BDA400 – Lecture 3
Introduction to Spark

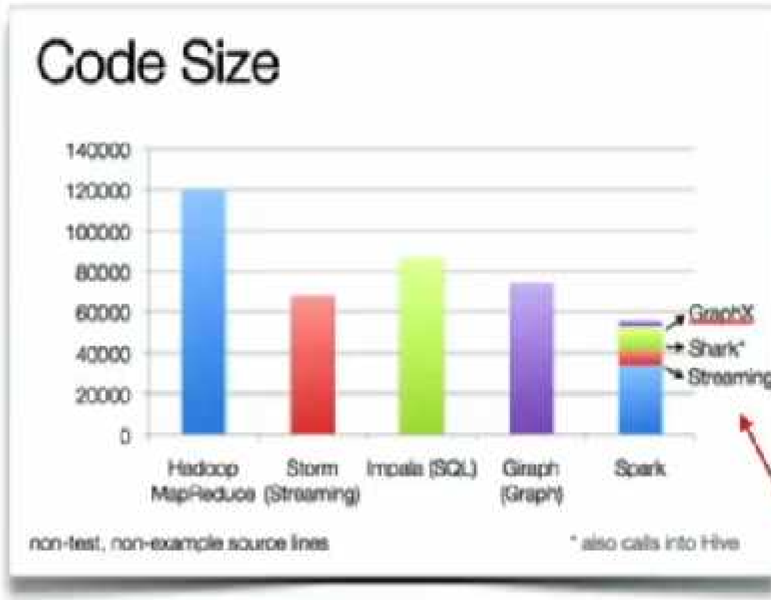
Big Data and Spark

- Data is increasing in volume, velocity, variety.
- The need to have faster results from analytics becomes increasingly important.
- Apache Spark is a computing platform designed to be *fast* and *general-purpose,* and *easy to use*
  - Speed
    - In-memory computations
    - Faster than MapReduce for complex applications on disk
  - Generality
    - Covers a wide range of workloads on one system
    - Batch applications (e.g. MapReduce)
    - Iterative algorithms
    - Interactive queries and streaming
  - Ease of use
    - APIs for Scala, Python, Java
    - Libraries for SQL, machine learning, streaming, and graph processing
    - Runs on Hadoop clusters or as a standalone
      - including the popular MapReduce model

Brief History of Spark

- 2002 – MapReduce @ Google
- 2004 – MapReduce paper
- 2006 – Hadoop @ Yahoo
- 2008 – Hadoop Summit
- 2010 – Spark paper
- 2014 – Apache Spark top-level

- MapReduce started a general batch processing paradigm
- Two limitations:
  1) Difficulty programming in MapReduce
  2) Batch processing did not fit many use cases
- Spawned a lot of specialized systems (Storm, Impala, Giraph, etc.)

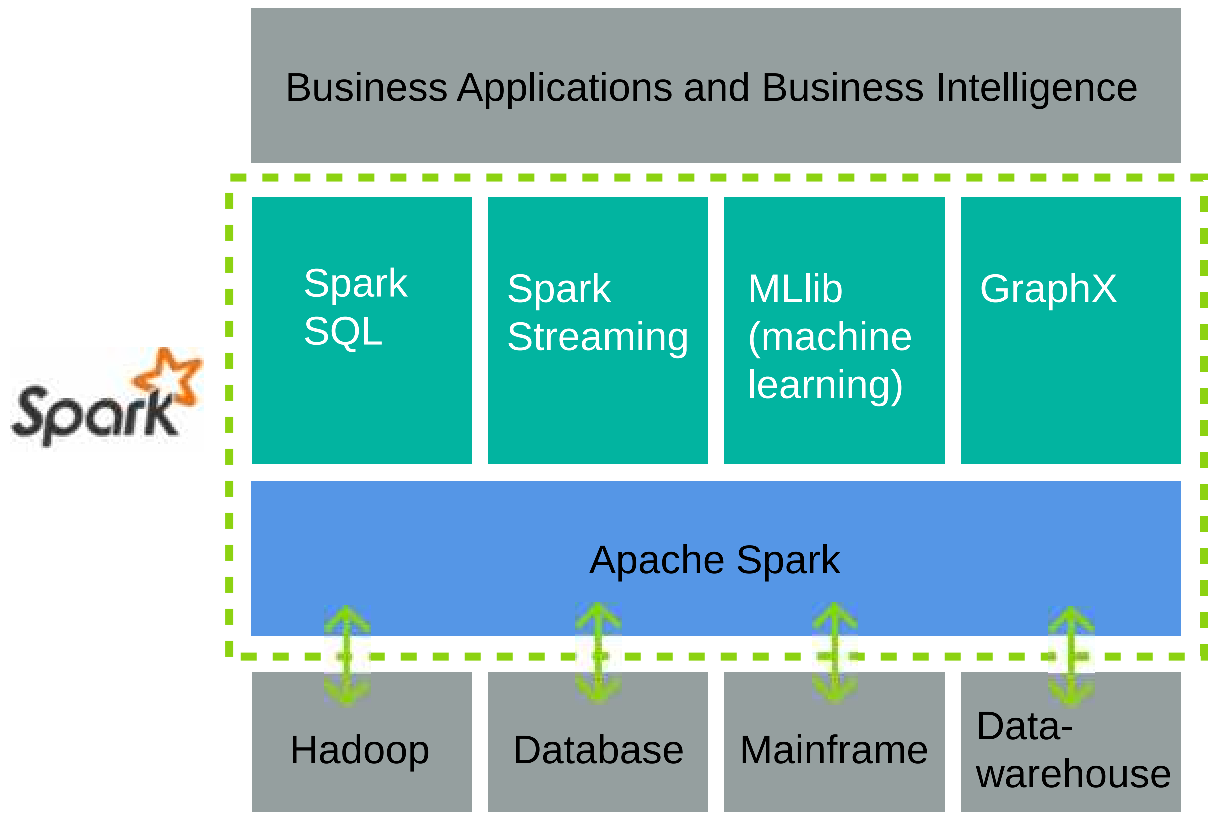


The State of Spark, and Where We're Going Next
Matei Zaharia
Spark Summit (2013)
youtu.be/nU6vO2EJAb4

used as libs, instead of specialized systems

Why Spark

Spark processes and analyzes data from ANY data source



Analytics for Apache Spark

Blends Multiple Data Types, Sources & Workloads

Spark SQL	Spark Streaming	MLlib Machine Learning	GraphX Graphing
Execute SQL Statements	Streaming Analytics via Micro-batch	M.L. and Statistical Algorithms	Distributed Graph Processing Framework

Spark Core

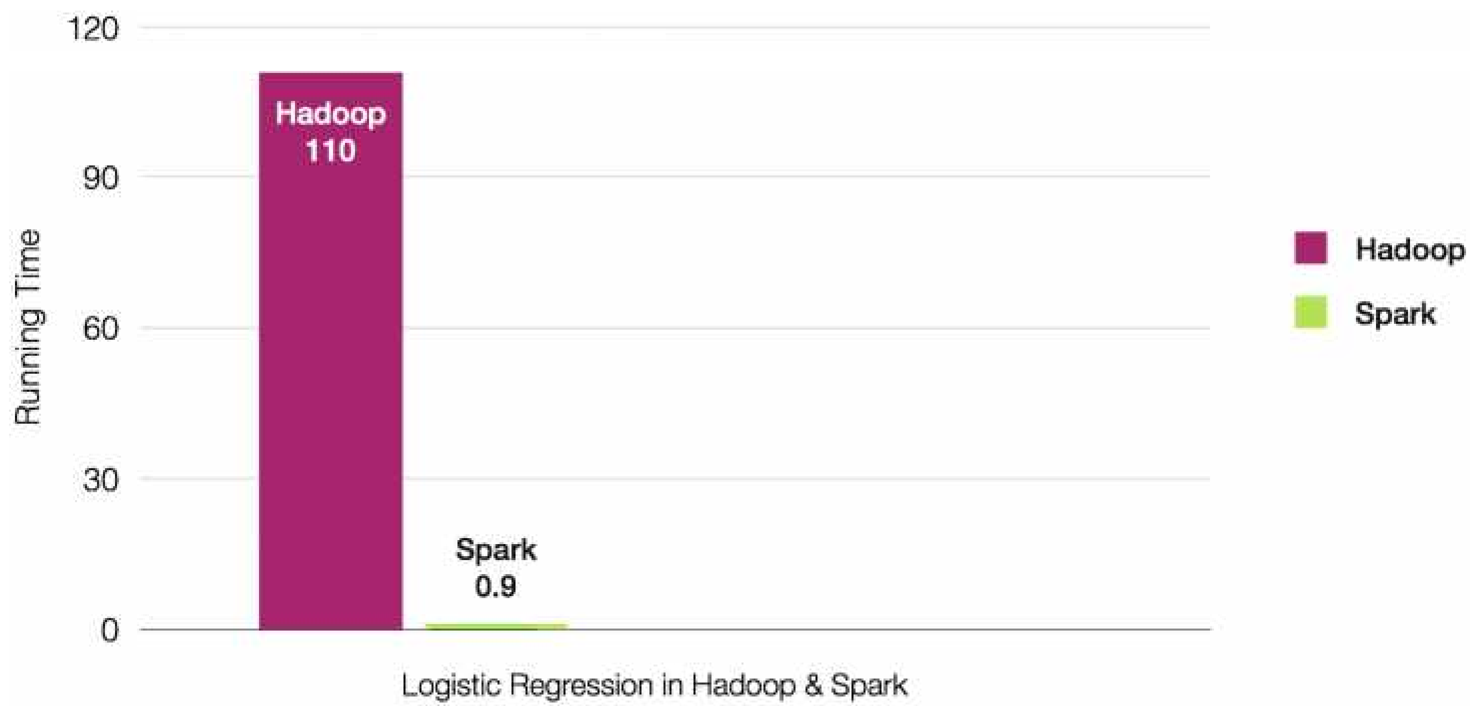
- General compute engine
- Basic I/O functions
- Task dispatching
- Scheduling

Data Sources

IBM Cloud	Public Cloud	Cloud Apps	On-Premises
BigInsights (HDFS), Cloudant (DBaaS), dashDB (Analytics), SQDB (Managed DB2), Swift (Object Storage)	amazon webservices S3, Cassandra, mongoDB, redis, rackspace, Microsoft Azure, MySQL, HIVE, PostgreSQL, HDFS, dBase, APACHE HBASE	NETSUITE, salesforce, CSV, JDBC, {JSON}, Parquet, elasticsearch, AVRO	ORACLE, SAP, IBM DB2

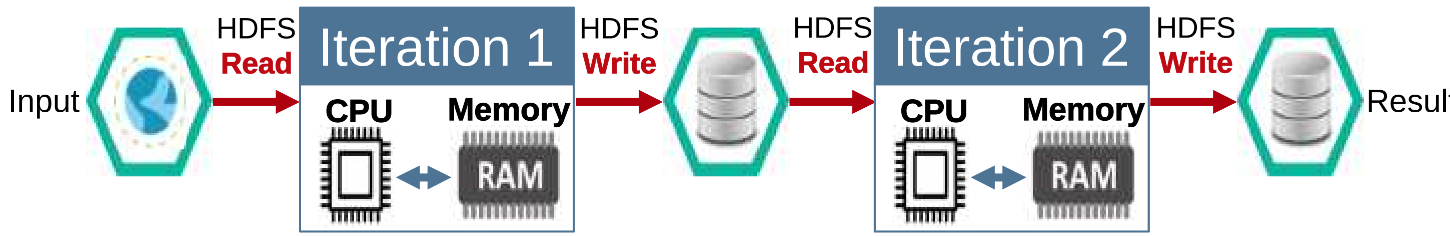
Spark and Hadoop

Spark is complementary to Hadoop, but much faster, with in-memory performance



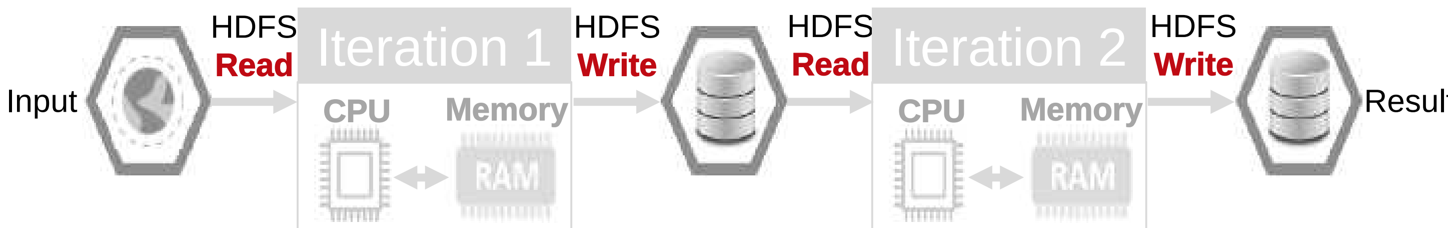
Motivation for Apache Spark

- Traditional Approach: MapReduce jobs for complex jobs, interactive query, and online event-hub processing involves lots of (slow) disk I/O

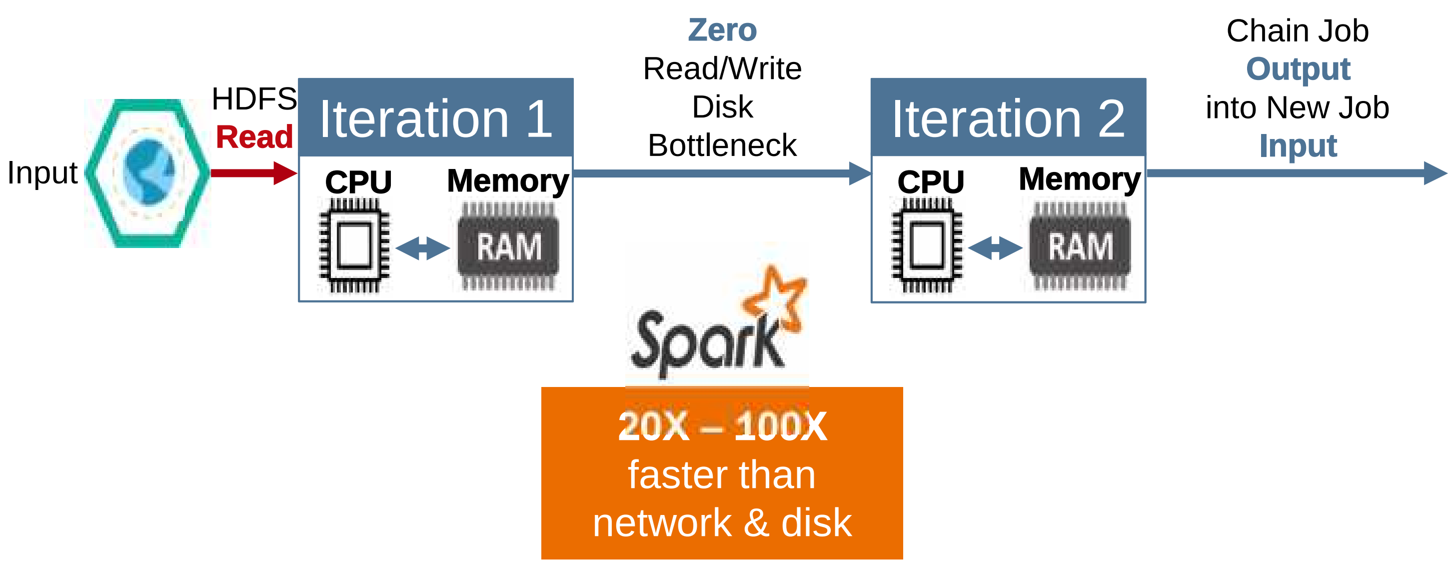


Motivation for Apache Spark

- Traditional Approach: MapReduce jobs for complex jobs, interactive query, and online event-hub processing involves lots of (slow) disk I/O



- Solution: Keep more data in-memory with a new distributed execution engine



Who uses Spark and why?

- Parallel distributed processing, fault tolerance on commodity hardware, scalability, in-memory computing, high level APIs, etc.
- Saves time and money
- Data scientist
  - Analyze and model the data to obtain insight using ad-hoc analysis
  - Transforming the data into a useable format
  - Statistics, machine learning, SQL
- Engineers
  - Develop a data processing system or application
  - Inspect and tune their applications
  - Programming with the Spark’s API
- Everyone else
  - Ease of use
  - Wide variety of functionality
  - Mature and reliable

How Users are Embracing Spark

Data **Scientist**

Derive insights which are immediately actionable with powerful Spark tools.

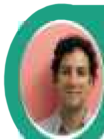
Application **Developer**

Integrate 100% open-standards Spark with any application, regardless of the platform.

Business **Analyst**

Self-service, rapid access to understanding of the business, without IT intervention.

Data **Engineer**



Assemble data pipelines with ease to power interactive dashboards and services.

Accessible	Integrated	Powerful

How the Market is Embracing Spark

Cloudera

- Selling a **proprietary Hadoop** distribution
- Have endorsed Spark
- Cloudera has announced a **One Platform Initiative**

Hortonworks

- Selling support and education for open source **Hadoop**
- Views Spark as a **threat**, but will reluctantly talk about
- **No** analytics capability: only storage & processing
- Cooperative with IBM: we both support **ODP**

ORACLE

- Selling **Exadata**, which can only handle structured data
- Launched Hadoop appliance
- Not much with Spark at this point

SAP

- Selling a **proprietary** in-memory database (HANA)
- Now beginning to embrace Hadoop and Spark with the announcement of **Vora**
- Prioritize applications over platform

IBM

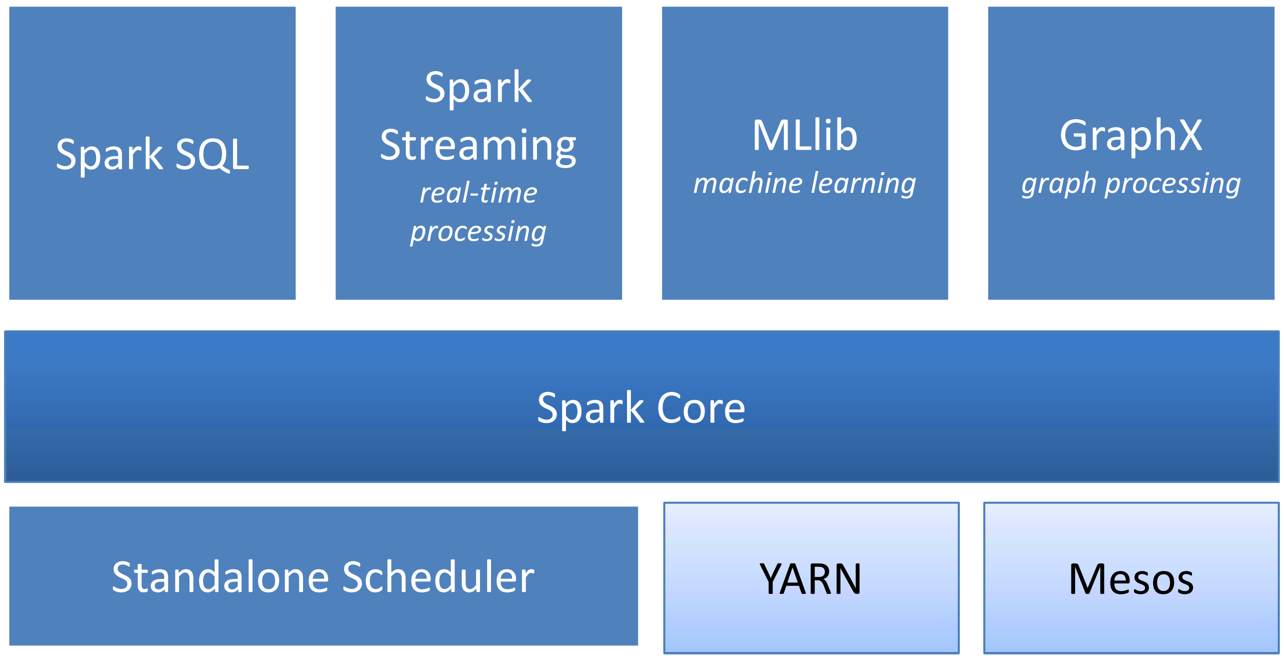
- Has embraced Spark fully
- New technology center in San Francisco with 300+ developers
- Incorporating Spark engine into several offerings
- Investing in SparkSQL, ML, Streaming and Graphing

Positioning Hadoop and Spark

	Spark	Hadoop MapReduce
Storage	No built-in storage (in-memory)	On-disk only
Operations	Map, Reduce, Join, Sample, etc.	Map and Reduce
Execution Model	Batch, interactive, streaming	Batch only
Programming Environments	Python, Scala, Java, and R	Java only

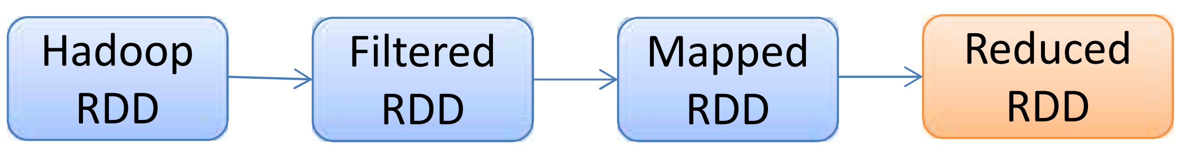
- Spark is a unified platform for data integration
  → Contrast with the many distinct distributions & tools for Hadoop
- Spark follows lazy evaluation of execution graphs
  → Optimizes jobs, reduces wait states, and allows easier pipelining of tasks
- Spark lowers the resource overhead for starting or shuffling jobs
  → Less expensive than MapReduce

Spark unified stack



Resilient Distributed Datasets (RDD)

- Spark's primary abstraction
- Distributed collection of elements
- Parallelized across the cluster
- Two types of RDD operations
  - Transformations
    - Creates a DAG
    - Lazy evaluations
    - No return value
  - Actions
    - Performs the transformations and the action that follows
    - Returns a value
- Fault tolerance
- Caching
- Example of RDD flow.



Resilient Distributed Dataset (RDD)

- Fault-tolerant collection of elements that can be operated on in parallel.
- Immutable
- Three methods for creating RDD
  - Parallelizing an existing collection
  - Referencing a dataset
  - Transformation from an existing RDD
- Two types of RDD operations
  - Transformations
  - Actions
- Dataset from any storage supported by Hadoop
  - HDFS
  - Cassandra
  - HBase
  - Amazon S3
  - etc.
- Types of files supported:
  - Text files
  - SequenceFiles
  - Hadoop InputFormat

Creating an RDD

- Launch the Spark shell
  ./bin/spark-shell

- Create some data
  val data = 1 to 10000

- Parallelize that data (creating the RDD)
  val distData = sc.parallelize(data)

- Perform additional transformations or invoke an action on it.
  distData.filter(…)

- Alternatively, create an RDD from an external dataset
  – val readmeFile = sc.textFile("Readme.md")

RDD operations - Basics

- Loading a file

```
val lines = sc.textFile(“hdfs://data.txt”)
```

- Applying transformation

```
val lineLengths = lines.map(s => s.length)
```

- Invoking action

```
val totalLengths = lineLengths.reduce((a,b) => a + b)
```

- MapReduce example:

```
val wordCounts = textFile.flatMap(line => line.split (" "))
.map(word => (word, 1))
.reduceByKey((a,b) => a + b)

wordCounts.collect()
```

Direct Acyclic Graph (DAG)

- View the DAG

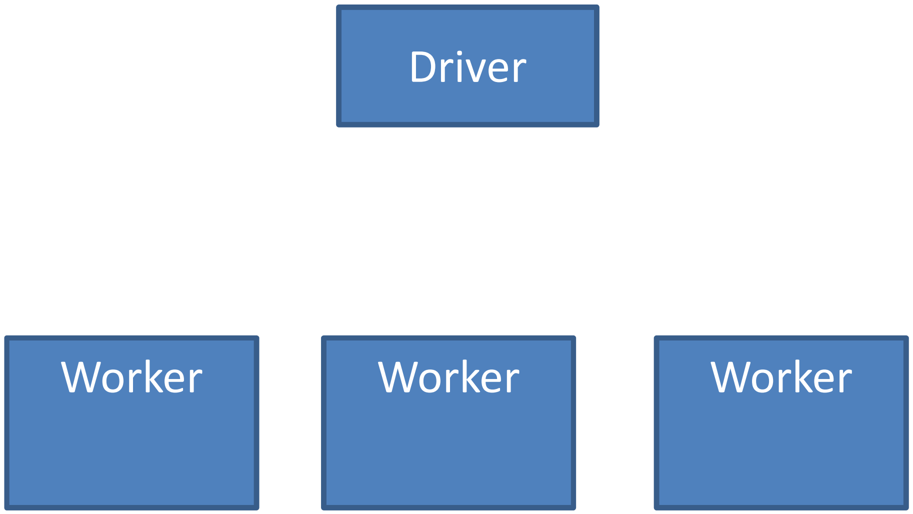
  linesLength.toDebugString

- Sample DAG

```
res5: String =
MappedRDD[4] at map at <console>:16 (3 partitions)
  MappedRDD[3] at map at <console>:16 (3 partitions)
    FilteredRDD[2] at filter at <console>:14 (3 partitions)
      MappedRDD[1] at textFile at <console>:12 (3 partitions)
        HadoopRDD[0] at textFile at <console>:12 (3 partitions)
```

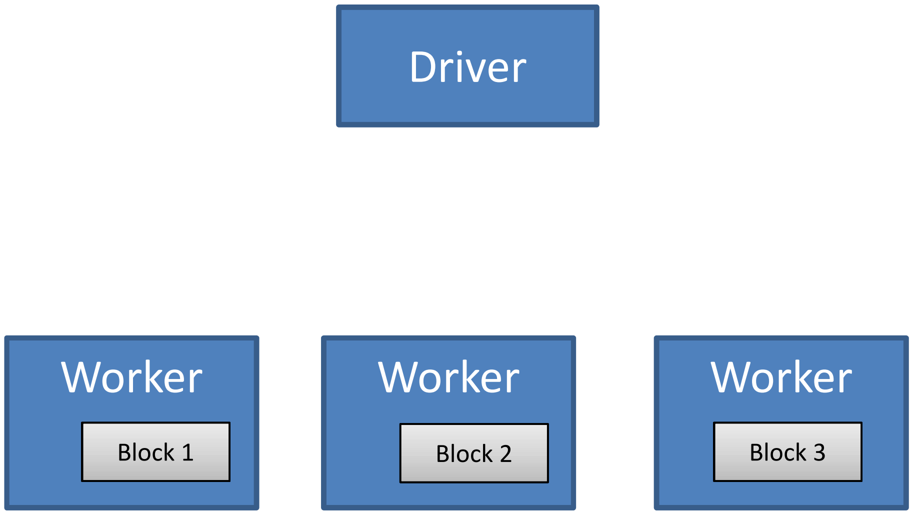
What happens when an action is executed?

```
// Creating the RDD
val logFile = sc.textFile(“hdfs://…”)
// Transformations
val errors = logFile.filter(_.startsWith(“ERROR”))
val messages = errors.map(_.split(“\t”)).map(r => r(1))
//Caching
messages.cache()
// Actions
messages.filter(_.contains(“mysql”)).count()
messages.filter(_.contains(“php”)).count()
```



What happens when an action is executed?

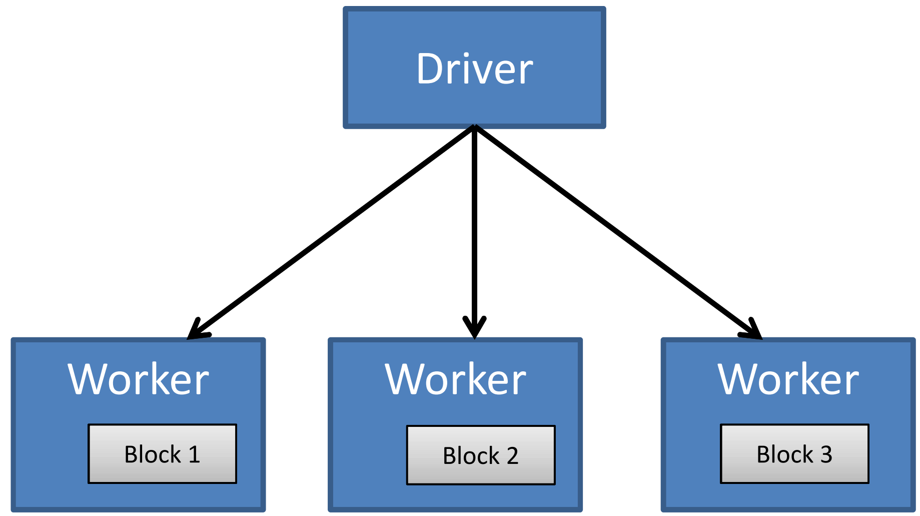
```
// Creating the RDD
val logFile = sc.textFile(“hdfs://…”)
// Transformations
val errors = logFile.filter(_.startsWith(“ERROR”))
val messages = errors.map(_.split(“\t”)).map(r => r(1))
//Caching
messages.cache()
// Actions
messages.filter(_.contains(“mysql”)).count()
messages.filter(_.contains(“php”)).count()
```



The data is partitioned into different blocks

What happens when an action is executed?

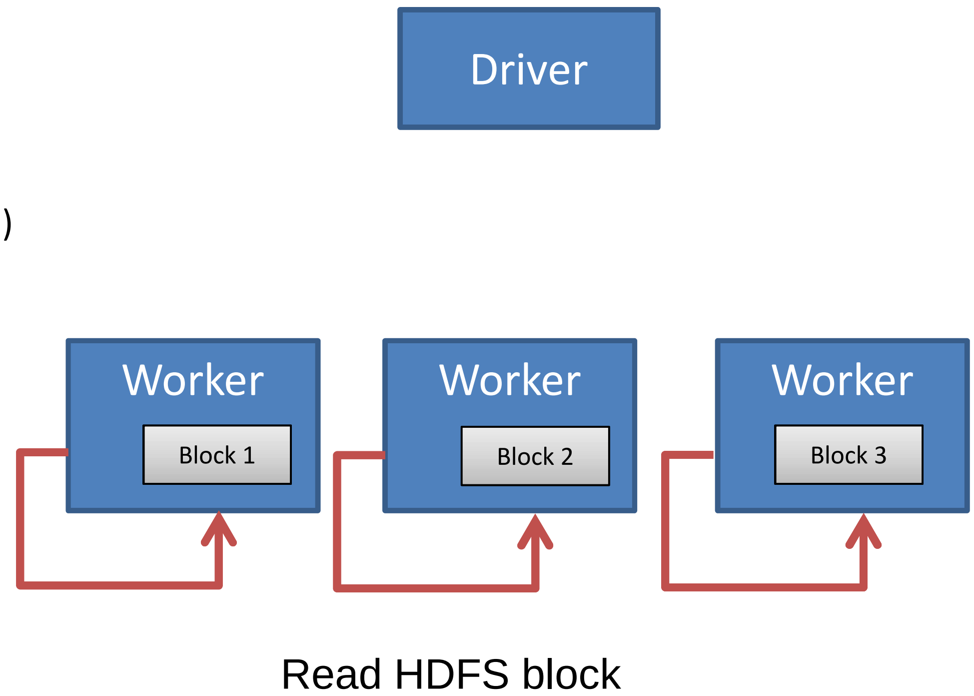
```
// Creating the RDD
val logFile = sc.textFile("hdfs://...")
// Transformations
val errors = logFile.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
//Cache
messages.cache()
// Actions
messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```



Driver sends the code to be executed on each block

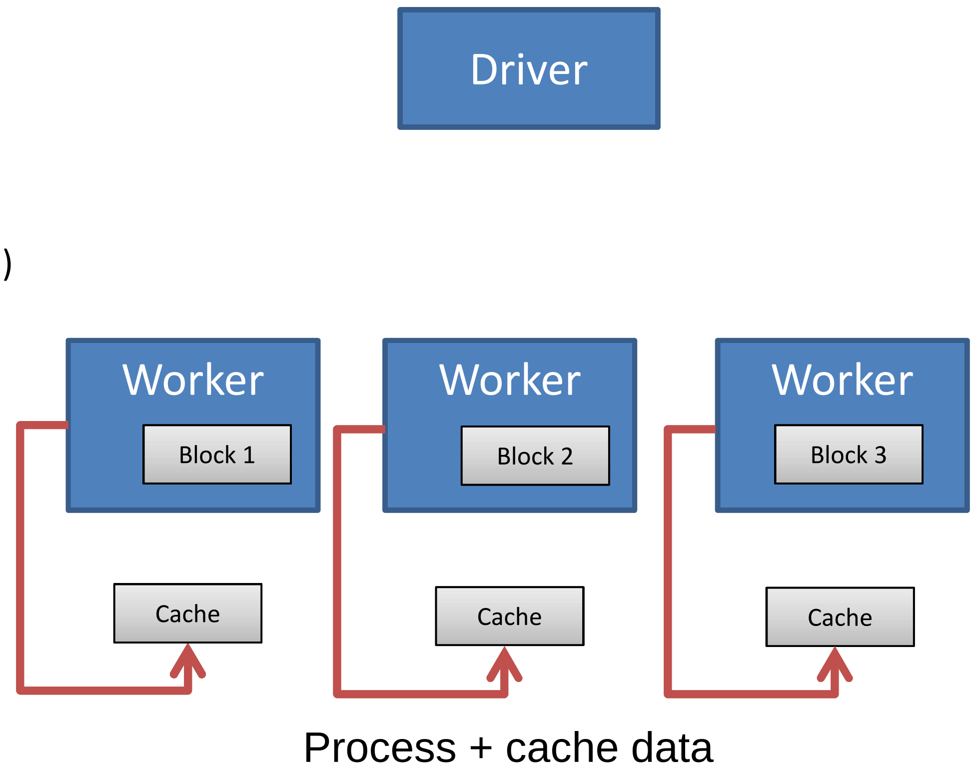
What happens when an action is executed?

```
// Creating the RDD
val logFile = sc.textFile("hdfs://…")
// Transformations
val errors = logFile.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
//Caching
messages.cache()
// Actions
messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```



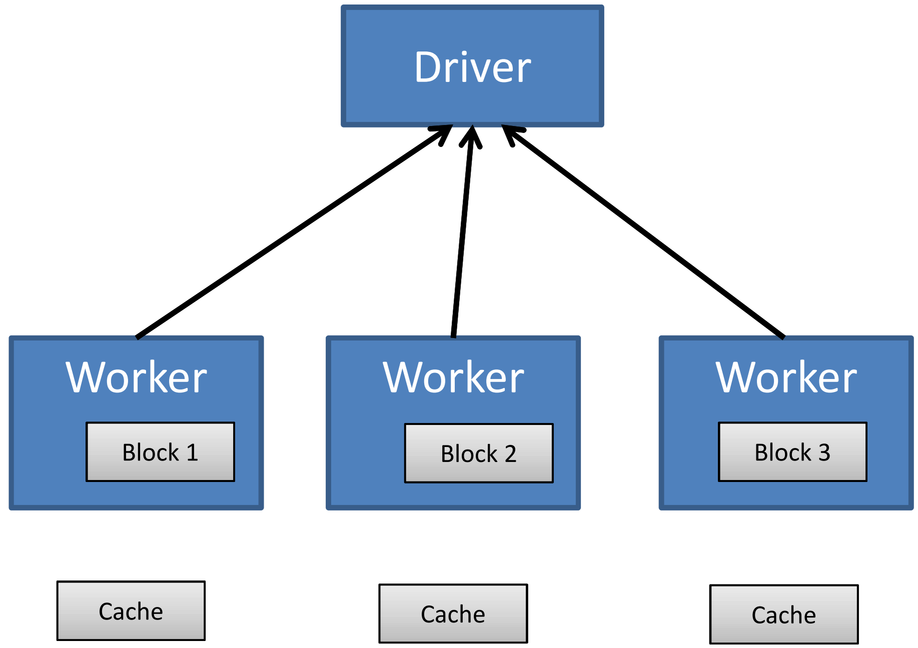
What happens when an action is executed?

```
// Creating the RDD
val logFile = sc.textFile("hdfs://...")
// Transformations
val errors = logFile.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
//Caching
messages.cache()
// Actions
messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```



What happens when an action is executed?

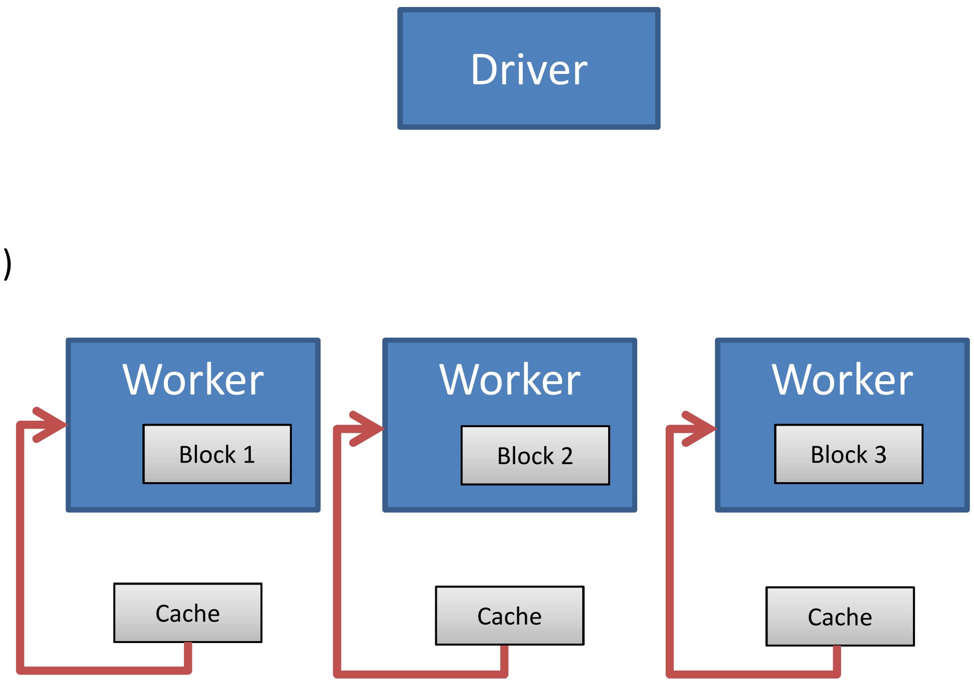
```
// Creating the RDD
val logFile = sc.textFile("hdfs://...")
// Transformations
val errors = logFile.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
//Caching
messages.cache()
// Actions
messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```



Send the data back
to the driver

What happens when an action is executed?

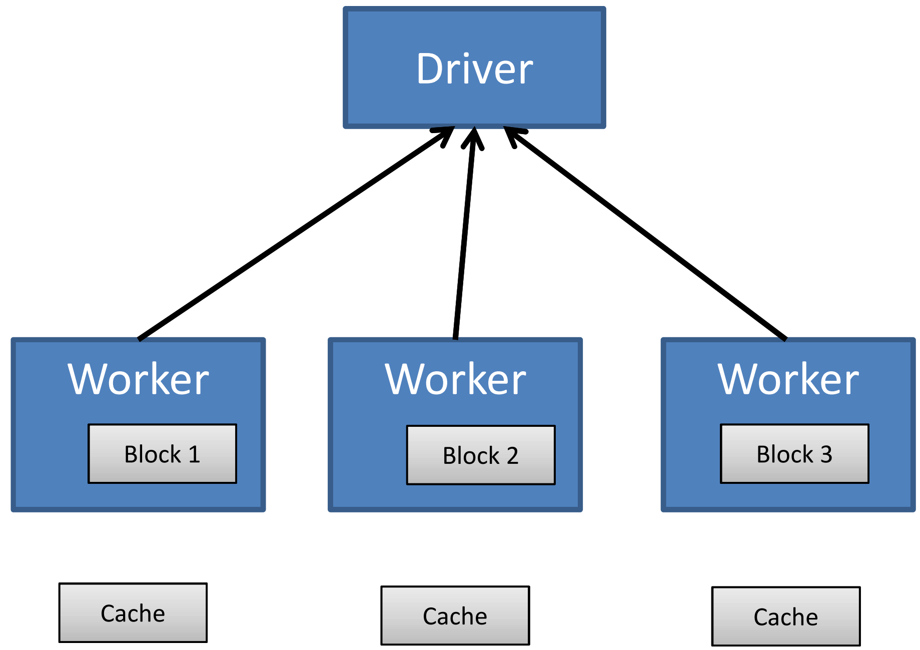
```
// Creating the RDD
val logFile = sc.textFile("hdfs://...")
// Transformations
val errors = logFile.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
//Caching
messages.cache()
// Actions
messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```



Process from cache

What happens when an action is executed?

```
// Creating the RDD
val logFile = sc.textFile("hdfs://...")
// Transformations
val errors = logFile.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
//Caching
messages.cache()
// Actions
messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```



Send the data back
to the driver

RDD operations - Transformations

- A subset of the transformations available. Full set can be found on Spark's website.
- Transformations are lazy evaluations
- Returns a pointer to the transformed RDD

Transformation	Meaning
map(func)	Return a new 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.
flatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items. So func should return a Seq rather than a single item
join(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.
reduceByKey(func)	When called on a dataset of (K, V) pairs, returns a dataset of (K,V) pairs where the values for each key are aggregated using the given reduce function *func*
sortByKey([ascending],[numTasks])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K,V) pairs sorted by keys in ascending or descending order.

RDD operations - Actions

- Actions returns values

Action	Meaning
collect()	Return all the elements of the dataset as an array of the driver program. This is usually useful after a filter or another operation that returns a sufficiently small subset of data.
count()	Return the number of elements in a dataset.
first()	Return the first element of the dataset
take(n)	Return an array with the first n elements of the dataset.
foreach(func)	Run a function func on each element of the dataset.

RDD persistence

- Each node stores any partitions of the cache that it computes in memory
- Reuses them in other actions on that dataset (or datasets derived from it)
  - Future actions are much faster (often by more than 10x)
- Two methods for RDD persistence
  - persist()
  - cache() → essentially just persist with MEMORY_ONLY storage

Storage Level	Meaning
MEMORY_ONLY	Store as deserialized Java objects in the JVM. If the RDD does not fit in memory, part of it will be cached. The other will be recomputed as needed. This is the default. The cache() method uses this.
MEMORY_AND_DISK	Same except also store on disk if it doesn't fit in memory. Read from memory and disk when needed.
MEMORY_ONLY_SER	Store as serialized Java objects (one bye array per partition). Space efficient, but more CPU intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_AND_DISK but stored as serialized objects.
DISK_ONLY	Store only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as above, but replicate each partition on two cluster nodes
OFF_HEAP (experimental)	Store RDD in serialized format in Tachyon.

Which storage level to choose?

- If your RDDs fit comfortably with the default storage level (MEMORY_ONLY), leave them that way. This is the most CPU-efficient option, allowing operations on the RDDs to run as fast as possible.

- If not, try using MEMORY_ONLY_SER and selecting a fast serialization library to make the objects much more space-efficient, but still reasonably fast to access.

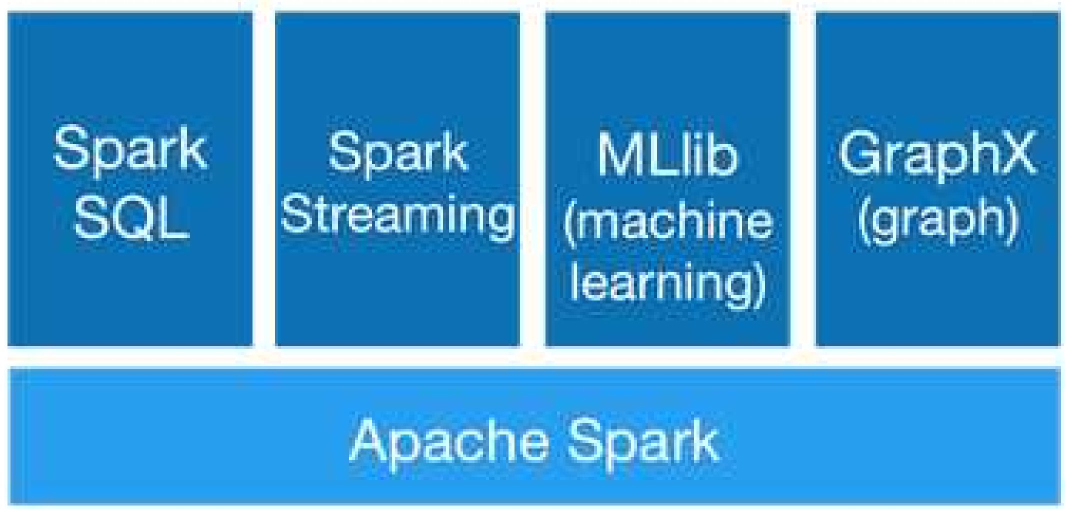
- Don't spill to disk unless the functions that computed your datasets are expensive, or they filter a large amount of the data. Otherwise, re-computing a partition may be as fast as reading it from disk.

- Use the replicated storage levels if you want fast fault recovery (e.g. if using Spark to serve requests from a web application). *All* the storage levels provide full fault tolerance by recomputing lost data, but the replicated ones let you continue running tasks on the RDD without waiting to re-compute a lost partition.

- In environments with high amounts of memory or multiple applications, the experimental OFF_HEAP mode has several advantages:
  - It allows multiple executors to share the same pool of memory in Tachyon.
  - It significantly reduces garbage collection costs.
  - Cached data is not lost if individual executors crash.

Spark libraries

- Extension of the core Spark API.
- Improvements made to the core are passed to these libraries.
- Little overhead to use with the Spark core



spark.apache.org

Spark Capabilities

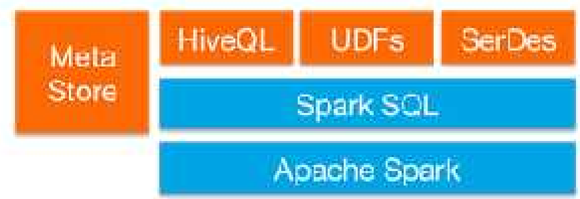
Spark Core	Component	Capability	Details
Spark Core	Spark Streaming	**Stream Processing** Near real-time data processing & analytics	• Micro-batch event processing for near real-time analytics • Process live streams of data (IoT, Twitter, Kafka) • No multi-threading or parallel processing required
	MLlib (machine learning)	**Machine Learning** Incredibly fast, easy to deploy algorithms	• Predictive and prescriptive analytics, and smart application design, from statistical and algorithmic models • Algorithms are pre-built
	Spark SQL	**Unified Data Access** Fast, familiar query language for all data	• Query your structured data sets with SQL or other dataframe APIs • Data mining, BI, and insight discovery • Get results faster due to performance
	GraphX (graph)	**Graph Analytics** Fast and integrated graph computation	• Represent data in a graph • Represent/analyze systems represented by nodes and interconnections between them • Transportation, person to person relationships, etc.

Spark SQL

- Allows relational queries expressed in
  - SQL
  - HiveQL
  - Scala
- SchemaRDD
  - Row objects
  - Schema
  - Created from:
    - Existing RDD
    - Parquet file
    - JSON dataset
    - HiveQL against Apache Hive
- Supports Scala, Java, and Python

Spark SQL

- Provide for relational queries expressed in SQL, HiveQL and Scala
- Seamlessly mix SQL queries with Spark programs
- DataFrame/Dataset provide a single interface for efficiently working with structured data including Apache Hive, Parquet and JSON files
- Leverages Hive frontend and metastore
  - Compatibility with Hive data, queries, and UDFs
  - HiveQL limitations may apply
  - Not ANSI SQL compliant
  - Little to no query rewrite optimization, automatic memo sophisticated workload management



- Graduated from alpha status with Spark 1.3
  - DataFrames API marked as experimental
- Standard connectivity through JDBC/ODBC

Spark SQL – Getting started

- SQLContext
  - Created from a SparkContext

Scala:

```
val sc: SparkContext // An existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

Java:

```
JavaSparkContext sc = ...; // An existing JavaSparkContext.
JavaSQLContext sqlContext = new org.apache.spark.sql.api.java.JavaSQLContext(sc);
```

Python:

```
from pyspark.sql import SQLContext sqlContext = SQLContext(sc)
```

- Import a library to convert an RDD to a SchemaRDD
  - Scala only: `import sqlContext.createSchemaRDD`
- SchemaRDD data sources:
  - Inferring the schema using reflection
  - Programmatic interface

Spark SQL – Inferring the schema using reflection

- The *case class* in Scala defines the schema of the table.

```
case class Person(name: String, age: Int)
```

- The arguments of the case class becomes the names of the columns.
- Create the RDD of the *Person* object

```
val people =
sc.textFile("examples/src/main/resources/people.txt")
.map(_.split(","))
.map(p => Person(p(0), p(1).trim.toInt))
```

- Register the RDD as a table

```
people.registerTempTable("people")
```

- Run SQL statements using the *sql* method provided by the SQLContext

```
val teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")
```

- The results of the queries are SchemaRDD. Normal RDD operations also work on them

```
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

Spark SQL – Programmatic interface

- Use when you cannot define the case classes ahead of time.
- Create the RDD:

```
val people = sc.textFile(…)
```

- Three steps to create the SchemaRDD:
  1. Create an RDD of ***Row***s from the original RDD

     ```
     val schemaString = “name age”
     ```

  2. Create the schema represented by a *StructType* matching the structure of the ***Row***s in the RDD from step 1.

     ```
     val schema = StructType( schemaString.split(" ").map(fieldName =>
     StructField(fieldName, StringType, true)))
     ```

  3. Apply the schema to the RDD of ***Row***s using the *applySchema* method.

     ```
     val rowRDD = people.map(_.split(",")).map(p => Row(p(0), p(1).trim))
     val peopleSchemaRDD = sqlContext.applySchema(rowRDD, schema)
     ```

- Then register the peopleSchemaRDD as a table

```
peopleSchemaRDD.registerTempTable("people")
```

- Run the sql statements using the sql method:

```
val results = sqlContext.sql("SELECT name FROM people")
results.map(t => "Name: " + t(0)).collect().foreach(println)
```

Spark Streaming

- Component of Spark
  - Project started in 2012
  - First alpha release in Spring 2013
  - Out of alpha with Spark 0.9.0

- Discretized Stream (DStream) programming abstraction
  - Represented as a sequence of RDDs (micro-batches)
  - RDD: set of records for a specific time interval
  - Supports Scala, Java, and Python (with limitations)

- Fundamental architecture: batch processing of datasets

Spark Streaming

- Scalable, high-throughput, fault-tolerant stream processing of live data streams
- Receives live input data and divides into small batches which are processed and returned as batches
- DStream – sequence of RDD
- Currently supports Scala and Java
- Basic Python support available in Spark 1.2.

- Receives data from:
  - Kafka
  - Flume
  - HDFS / S3
  - Kinesis
  - Twitter
- Pushes data out to:
  - HDFS
  - Databases
  - Dashboard

Current Spark Streaming I/O

- Input Sources
  - Kafka, Flume, Twitter, ZeroMQ, MQTT, TCP sockets
  - Basic sources: sockets, files, Akka actors
  - Other sources require receiver threads

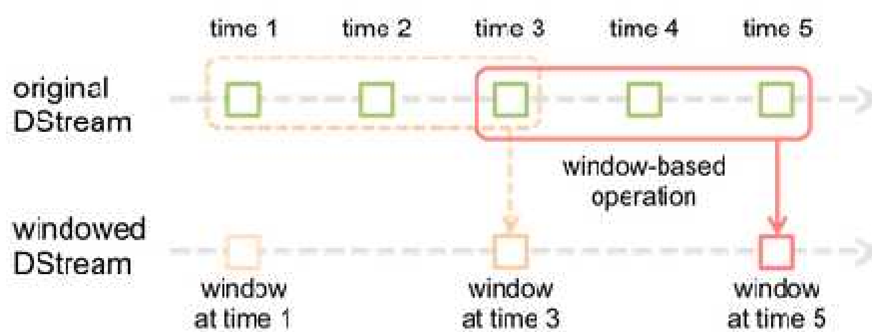
- Output operations
  - Print(), saveAsTextFiles(), saveAsObjectFiles(), saveAsHadoopFiles(), foreachRDD()
  - foreachRDD can be used for message queues, DB operations and more

Spark Streaming - Internals

- The input stream (DStream) goes into Spark Steaming
- Breaks up into batches
- Feeds into the Spark engine for processing
- Generate the final results in streams of batches

spark.apache.org

- Sliding window operations
  - Windowed computations
    - Window length
    - Sliding interval
    - `reduceByKeyAndWindow`



spark.apache.org

Spark Streaming – Getting started

- Scenario: Count the number of words coming in from the TCP socket.
- Import the Spark Streaming classes and some implicit conversions

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._
```

- Create the StreamingContext object

```
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
```

- Create a DStream

```
val lines = ssc.socketTextStream("localhost", 9999)
```

- Split the lines into words

```
val words = lines.flatMap(_.split(" "))
```

- Count the words

```
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
```

- Print to the console:

```
wordCounts.print()
```

Spark Streaming – Continued

- No real processing happens until you tell it:

```
ssc.start() // Start the computation
ssc.awaitTermination() // Wait for the computation to terminate
```

- The entire code and application can be found in the NetworkWordCount example
- Run the full example:
  - Run netcat to start the data stream
  - In a different terminal, run the application

```
./bin/run-example streaming.NetworkWordCount localhost 9999
```

Spark ML

- Spark ML for machine learning library
  - RDD-based package `spark.mllib` now in maintenance mode
  - The primary API is now the DataFrame-based package `spark.ml`
  - Parity of `spark.ml` estimated for Spark 2.2
- Provides common algorithm and utilities
  - Classification
  - Regression
  - Clustering
  - Collaborative filtering
  - Dimensionality reduction
- Leverages iteration and yields better results than one-pass approximations sometimes used with MapReduce

Spark MLLib

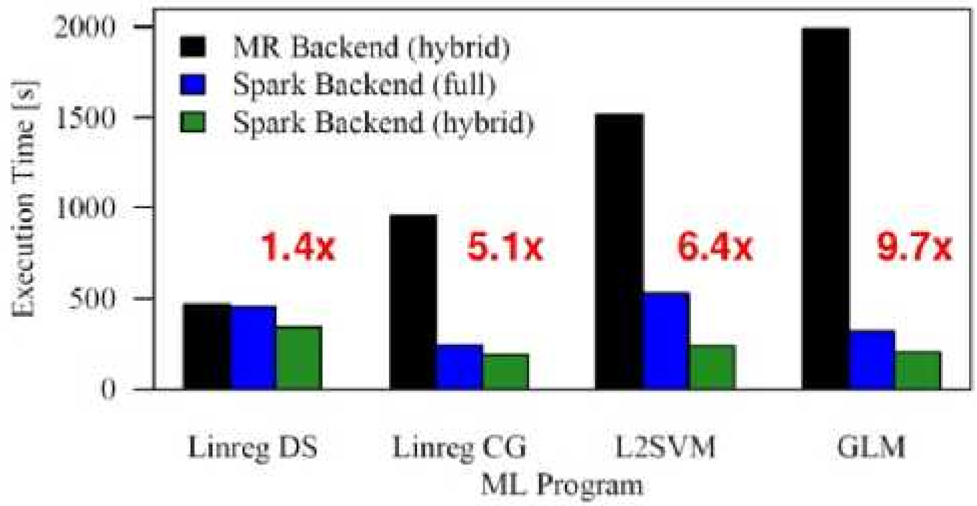
Basic Statistics	Clustering	Frequent Pattern Mining
Summary Statistics	K-means	FP-growth
Correlations	Gaussian mixture	Association rules
Stratified sampling	Power iteration clustering	PreficSpan
Hypothesis testing	Latent Dirichlet allocation	**Optimization (developer)**
Streaming significant testing	Bisecting k-means	Stochastic gradient descent
Random data generation	Streaming k-means	Limited memory BFGS
Classification&Regression	**Collaborative Filtering**	**Others**
Linear models	Alternating least squares	Evaluation metrics
Naïve Bayes	**Dimensionality Reduction**	PMML model export
Decision trees	Singular value decomposition	Feature extraction and transformation
Ensemble trees	Principal component analysis	
Isotonic regression		

Spark ML

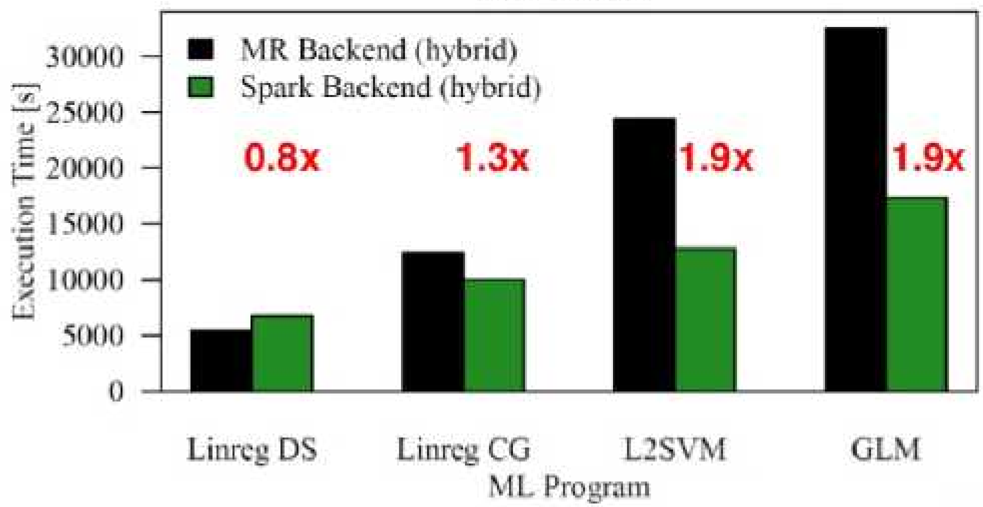
Classification/Regression	Clustering	Feature Extractors
Logistic regression	K-means	TF-IDF
Decision tree classifier	Gaussian mixture	Word2Vec
Random forest (class/reg)	Latent Dirichlet allocation	CountVectorizer
Gradient boosted tree	Bisecting k-means	**Feature Transformers**
Multilayer perceptron classifier	**Collaborative Filtering**	Tokenizer
One-vs-rest classifier	Alternating least squares	StopWordRemover
Linear regression	**Feature Selectors**	N-gram
Generalized linear reg.	VectorSlicer	Binarizer
Naïve Bayes	RFormula	PCA
Decision trees (class/reg)	ChiSqSelector	PolynomialExpansion
Survival regression		StringIndexer
Isotonic regression		---- 13 more ---

Machine Learning with Spark: Backend performance

In-Memory Data Set (160 GB)



Large-Scale Data Set (1.6 TB)

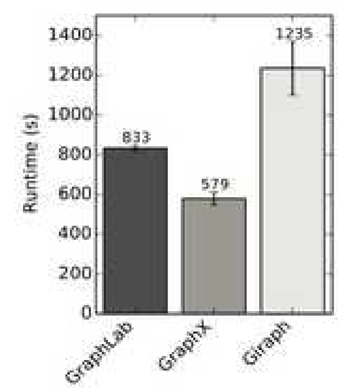


Spark R

- Spark R is an R package that provides a light-weight front-end to use Apache Spark from R
- Spark R exposes the Spark API through the RDD class and allows users to interactively run jobs from the R shell on a cluster.
- Goals
  - Make Spark R production ready
    - Efforts from AlteryX and DataBricks
  - Integration with MLlib
  - Consolidations to the data frame and RDD concepts

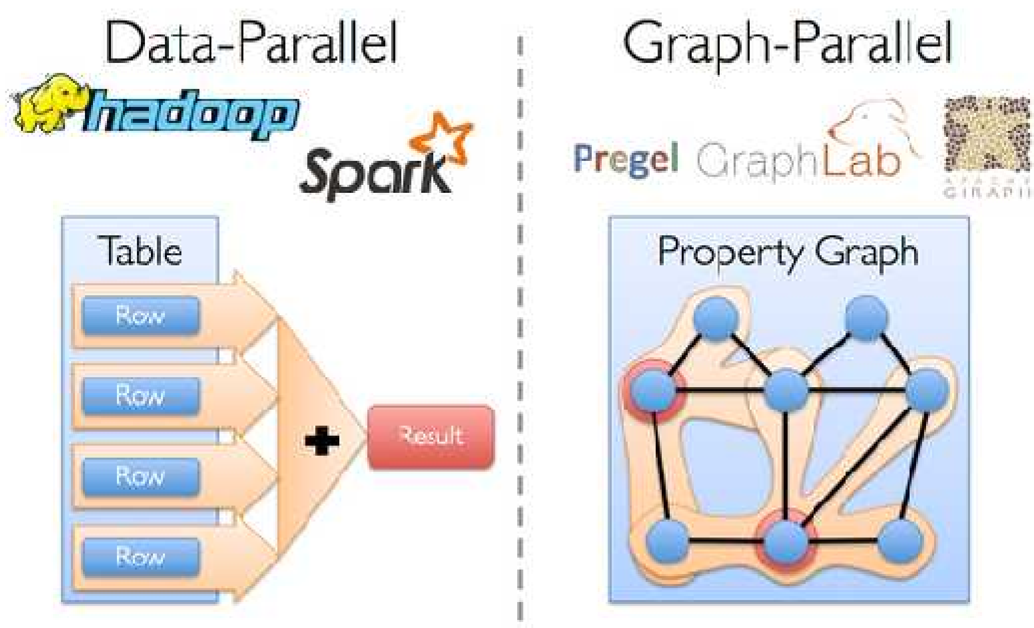
Spark GraphX

- Flexible Graphing
  - GraphX unifies ETL, exploratory analysis, and iterative graph computation
  - You can view the same data as both graphs and collections, transform and join graphs with RDDs efficiently, and write custom iterative graph algorithms with the API
- Speed
  - Comparable performance to the fastest specialized graph processing systems.
- Algorithms
  - Choose from a growing library of graph algorithms
  - In addition to a highly flexible API, GraphX comes with a variety of graph algorithms



Spark GraphX

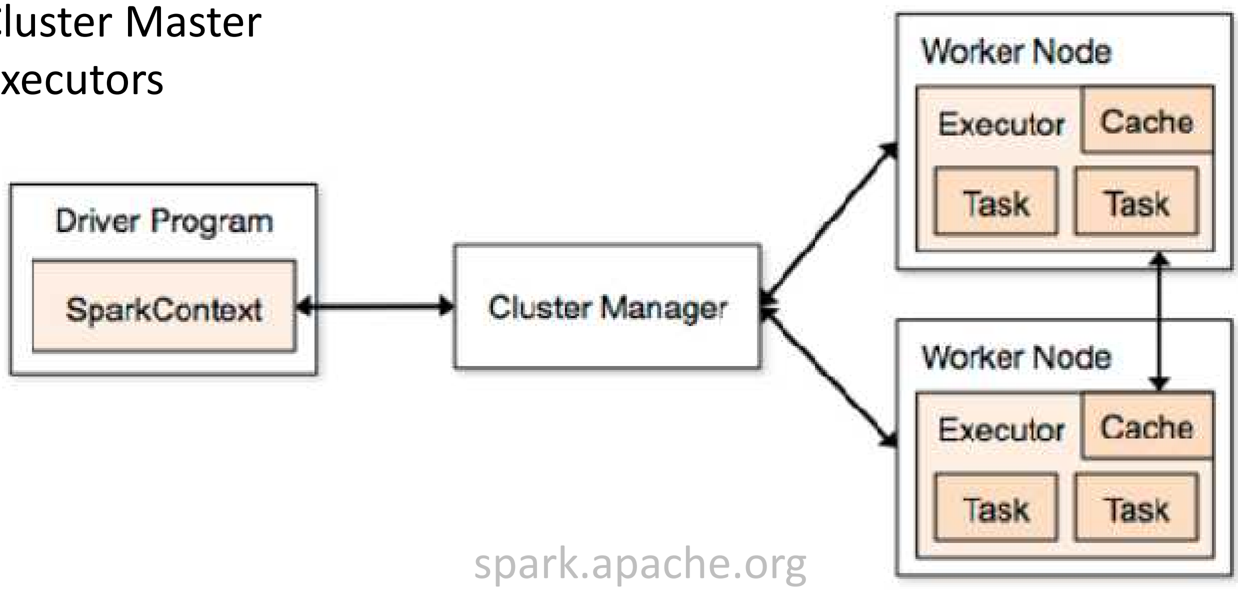
- GraphX for graph processing
  - Graphs and graph parallel computation
  - Social networks and language modeling
- Lab exercise will be on finding attributes associated with the tops users.



https://spark.apache.org/docs/latest/graphx-programming-guide.html#overview

Spark cluster overview

- Components
  - Driver
  - Cluster Master
  - Executors



spark.apache.org

- Cluster manager:
  - Standalone
  - Apache Mesos
  - Hadoop YARN

SparkContext

- The main entry point for Spark functionality
- Represents the connection to a Spark cluster
- Create RDDs, accumulators, and broadcast variables on that cluster
- In the Spark shell, the SparkContext, sc, is automatically initialized for you to use
- In a Spark program, import some classes and implicit conversions into your program:

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
```

Spark configuration

- Three locations for configuration:
  - Spark properties
  - Environment variables
    - `conf/spark-env.sh`
  - Logging
    - `log4j.properties`
- Override default configuration directory (`SPARK_HOME/conf`)
  - `SPARK_CONF_DIR`
    - spark-defaults.conf
    - spark-env.sh
    - log4j.properties
    - etc.
- Spark shell can be verbose
  - To view ERRORs only, changed the INFO value to ERROR in the log4j.properties
    - `$SPARK_HOME/conf/log4j.properties`

Spark configuration – Spark properties

- Set application properties via the SparkConf object.

```
val conf = new SparkConf()
      .setMaster("local")
      .setAppName("CountingSheep")
      .set("spark.executor.memory", "1g")
val sc = new SparkContext(conf)
```

- Dynamically setting Spark properties
  - Create a SparkContext with an empty conf

    ```
    val sc = new SparkContext(new SparkConf())
    ```

  - Supply the configuration values during runtime

    ```
    ./bin/spark-submit --name "My app" --master local[4] --conf
    spark.shuffle.spill=false --conf "spark.executor.extraJavaOptions=-
    XX:+PrintGCDetails -XX:+PrintGCTimeStamps" myApp.jar
    ```

  - conf/spark-defaults.conf

- Application web UI

```
http://<driver>:4040
```

Spark tuning

- Data serialization
  - Java serialization – harder, but, more complete
  - Kyro serialization – simpler, but, not as complete

```
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer”)
```

- Memory tuning
  - Amount of memory used by the objects
    - Avoid Java features that add overhead
    - Go with arrays or primitive types
    - Avoid nested structures when possible
  - Cost of accessing those objects
    - Serialized RDD storage
  - Overhead of garbage collection
    - Analyze the garbage collection
    - SPARK_JAVA_OPTS

```
-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps to your SPARK_JAVA_OPTS
```

Spark tuning – other considerations

- Level of parallelism
  - Automatically set according to the file size
  - Optional parameters such as SparkContext.textFile
  - spark.default.parallelism
  - 2-3 tasks per CPU core in the cluster
- Memory usage of reduce tasks
  - OutOfMemoryError can be resolved by increasing the level of parallelism
- Broadcasting large variables
- Serialized size of each tasks are located on the master.
  - Tasks > 20 KB worth optimizing

Spark monitoring

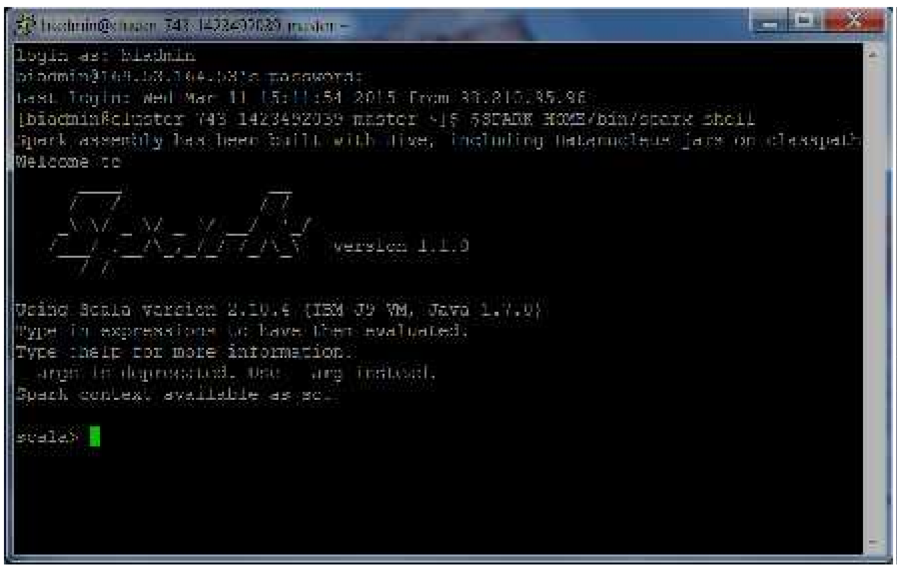
- Three ways to monitor Spark applications
  1. Web UI
     - Port 4040 (lab exercise on port 8088)
     - Available for the duration of the application
  2. Metrics
     - Based on the Coda Hale Metrics Library
     - Report to a variety of sinks (HTTP, JMX, and CSV)
     - /conf/metrics.properties
  3. External instrumentations
     - Ganglia
     - OS profiling tools (dstat, iostat, iotop)
     - JVM utilities (jstack, jmap, jstat, jconsole)

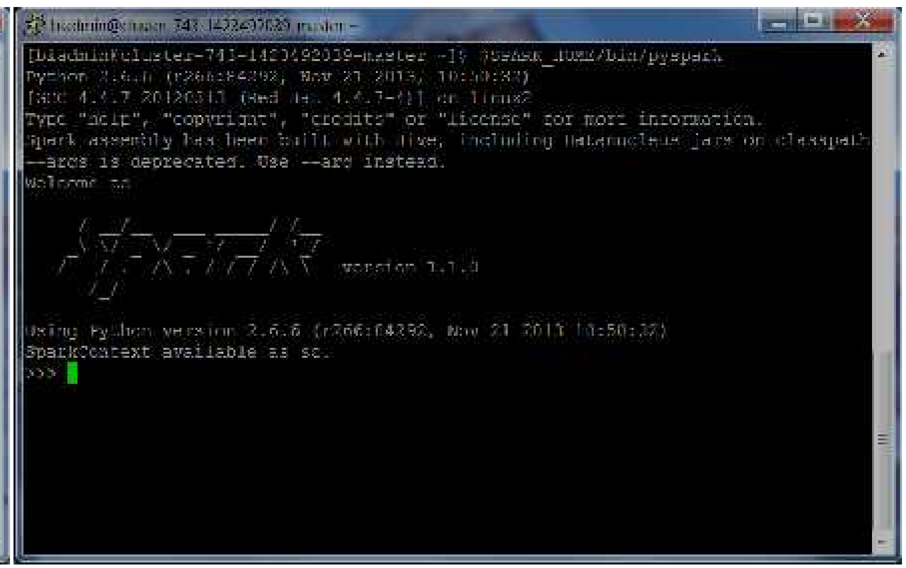
Spark monitoring – Web UI / history server

- Port 4040
- Shows current application
- Contains the following information
  - A list of scheduler stages and tasks
  - A summary of RDD sizes and memory usage
  - Environmental information.
  - Information about the running executors
- Viewing the history (on Mesos or YARN): `./sbin/start-history-server.sh`
- Configure the history server to set
  - Memory allocated
  - JVM options
  - Public address for the server
  - Various properties

Spark jobs and shell

- Spark jobs can be written in Scala, Python, or Java.
- Spark shells for Scala and Python
- APIs are available for all three.
- Must adhere to the appropriate versions for each Spark release.
- Spark's native language is Scala, so it is natural to write Spark applications using Scala.
- The course will cover code examples from Scala, Python and Java.





Application Development with Spark

Brief overview of Scala

- Everything is an Object:
  - Primitive types such as numbers or boolean
  - Functions
- Numbers are objects
  - 1 + 2 * 3 / 4 ➔ (1).+(((2).*(3))./(x)))
  - Where the +, *, / are valid identifiers in Scala
- Functions are objects
  - Pass functions as arguments
  - Store them in variables
  - Return them from other functions
- Function declaration
  - def functionName ([list of parameters]) : [return type]

Scala - anonymous functions

- Functions without a name created for one-time use to pass to another function
- Left side of the right arrow => is where the argument resides (no arguments in the example)
- Right side of the arrow is the body of the function (the println statement)

```scala
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def timeFlies() {
    println("time flies like an arrow...")
  }
  def main(args: Array[String]) {
    oncePerSecond(timeFlies)
  }
}
```

http://docs.scala-lang.org/tutorials/scala-for-java-programmers.html

```scala
object TimerAnonymous {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def main(args: Array[String]) {
    oncePerSecond(() =>
      println("time flies like an arrow..."))
  }
}
```

Spark’s Scala and Python shell

- Spark’s shell provides a simple way to learn the API
- Powerful tool to analyze data interactively.
- The Scala shell runs on the Java VM
  - A good way to use existing Java libraries
- Scala:
  - To launch the Scala shell:

    ```
    ./bin/spark-shell
    ```

  - To read in a text file:

    ```
    scala> val textFile = sc.textFile("README.md")
    ```
- Python:
  - To launch the Python shell:

    ```
    ./bin/pyspark
    ```

  - To read in a text file:

    ```
    >>> textFile = sc.textFile("README.md")
    ```

Shared variables and key-value pairs

- When a function is passed from the driver to a worker, normally a separate copy of the variables are used.
- Two types of variables:
  - Broadcast variables
    - Read-only copy on each machine
    - Distribute broadcast variables using efficient broadcast algorithms
  - Accumulators
    - Variables added through an associative operation
    - Implement counters and sums
    - Only the driver can read the accumulators value
    - Numeric types accumulators. Extend for new types.

Scala: key-value pairs

```
val pair = (‘a’, ‘b’)
pair._1 // will return ‘a’
pair._2 // will return ‘b’
```

Python: key-value pairs

```
pair = (‘a’, ‘b’)
pair[0] # will return ‘a’
pair[1] # will return ‘b’
```

Java: key-value pairs

```
Tuple2 pair = new Tuple2(‘a’, ‘b’);
pair._1 // will return ‘a’
pair._2 // will return ‘b’
```

Programming with key-value pairs

- There are special operations available on RDDs of key-value pairs
  - Grouping or aggregating elements by a key
- Tuple2 objects created by writing (a, b)
  - Must import org.apache.spark.SparkContext._
- PairRDDFunctions contains key-value pair operations
  - reduceByKey((a, b) => a + b)
- Custom objects as key in key-value pair requires a custom equals() method with a matching hashCode() method.
- Example:

```
val textFile = sc.textFile("...")
val readmeCount = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)
```

Linking with Spark - Scala

- Spark applications requires certain dependencies.
- Must have a compatible Scala version to write applications.
  - e.g Spark 1.1.1 uses Scala 2.10.
- To write a Spark application, you need to add a Maven dependency on Spark.
  - Spark is available through Maven Central at:

```
groupId = org.apache.spark
artifactId = spark-core_2.10
version = 1.1.1
```

- To access a HDFS cluster, you need to add a dependency on *hadoop-client* for your version of HDFS

```
groupId = org.apache.hadoop
artifactId = hadoop-client
version = <your-hdfs-version>
```

Linking with Spark - Python

- Spark 1.1.1 works with Python 2.6 or higher (but not Python 3)
- Uses the standard CPython interpreter, so C libraries like NumPy can be used.
- To run Spark applications in Python, use the *bin/spark-submit* script located in the Spark directory.
  - Load Spark's Java/Scala libraries
  - Allow you to submit applications to a cluster
- If you wish to access HDFS, you need to use a build of PySpark linking to your version of HDFS.
- Import some Spark classes

```
from pyspark import SparkContext, SparkConf
```

Linking with Spark - Java

- Spark 1.1.1 works with Java 6 and higher.
  - Java 8 supports lambda expressions
- Add a dependency on Spark
  - Available through Maven Central at:

```
groupId = org.apache.spark
artifactId = spark-core_2.10
version = 1.1.1
```

- If you wish to access an HDFS cluster, you must add the dependency as well.

```
groupId = org.apache.hadoop
artifactId = hadoop-client
version = <your-hdfs-version>
```

- Import some Spark classes

```
import org.apache.spark.api.java.JavaSparkContext
import org.apache.spark.api.java.JavaRDD
import org.apache.spark.SparkConf
```

Initializing Spark - Scala

- Build a SparkConf object that contains information about your application

```
val conf = new SparkConf().setAppName(appName).setMaster(master)
```

- The *appName* parameter → Name for your application to show on the cluster UI
- The *master* parameter → is a Spark, Mesos, or YARN cluster URL (or a special “local” string to run in local mode)
  - In testing, you can pass “local” to run Spark.
  - local[16] will allocate 16 cores
  - In production mode, do not hardcode *master* in the program. Launch with *spark-submit* and provide it there.
- Then, you will need to create the SparkContext object.

```
new SparkContext(conf)
```

Initializing Spark - Python

- Build a SparkConf object that contains information about your application
```
conf = SparkConf().setAppName(appName).setMaster(master)
```
- The *appName* parameter → Name for your application to show on the cluster UI
- The *master* parameter → is a Spark, Mesos, or YARN cluster URL (or a special "local" string to run in local mode)
  - In production mode, do not hardcode *master* in the program. Launch with *spark-submit* and provide it there.
  - In testing, you can pass "local" to run Spark.
- Then, you will need to create the SparkContext object.
```
sc = SparkContext(conf=conf)
```

Initializing Spark - Java

- Build a SparkConf object that contains information about your application
```
SparkConf conf = new SparkConf().setAppName(appName).setMaster(master)
```
- The *appName* parameter → Name for your application to show on the cluster UI
- The *master* parameter → is a Spark, Mesos, or YARN cluster URL (or a special “local” string to run in local mode)
  - In production mode, do not hardcode *master* in the program. Launch with *spark-submit* and provide it there.
  - In testing, you can pass “local” to run Spark.
- Then, you will need to create the JavaSparkContext object.
```
JavaSparkContext sc = new JavaSparkContext(conf);
```

Passing functions to Spark

- Spark's API relies on heavily passing functions in the driver program to run on the cluster
- Three methods
  - Anonymous function syntax

```
(x: Int) => x + 1
```

  - Static methods in a global singleton object

```
object MyFunctions {
       – def func1 (s: String): String = {…}
}
myRdd.map(MyFunctions.func1)
```

  - Passing by reference
    - To avoid sending the entire object, consider copying the function to a local variable.
    - Example:

```
val field = "Hello"
```

    - Avoid:

```
def doStuff(rdd: RDD[String]):RDD[String] = {rdd.map(x => field + x)}
```

    - Consider:

```
def doStuff(rdd: RDD[String]):RDD[String] = {
val field_ = this.field
rdd.map(x => field_ + x) }
```

Programming the business logic

- Spark's API available in Scala, Java, or Python.
- Create the RDD from an external dataset or from an existing RDD.
- Transformations and actions to process the data.
- Use RDD persistence to improve performance
- Use broadcast variables or accumulators for specific use cases

```scala
package org.apache.spark.examples

import org.apache.spark._


object HdfsTest {

  /** Usage: HdfsTest [file] */
  def main(args: Array[String]) {
    if (args.length < 1) {
      System.err.println("Usage: HdfsTest <file>")
      System.exit(1)
    }
    val sparkConf = new SparkConf().setAppName("HdfsTest")
    val sc = new SparkContext(sparkConf)
    val file = sc.textFile(args(0))
    val mapped = file.map(s => s.length).cache()
    for (iter <- 1 to 10) {
      val start = System.currentTimeMillis()
      for (x <- mapped) { x + 2 }
      val end = System.currentTimeMillis()
      println("Iteration " + iter + " took " + (end-start) + " ms")
    }
    sc.stop()
  }
}
```

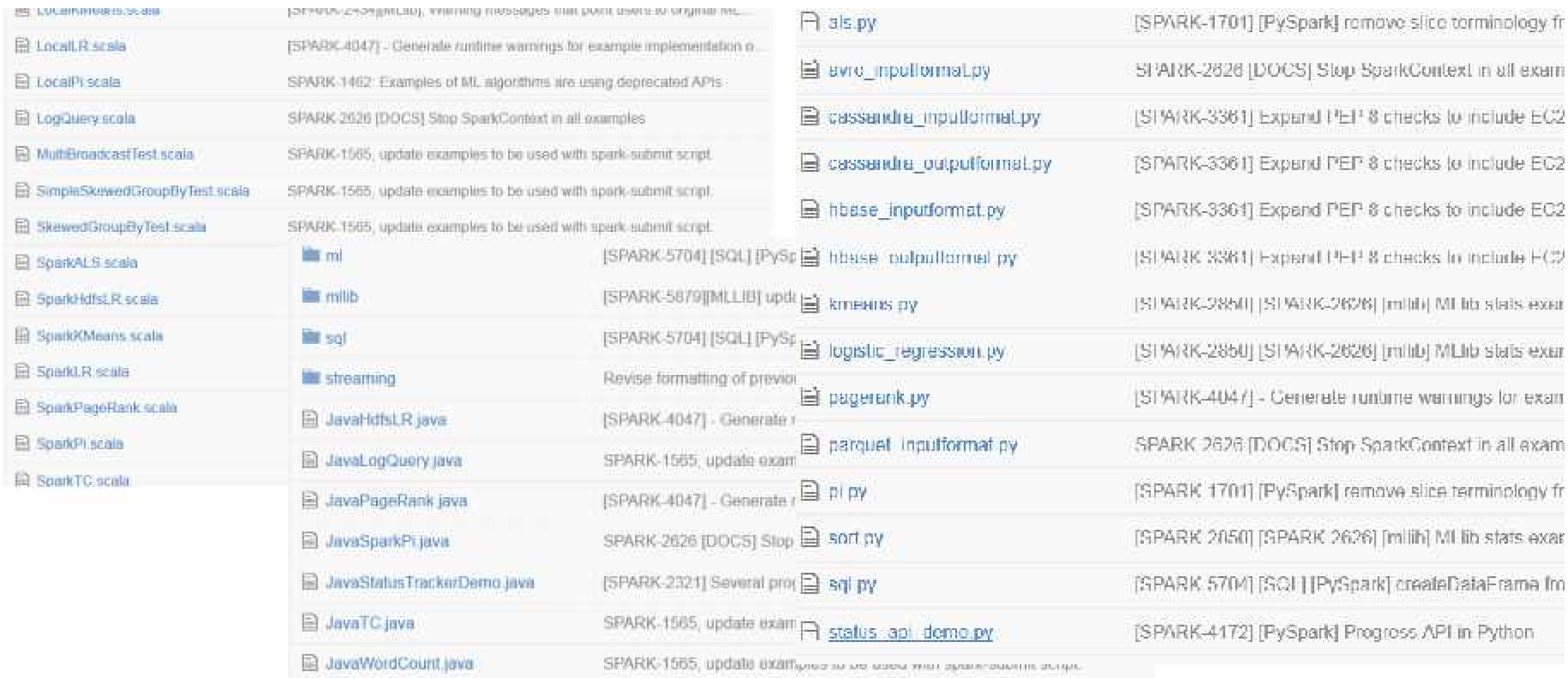
Running Spark examples

- Spark samples available in the *examples* directory
- Run the examples:

  `./bin/run-example SparkPi`

  where *SparkPi* is the name of the sample application
- In Python:

  `./bin/spark-submit examples/src/main/python/pi.py`



Create Spark standalone applications - Scala

```
/* SimpleApp.scala */
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "YOUR_SPARK_HOME/README.md" // Should be some file on your system
    val conf = new SparkConf().setAppName("Simple Application")
    val sc = new SparkContext(conf)
    val logData = sc.textFile(logFile, 2).cache()
    val numAs = logData.filter(line => line.contains("a")).count()
    val numBs = logData.filter(line => line.contains("b")).count()
    println("Lines with a: %s, Lines with b: %s".format(numAs, numBs))
  }
}
```

Import statements

SparkConf and SparkContext

Transformations + Actions

Create Spark standalone applications – Python

Import statement

```
"""SimpleApp.py"""
from pyspark import SparkContext

logFile = "YOUR_SPARK_HOME/README.md"  # Should be some file on your system
sc = SparkContext("local", "Simple App")
logData = sc.textFile(logFile).cache()

numAs = logData.filter(lambda s: 'a' in s).count()
numBs = logData.filter(lambda s: 'b' in s).count()

print "Lines with a: %i, lines with b: %i" % (numAs, numBs)
```

SparkContext

Transformations + Actions

Create Spark standalone applications – Java

```
/* SimpleApp.java */
import org.apache.spark.api.java.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.function.Function;

public class SimpleApp {
  public static void main(String[] args) {
    String logFile = "YOUR_SPARK_HOME/README.md"; // Should be some file on your system
    SparkConf conf = new SparkConf().setAppName("Simple Application");
    JavaSparkContext sc = new JavaSparkContext(conf);
    JavaRDD<String> logData = sc.textFile(logFile).cache();

    long numAs = logData.filter(new Function<String, Boolean>() {
      public Boolean call(String s) { return s.contains("a"); }
    }).count();

    long numBs = logData.filter(new Function<String, Boolean>() {
      public Boolean call(String s) { return s.contains("b"); }
    }).count();

    System.out.println("Lines with a: " + numAs + ", lines with b: " + numBs);
  }
}
```

Import statements

JavaSparkContext

Transformations + Actions

Run standalone applications

- Define the dependencies – can use any system builds (Ant, sbt, Maven)
- Example:
  - Scala → simple.sbt
  - Java → pom.xml
  - Python → --py-files argument (not needed for SimpleApp.py)
- Create the typical directory structure with the files

Scala using SBT :	Java using Maven:
`./simple.sbt`	`./pom.xml`
`./src`	`./src`
`./src/main`	`./src/main`
`./src/main/scala`	`./src/main/java`
`./src/main/scala/SimpleApp.scala`	`./src/main/java/SimpleApp.java`

- Create a JAR package containing the application's code.
  - Scala: sbt
  - Java: mvn
  - Python: submit-spark
- Use spark-submit to run the program

Submit applications to the cluster

- Package application into a JAR (Scala/Java) or set of .py or .zip (for Python)
- Use spark-submit under the $SPARK_HOME/bin directory

```
./bin/spark-submit \
--class <main-class> \
--master <master-url> \
--deploy-mode <deploy-mode> \
--conf <key>=<value> \
... # other options
<application-jar> \
[application-arguments]
```

- `spark-submit --help` will show you the other options
- Example of running an application locally on 8 cores:

```
./bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master local[8] \
/path/to/examples.jar \
100
```

Downloading and installing Spark standalone

- Runs on both Windows and Unit-like systems (e.g Linux, Mac OS)
- To run locally on one machine, all you need is to have Java installed on your system PATH or the JAVA_HOME pointing to a valid Java installation.
- Visit this page to download: http://spark.apache.org/downloads.html
  - Select the Hadoop distribution you require under the "Pre-built packages"
  - Place a compiled version of Spark on each node on the cluster.
- Manually start the cluster by executing:
  - *./sbin/start-master.sh*
- Once started, the master will print out a spark://HOST:PORT URL for itself, which you can use to connect workers to it.
  - The default master's web UI is http://localhost:8080
- Check out Spark's website for more information
  - http://spark.apache.org/docs/latest/spark-standalone.html