Intro to Apache Spark

http://databricks.com/

download slides:
http://cdn.liber118.com/workshop/itas_workshop.pdf

DATABRICKS

Spark

00: Getting Started

Introduction

installs + intros, while people arrive: 20 min

Intro: *Online Course Materials*

Best to download the slides to your laptop:

cdn.liber118.com/workshop/itas_workshop.pdf

Be sure to complete the course survey:
http://goo.gl/QpBSnR

In addition to these slides, all of the code samples are available on GitHub gists:

- **gist.github.com/ceteri/f2c3486062c9610eac1d**
- **gist.github.com/ceteri/8ae5b9509a08c08a1132**
- **gist.github.com/ceteri/11381941**

Intro: *Success Criteria*

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

- open a Spark Shell
- use of some ML algorithms
- explore data sets loaded from HDFS, etc.
- review Spark SQL, Spark Streaming, Shark
- review advanced topics and BDAS projects
- follow-up courses and certification
- developer community resources, events, etc.
- return to workplace and demo use of Spark!

Intro: *Preliminaries*

- intros – what is your background?
- who needs to use AWS instead of laptops?
- PEM key, if needed? See tutorial:
  Connect to Your Amazon EC2 Instance from Windows Using PuTTY

01: Getting Started

Installation

hands-on lab: 20 min

Installation:

Let’s get started using Apache Spark,
in just four easy steps…

spark.apache.org/docs/latest/

(for class, please copy from the USB sticks)

Step 1: *Install Java JDK 6/7 on MacOSX or Windows*

oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html

- follow the license agreement instructions
- then click the download for your OS
- need JDK instead of JRE (for Maven, etc.)

(for class, please copy from the USB sticks)

Step 1: *Install Java JDK 6/7 on Linux*

this is much simpler on Linux…

```
sudo apt-get -y install openjdk-7-jdk
```

Step 2: *Download Spark*

we'll be using Spark 1.0.0

see **spark.apache.org/downloads.html**

1. download this URL with a browser
2. double click the archive file to open it
3. connect into the newly created directory

(for class, please copy from the USB sticks)

Step 3: *Run Spark Shell*

we'll run Spark's interactive shell…

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

then from the "scala>" REPL prompt,
let's create some data…

```
val data = 1 to 10000
```

Step 4: *Create an RDD*

create an **RDD** based on that data…

```
val distData = sc.parallelize(data)
```

then use a filter to select values less than 10…

```
distData.filter(_ < 10).collect()
```

Step 4: *Create an RDD*

create an

```
val distData = sc.parallelize(data)
```

then use a filter to select values less than 10 …

```
d
```

Checkpoint:
what do you get for results?

gist.github.com/ceteri/f2c3486062c9610eac1d#file-01-repl-txt

Installation: *Optional Downloads: Python*

For Python 2.7, check out *Anaconda* by Continuum Analytics for a full-featured platform:

store.continuum.io/cshop/anaconda/

Anaconda

Installation: *Optional Downloads: Maven*

Java builds later also require Maven, which you can download at:

maven.apache.org/download.cgi

maven

03: Getting Started

Spark Deconstructed

lecture: 20 min

Spark Deconstructed:

Let’s spend a few minutes on this Scala thing…

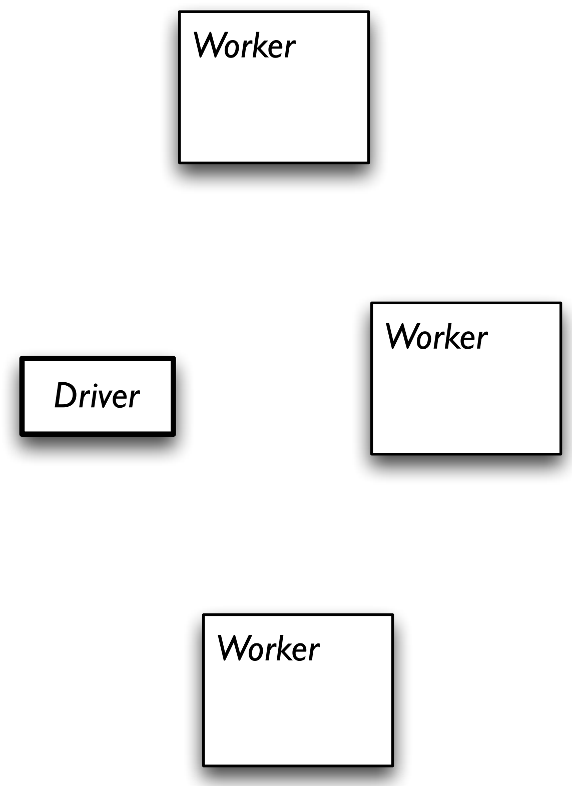
scala-lang.org/

Spark Deconstructed: *Log Mining Example*

```scala
// load error messages from a log into memory
// then interactively search for various patterns
// https://gist.github.com/ceteri/8ae5b9509a08c08a1132

// base RDD
val lines = sc.textFile("hdfs://...")

// 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()
```

Spark Deconstructed: *Log Mining Example*

We start with Spark running on a cluster…
submitting code to be evaluated on it:



Spark Deconstructed: *Log Mining Example*

```
// base RDD
val lines = sc.textFile("hdfs://...")

// 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()
```

discussing the other part

Spark Deconstructed: *Log Mining Example*

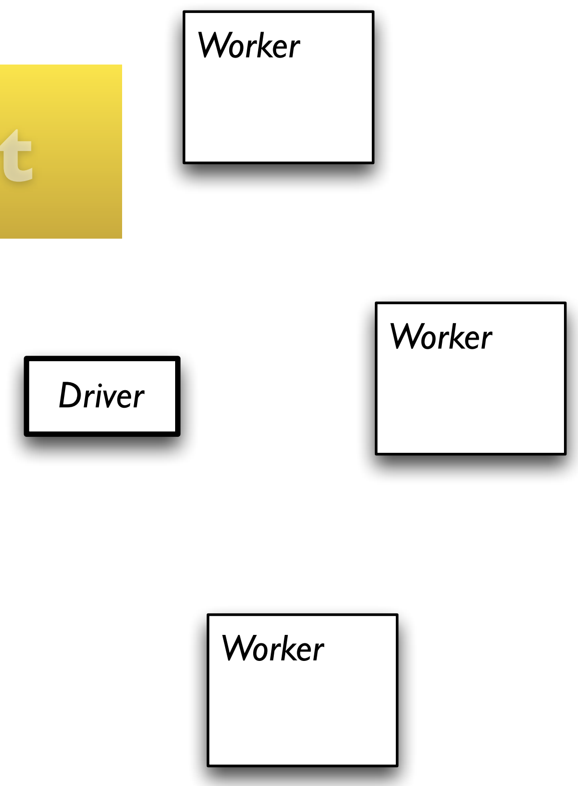
At this point, take a look at the transformed RDD *operator graph*:

```
scala> messages.toDebugString
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)
```

Spark Deconstructed: *Log Mining Example*

```
// base RDD
val lines = sc.textFile("hdfs://...")

// 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()
```

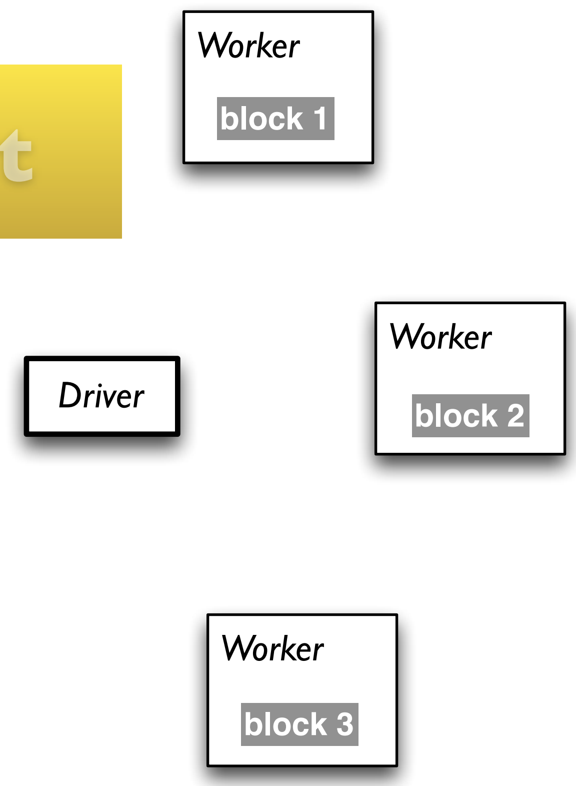
discussing the other part



Spark Deconstructed: *Log Mining Example*

```
// base RDD
val lines = sc.textFile("hdfs://...")

// 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()
```

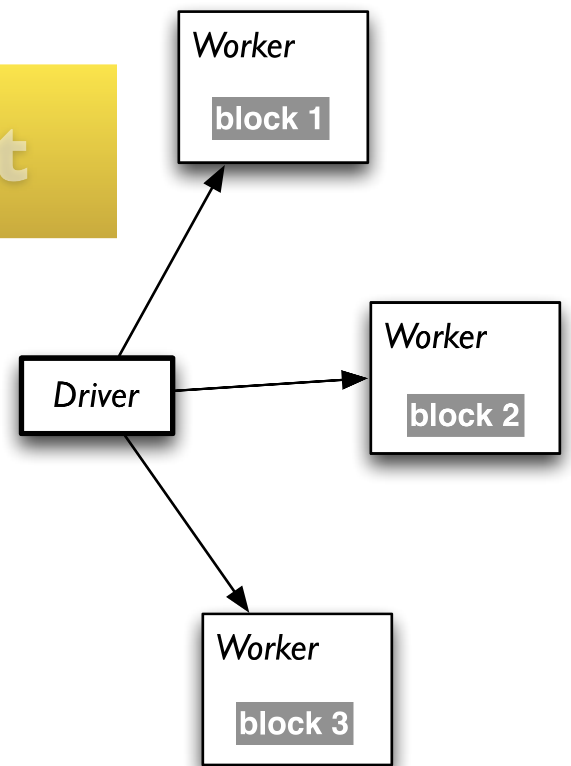
discussing the other part



Spark Deconstructed: *Log Mining Example*

```
// base RDD
val lines = sc.textFile("hdfs://...")

// 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()
```

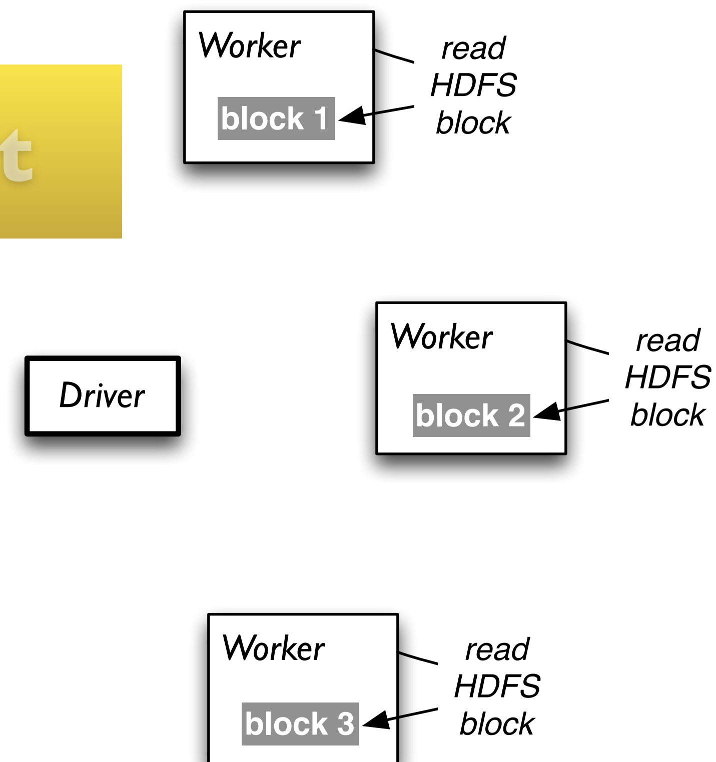
discussing the other part



Spark Deconstructed: *Log Mining Example*

```
// base RDD
val lines = sc.textFile("hdfs://...")

// 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()
```

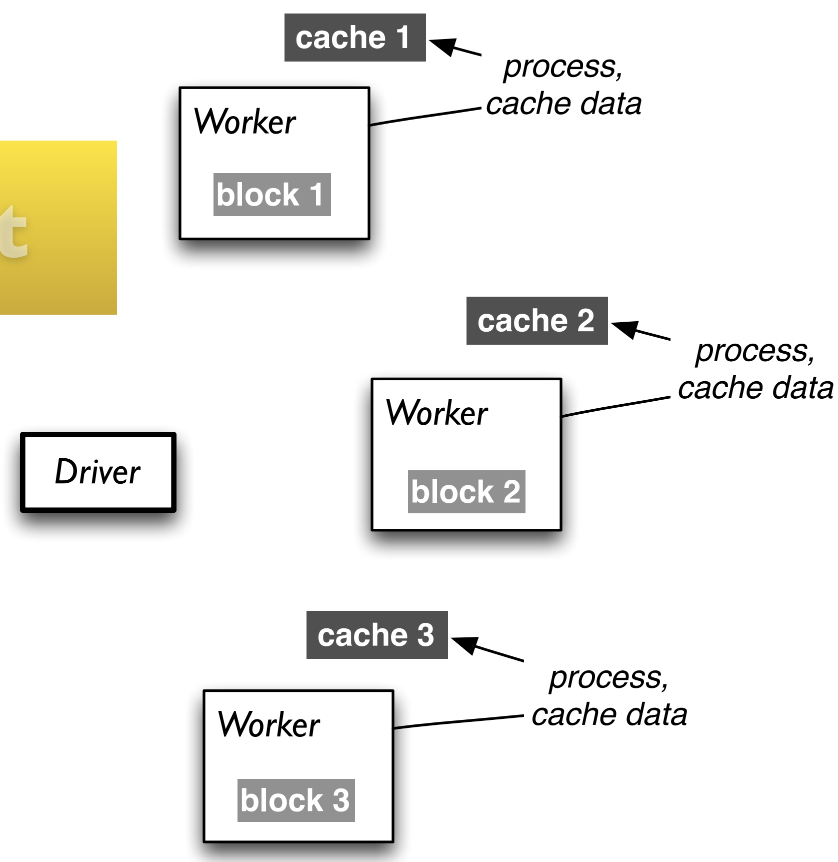
discussing the other part



Spark Deconstructed: *Log Mining Example*

```
// base RDD
val lines = sc.textFile("hdfs://...")

// 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()
```

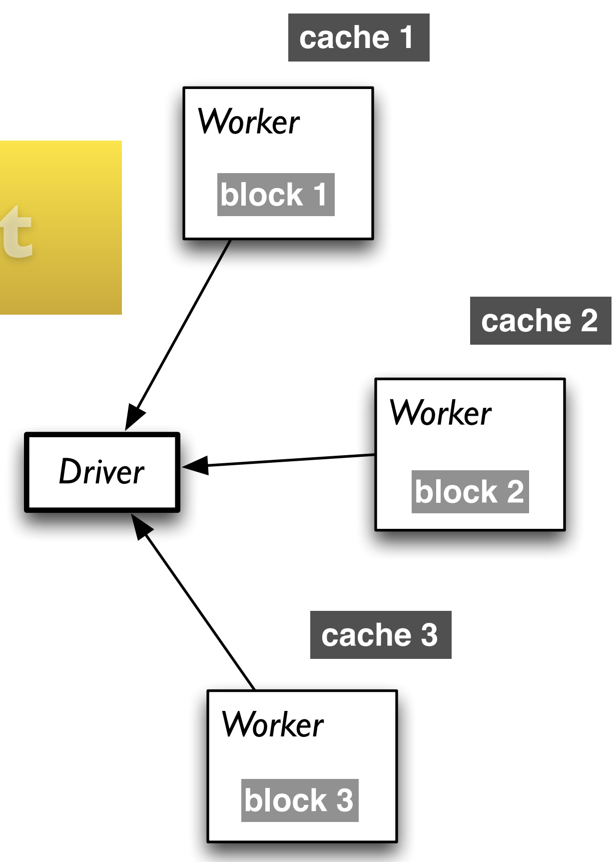
discussing the other part



Spark Deconstructed: *Log Mining Example*

```
// base RDD
val lines = sc.textFile("hdfs://...")

// 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()
```

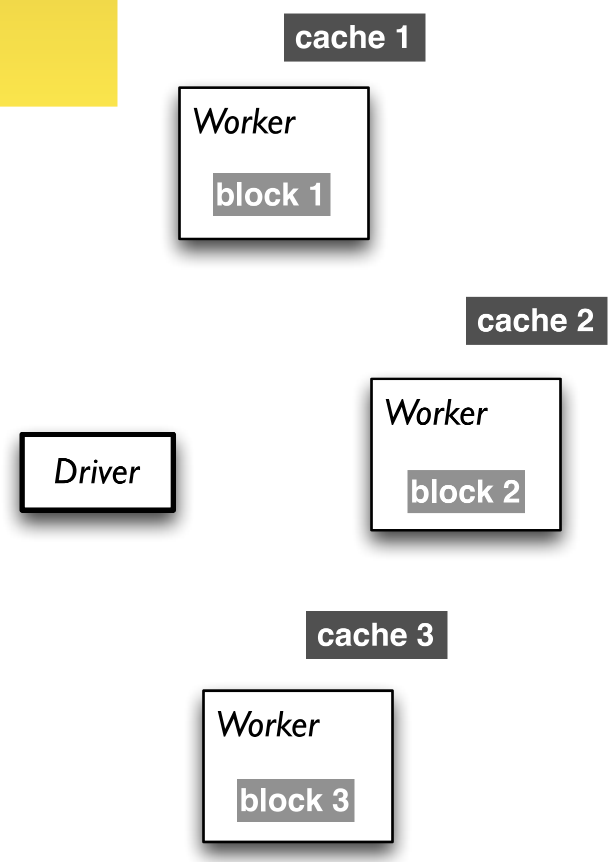
discussing the other part



Spark Deconstructed: *Log Mining Example*

discussing the other part

```
// base RDD
val lines = sc.textFile("hdfs://...")

// 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()
```

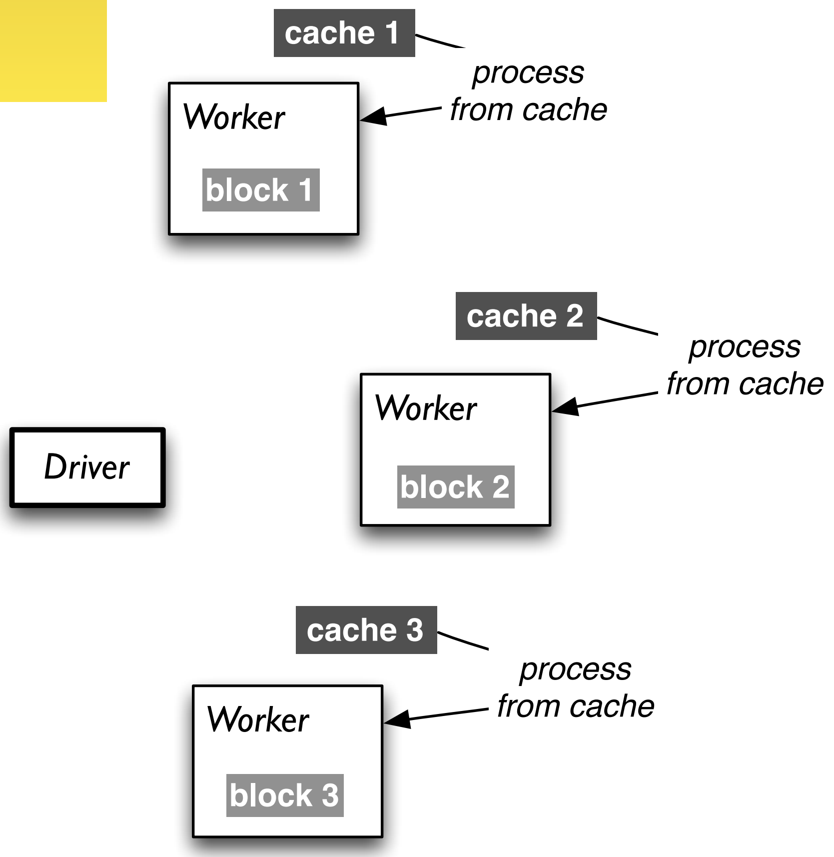


Spark Deconstructed: *Log Mining Example*

```
// base RDD
val lines = sc.textFile("hdfs://...")

// 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()
```

discussing the other part

```
// action 2
messages.filter(_.contains("php")).count()
```

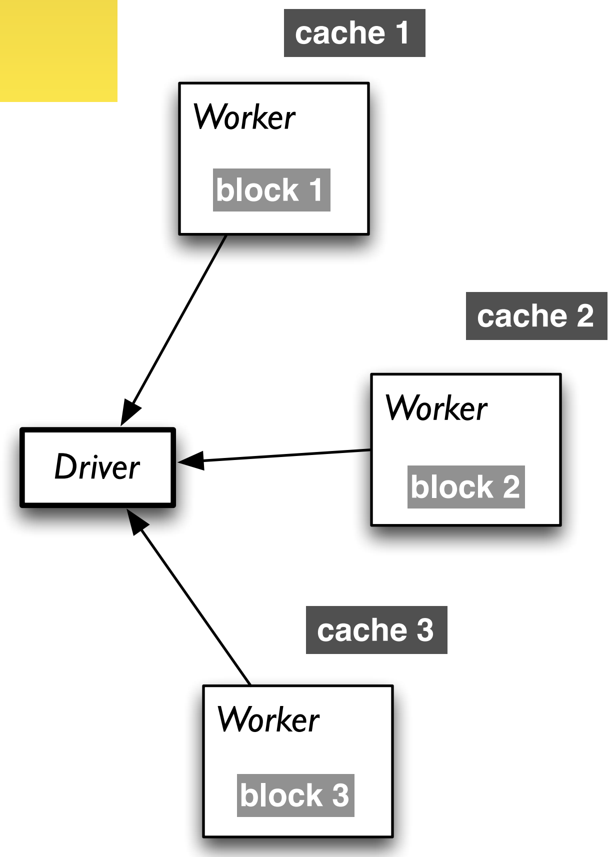


Spark Deconstructed: *Log Mining Example*

```
// base RDD
val lines = sc.textFile("hdfs://...")

// 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()
```

discussing the other part

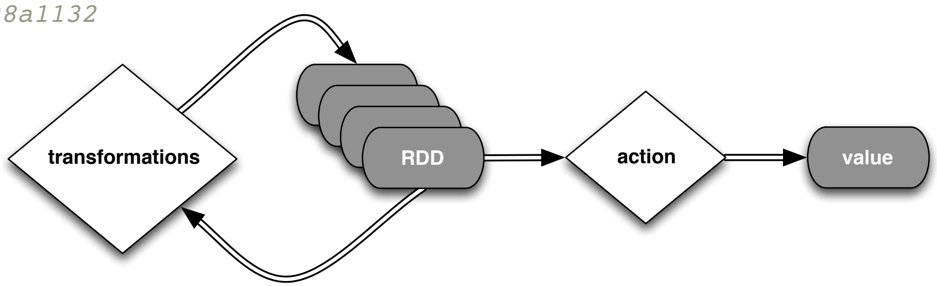
```
// action 2
messages.filter(_.contains("php")).count()
```



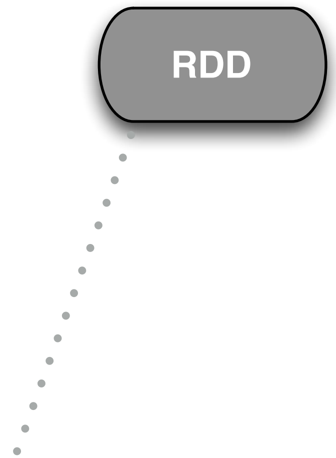
Spark Deconstructed:

Looking at the RDD transformations and actions from another perspective…

```
// load error messages from a log into memory
// then interactively search for various patterns
// https://gist.github.com/ceteri/8ae5b9509a08c08a1132

// base RDD
val lines = sc.textFile("hdfs://...")

// 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()
```

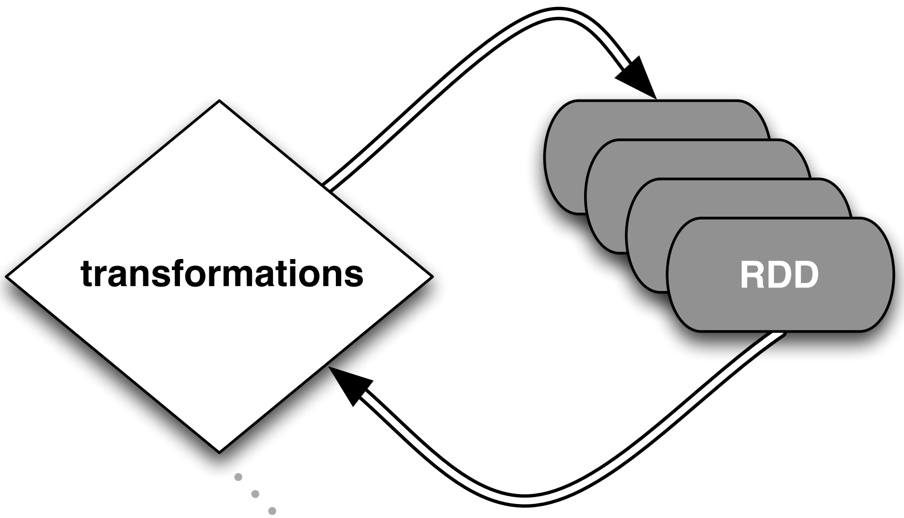


Spark Deconstructed:



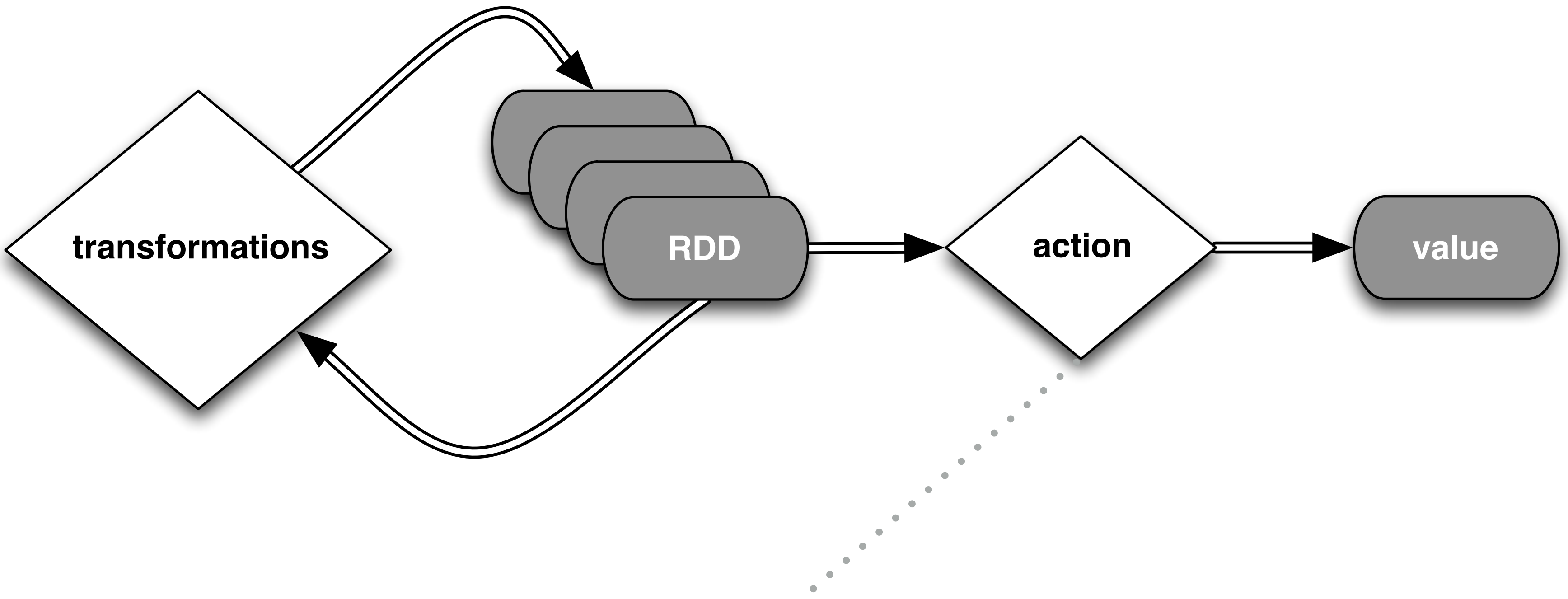

```
// base RDD
val lines = sc.textFile("hdfs://...")
```

Spark Deconstructed:




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

Spark Deconstructed:




```
// action 1
messages.filter(_.contains("mysql")).count()
```

04: Getting Started

Simple Spark Apps

lab: 20 min

Simple Spark Apps: *WordCount*

Definition:

count how often each word appears in a collection of text documents

This simple program provides a good test case for parallel processing, since it:

- requires a minimal amount of code
- demonstrates use of both symbolic and numeric values
- isn't many steps away from search indexing
- serves as a "Hello World" for Big Data apps

A distributed computing framework that can run WordCount **efficiently in parallel at scale** can likely handle much larger and more interesting compute problems

```
void map (String doc_id, String text):
  for each word w in segment(text):
    emit(w, "1");


void reduce (String word, Iterator group):
  int count = 0;

  for each pc in group:
    count += Int(pc);

  emit(word, String(count));
```

Simple Spark Apps: *WordCount*

Scala:

```scala
val f = sc.textFile("README.md")
val wc = f.flatMap(l => l.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)
wc.saveAsTextFile("wc_out.txt")
```

Python:

```python
from operator import add
f = sc.textFile("README.md")
wc = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1)).reduceByKey(add)
wc.saveAsTextFile("wc_out.txt")
```

Simple Spark Apps: *WordCount*

Scala:

```
val f = sc.textFile(
val wc
wc.saveAsTextFile(
```

Checkpoint:
how many "Spark" keywords?

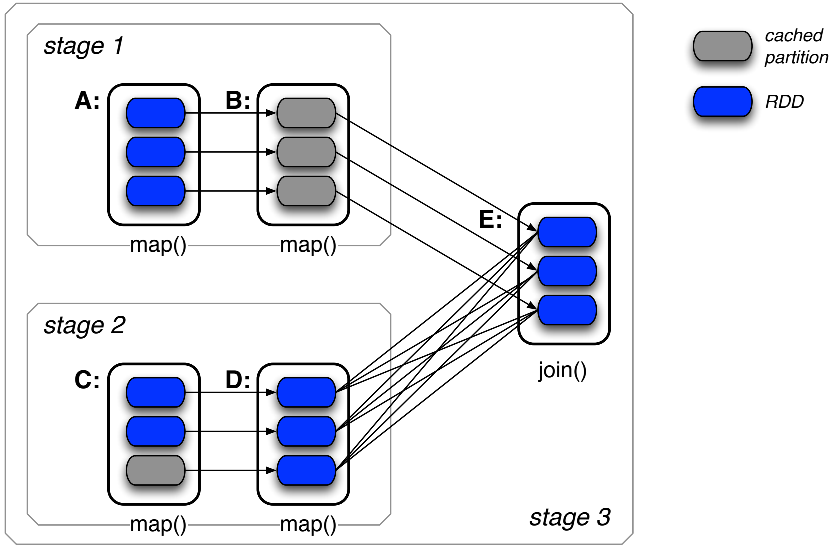
Pythor

```
from operator
f = sc
wc = f
wc.saveAsTextFile(
```

Simple Spark Apps: *Code + Data*

The code + data for the following example of a join is available in:

gist.github.com/ceteri/11381941

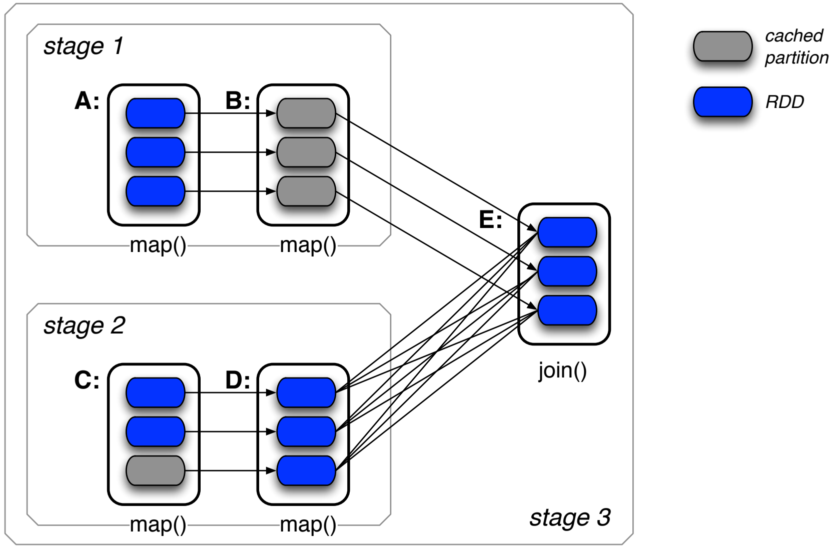


Simple Spark Apps: *Source Code*

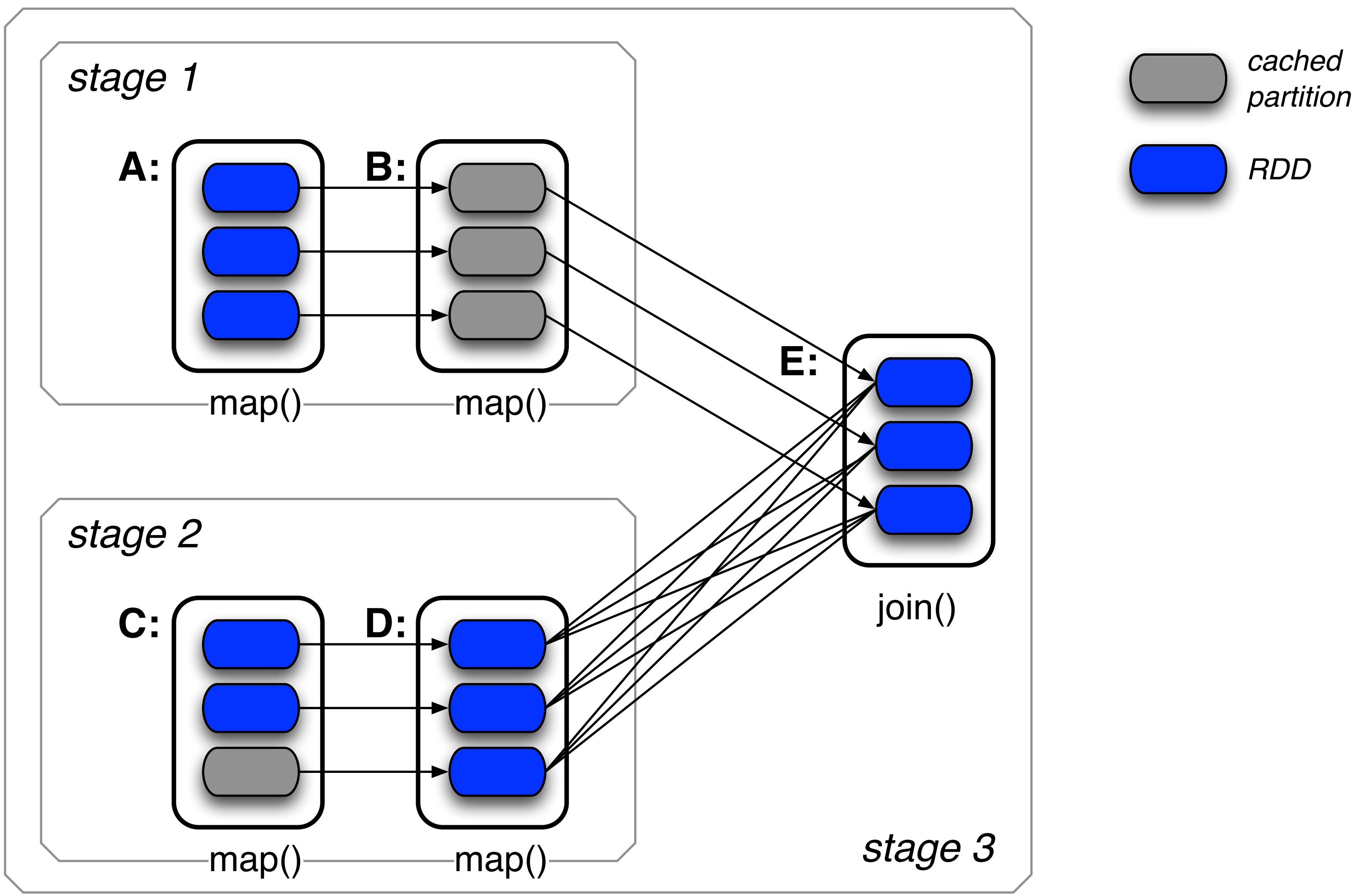
```
val format = new java.text.SimpleDateFormat("yyyy-MM-dd")

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("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("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)
```

Simple Spark Apps: *Operator Graph*

```
scala> reg.join(clk).toDebugString
res5: String =
FlatMappedValuesRDD[46] at join at <console>:23 (1 partitions)
  MappedValuesRDD[45] at join at <console>:23 (1 partitions)
    CoGroupedRDD[44] at join at <console>:23 (1 partitions)
      MappedRDD[36] at map at <console>:16 (1 partitions)
        MappedRDD[35] at map at <console>:16 (1 partitions)
          MappedRDD[34] at textFile at <console>:16 (1 partitions)
            HadoopRDD[33] at textFile at <console>:16 (1 partitions)
      MappedRDD[40] at map at <console>:16 (1 partitions)
        MappedRDD[39] at map at <console>:16 (1 partitions)
          MappedRDD[38] at textFile at <console>:16 (1 partitions)
            HadoopRDD[37] at textFile at <console>:16 (1 partitions)
```



Simple Spark Apps: *Operator Graph*



Simple Spark Apps: *Assignment*

Using the `README.md` and `CHANGES.txt` files in the Spark directory:

1. create RDDs to filter each line for the keyword "Spark"
2. perform a WordCount on each, i.e., so the results are (K,V) pairs of (word, count)
3. join the two RDDs

Simple Spark Apps: *Assignment*

Using the
the Spark directory:

1. create RDDs to filter each file for the keyword "Sp

2. per
res

Checkpoint:
how many "Spark" keywords?

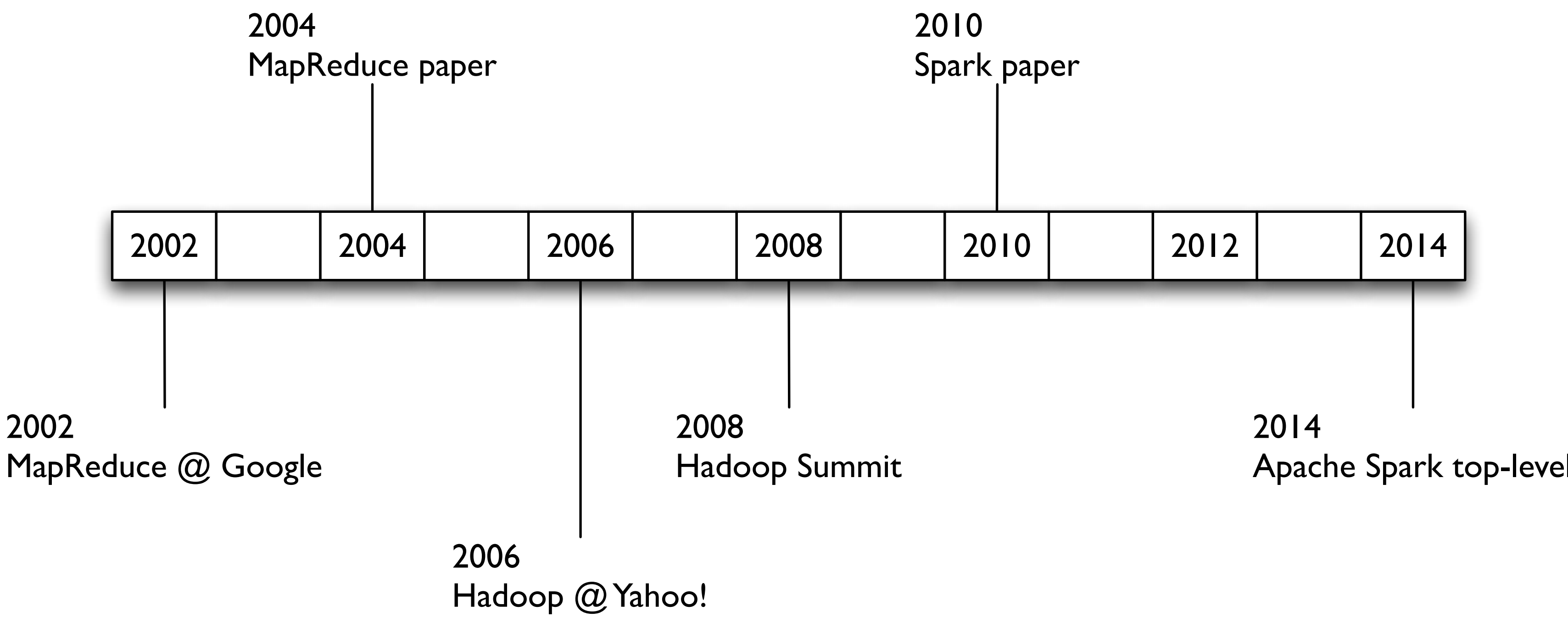
3. join the two RDDs

05: Getting Started

A Brief History

lecture: 35 min

A Brief History:



A Brief History: *MapReduce*

circa 1979 – **Stanford, MIT, CMU**, etc.
set/list operations in LISP, Prolog, etc., for parallel processing
www-formal.stanford.edu/jmc/history/lisp/lisp.htm

circa 2004 – **Google**
MapReduce: Simplified Data Processing on Large Clusters
Jeffrey Dean and Sanjay Ghemawat
research.google.com/archive/mapreduce.html

circa 2006 – **Apache**
Hadoop, originating from the Nutch Project
Doug Cutting
research.yahoo.com/files/cutting.pdf

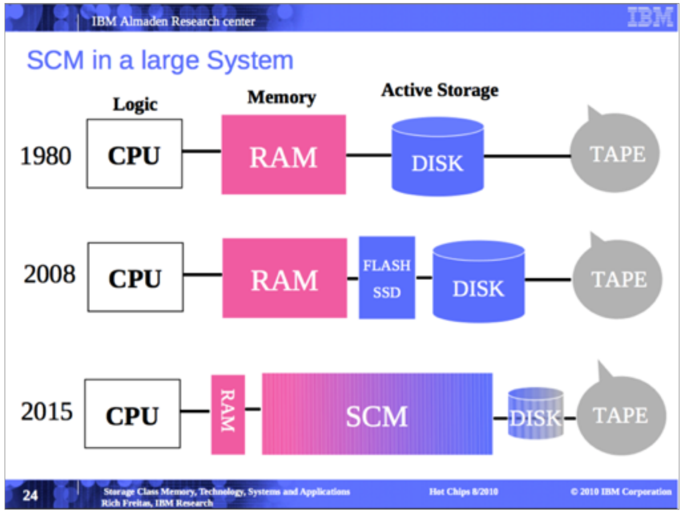
circa 2008 – **Yahoo**
web scale search indexing
Hadoop Summit, HUG, etc.
developer.yahoo.com/hadoop/

circa 2009 – **Amazon AWS**
Elastic MapReduce
Hadoop modified for EC2/S3, plus support for Hive, Pig, Cascading, etc.
aws.amazon.com/elasticmapreduce/

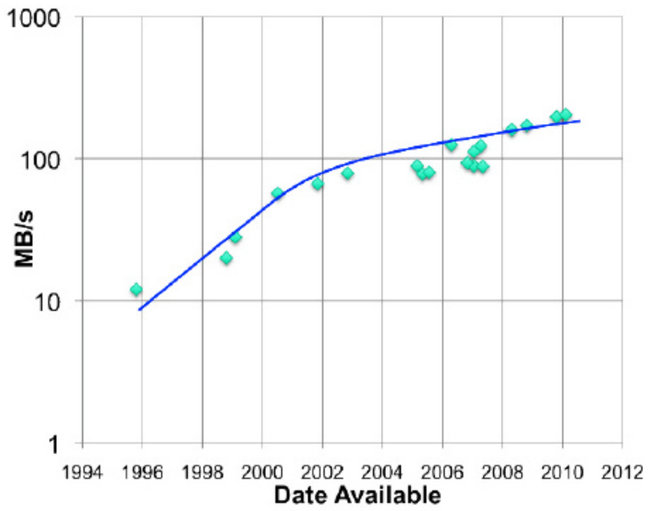
Open Discussion:

Enumerate several changes in data center technologies since 2002…

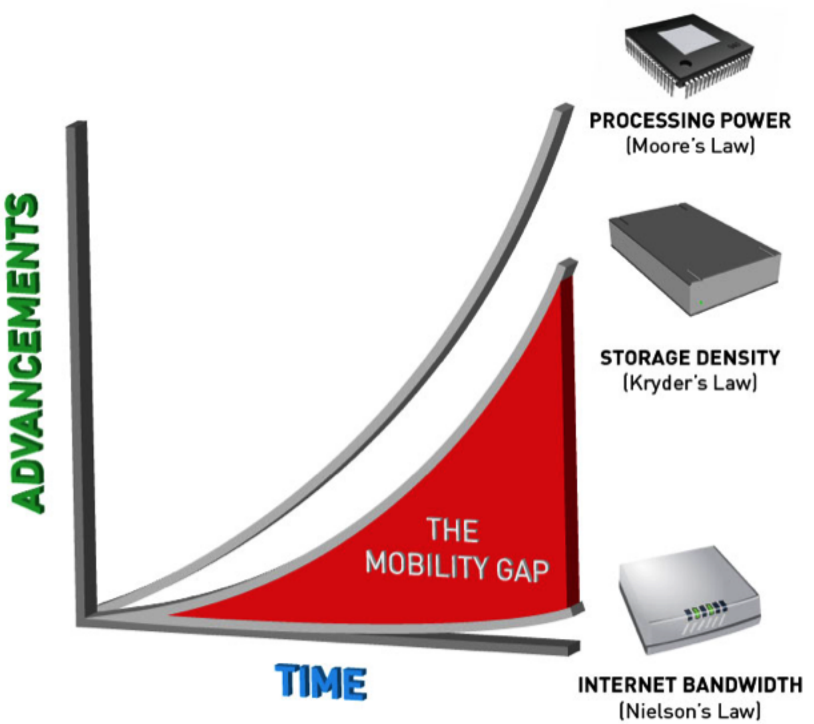
A Brief History: *MapReduce*



Rich Freitas, **IBM Research**



storagenewsletter.com/rubriques/hard-disk-drives/hdd-technology-trends-ibm/



pistoncloud.com/2013/04/storage-and-the-mobility-gap/

meanwhile, spinny disks haven't changed all that much…

A Brief History: *MapReduce*

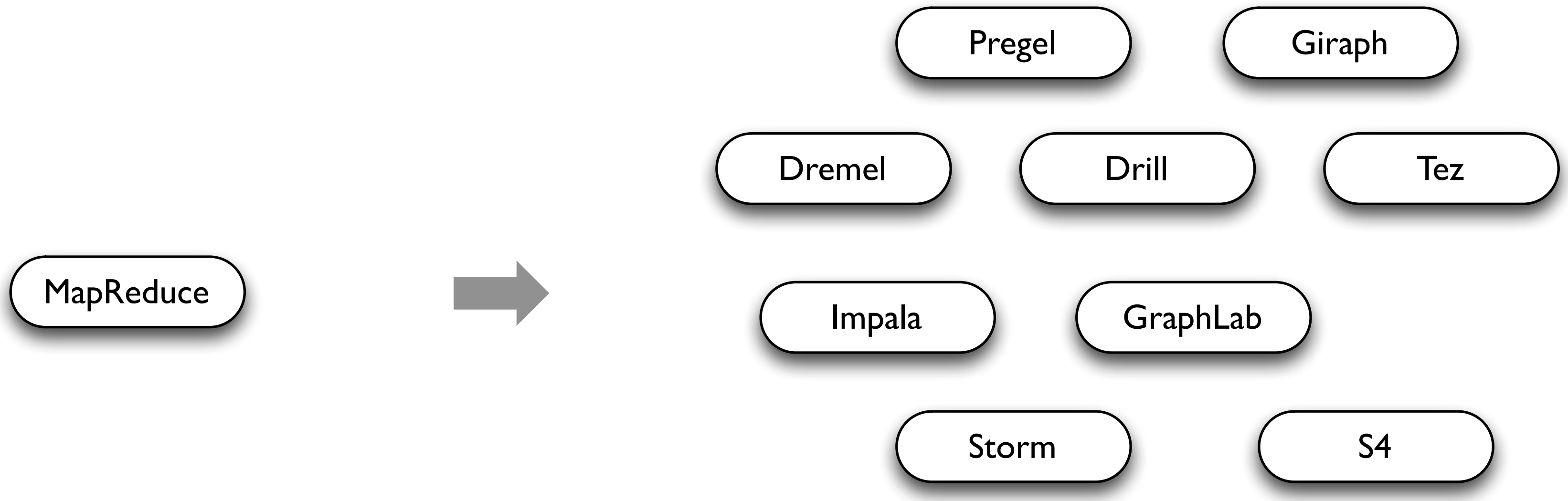
MapReduce use cases showed two major limitations:

1. difficultly of programming directly in MR
2. performance bottlenecks, or batch not fitting the use cases

In short, MR doesn’t compose well for large applications

Therefore, people built *specialized systems* as workarounds…

A Brief History: *MapReduce*

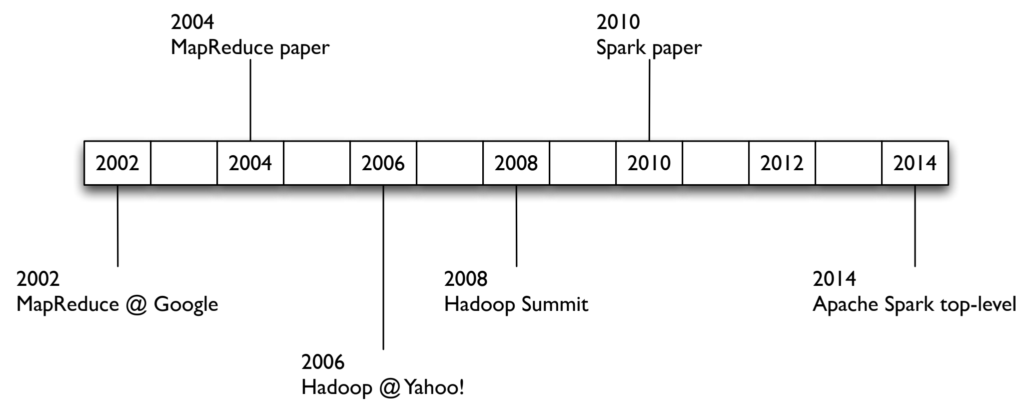


General Batch Processing

Specialized Systems:
iterative, interactive, streaming, graph, etc.

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

A Brief History: *Spark*



Spark: Cluster Computing with Working Sets
Matei Zaharia, Mosharaf Chowdhury,
Michael J. Franklin, Scott Shenker, Ion Stoica
USENIX HotCloud (2010)
people.csail.mit.edu/matei/papers/2010/hotcloud_spark.pdf

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for
In-Memory Cluster Computing
Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave,
Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
NSDI (2012)
usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

A Brief History: *Spark*

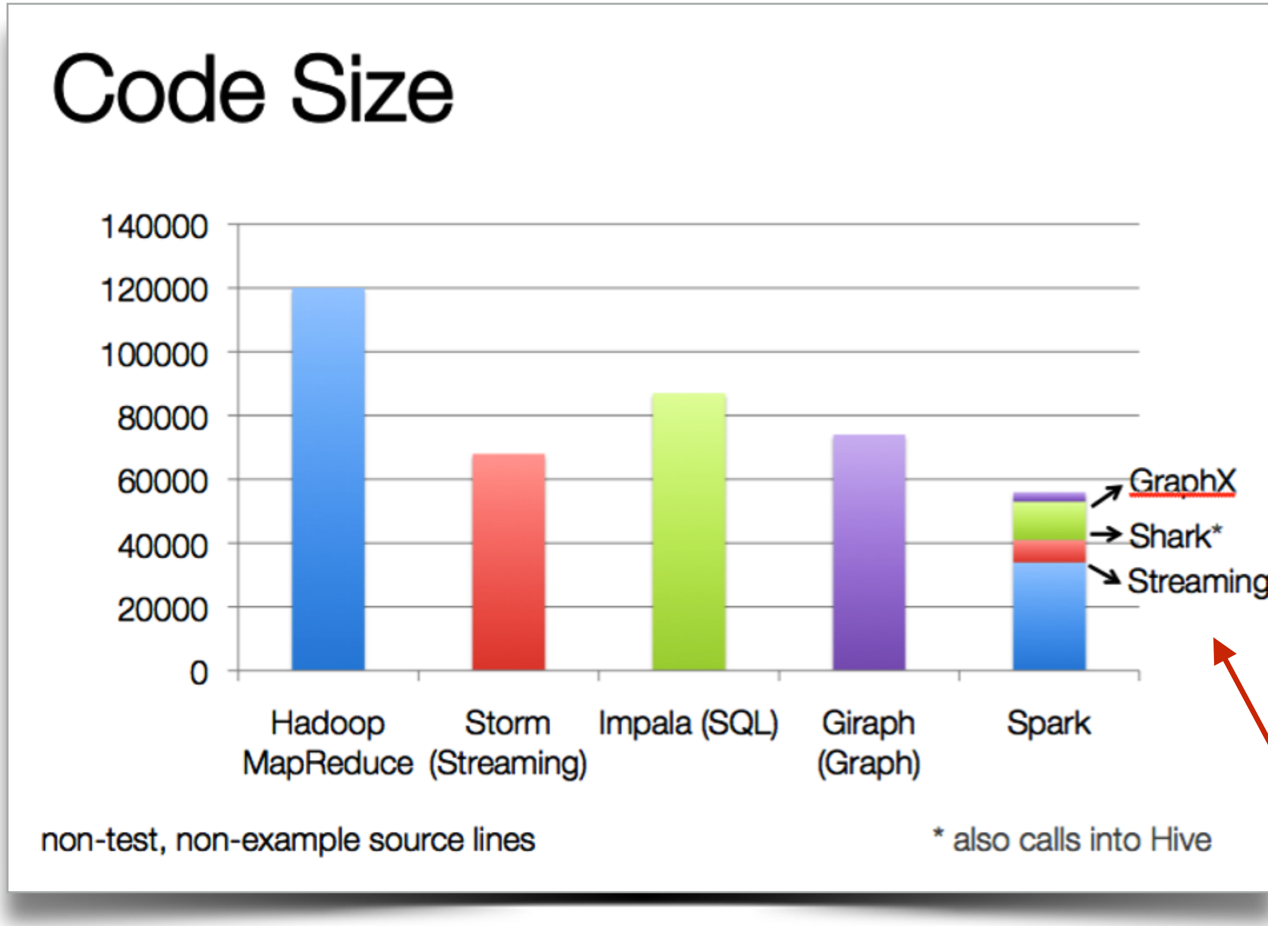
Unlike the various specialized systems, Spark's goal was to *generalize* MapReduce to support new apps within same engine

Two reasonably small additions are enough to express the previous models:

- *fast data sharing*
- *general DAGs*

This allows for an approach which is more efficient for the engine, and much simpler for the end users

A Brief History: *Spark*



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

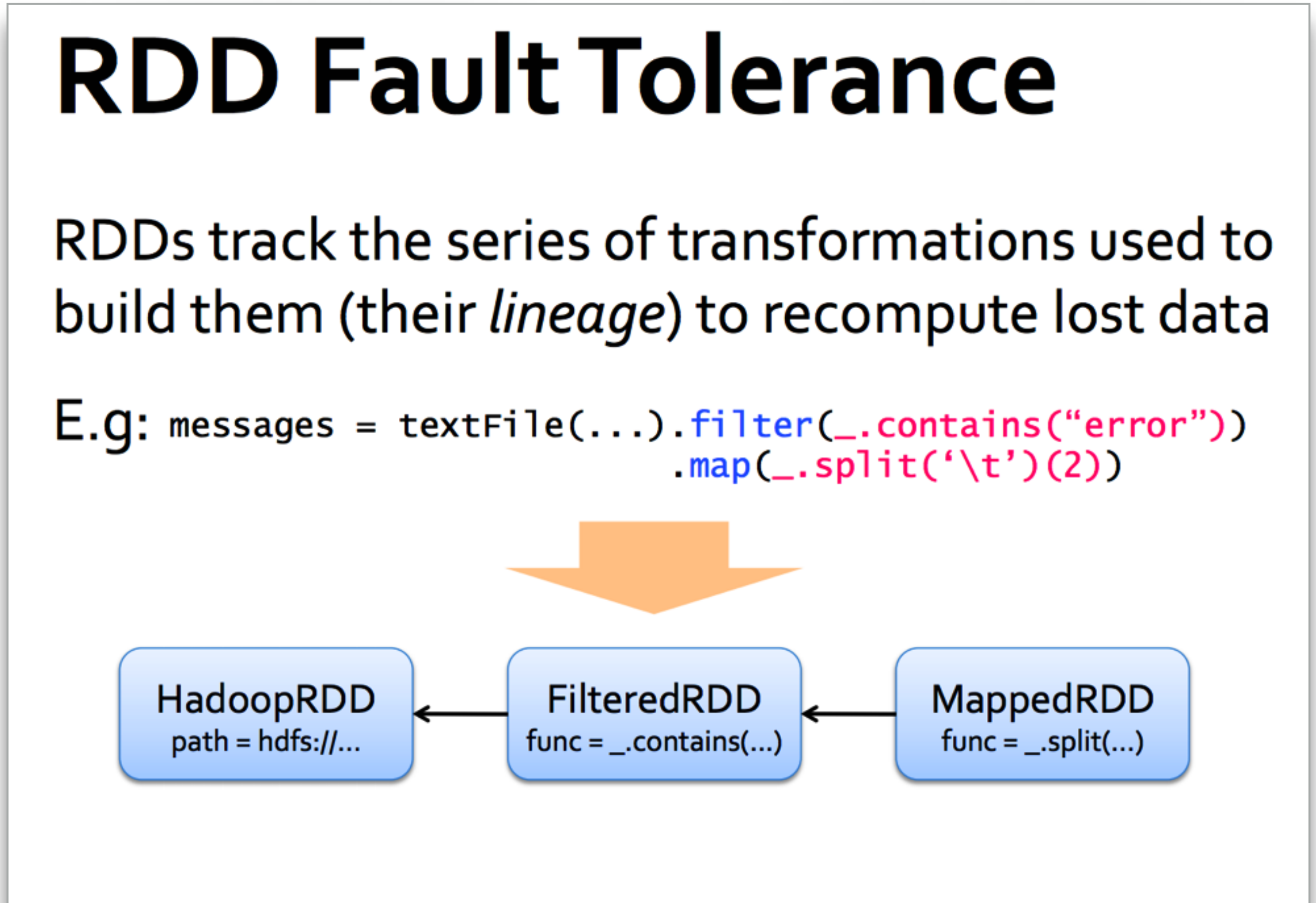
used as libs, instead of specialized systems

A Brief History: *Spark*

Some key points about Spark:

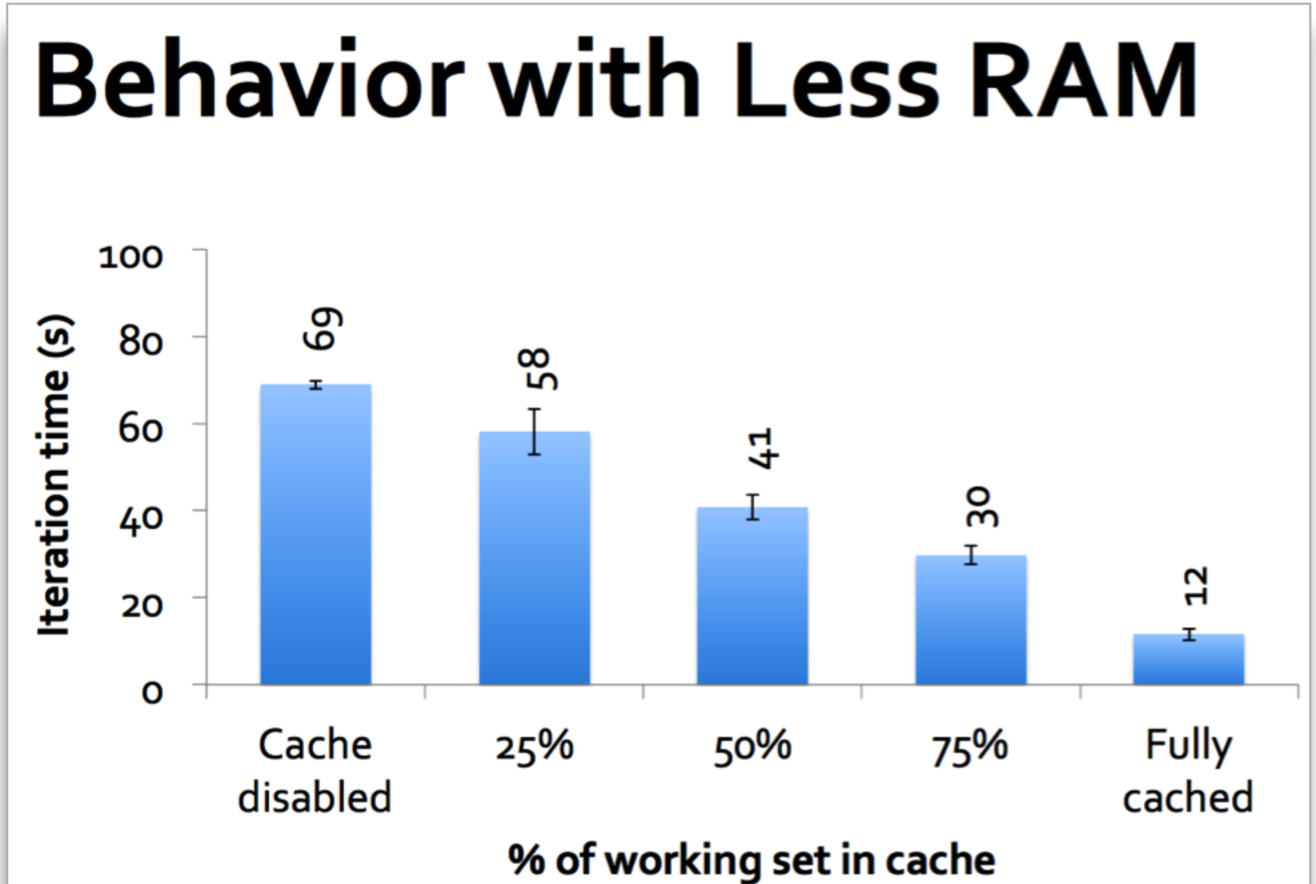
- handles batch, interactive, and real−time within a single framework
- native integration with Java, Python, Scala
- programming at a higher level of abstraction
- more general: map/reduce is just one set of supported constructs

A Brief History: *Spark*



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

A Brief History: *Spark*



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

(break)

break: 15 min

03: Intro Spark Apps

Spark Essentials

lecture/lab: 45 min

Spark Essentials:

Intro apps, showing examples in both Scala and Python…

Let's start with the basic concepts in:

spark.apache.org/docs/latest/scala-programming-guide.html

using, respectively:

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

```
./bin/pyspark
```

alternatively, with IPython Notebook:

```
IPYTHON_OPTS="notebook --pylab inline" ./bin/pyspark
```

Spark Essentials: *SparkContext*

First thing that a Spark program does is create a `SparkContext` object, which tells Spark how to access a cluster

In the shell for either Scala or Python, this is the `sc` variable, which is created automatically

Other programs must use a constructor to instantiate a new `SparkContext`

Then in turn `SparkContext` gets used to create other variables

Spark Essentials: *SparkContext*

Scala:

```
scala> sc
res: spark.SparkContext = spark.SparkContext@470d1f30
```

Python:

```
>>> sc
<pyspark.context.SparkContext object at 0x7f7570783350>
```

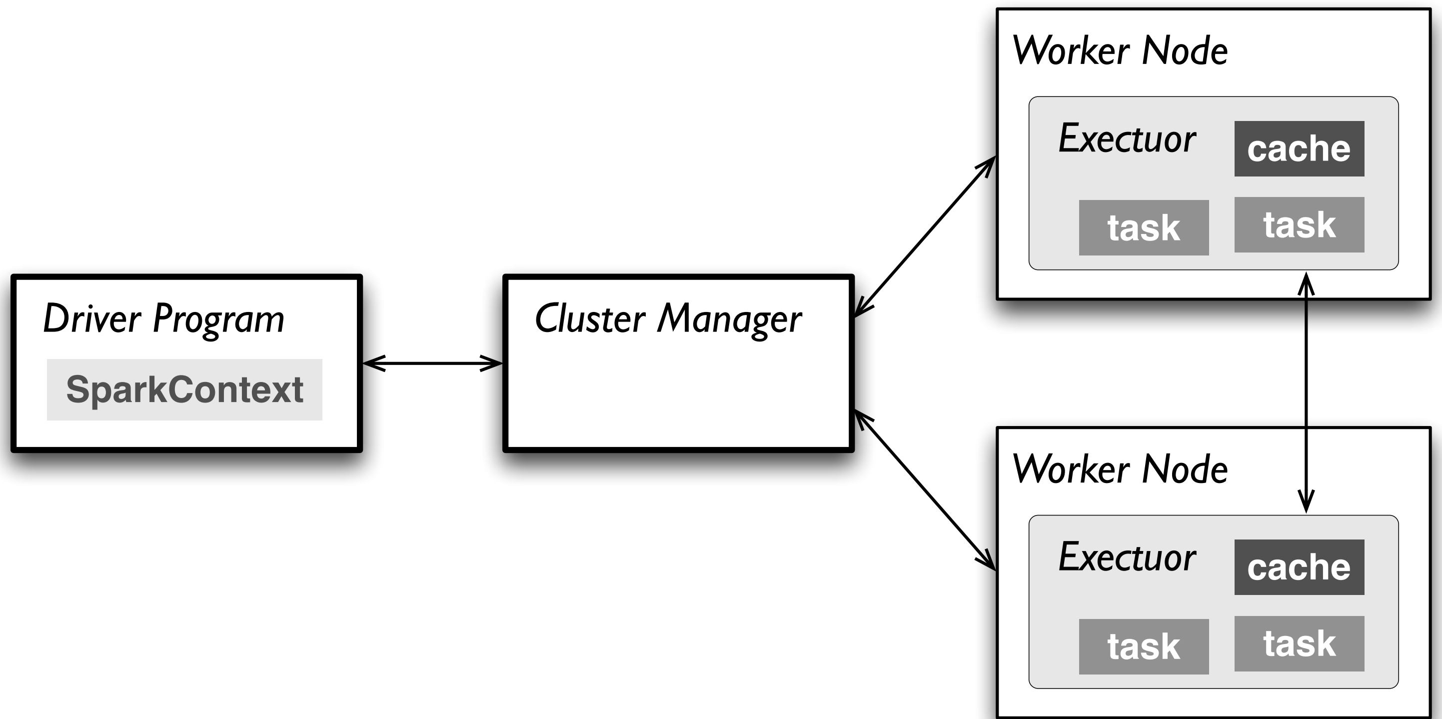
Spark Essentials: *Master*

The `master` parameter for a `SparkContext` determines which cluster to use

master	*description*
`local`	run Spark locally with one worker thread (no parallelism)
`local[K]`	run Spark locally with K worker threads (ideally set to # cores)
`spark://HOST:PORT`	connect to a Spark standalone cluster; PORT depends on config (7077 by default)
`mesos://HOST:PORT`	connect to a Mesos cluster; PORT depends on config (5050 by default)

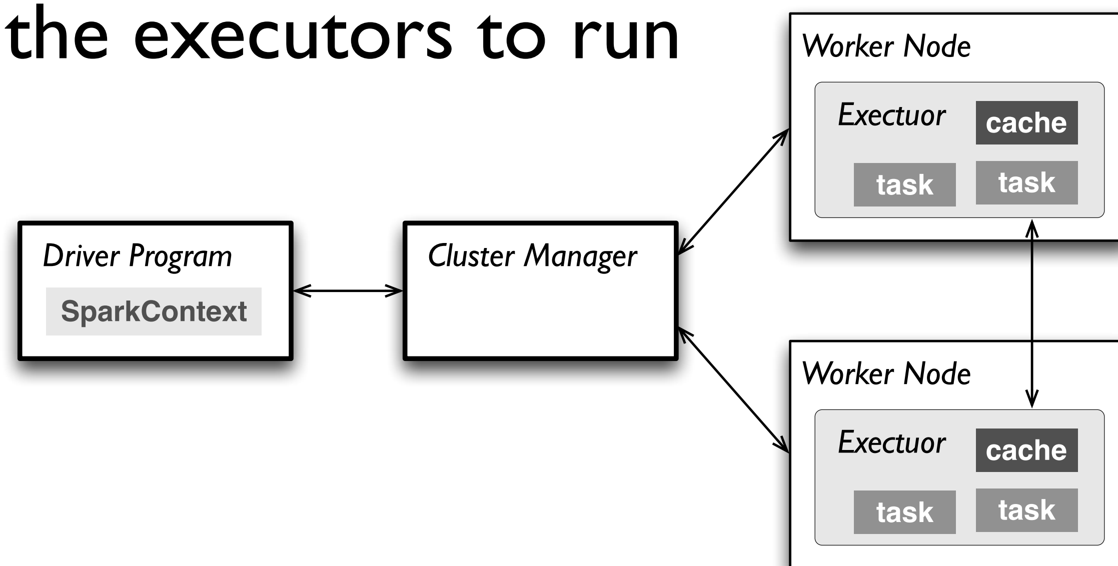
Spark Essentials: *Master*

spark.apache.org/docs/latest/cluster-overview.html



Spark Essentials: *Master*

1. connects to a *cluster manager* which allocate resources across applications
2. acquires *executors* on cluster nodes – worker processes to run computations and store data
3. sends *app code* to the executors
4. sends *tasks* for the executors to run



Spark Essentials: *RDD*

Resilient **D**istributed **D**atasets (RDD) are the primary abstraction in Spark – a fault-tolerant collection of elements that can be operated on in parallel

There are currently two types:

- *parallelized collections* – take an existing Scala collection and run functions on it in parallel
- *Hadoop datasets* – run functions on each record of a file in Hadoop distributed file system or any other storage system supported by Hadoop

Spark Essentials: *RDD*

- two types of operations on RDDs: *transformations* and *actions*
- transformations are lazy (not computed immediately)
- the transformed RDD gets recomputed when an action is run on it (default)
- however, an RDD can be *persisted* into storage in memory or disk

Spark Essentials: *RDD*

Scala:

```
scala> val data = Array(1, 2, 3, 4, 5)
data: Array[Int] = Array(1, 2, 3, 4, 5)

scala> val distData = sc.parallelize(data)
distData: spark.RDD[Int] = spark.ParallelCollection@10d13e3e
```

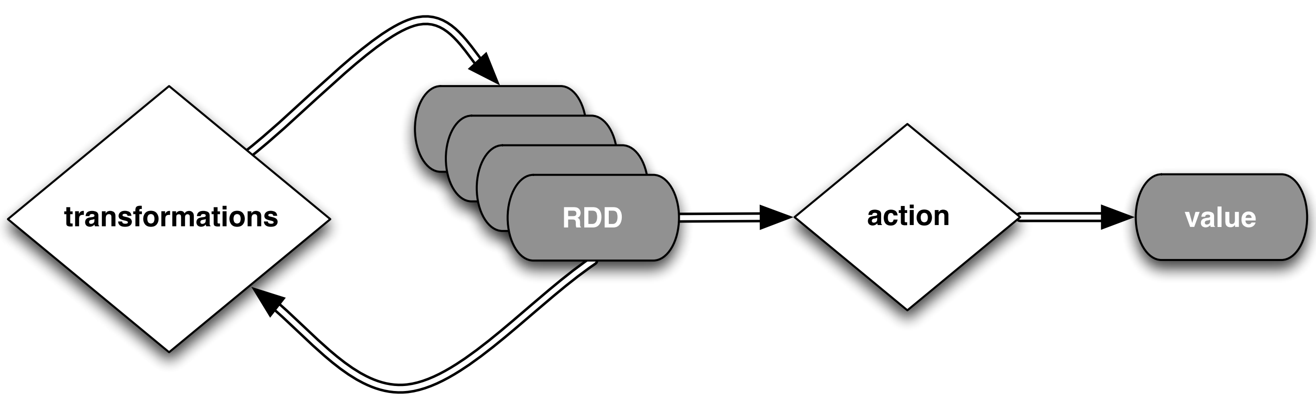
Python:

```
>>> data = [1, 2, 3, 4, 5]
>>> data
[1, 2, 3, 4, 5]

>>> distData = sc.parallelize(data)
>>> distData
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:229
```

Spark Essentials: *RDD*

Spark can create RDDs from any file stored in HDFS or other storage systems supported by Hadoop, e.g., local file system, Amazon S3, Hypertable, HBase, etc.

Spark supports text files, SequenceFiles, and any other Hadoop `InputFormat`, and can also take a directory or a glob (e.g. `/data/201404*`)



Spark Essentials: *RDD*

Scala:

```
scala> val distFile = sc.textFile("README.md")
distFile: spark.RDD[String] = spark.HadoopRDD@1d4cee08
```

Python:

```
>>> distFile = sc.textFile("README.md")
14/04/19 23:42:40 INFO storage.MemoryStore: ensureFreeSpace(36827) called
with curMem=0, maxMem=318111744
14/04/19 23:42:40 INFO storage.MemoryStore: Block broadcast_0 stored as
values to memory (estimated size 36.0 KB, free 303.3 MB)
>>> distFile
MappedRDD[2] at textFile at NativeMethodAccessorImpl.java:-2
```

Spark Essentials: *Transformations*

Transformations create a new dataset from an existing one

All transformations in Spark are *lazy*: they do not compute their results right away – instead they remember the transformations applied to some base dataset

- optimize the required calculations
- recover from lost data partitions

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: *Transformations*

Scala:

```
val distFile = sc.textFile("README.md")
distFile.map(l => l.split(" ")).collect()
distFile.flatMap(l => l.split(" ")).collect()
```

distFile is a collection of lines

Python:

```
distFile = sc.textFile("README.md")
distFile.map(lambda x: x.split(' ')).collect()
distFile.flatMap(lambda x: x.split(' ')).collect()
```

Spark Essentials: *Transformations*

Scala:

```
val distFile = sc.textFile("README.md")
distFile.map(l => l.split(" ")).collect()
distFile.flatMap(l => l.split(" ")).collect()
```

closures

Python:

```
distFile = sc.textFile("README.md")
distFile.map(lambda x: x.split(' ')).collect()
distFile.flatMap(lambda x: x.split(' ')).collect()
```

Spark Essentials: *Transformations*

Scala:

```
val distFile = sc.textFile("README.md")
distFile.map(l => l.split(" ")).collect()
distFile.flatMap(l => l.split(" ")).collect()
```

closures

Python:

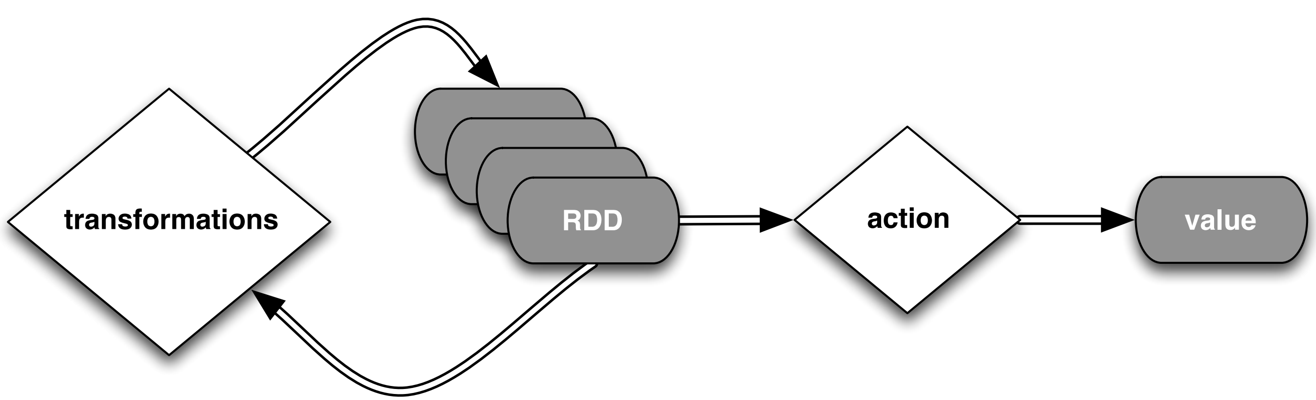
```
distFile = sc.textFile("README.md")
distFile.map(lambda x: x.split(' ')).collect()
distFile.flatMap(lambda x: x.split(' ')).collect()
```

looking at the output, how would you compare results for map() vs. flatMap() ?

Spark Essentials: *Transformations*

Using closures is now possible in Java 8 with *lambda expressions* support, see the tutorial:

databricks.com/blog/2014/04/14/Spark-with-Java-8.html



Spark Essentials: *Transformations*

Java 7:

```
JavaRDD<String> distFile = sc.textFile("README.md");

// Map each line to multiple words
JavaRDD<String> words = distFile.flatMap(
  new FlatMapFunction<String, String>() {
    public Iterable<String> call(String line) {
      return Arrays.asList(line.split(" "));
    }
});
```

Java 8:

```
JavaRDD<String> distFile = sc.textFile("README.md");
JavaRDD<String> words =
    distFile.flatMap(line -> Arrays.asList(line.split(" ")));
```

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: *Actions*

Scala:

```
val f = sc.textFile("README.md")
val words = f.flatMap(l => l.split(" ")).map(word => (word, 1))
words.reduceByKey(_ + _).collect.foreach(println)
```

Python:

```
from operator import add
f = sc.textFile("README.md")
words = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1))
words.reduceByKey(add).collect()
```

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*

Scala:

```scala
val f = sc.textFile("README.md")
val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
w.reduceByKey(_ + _).collect.foreach(println)
```

Python:

```python
from operator import add
f = sc.textFile("README.md")
w = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1)).cache()
w.reduceByKey(add).collect()
```

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:

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

Python:

```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:

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

accum.value
```

Python:

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

rdd.foreach(f)

accum.value
```

Spark Essentials: *Accumulators*

Scala:

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

accum.value
```

driver-side

Python:

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

rdd.foreach(f)

accum.value
```

Spark Essentials: *(K,V) pairs*

Scala:

```
val pair = (a, b)

    pair._1 // => a
    pair._2 // => b
```

Python:

```
pair = (a, b)

    pair[0] # => a
    pair[1] # => b
```

Java:

```
Tuple2 pair = new Tuple2(a, b);

    pair._1 // => a
    pair._2 // => b
```

Spark Essentials: *API Details*

For more details about the Scala/Java API:

spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.package

For more details about the Python API:

spark.apache.org/docs/latest/api/python/

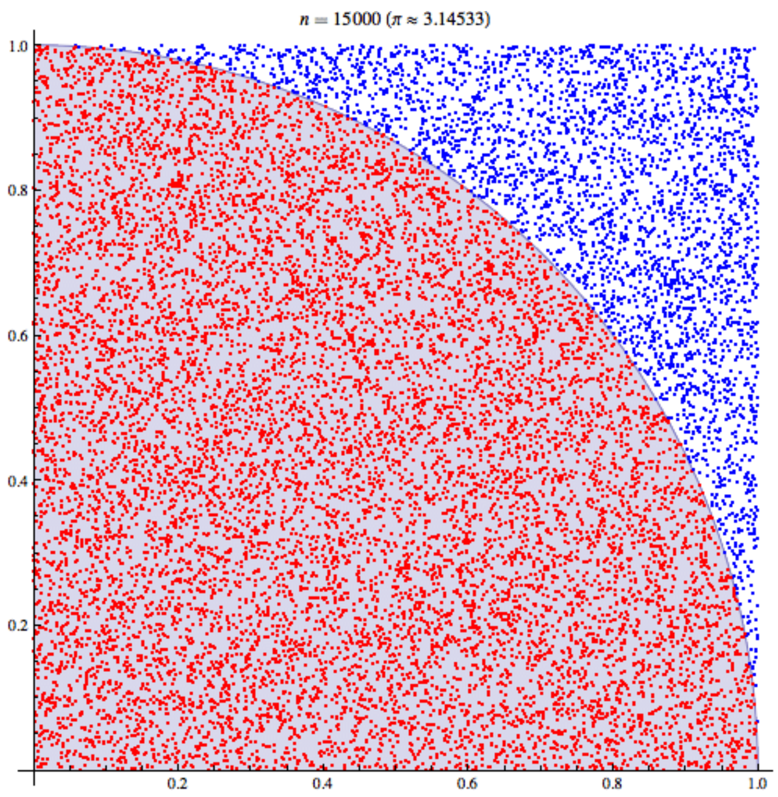
Spark Examples

lecture/lab: 10 min

Spark Examples: *Estimate Pi*

Next, try using a **Monte Carlo method** to estimate the value of Pi

```
./bin/run-example SparkPi 2 local
```



wikipedia.org/wiki/Monte_Carlo_method

Spark Examples: *Estimate Pi*

```scala
import scala.math.random
import org.apache.spark._

/** Computes an approximation to pi */
object SparkPi {
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("Spark Pi")
    val spark = new SparkContext(conf)

    val slices = if (args.length > 0) args(0).toInt else 2
    val n = 100000 * slices

    val count = spark.parallelize(1 to n, slices).map { i =>
      val x = random * 2 - 1
      val y = random * 2 - 1
      if (x*x + y*y < 1) 1 else 0
    }.reduce(_ + _)

    println("Pi is roughly " + 4.0 * count / n)
    spark.stop()
  }
}
```

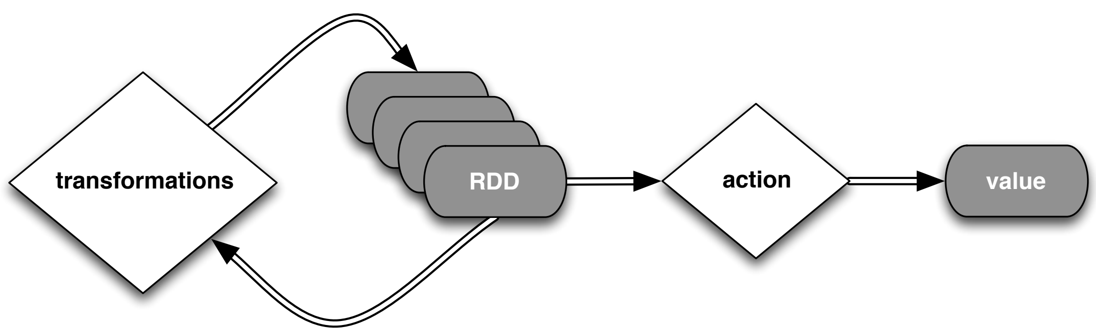
Spark Examples: *Estimate Pi*

```scala
val count = sc.parallelize(1 to n, slices)

  .map { i =>
    val x = random * 2 - 1
    val y = random * 2 - 1
    if (x*x + y*y < 1) 1 else 0
  }

  .reduce(_ + _)
```

base RDD

transformed RDD

action



Spark Examples: *Estimate Pi*

```
val count

 .map
  val
  val
  if
 }

 .reduc
```

base RDD

transformed RDD

Checkpoint:
what estimate do you get for Pi?