Intro To Apache Spark

Gurumurthy Yelaswarapu

Andrew Zhao

Target Audience

- Software Engineers
- Data Engineers
- ETL Developers

Course Prerequisites

- No prior knowledge of Spark, Hadoop or distributed programming concepts is required
- Requirements
  - Basic familiarity with Linux or Unix
  - Intermediate-level programming skills in either Scala or Python

Success Criteria

By end of day, participants will be comfortable with the following:

- open a Spark Shell
- explore data sets loaded from HDFS, etc.
- review Spark SQL, Spark Streaming,
- use of some ML algorithms
- return to workplace and demo use of Spark!

Outline Basic

- What is Spark(20m)
- Tour of Spark operations(40m)
- Spark Run Runtime: Job execution(20m)
- Lab1: Log Mining Example(30m)
- Lab2: Join Examples(20m)
- Lab3: Spark SQL(30m)
- Lab4: Spark Streaming (30m)
- Lab5:Spark MLlib(30m)

What is Spark

What is Spark?

- **Fast, expressive cluster computing system compatible with Apache Hadoop**
  - Works with any Hadoop-supported storage system (HDFS, S3, Avro, …)
- **Improves efficiency through:**
  - In-memory computing primitives
  - General computation graphs

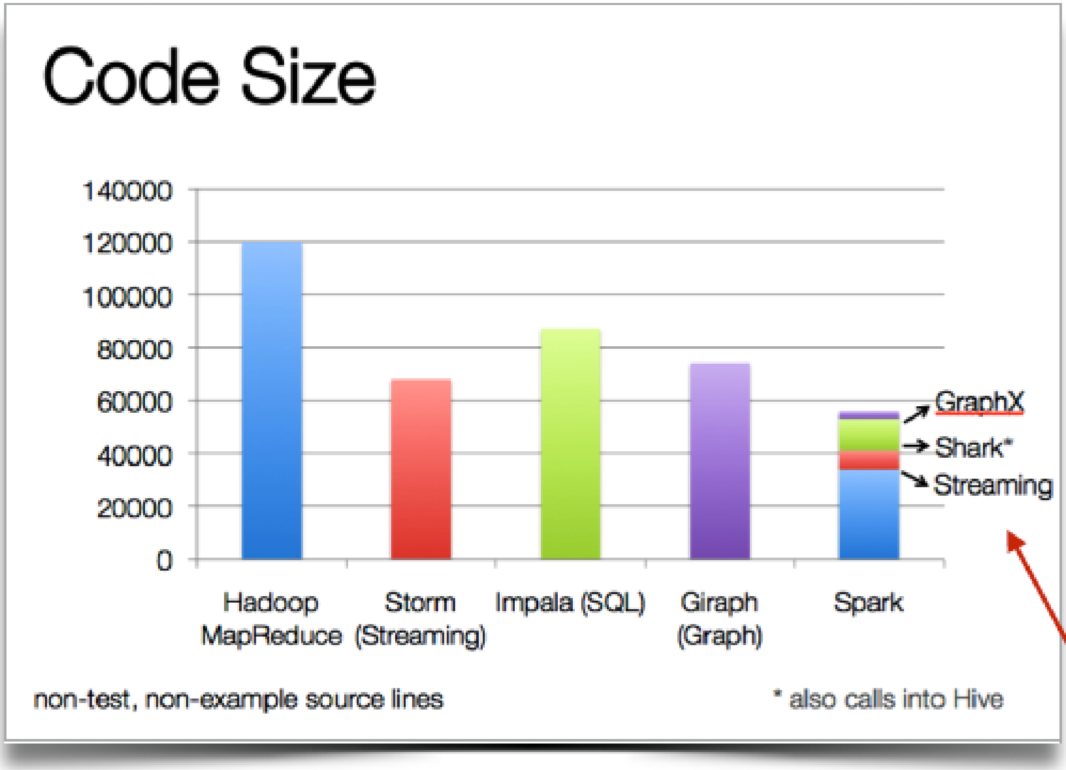
→ **Up to 100x faster**

- **Improves usability through:**
  - Rich APIs in Java, Scala, Python,R
  - Interactive shell

→ **Often 2-10x less code**

- **Handles batch, interactive, and real-time within a single framework**
  - Spark SQL : For SQL and unstructured data processing
  - Spark Streaming: Stream processing of live data streams
  - Spark Mllib: Machine Learning Algorithms
  - GraphX: Graph Processing

What is Spark?

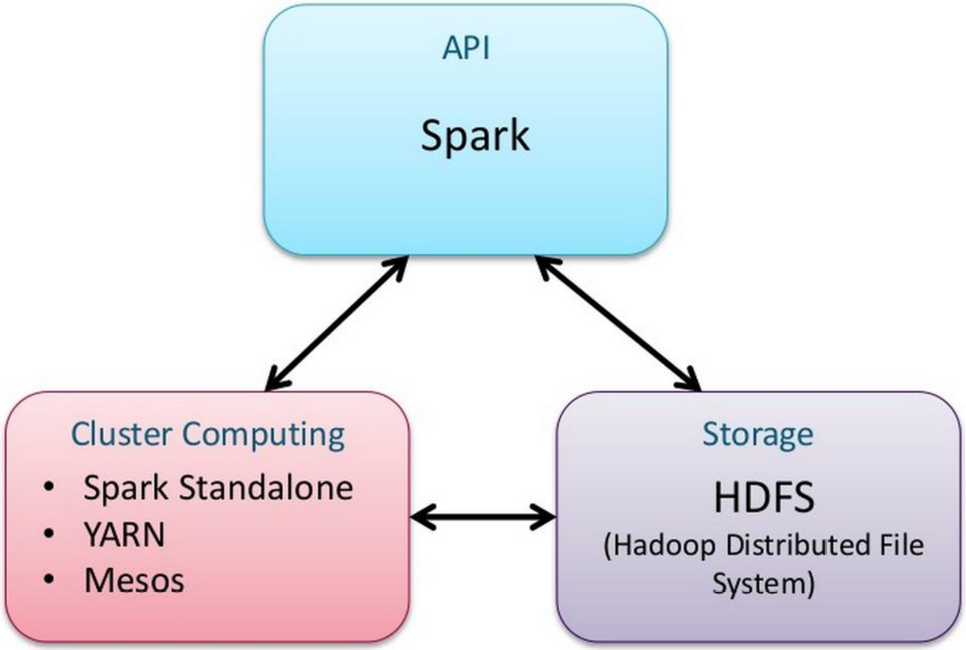


The State of Spark, and Where We're Going Next
Matei Zaharia
Spark Summit (2013)
youtu.be/nU6vO2EJAb4

used as libs, instead of specialized systems

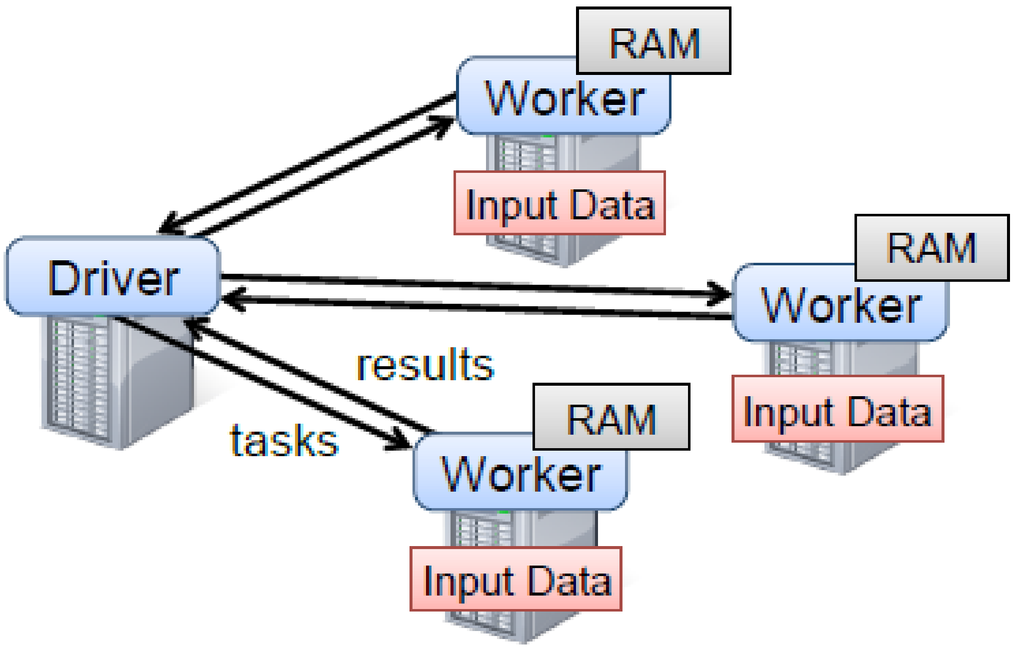
How to Run It ?

- Local multicore: just a library in your program
- EC2: scripts for launching a Spark cluster
- Private cluster: Mesos, YARN, Standalone Mode



Spark Runtime

- The user's driver program launches multiple workers,
- The user's driver read data blocks from a distributed file system
- Can persist computed RDD partitions in memory.



Languages

- APIs in Java, Scala, Python, R→ for large scale data processing
- Interactive shells in Scala and Python→ for learning or data exploration

Python Shell

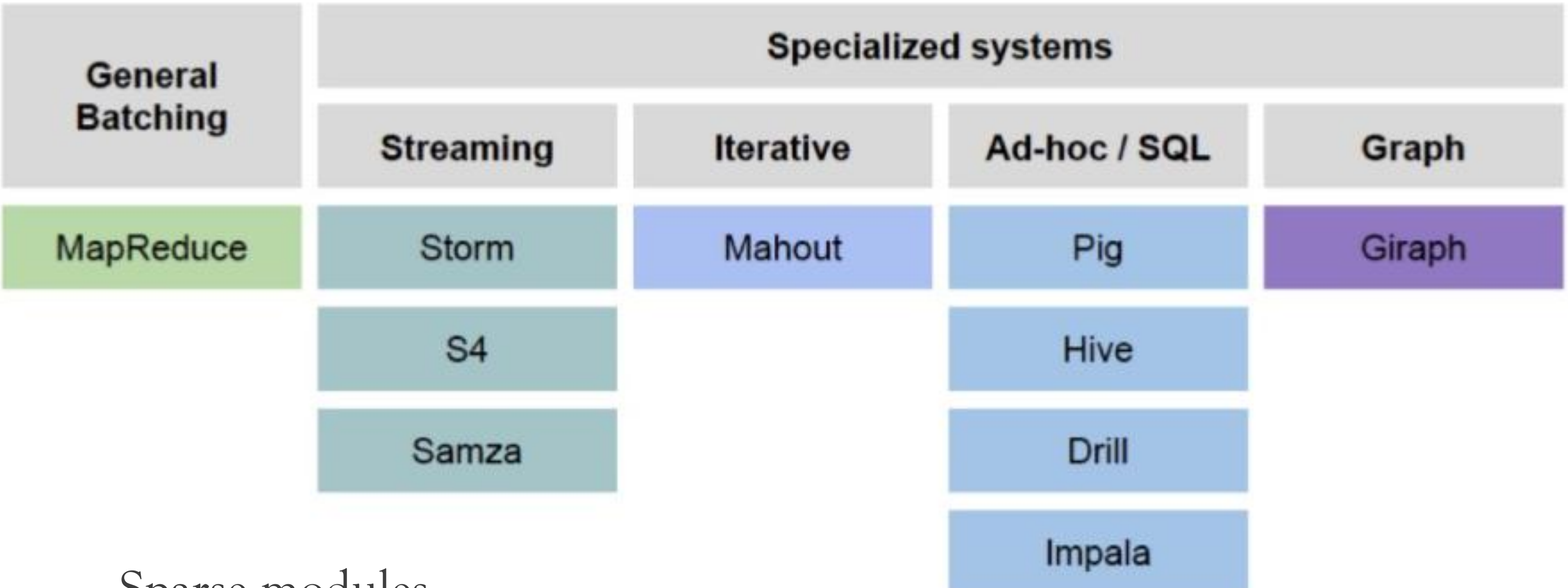
```
$ pyspark

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 0.9.1
      /_/

Using Python version 2.6.6 (r266:84292, Jan
22 2014 09:42:36)
Spark context available as sc.

>>>
```

Scala Shell

```
$ spark-shell

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 0.9.1
      /_/

Using Scala version 2.10.3 (Java HotSpot(TM)
64-Bit Server VM, Java 1.7.0_51)
Created spark context..
Spark context available as sc.

scala>
```

Why Spark

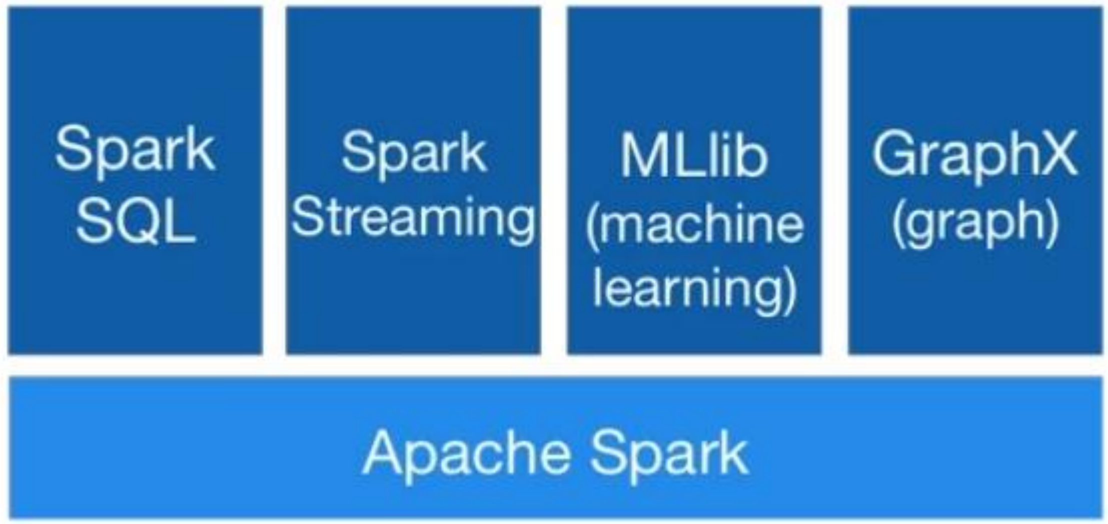
Hadoop: No Unified Vision



- Sparse modules
- Diversity APIs
- High operational costs

Why Spark

Spark: A Unified Pipeline



- Spark SQL : For SQL and unstructured data processing
- Spark Streaming: Stream processing of live data streams
- Spark Mllib: Machine Learning Algorithms
- GraphX: Graph Processing

Key Idea

- **Work with distributed collections as you would with local ones**
- Concept: resilient distributed datasets (RDDs)
  - Immutable collections of objects spread across a cluster
  - Built through parallel transformations (map, filter, etc)
  - Automatically rebuilt on failure
  - Controllable persistence (e.g. caching in RAM)

Operations

- Transformations (e.g. map, filter, groupBy, join)
  - Lazy operations to build RDDs from other RDDs
- Actions (e.g. count, collect, save)
  - Return a result or write it to storage

Which Language Should I Use?

- **Standalone programs can be written in any, but console is only Python & Scala**
- **Python developers:** can stay with Python for both
- **Java developers:** consider using Scala for console (to learn the API)
- **Performance:** Java / Scala will be faster (statically typed), but Python can do well for numerical work with NumPy

Scala Cheat Sheet

Variables:

```
var x: Int = 7
var x = 7       // type inferred

val y = “hi”   // read-only
```

Functions:

```
def square(x: Int): Int = x*x

def square(x: Int): Int = {
  x*x   // last line returned
}
```

Collections and closures:

```
val nums = Array(1, 2, 3)

nums.map((x: Int) => x + 2) // => Array(3, 4, 5)

nums.map(x => x + 2)  // => same
nums.map(_ + 2)       // => same

nums.reduce((x, y) => x + y) // => 6
nums.reduce(_ + _)           // => 6
```

Java interop:

```
import java.net.URL

new URL(“http://cnn.com”).openStream()
```

More details:
scala-lang.org

Tour of Spark operations

Learning Spark

- Easiest way: Spark interpreter (spark-shell or pyspark)
  - Special Scala and Python consoles for cluster use
- Runs in local mode on 1 thread by default, but can control with MASTER environment var:

```
MASTER=local      ./spark-shell            # local, 1 thread
MASTER=local[2]   ./spark-shell            # local, 2 threads
MASTER=spark://host:port ./spark-shell    # Spark standalone cluster
```

First Stop: SparkContext

- Main entry point to Spark functionality
- Created for you in Spark shells as variable sc
- In standalone programs, you'd make your own (see later for details)

Creating RDDs

```
# Turn a local collection into an RDD
sc.parallelize([1, 2, 3])

# Load text file from local FS, HDFS, or S3
sc.textFile("file.txt")
sc.textFile("directory/*.txt")
sc.textFile("hdfs://namenode:9000/path/file")

# Use any existing Hadoop InputFormat
sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```

Basic Transformations

```
nums = sc.parallelize([1, 2, 3])

# Pass each element through a function
squares = nums.map(lambda x: x*x)   # => {1, 4, 9}

# Keep elements passing a predicate
even = squares.filter(lambda x: x % 2 == 0) # => {4}

# Map each element to zero or more others
nums.flatMap(lambda x: range(0, x))  # => {0, 0, 1, 0, 1, 2}
```

Range object (sequence of numbers 0, 1, …, x-1)

Basic Actions

```
nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local collection
nums.collect() # => [1, 2, 3]

# Return first K elements
nums.take(2)   # => [1, 2]

# Count number of elements
nums.count()   # => 3

# Merge elements with an associative function
nums.reduce(lambda x, y: x + y)  # => 6

# Write elements to a text file
nums.saveAsTextFile("hdfs://file.txt")
```

Working with Key-Value Pairs

- **Spark's "distributed reduce" transformations act on RDDs of key-value pairs**
- Python:
```
pair = (a, b)
pair[0] # => a
pair[1] # => b
```
- Scala:
```
val pair = (a, b)
pair._1 // => a
pair._2 // => b
```
- Java:
```
Tuple2 pair = new Tuple2(a, b);  // class scala.Tuple2
pair._1 // => a
pair._2 // => b
```

Some Key-Value Operations

```
pets = sc.parallelize([("cat", 1), ("dog", 1), ("cat", 2)])

pets.reduceByKey(lambda x, y: x + y)
# => {(cat, 3), (dog, 1)}

pets.groupByKey()
# => {(cat, Seq(1, 2)), (dog, Seq(1)}

pets.sortByKey()
# => {(cat, 1), (cat, 2), (dog, 1)}
```

Multiple Datasets

```
visits = sc.parallelize([(“index.html”, “1.2.3.4”),
                         (“about.html”, “3.4.5.6”),
                         (“index.html”, “1.3.3.1”)])

pageNames = sc.parallelize([(“index.html”, “Home”), (“about.html”, “About”)])

visits.join(pageNames)
# (“index.html”, (“1.2.3.4”, “Home”))
# (“index.html”, (“1.3.3.1”, “Home”))
# (“about.html”, (“3.4.5.6”, “About”))

visits.cogroup(pageNames)
# (“index.html”, (Seq(“1.2.3.4”, “1.3.3.1”), Seq(“Home”)))
# (“about.html”, (Seq(“3.4.5.6”), Seq(“About”)))
```

Using Local Variables

- **External variables you use in a closure will automatically be shipped to the cluster:**

```
query = raw_input("Enter a query:")
pages.filter(lambda x: x.startswith(query)).count()
```

- **Some caveats:**
  - Each task gets a new copy (updates aren't sent back)
  - Variable must be Serializable (Java/Scala) or Pickle-able (Python)
  - Don't use fields of an outer object (ships all of it!)

Closure Mishap Example

```
class MyCoolRddApp {
  val param = 3.14
  val log = new Log(...)
  ...

  def work(rdd: RDD[Int]) {
    rdd.map(x => x + param)
       .reduce(...)
  }
}
```

NotSerializableException: MyCoolRddApp (or Log)

How to get around it:

```
class MyCoolRddApp {
  ...

  def work(rdd: RDD[Int]) {
    val param_ = param
    rdd.map(x => x + param_)
       .reduce(...)
  }
}
```

References only local variable instead of `this.param`

Spark Essentials: Transformations

transformation	*description*
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
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)
sample(*withReplacement, fraction, seed*)	sample a fraction *fraction* of the data, with or without replacement, using a given random number generator *seed*
union(*otherDataset*)	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks*]))	return a new dataset that contains the distinct elements of the source dataset

Spark Essentials: Transformations

transformation	*description*
`groupByKey([numTasks])`	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, Seq[V])` pairs
`reduceByKey(func, [numTasks])`	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
`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, as specified in the boolean ascending argument
`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
`cogroup(otherDataset, [numTasks])`	when called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, Seq[V], Seq[W])` tuples – also called `groupWith`
`cartesian(otherDataset)`	when called on datasets of types `T` and `U`, returns a dataset of `(T, U)` pairs (all pairs of elements)

Spark Essentials: Actions

action	*description*
`reduce(func)`	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also 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 – usually useful after a filter or other operation that returns a sufficiently small subset of the data
`count()`	return the number of elements in the dataset
`first()`	return the first element of the dataset – similar to *take(1)*
`take(n)`	return an array with the first *n* elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
`takeSample(withReplacement, fraction, seed)`	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

Spark Essentials: Actions

action	*description*
`saveAsTextFile(`*`path`*`)`	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call `toString` on each element to convert it to a line of text in the file
`saveAsSequenceFile(`*`path`*`)`	write the elements of the dataset as a Hadoop `SequenceFile` in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
`countByKey()`	only available on RDDs of type `(K, V)`. Returns a \`Map\` of `(K, Int)` pairs with the count of each key
`foreach(`*`func`*`)`	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

Spark Essentials: Persistence

- Spark can persist (or cache) a dataset in memory across operations
- Each node stores in memory any slices of it that it computes and reuses them in other actions on that dataset – often making future actions more than 10x faster
- The cache is fault-tolerant: if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it

Spark Essentials: Persistence

transformation	*description*
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.

Spark Essentials: Persistence

transformation	*description*
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.

Spark Essentials: Broadcast Variables

- Broadcast variables let programmer keep a read-only variable cached on each machine rather than shipping a copy of it with tasks
- For example, to give every node a copy of a large input dataset efficiently
- Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost

Spark Essentials: Broadcast Variables

Scala:

```
val broadcastvar = sc.broadcast(Array(1, 2, 3))
broadcastvar.value
```

Python:

```
broadcastvar = sc.broadcast(list(range(1, 4)))
broadcastvar.value
```

Spark Essentials: Accumulators

- Accumulators are variables that can only be “added” to through an associative operation Used to implement counters and sums, efficiently in parallel
- Spark natively supports accumulators of numeric value types and standard mutable collections, and programmers can extend for new types
- Only the driver program can read an accumulator’s value, not the tasks

Spark Essentials: Accumulators

Scala:

```
val accum = sc.accumulator(0)
sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)

accum.value
```

Python:

```
accum = sc.accumulator(0)
rdd = sc.parallelize([1, 2, 3, 4])
def f(x):
    global accum
    accum += x
rdd.foreach(f)

accum.value
```

Driver Side

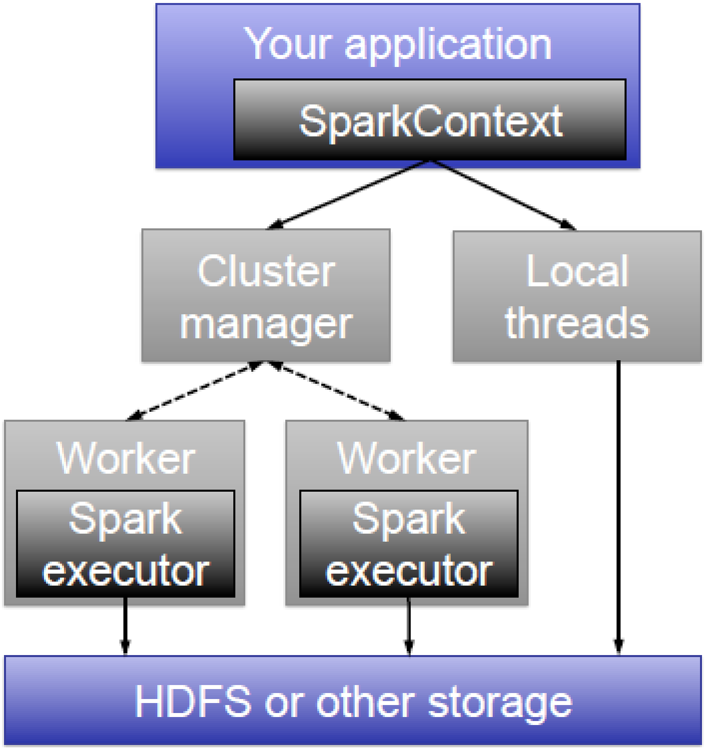
More Details

- Spark supports lots of other operations!
- Full programming guide: spark-project.org/documentation

Spark Run Runtime:
Job execution

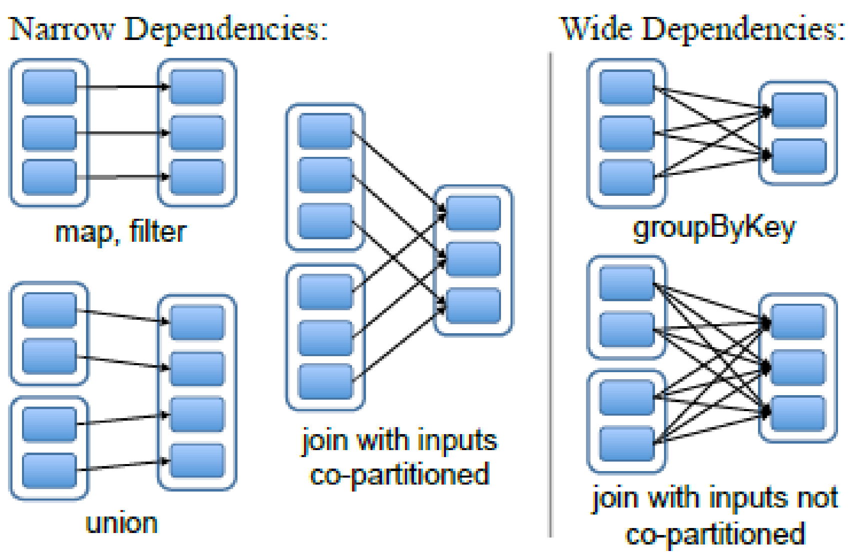
Software Components

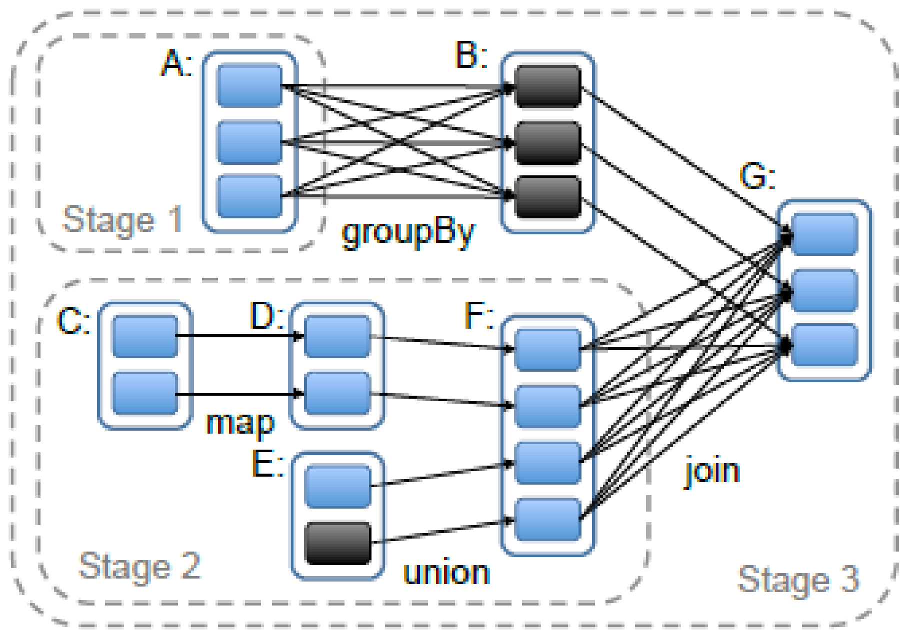
- Spark runs as a library in your program
  - One instance per app
- Runs tasks locally or on a cluster
  - Standalone deploy cluster, Mesos or YARN
- Accesses storage via Hadoop InputFormat API
  - Can use HBase, HDFS, S3, …



Task Scheduler

- Supports general task graphs
- Pipelines functions where possible
- Cache-aware data reuse & locality
- Partitioning-aware to avoid shuffles





Hadoop Compatibility

- Spark can read/write to any storage system / format that has a plugin for Hadoop!
  - Examples: HDFS, S3, HBase, Cassandra, Avro, SequenceFile
  - Reuses Hadoop's InputFormat and OutputFormat APIs
- APIs like SparkContext.textFile support filesystems, while SparkContext.hadoopRDD allows passing any Hadoop JobConf to configure an input source

Log Mining Example

Log Mining Example

```
// load error messages from a log into memory
// then interactively search for various patterns
// base RDD
val lines = sc.textFile("/andrew/data/basic/error_log.txt")
// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()
// action 1
messages.filter(_.contains("mysql")).count()
// action 2
messages.filter(_.contains("php")).count()
```

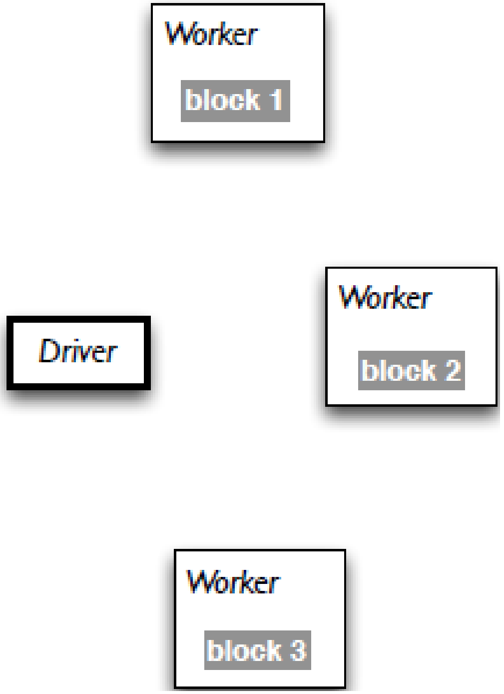
Log Mining Example

At this point, take a look at the transformed RDD *operator graph*:

messages.toDebugString

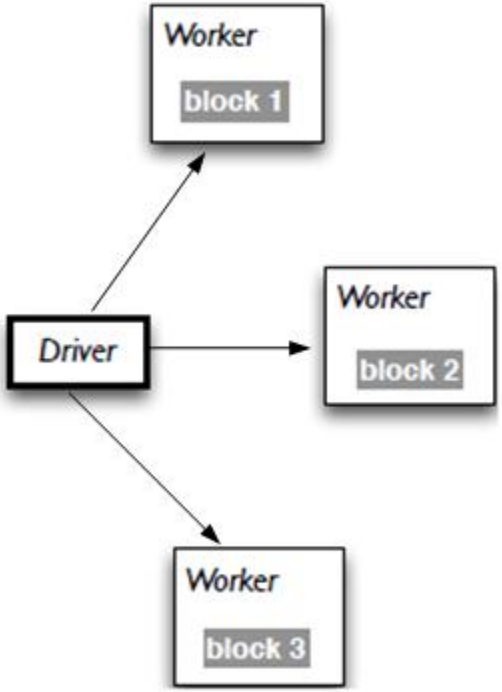
Log Mining Example

```
// base RDD
val lines = sc.textFile("/andrew/data/basic/error_log.txt")
// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()
// action 1
messages.filter(_.contains("mysql")).count()
```



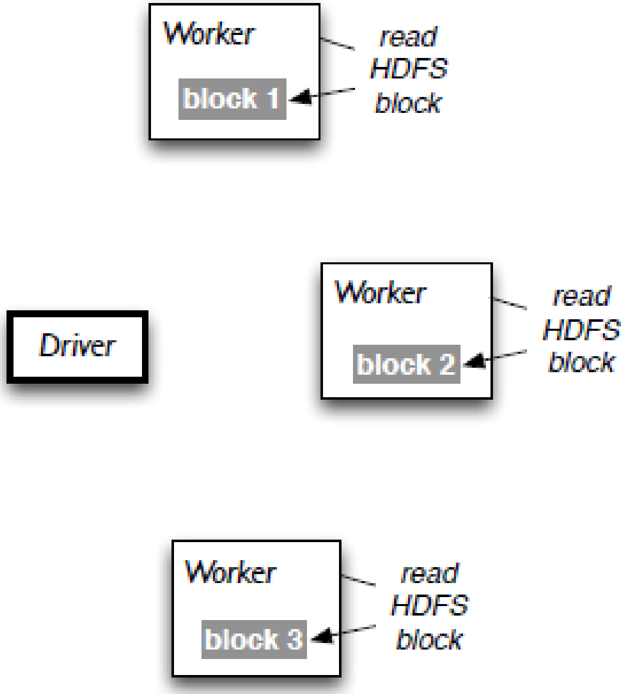
Log Mining Example

```
// base RDD
val lines = sc.textFile("/andrew/data/basic/error_log.txt")
// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()
// action 1
messages.filter(_.contains("mysql")).count()
```



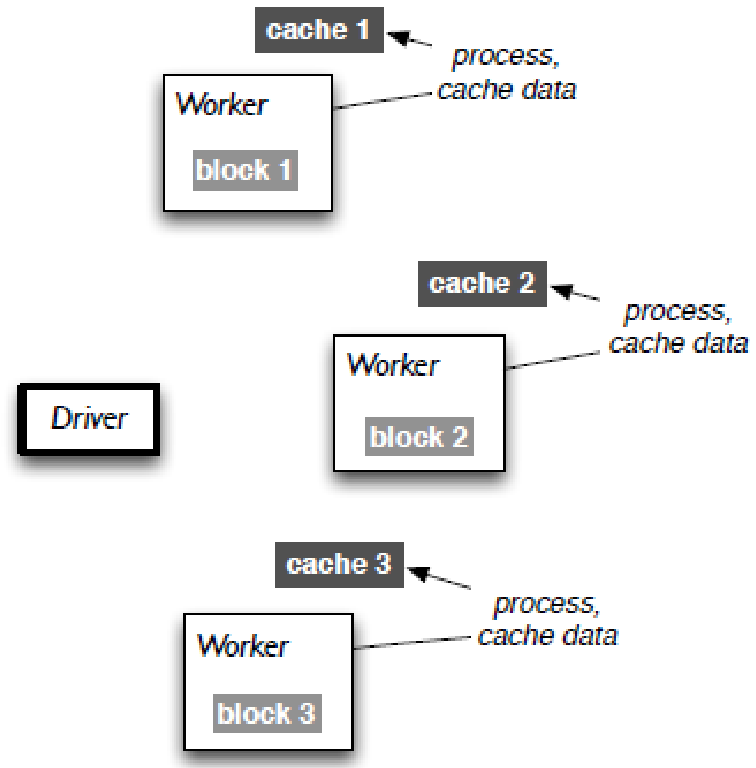
Log Mining Example

```
// base RDD
val lines = sc.textFile("/andrew/data/basic/error_log.txt")
// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()
// action 1
messages.filter(_.contains("mysql")).count()
```



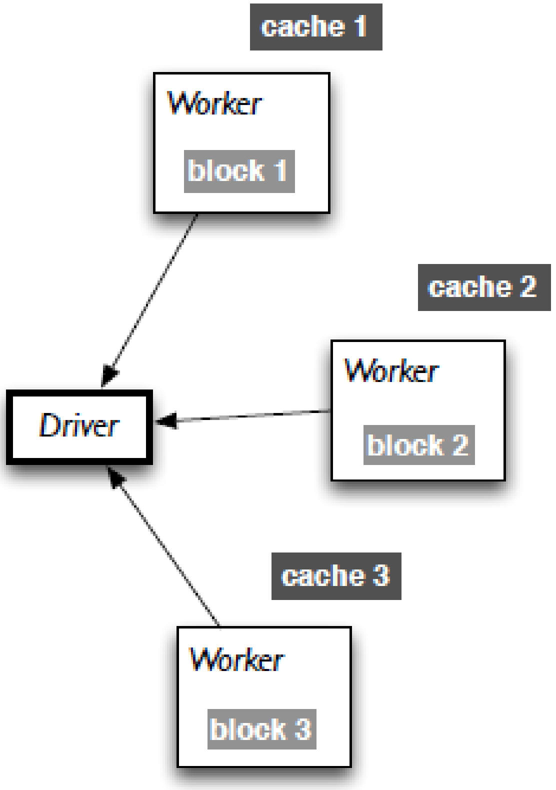
Log Mining Example

```
// base RDD
val lines = sc.textFile("/andrew/data/basic/error_log.txt")
// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r =>
messages.cache()
// action 1
messages.filter(_.contains("mysql")).count()
```



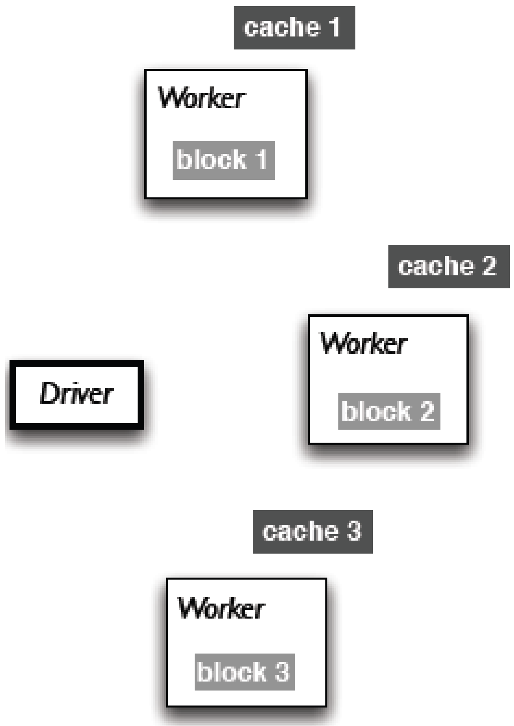
Log Mining Example

```
// base RDD
val lines = sc.textFile("/andrew/data/basic/error_log.txt")
// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()
// action 1
messages.filter(_.contains("mysql")).count()
```



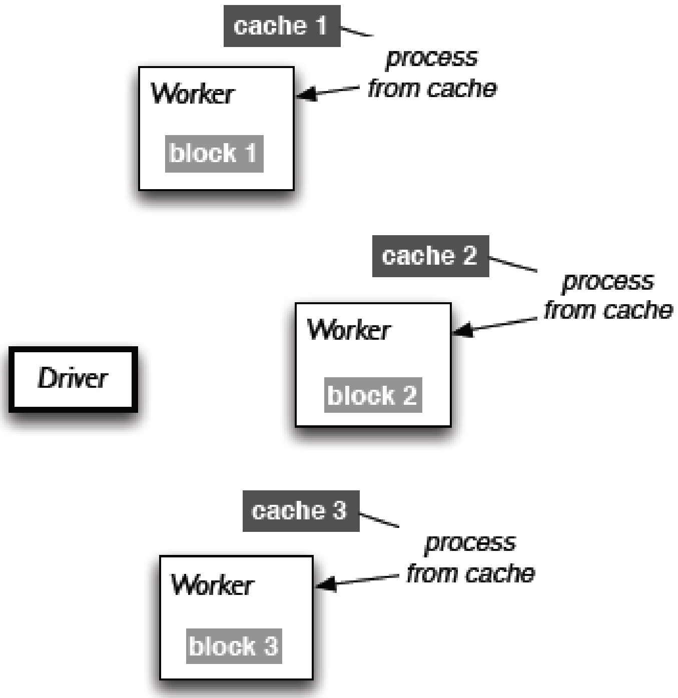
Log Mining Example

```
// base RDD
val lines = sc.textFile("/andrew/data/basic/error_log.txt")
// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()
// action 1
messages.filter(_.contains("mysql")).count()
// action 2
messages.filter(_.contains("php")).count()
```



Log Mining Example

```
// base RDD
val lines = sc.textFile(“/andrew/data/basic/error_log.txt")
// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()
// action 1
messages.filter(_.contains("mysql")).count()
// action 2
messages.filter(_.contains("php")).count()
```



Log Mining Example

// base RDD

val lines = sc.textFile("/andrew/data/basic/error_log.txt")

// transformed RDDs

val errors = lines.filter(_.startsWith("ERROR"))

val messages = errors.map(_.split("\t")).map(r => r(1))

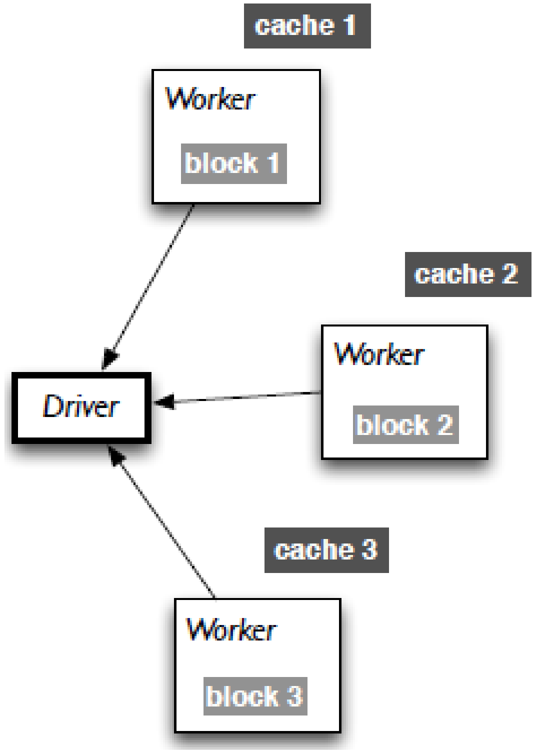
messages.cache()

// action 1

messages.filter(_.contains("mysql")).count()

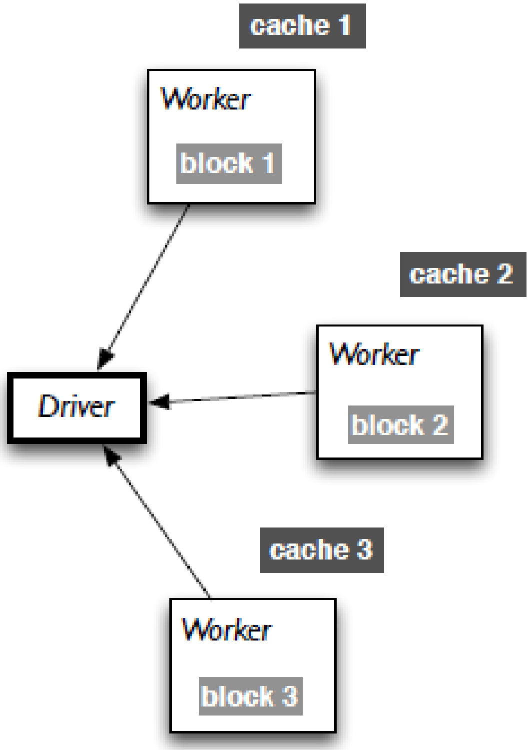
// action 2

messages.filter(_.contains("php")).count()



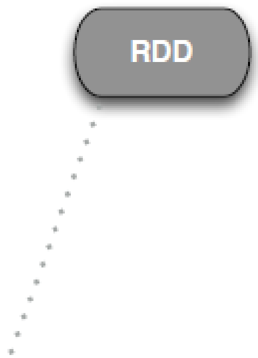
Log Mining Example

```
// base RDD
val lines = sc.textFile("/andrew/data/basic/error_log.txt")
// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()
// action 1
messages.filter(_.contains("mysql")).count()
// action 2
messages.filter(_.contains("php")).count()
```



Log Mining Example

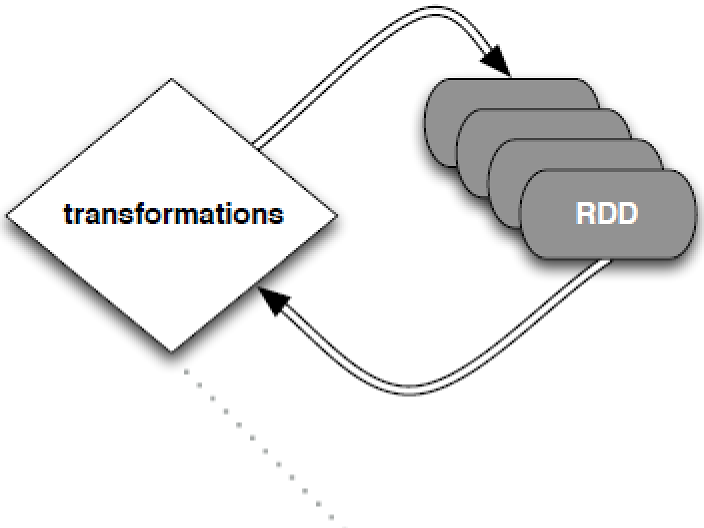
Looking at the RDD transformations and actions from another perspective…




```
// base RDD
val lines = sc.textFile("/andrew/data/basic/error_log.txt")
```

Log Mining Example

Looking at the RDD transformations and actions from another perspective…

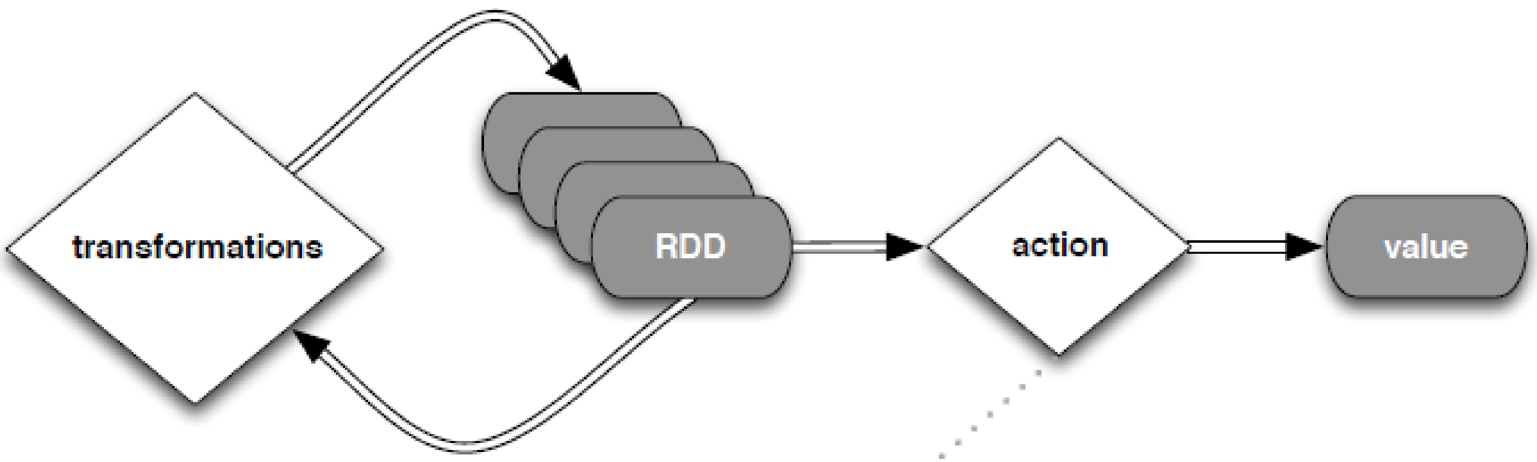


// transformed RDDs

```
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
```

Log Mining Example

Looking at the RDD transformations and actions from another perspective…



// action 1

```
messages.filter(_.contains("mysql")).count()
```

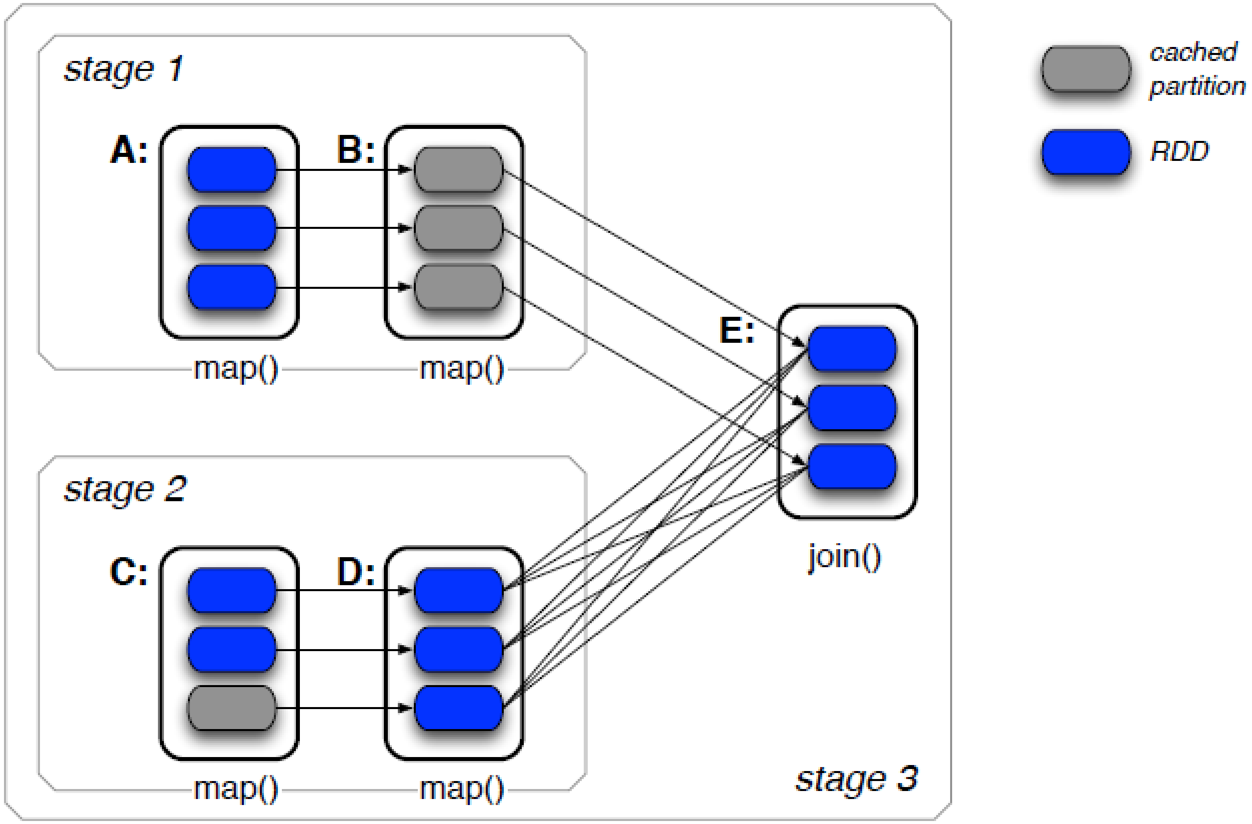
Join Example

Join Example~ *Source Code*

```
val format = new java.text.SimpleDateFormat("yyyy-MM-dd")

case class Register (d: java.util.Date, uuid: String, cust_id: String, lat: Float, lng: Float)

case class Click (d: java.util.Date, uuid: String, landing_page: Int)

val reg = sc.textFile("/andrew/data/join/reg.tsv").map(_.split("\t")).map(
  r => (r(1), Register(format.parse(r(0)), r(1), r(2), r(3).toFloat, r(4).toFloat))
)

val clk = sc.textFile("/andrew/data/join/clk.tsv").map(_.split("\t")).map(
  c => (c(1), Click(format.parse(c(0)), c(1),   c(2).trim.toInt))
)

reg.join(clk).take(2)
```

Join Example- *Operator Graph*

reg.join(clk).toDebugString



Spark SQL

Data Workflows: Spark SQL

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
import sqlContext._
// Define the schema using a case class.
case class Person(name: String, age: Int)
// Create an RDD of Person objects and register it as a table.
val people = sc.textFile(“andre/data/basic/people.txt").map(_.split(",")).map(p =>
Person(p(0), p(1).trim.toInt))

people.registerAsTable("people")

// SQL statements can be run by using the sql methods provided by sqlContext.
val teenagers = sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

// The results of SQL queries are SchemaRDDs and support all the
// normal RDD operations.
// The columns of a row in the result can be accessed by ordinal.
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

Data Workflows: Spark SQL:Parquet

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
import sqlContext._
// Define the schema using a case class.
case class Person(name: String, age: Int)
// Create an RDD of Person objects and register it as a table.
val people = sc.textFile("/andrew/data/basic/people.txt").
map(_.split(",")).map(p => Person(p(0), p(1).trim.toInt))
people.registerAsTable("people")
// The RDD is implicitly converted to a SchemaRDD  allowing it to be stored using parquet.
people.saveAsParquetFile("people.parquet")
// Read in the parquet file created above. Parquet files are  self-describing so the schema is preserved.
// The result of loading a parquet file is also a JavaSchemaRDD.
val parquetFile = sqlContext.parquetFile("people.parquet")
//Parquet files can also be registered as tables and then used in  SQL statements.
parquetFile.registerAsTable("parquetFile")
val teenagers = sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")
teenagers.collect().foreach(println)
```

Data Workflows: Spark SQL:DSL

- Spark SQL also provides a DSL for queries
- Scala symbols represent columns in the underlying table, which are identifiers prefixed with a tick ('))

Data Workflows: Spark SQL:DSL

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
import sqlContext._
// Define the schema using a case class.

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

// Create an RDD of Person objects and register it as a table.

val people = sc.textFile("/andrew/data/basic/people.txt").map(_.split(",")).map(p => Person(p(0), p(1).trim.toInt))

people.registerAsTable("people")

// The following is the same as

// 'SELECT name FROM people WHERE age >= 13 AND age <= 19'

val teenagers = people.where('age >= 13).where('age <= 19).select('name)


// The results of SQL queries are SchemaRDDs and support all the  normal RDD operations.

// The columns of a row in the result can be accessed by ordinal.

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

Spark Streaming

Data Workflows: Spark Streaming

Spark Streaming extends the core API to allow high-throughput, fault-tolerant stream processing of live data streams

Data Workflows: Spark Streaming

Spark Streaming extends the core API to allow high-throughput, fault-tolerant stream processing of live data streams,

- Data can be ingested from many sources: **Kafka**, **Flume**, **Twitter**, **ZeroMQ**, TCP sockets, etc.
- Results can be pushed out to filesystems, databases, live dashboards, etc.
- Spark's built-in machine learning algorithms and graph processing algorithms can be applied to data streams

Data Workflows: Spark Streaming

```
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._
// Create a StreamingContext with a SparkConf configuration
val ssc = new StreamingContext(sparkConf, Seconds(10))
// Create a DStream that will connect to serverIP:serverPort
val lines = ssc.socketTextStream("127.0.0.1", 9999)
// Split each line into words
val words = lines.flatMap(_.split(" "))
// Count each word in each batch
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
// Print a few of the counts to the console
wordCounts.print()
ssc.start() // Start the computation
ssc.awaitTermination() // Wait for the computation to terminate
```

Data Workflows: Spark Streaming

```
# in one terminal run the NetworkWordCount example in Spark Streaming
# expecting a data stream on the localhost:9999 TCP socket
./bin/run-example org.apache.spark.examples.streaming.NetworkWordCount localhost  9999


# in another terminal use Netcat http://nc110.sourceforge.net/
# to generate a data stream on the localhost:9999 TCP socket
$ nc -lk 9999
hello world
hi there fred
what a nice world there
```

Spark ML

Data Workflows: Spark MLlib

Transformers

A Transformer is an abstraction which includes feature transformers and learned models.
→transform()

- A feature transformer might take a dataset, read a column (e.g., text), convert it into a new column (e.g., feature vectors), append the new column to the dataset, and output the updated dataset.
- A learning model might take a dataset, read the column containing feature vectors, predict the label for each feature vector, append the labels as a new column, and output the updated dataset.

Estimators

An Estimator abstracts the concept of a learning algorithm or any algorithm which fits or trains on data. →fit()

Example,

A learning algorithm such as LogisticRegression is an Estimator;

Calling fit() trains a LogisticRegressionModel, which is a Transformer;

Data Workflows: Spark MLlib

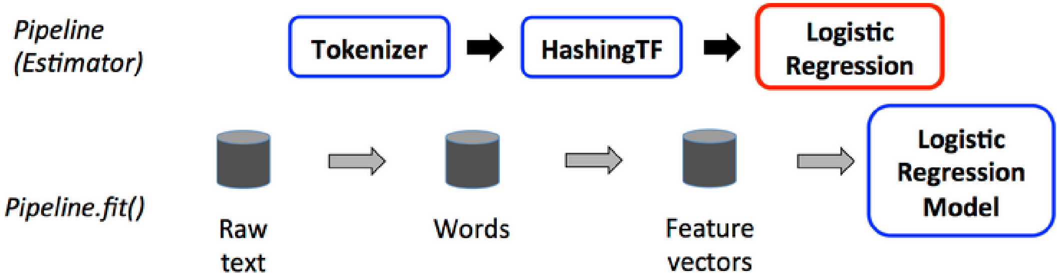
Pipeline

In machine learning, it is common to run a sequence of algorithms to process and learn from data. E.g., a simple text document processing workflow might include several stages:

- Split each document's text into words.
- Convert each document's words into a numerical feature vector.
- Learn a prediction model using the feature vectors and labels.

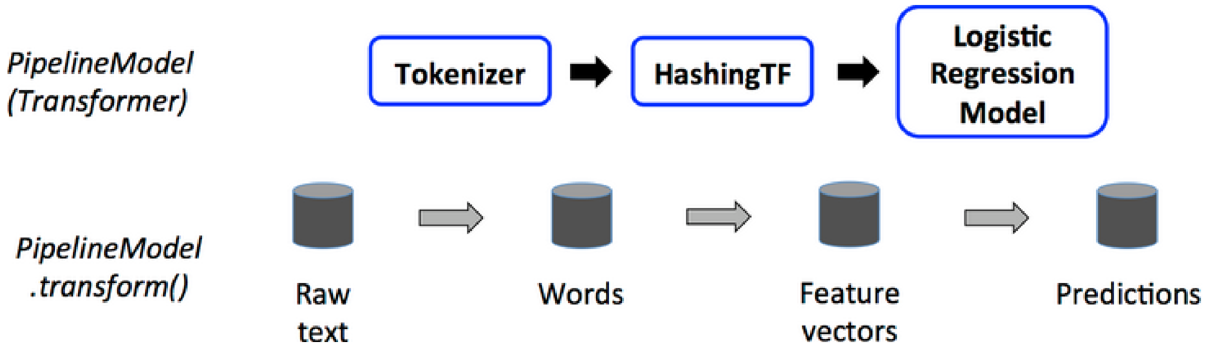
Data Workflows: Spark MLlib

```
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.feature.{HashingTF, Tokenizer}
import org.apache.spark.mllib.linalg.Vector
import org.apache.spark.sql.Row
// Prepare training documents from a list of (id, text, label) tuples.
val training = sqlContext.createDataFrame(Seq(
  (0L, "a b c d e spark", 1.0),
  (1L, "b d", 0.0),
  (2L, "spark f g h", 1.0),
  (3L, "hadoop mapreduce", 0.0)
)).toDF("id", "text", "label")
// Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
val tokenizer = new Tokenizer().setInputCol("text").setOutputCol("words")
val hashingTF = new
HashingTF().setNumFeatures(1000).setInputCol(tokenizer.getOutputCol).setOutputCol("features")
val lr = new LogisticRegression().setMaxIter(10).setRegParam(0.01)
val pipeline = new Pipeline().setStages(Array(tokenizer, hashingTF, lr))
// Fit the pipeline to training documents.
val model = pipeline.fit(training)
```



Data Workflows: Spark MLlib

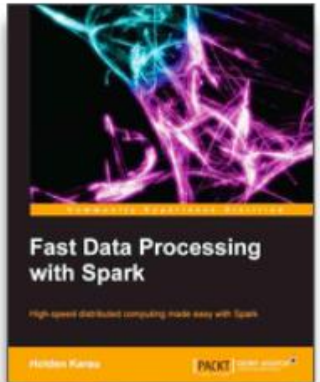
```
// Prepare test documents, which are unlabeled (id, text) tuples.
val test = sqlContext.createDataFrame(Seq(
  (4L, "spark i j k"),
  (5L, "l m n"),
  (6L, "mapreduce spark"),
  (7L, "apache hadoop")
)).toDF("id", "text")

// Make predictions on test documents.
model.transform(test)
.select("id", "text", "probability", "prediction")
.collect()
.foreach { case Row(id: Long, text: String, prob: Vector, prediction: Double) =>
  println(s"($id, $text) --> prob=$prob, prediction=$prediction")
 }
```



Suggested Books

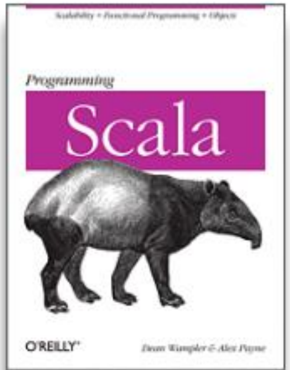
Q&A

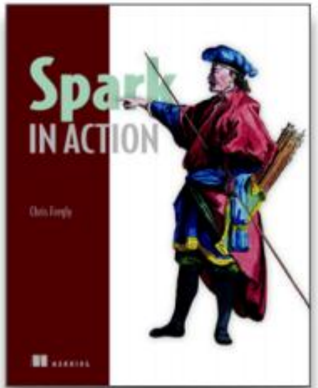
Suggested Books



Fast Data Processing with Spark
Holden Karau
Packt (2013)
shop.oreilly.com/product/9781782167068.do

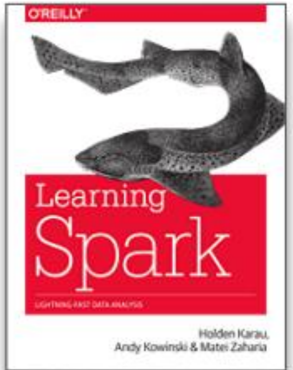
Programming Scala
Dean Wampler, Alex Payne
O'Reilly (2009)
shop.oreilly.com/product/9780596155964.do





Spark in Action
Chris Fregly
Manning (2015*)
sparkinaction.com/

Learning Spark
Holden Karau, Andy Kowinski, Matei Zaharia
O'Reilly (2015*)
shop.oreilly.com/product/0636920028512.do



Next Meetup

- Lab6: Spark Graphx(30m)
- Spark in production: build(20m)
- Spark in production: deploy(30m)
- Spark in production: monitor(20m)
- Optimizing Transformations & Actions(60m)
- Caching and Serialization(20m)
- Advanced Data Sources(30m)
- Case Studies(30m)