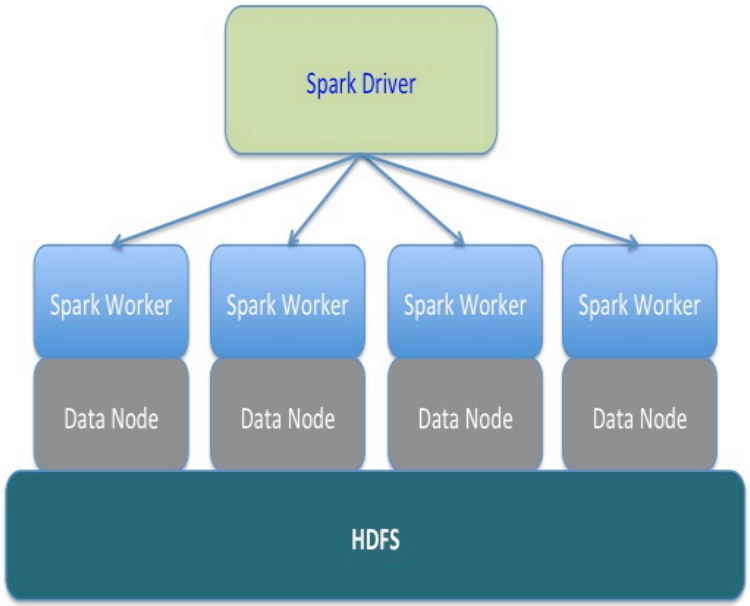
- Spark local
- Spark with HDFS and YARN or MESOS Res. Mgrs.
- Spark on HPC (Thursday)
- Spark and Tungsten (new research)

Spark Standalone

- Core Spark
- Local file system, compute and memory space
- Practical for learning and monster CPU/RAM machines

Spark deployment on Cluster

- In cluster mode, the Spark runtime environment consists of a driver program, a cluster manager, and workers on compute nodes.
- Cluster manager is responsible for allocating and starting workers as well as coordinating communication between the driver program and workers.



Spark Architecture

- Spark supports YARN, Mesos, and standalone mode.

Spark deployment on Cluster

- At NICS, we provides “Spark on demand” within the context of a PBS batch or interactive job.
- The key tool is pbs-spark-submit (http://svn.nics.tennessee.edu/repos/pbstools/trunk/bin/pbs-spark-submit)
- pbs-spark-submit work flow:

1. Verifies execution is inside a PBS job
2. Sets default value for a number of environment variables

Spark deployment on Cluster

- pbs-spark-submit work flow:

3. Determines what shared and node-local directories to use.

4. Parses its command line options and updates settings

5. Parses any Java property files found in its configuration directory

6. Launches the Spark master and worker daemons

7. Executes user's Spark driver program

Spark deployment on Cluster

For more details, please attend the talk on "Integrating Apache Spark Into PBS-Based HPC Environments" by Edmon Begoli

Introduction to Spark SQL and Streaming

Spark SQL

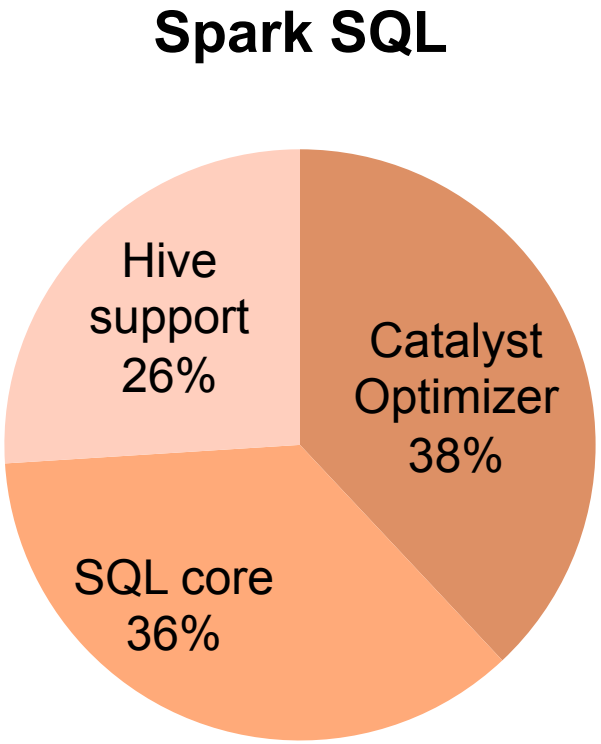
- Load data from a variety of structured sources
  - JSON, Hive, and Parquet
- Query data using SQL
  - From inside a Spark program
  - From external tools that connect through JDBC/ODBC
- Rich integration between SQL and Scala/Java/Python
  - Join RDDs and SQL tables
  - Custom functions in SQL

Spark SQL

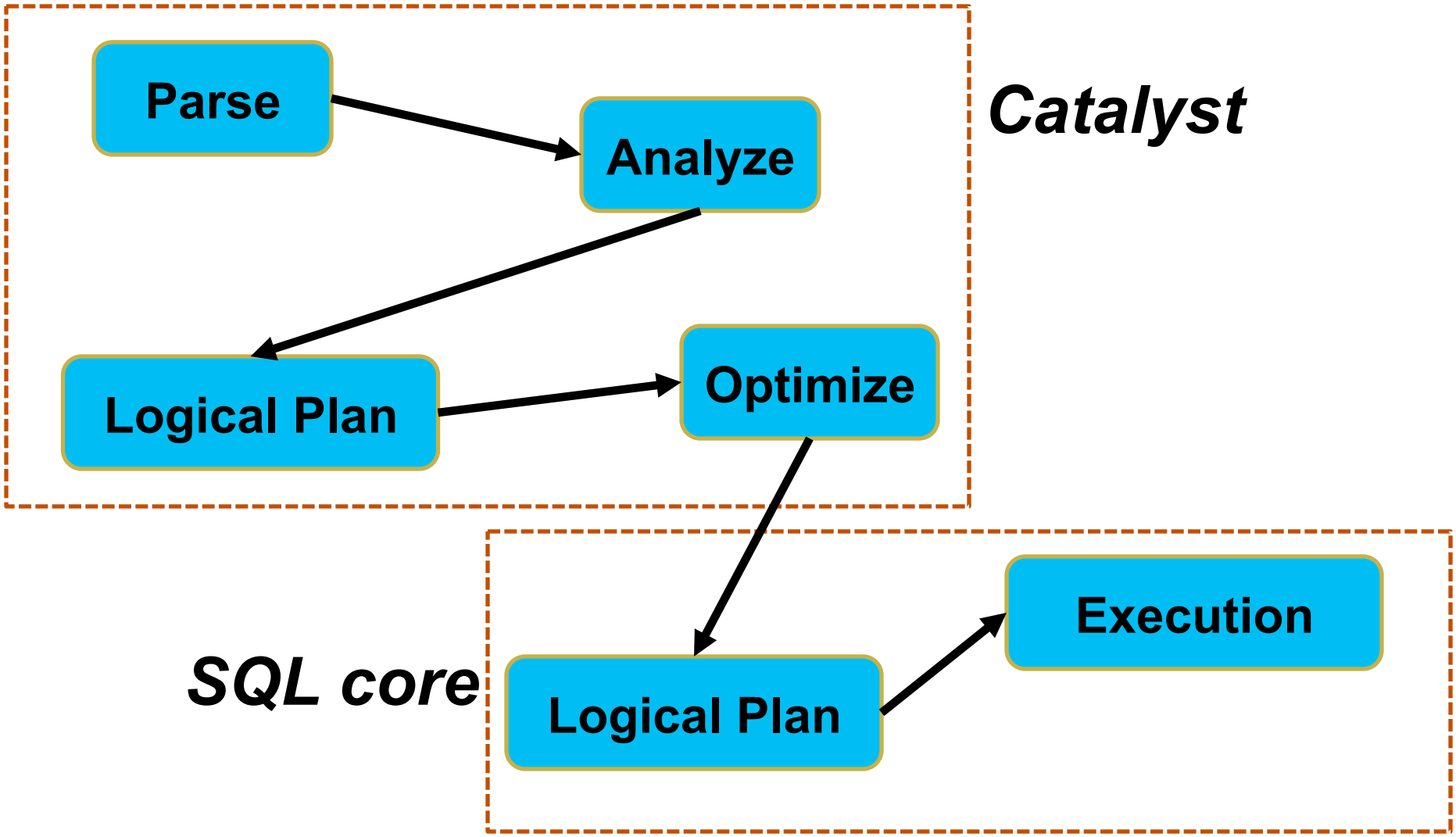
- Shark, a backend modified Hive running over Spark.
  - Limited integration with Spark
  - Hive optimizer not designed for Spark
- Spark SQL reuses parts of Shark,
  - Hive data loading
  - In-memory column store
- Spark SQL also adds
  - RDD-aware optimizer
  - Rich language interfaces

Spark SQL components

- **Catalyst Optimizer**
  - Relational algebra plus expressions
  - Query optimization
- **Spark SQL core**
  - Execution of queries as RDDs
  - Reading in Parquet, JSON, etc
- **Hive Support**
  - HQL, MetaStore, SerDes, UDFs



Query execution steps



Using Spark SQL

- **SQLContext**
  - **Entry point for all SQL functionality**
  - **Wraps/extends existing spark context**

```
val sc: SparkContext // An existing SparkContext
val sqlContext = new
org.apache.spark.sql.SQLContext(sc)
// importing SQL context gives access to all SQL
functions
import sqlContext._
```

DataFrame

- A DataFrame is a distributed collection of data organized into named columns
- Equivalent to table in relational database or data frame in R/Python, but with richer optimizations
- DataFrame API is available in Scale/Java/Python
- A DataFrame can be created from an existing RDD, a Hive table, or data sources.

DataFrame operations

```
// Create the DataFrame from data source
val df = sqlContext.jsonFile("people.json")
// show the content of the DataFrame
df.show()
//age name
// null Michael
// 30 Andy
// print the schema in  a tree format
df.printSchema()
//root
// /-- age: long (nullable = true)
// /-- name: string (nullable = true)
```

DataFrame operations

```
// select only the “name” column
df.select(“name”).show()
//name
//Michael
// Andy
// select people older than 21
df.filter(df(“age”) > 21).show()
//age name
// 30 Andy
// count people by age
df.groupBy(“age”).count().show()
//age name
```

Turning RDD into DataFrame

```
// Define the schema using a case class
case class Person(name: String, age: Int)

// Create an RDD of Person objects
val people =
    sc.textFile("people.txt")
      .map(_.split(","))
      .map(p => Person( p(0), p(1).trim.toInt)).toDF()

//register RDD as an table
people.registerTempTable("people")
```

Querying using SQL

```
// SQL statements are run with the sql method from
// sqlContext
val teenagers = sql("SELECT name FROM people
                     WHERE age>=10 AND age<=19")

// The results of query are SchemaRDDs but also
// support normal RDD operations.
// The colums of a row in the result are accessed by
// ordinal
val nameList= teenagers.map(t => "Name: " + t(0))
                       .collect().foreach(println)
```

Querying using Scala DSL

- **Express queries using functions, instead of SQL strings**

```
// The following is the same as:
//  SELECT name FROM people
//  WHERE age >= 10 AND age <=19

val teenagers = people.where('age >= 10)
                      .where('age <= 19)
                      .select('name)
```

Caching Table in memory

```
// Spark SQL can cache tables using in memory
//  columnar format:

cacheTable("people") or people.cache()
```

- **Scan only required columns**
- **Fewer allocated objects**
- **Automatically selects best compression**

Parquet Compatibility

- **Native support for reading data in Parquet**
  - **Columnar storage avoids reading unneeded data**
  - **RDDs can be written to parquet files, preserving the schema**

```
// SchemaRDD can be stored as Parquet
people.write.parquet("people.parquet")
// Parquet files are self-describing
val parquetFile=sqlContext.parquetFile("people.parquet")

//Parquet files can be used in SQL statements
parquetFile.registerTempTable("people")
val teenagers = sql("SELECT name FROM people
                     WHERE age>=10 AND age<=19")
```

Hive compatibility

- Support for writing queries in HQL
- Catalog info from Hive MetaStore
- Tablescan operator that uses Hive SerDes
- Wrappers for Hive UDFs, UDAFs, UDTFs

```
val hiveCtx = new org.apache.spark.sql.hive.HiveContext(sc)
hiveCtx.sql("CREATE TABLE IF NOT EXISTS src (key INT,
value STRING)")
hiveCtx.sql("LOAD DATA LOCAL INPATH ''kv1.txt' INTO
TABLE src)
hiveCtx.sql("FROM src SELECT key, value").collect()
```

Spark SQL UDFs

- **Build-in method to easily register UDFs**

```
//Make a UDF to tell us how long some text is
sqlContext.udf.register("strLen", (s:String) => s.length() )

sqlContext.sql( " SELECT strLen('name') FROM
people).collect().foreach(println)
```

- **Use existing Hive UDFs**

```
hiveCtx.sql("CREATE TEMPORARY FUNCTION name
                AS class.function")
```

JDBC server and Beeline client

- **Spark SQL provides JDBC connectivity, which is useful for connecting MySQL, PostgreSQL, etc database and business intelligence tools.**

```
//lanching the JDBC server
start-thrifserver.sh --master local
//connecting to the JDBC server with Beeline
 beeline -u jdbc:hive2://localhost:10000
```

- **Beeline client supports HiveQL commands to create, list and query tables (see example in Hive compatibility slide)**

Performance Tuning

```
// Performance options in Spark SQL
Option                                          Defaut
spark.sql.codegen                               false

spark.sql.
inMemoryColumnarStorage                         false
.compressed

spark.sql.
inMemoryColumnarStorage                         1000
.batchSize

spark.sql.parquet.compression.codec             snappy
```

Spark Streaming

- Why Streaming?
  - Many big-data applications need to process large data streams in real-time
  - Website monitoring; Fraud detection; Ads monetization
- Why Spark Streaming?
  - Easy of use: build applications through high-level operators
  - Fault tolerance: stateful exactly-once semantics out of the box
  - Spark integration: combine streaming with batch and interactive queries

Other Streaming Systems

- Storm
  - Replys record if not processed by a node
  - Processes each record at least once
  - May update mutable state twice
  - Mutable state can be lost due to failure
- Trident
  - Processes each record exactly once
  - Use transactions to update state
  - Per-state transaction to external database is slow

Spark Streaming

Input sources

- **File stream, e.g. monitoring a log directory**

```
// The files have to be created atomically:
val ssc = new StreamingContext(conf, Seconds(5))
val logData = ssc.textFileStream(logDirectory)
```

- **Socket**

```
// The files have to be created atomically:
val lines = ssc.sockeTextStream(“localhost”, 7777)
```

Input sources

- **File stream, e.g. monitoring a log directory**

```
// The files have to be created atomically:
val ssc = new StreamingContext(conf, Seconds(5))
val logData = ssc.textFileStream(logDirectory)
```

- **Socket**

```
// The files have to be created atomically:
val lines = ssc.sockeTextStream("localhost", 7777)
```

Input sources

- **Apache Kafka**

```
import org.apache.spark.streaming.kafka._
// create a map of topic
val topics = List(…).toMap
val topicLines = KafkaUtils.createStream(ssc, host,
group, topics)
```

- **Apache Flume**

```
// FlumeUtils agent
val events = FlumeUtils.createStream(ssc,
receiverHostname, receiverPort)
val lines = events.map{…}
```

Stateless Transformations

- **Transformations of the data batch does not depend on the data of its previous batches.**

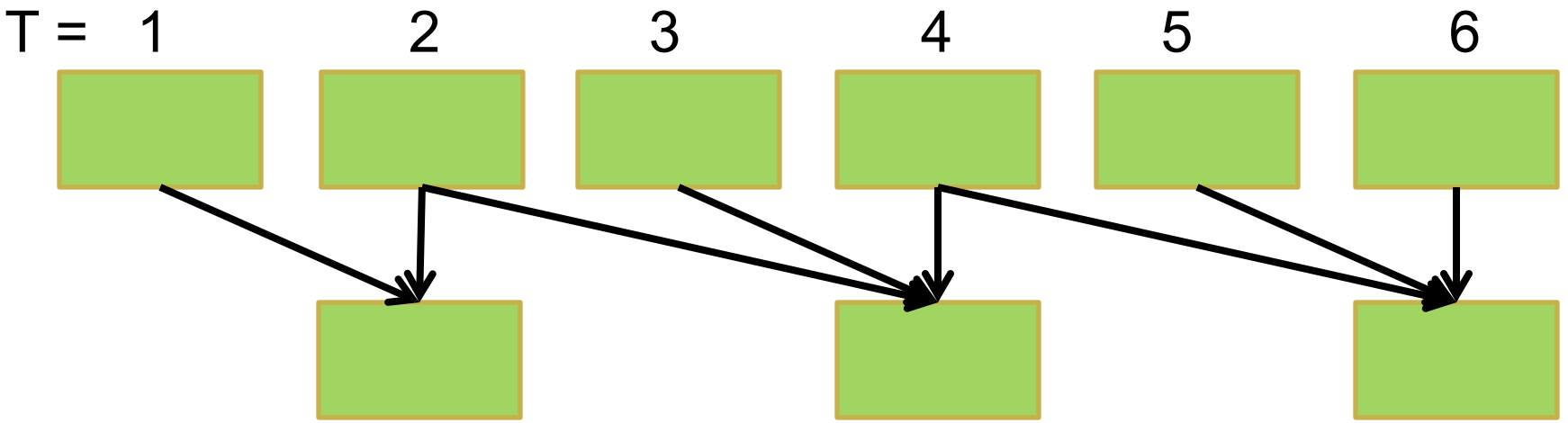
Function	*example*
map()	ds.map(x => x + 1)
flatMap()	ds.flatMap(x => x.split(“ “))
filter()	ds.filter(x => x != 1)
repartition()	ds.repartition(10)
reduceByKey()	ds.reduceByKey((x,y) => x + y)
groupByKey()	ds.groupByKey()

Stateful Transformations

- **Transformations use data or intermediate results from previous batches**

Function	*example*
window()	ds.window(Secons(10), Seconds(5))
reduceByWindow()	ds.flatMap(x => x.split(“ “))
reduceByKeyAndWindow()	ds.filter(x => x != 1)
CountByWindow()	ds.repartition(10)
CountByValueAndWindow()	ds.reduceByKey((x,y) => x + y)

Stateful Transformations



Windowed Stream: window = 3 ; slide = 2

```
// source DStream with a batch interval of 10 seconds
val logWindow = logDStream.window(Seconds(30),
                                  Seconds(10))
val windowCounts = logWindow.count()
```


NICS

Checkpointing and Fault Tolerance

- **Periodically save data about the application to a reliable storage system.**
- **Limiting the state that must be recomputed on failure.**
- **Providing fault tolerance for the driver.**

```
// setting up checkpointing by passing a path :
// either HDFS, S3, or local filesystem
ssc.checkpoint("hdfs://…")
```

Checkpointing and Fault Tolerance

- **Driver fault tolerance requires a special way of creating StreamingContext.**

```
// setting up fault tolerance driver :
def createStreamingContext() = {
        …
        val sc = new SparkContext(conf)
        val ssc = new StreamingContext(sc, Seconds(5))
        ssc.checkpoint(checkpointDir)
}
val ssc = StreamingContext.getOrCreate
        (checkpointDir, createStreamingContext _)
```

Performance Considerations

- Batch and window sizes have big impact on performance. Start with a larger batch size or slide interval (say, 10 seconds), and gradually decrease to a smaller size.
- Increase parallelism: number of receivers (create multiple input DStreams and then merge them into one);repartion received data (Dstream.repartition); parallel aggregation (specify parallelism as a second parameter for function such as reduceByKey())
- Use concurrent Mark-Sweep garbage collector

Hands-on session: Spark deployment

- Instructions for installing spark on Linux(Ubuntu)

1. Install java

```
sudo apt-get install oracle-java7-installer
```

2. Download spark 1.4.0 prebuild binary

```
wget
http://mirror.symnds.com/software/Apache/spark/
spark-1.4.0/spark-1.4.0-bin-hadoop2.4.tgz
```

3. Add spark bin and sbin to your PATH

```
tar -xf  spark-1.4.0-bin-hadoop2.4.tgz

export PATH=`pwd`/spark-1.4.0-bin-hadoop2.4/bin:
`pwd`/spark-1.4.0-bin-hadoop2.4/sbin:$PATH
```

Hands-on session: Spark deployment

- Instructions for installing spark on Linux(Ubuntu)

4. Install sbt

```
echo "deb http://dl.bintray.com/sbt/debian /" | sudo tee -
a /etc/apt/sources.list.d/sbt.list

sudo apt-get update

sudo apt-get install sbt
```

5. Run spark-shell in terminal

Spark installation for Mac OS X

- Java installation: https://www.java.com/en/download/help/mac_install.xml
- Spark installation: http://spark.apache.org/downloads.html
  - (1) Choose a Spark release: 1.4.0
  - (2) Choose a package type: Pre-built for Hadoop 2.6 and later
  - (3) Choose a download type: Direct Download
  - (4) Download Spark:spark-1.4.0-bin-hadoop2.6.tgz [Click to begin download]
  - (5) > tar xvf spark-1.4.0-bin-hadoop2.6.tgz
  - (6) > export PATH=location_of_spark_bin_dir:$PATH
  - (7) > spark-shell

Scala SBT(Simple Build Tool) installation for Mac OS X

- If you don't have *brew* install then download from http://brew.sh/

  > brew install sbt

  OR

- If you don't have Mac Ports then follow instuctions given here to install it : https://www.macports.org/install.php

  > port install sbt

- To build Spark and its example programs, run:

  > ./sbt/sbt assembly

Hands-on session: Spark deployment

- Instructions for installing spark on Windows

1. Install java

http://www.java.com/en/download/help/windows_manual_download.xml

2. Download spark 1.4.0 prebuild binary

http://mirror.symnds.com/software/Apache/spark/spark-1.4.0/spark-1.4.0-bin-hadoop2.4.tgz

(2). Download and install 7-Zip to unzip spark tar ball

Hands-on session: Spark deployment

3. Add “spark-1.4.0-bin-hadoop2.4/bin” and “C:\Windows\system32” to the PATH

Control Panel->System and Security->System->Change settings -> Advanced -> Environment Variables

4. Install SBT tool

http://www.scala-sbt.org/release/tutorial/Installing-sbt-on-Windows.html

6. Install ncat tool

https://nmap.org/dist/nmap-6.49BETA4-setup.exe

5. run spark-shell in “cmd”

Hands-on session: SQL and Streaming

- Spark shell practice SQL with data file people.json
- Spark Streaming examples:

a) HdfsWordCount (file stream)

b) NetworkWordCount (socket stream)

c) SqlNetworkWordCount (combine SQL with Streaming)

- Build spark standalone applications

1. Create a project directory, say “MySparkExample”
2. mkdir -p src/main/scala
3. cp spark-1.4.0-bin-hadoop2.4/examples/src/main/scala/org/apache/spark/examples/streaming/(a,b,c and StreamingExamples.scala) src/main/scala

Hands-on session: SQL and Streaming

4. cat build.sbt

```
name := "MySparkExample"

version := "1.0"

scalaVersion := "2.10.4"

libraryDependencies += "org.apache.spark" %% "spark-core" % "1.4.0"

libraryDependencies += "org.apache.spark" %% "spark-streaming" % "1.4.0"

libraryDependencies += "org.apache.spark" %% "spark-sql" % "1.4.0"
```

Hands-on session: SQL and Streaming

5. sbt clean package

6. Run the applications

a) spark-submit --master local[2] --class org.apache.spark.examples.streaming.HdfsWordCount target/scala-2.10/mysparkexample_2.10-1.0.jar file:///directory/of/steaming/files

nc -lk 9999 for linux/mac; ncat -lk 9999 for Windows

b) spark-submit --master local[2] --class org.apache.spark.examples.streaming.NetworkWordCount target/scala-2.10/mysparkexample_2.10-1.0.jar localhost 9999

Hands-on session: SQL and Streaming

6. Run the applications

c) spark-submit --master local[2] --class org.apache.spark.examples.streaming.SqlNetworkWordCount target/scala-2.10/mysparkexample_2.10-1.0.jar localhost 9999

Spark machine learning and graph libraries

Spark: MLlib/GraphX Applications

- PYA Analytics is a Knoxville based data science company
- Currently using Spark MLlib and GraphX to do large scale analysis of PubMed articles
- Data set is 65Gb of unstructured text, nearly 10M articles
- Other uses: text analysis, general statistical model development, etc.

MLlib: Spark Machine Learning library

- It is build on Spark.
- It supports many algorithms:
  - Classifications (logistic regression, SVM and naïve Bayes)
  - Regression (GLM)
  - Collaborative filtering (Alternating least squares)
  - Clustering (K-means)
  - Decomposition (SVD, PCA)
- Scalability
- User-friendly APIs

MLlib: K-means

```
# Load and parse the data
data = sc.textFile("data/mllib/kmeans_data.txt") parsedData =
data.map(lambda line: array([float(x) for x in line.split(' ')]))

# Build the model (cluster the data)
clusters = KMeans.train(parsedData, 2, maxIterations=10, runs=10,
initializationMode="random")

# Evaluate clustering by computing Within Set Sum of Squared Errors
def error(point):
        center = clusters.centers[clusters.predict(point)]
        return sqrt(sum([x**2 for x in (point - center)]))
         WSSSE = parsedData.map(lambda point:
error(point)).reduce(lambda x, y: x + y) print("Within Set Sum of Squared
Error = " + str(WSSSE))
```

MLlib – Word2Vec

- **Using Spark's Word2Vec model to extract features from text.**

```
val sparkConf = new SparkConf()
      .setAppName("PMWord2Vec")
      .setMaster(sparkMaster)

val sc = new SparkContext(sparkConf)

val lines = sc.textFile(abstractsFile)
      .map(_.split("\\s+").toSeq)

val w2v = new Word2Vec()
      .setVectorSize(300)
      .setMinCount(30)
```

```
val w2vModel = w2v.fit(lines)

val oStream = new
      ObjectOutputStream(new
      FileOutputStream(modelFile))

oStream.writeObject(w2vModel)

val synonyms =
    w2vModel.findSynonyms(”heart", 20)

synonyms map {x => println(x.toString)}

sc.stop()
```

MLlib – Use Case (Word2Vec)

- **Vectors near "heart" by cosine similarity**

((HF),,0.606659531593322 8)	Heart Failure
((HF).,0.6064708828926086)	Heart Failure
((RHR),0.594153702259063 7)	Resting Heart Rate
(Congestive,0.588003575801 8494)	Congestive
((CHD),0.585739433765411 4)	Congenital Heart Defect
((ADHF),,0.582550168037414 6)	Acute Decompensated HF
((IHD),0.578753530979156 5)	Ischemic Heart Disease
((CHD),,0.578229904174804 7)	Congenital Heart Defect
(valvular,0.561195492744445 8)	Valvular
((heart,0.555529534816741 9)	Heart
((AHF),0.554232418537139 9)	Acute Heart Failure
((ADHF).,0.553462088108062 7)	Acute Decompensated HF
((AHF).,0.553108870983123 8)	Acute Heart Failure
((HF),0.533207356929779)	Heart Failure
(failure,,0.532746374607086 2)	Failure

The model has learned several abbreviations for heart problems

GraphX

- Graph-parallel computation
- Fundamental operators
  - Subgraph
  - joinVertices
  - aggregateMessages
- Collection of graph algorithms
- Still emerging

GraphX: PageRank

```
//Load and initialize the graph!
val graph = GraphLoader.edgeListFile("hdfs://web.txt")! val prGraph =
graph.joinVertices(graph.outDegrees)!

// Implement and Run PageRank!
val pageRank = !
prGraph.pregel(initialMessage = 0.0, iter = 10)(! (oldV, msgSum) =>
0.15 + 0.85 * msgSum,! triplet => triplet.src.pr / triplet.src.deg,!
(msgA, msgB) => msgA + msgB)!

// Get the top 20 pages! pageRank.vertices.top(20))
(Ordering.by(_._2)).foreach(println)!
```

GraphX: Use Case

- Each PubMed article contains a list of papers cited and a list of authors
- We use GraphX to build a graph with vertices for authors and papers

```
sealed trait VertexProperty()
case class AuthorProperty(name: String) extends VertexProperty
case class PaperProperty(pmid: Int) extends VertexProperty
```

- Paper -> Paper edges show citation
- Paper -> Author edges show authorship

GraphX

- **With GraphX we can create a new edge between authors for each paper they have in common:**

```
val authEdges: RDD[Edge[Int]] = graph.collectNeighborIds(EdgeDirection.Out)
      .flatMap(vArray => (vArray._2.toTraversable cross vArray._2.toTraversable)
                         .map(xs => xs match {case (x, y) => Edge(x, y, 1)}))
```

- **Next, we aggregate edges between authors to end up with a weighted edge showing how many papers they have coauthored:**

```
val fullGraph = Graph(graph.vertices, graph.edges ++ authEdges)
       .groupEdges((x, y) => x + y)
```

Reference

- https://spark.apache.org/
- http://radar.oreilly.com/2015/02/processing-frameworks-for-hadoop.html
- http://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf
- http://horicky.blogspot.com/2013/12/spark-low-latency-massively-parallel.html
- MLlib: Scalable Machine Learning on Spark Xiangrui Meng
- MapReduce: Simplified Data Processing on Large Clusters Jeffrey Dean and Sanjay Ghemawat

GraphX- Optimization

- Incremental updates to mirror caches
- Join elimination
- Index and routing table reuse
- Index scanning for Active Sets

MLlib- Exercise

GraphX- Exercise

Reference

- https://spark.apache.org/docs/latest/mllib-clustering.html
- https://spark.apache.org/docs/latest/graphx-programming-guide.html#graph-operators
- GRAPHX: UNIFIED GRAPH ANALYTICS ON SPARK by DATABRICKS

Important Note:

These slides are in-progress. We will add more slides of use cases, exercises and performance tuning/optimization soon.

Thank you.