Cloud Computing and Big Data

Apache Spark and More

Oxford University
Software Engineering
Programme
July 2019

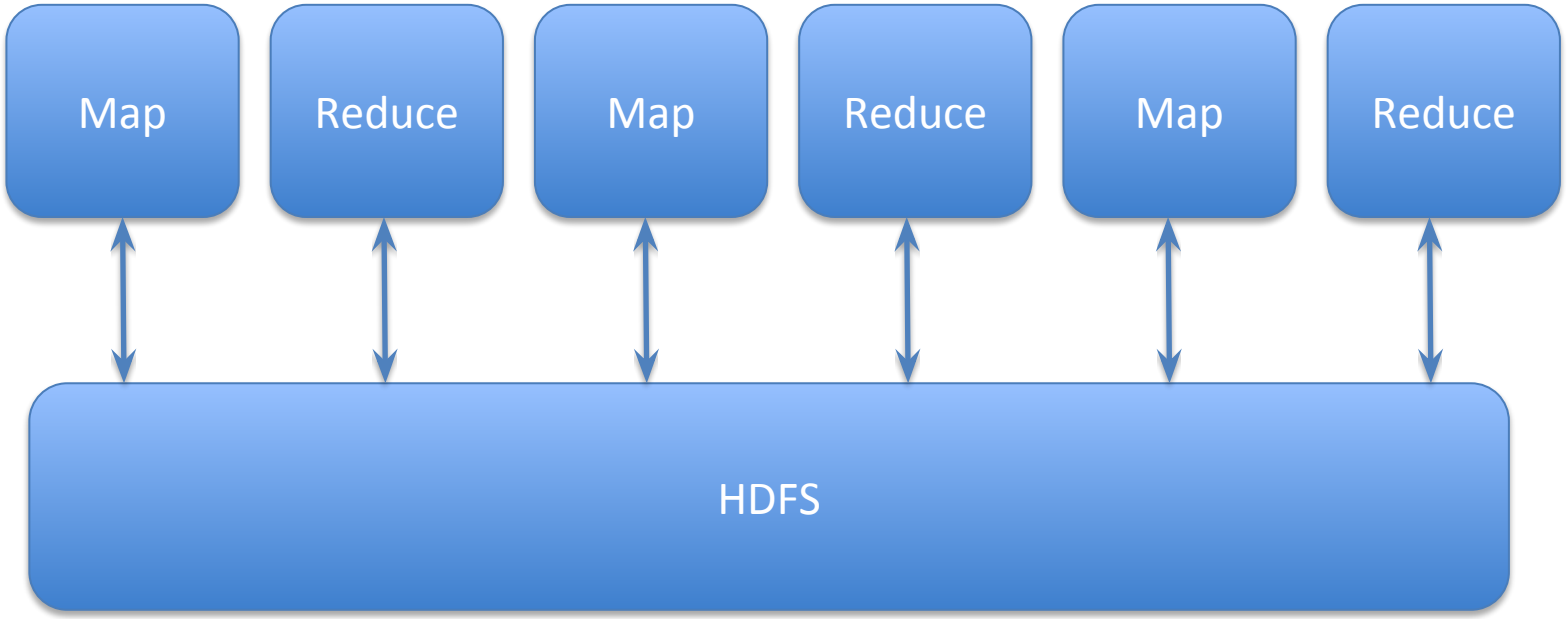
Contents

- What is wrong with Hadoop?
- Apache Spark
- PySpark / Python
- SparkSQL and Hive
- SparkR
- Spark and Yarn
- Spark and Mesos

Issues with Hadoop

- Hadoop is fundamentally all about Map Reduce
  - Though v2 did allow for other approaches
- Based on cheap commodity hardware
- But….
  - Not based on cheap commodity hardware with lots of memory!

Hadoop Model



Hadoop and Disk

- Hadoop does everything via replicated disk images
- Intermediate results are stored on disk
  - Slow for many operations
  - Including Machine Learning
  - No support for interactive processing

Improved Approach

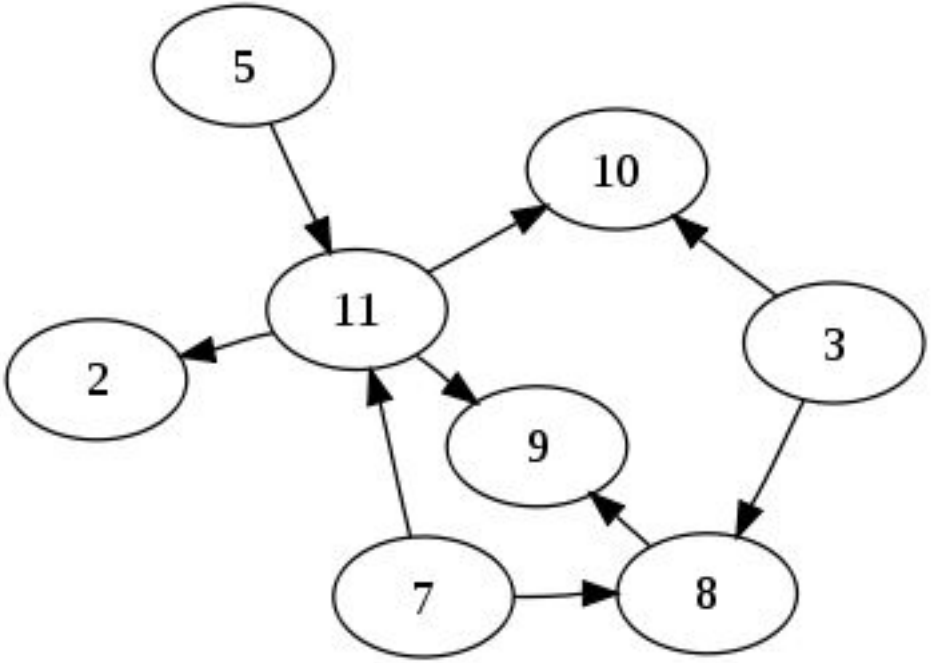
- A new model based on memory
  - Based on Directed Acyclic Graphs
  - And partitions

- What about reliability?

DAG

Directed Acyclic Graph

No Loops!



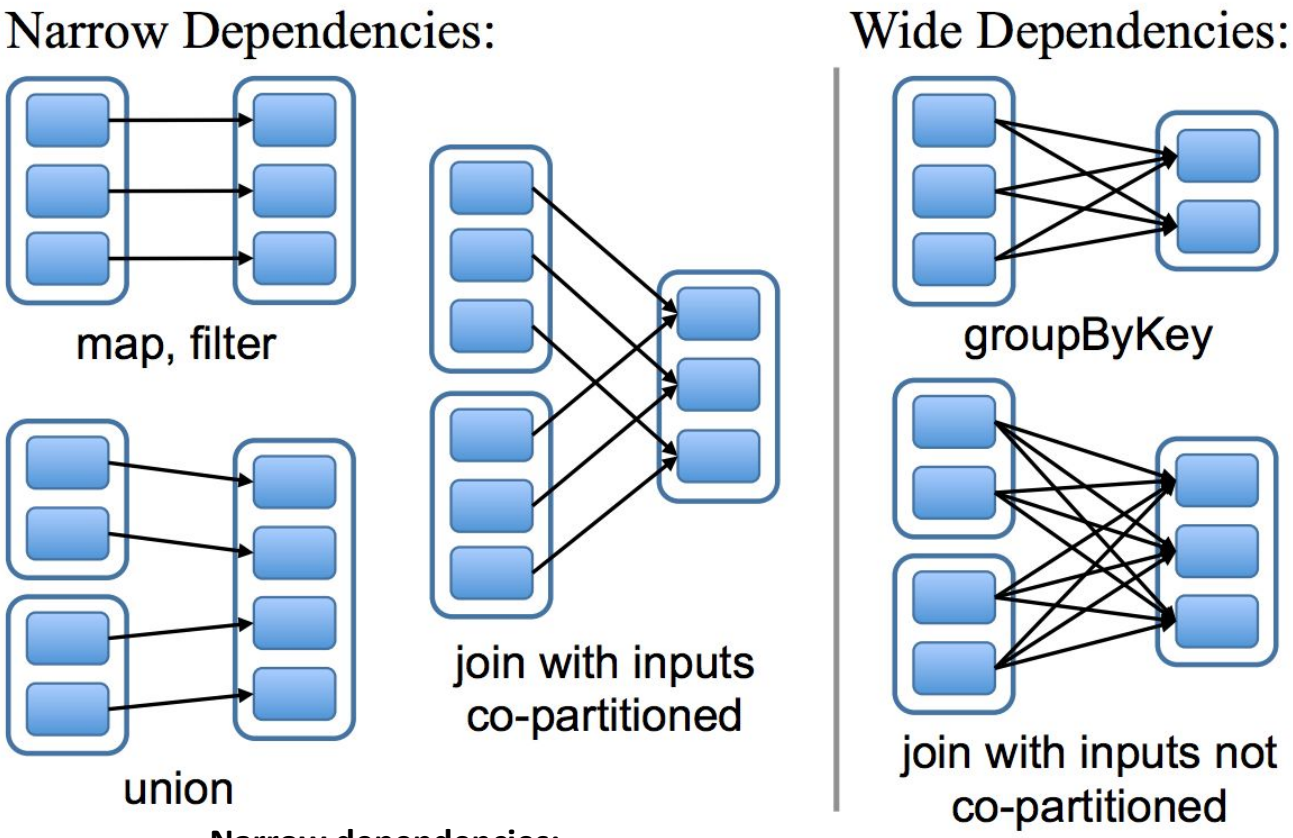
Apache Spark

- Started in 2009 at UC Berkeley
- Donated to Apache in 2013
- Written on top of JVM mainly in Scala
- 10x-100x faster than Hadoop
- Supports coding in:
  - Scala
  - Java
  - Python
  - R
- Supports an interactive shell
- More details in this paper:
  - http://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

Resilient Distributed Datasets

- A logical collection of data
  - Partitioned across multiple machines
- Logs the lineage of the current data
  - If there is a failure, recreate the data
  - Solves the reliability problem
- Developers can specify the *persistence* and *partitioning* of RDDs

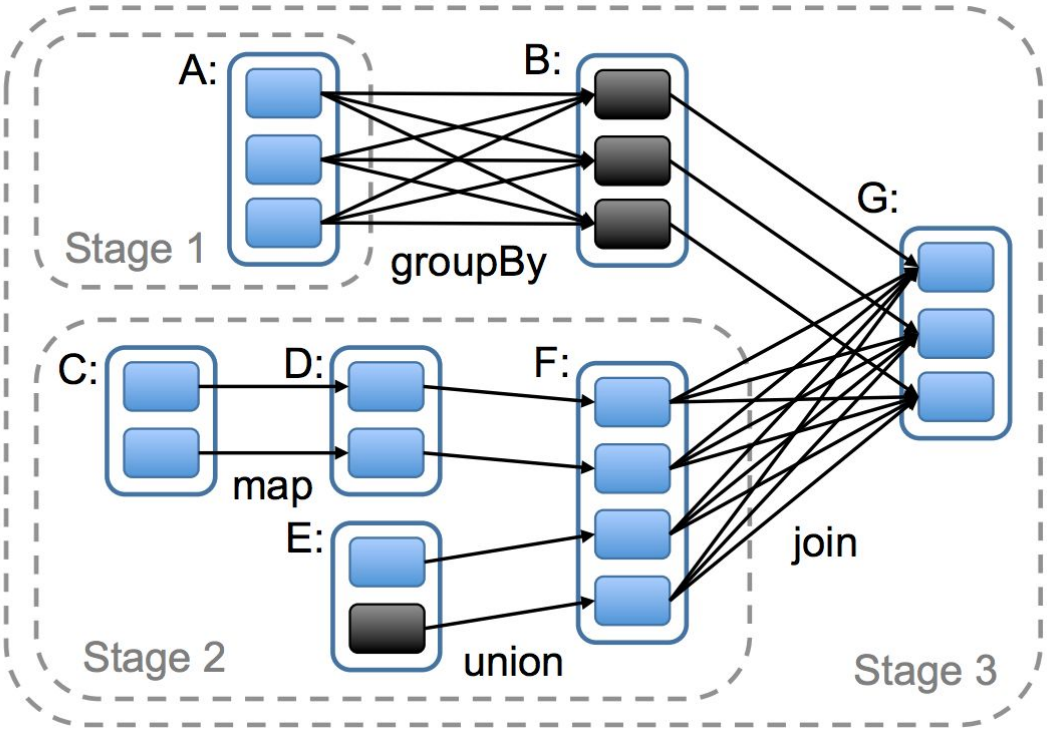
Narrow and Wide dependencies



Narrow dependencies:
Each partition of the parent is used by one child partition
Wide Dependencies:
multiple child dependencies depend upon it

Source: http://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

How Spark computes jobs



Boxes with solid outlines are **RDDs.**
Partitions are shaded rectangles, in black if they are **already in memory**.

To run an action on RDD G, build **stages** at wide dependencies and **pipeline** narrow transformations inside each stage.

In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

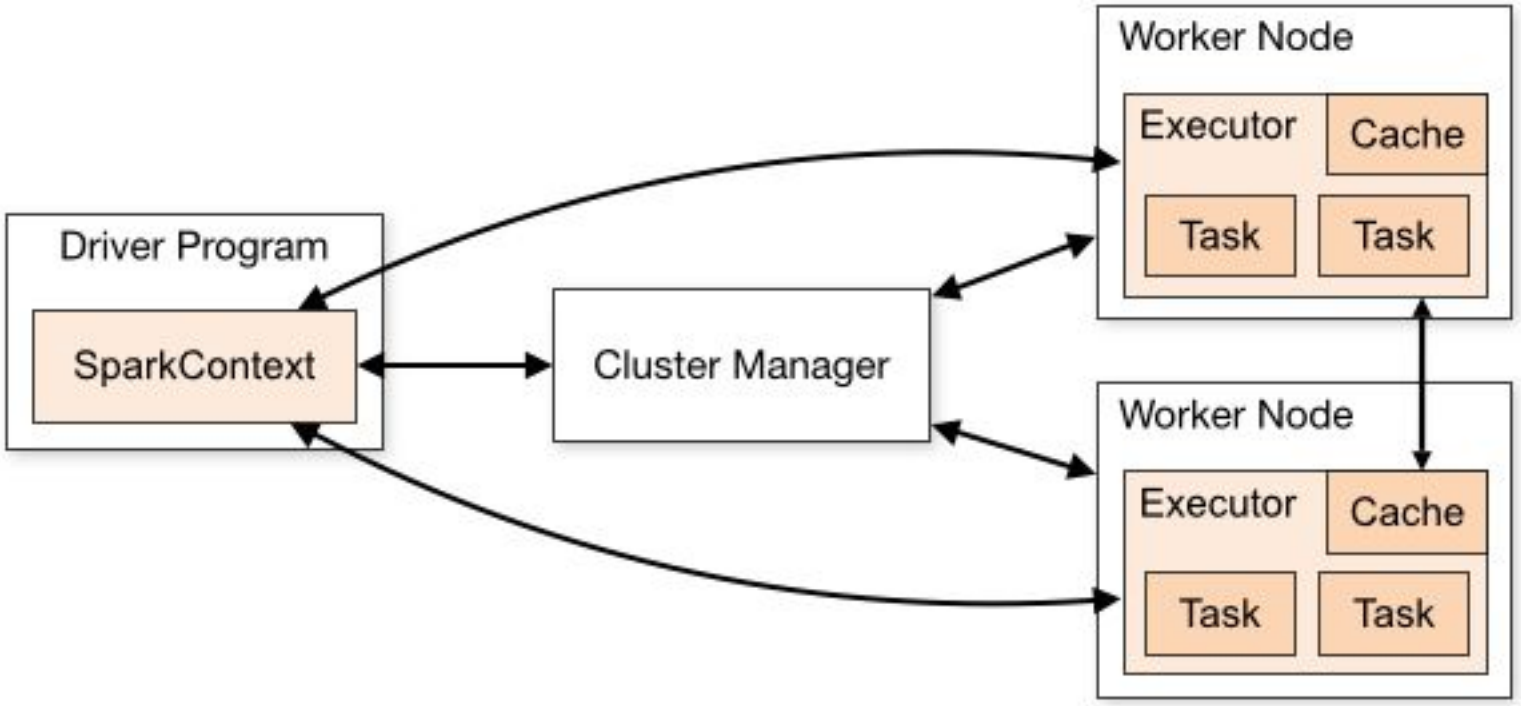
Source: http://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

Hadoop vs Spark sorting

	Hadoop World Record	Spark 100 TB *	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400	6592	6080
# Reducers	10,000	29,000	250,000
Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min
Sort Benchmark Daytona Rules	Yes	Yes	No
Environment	dedicated data center	EC2 (i2.8xlarge)	EC2 (i2.8xlarge)

* not an official sort benchmark record

Apache Spark cluster model



Spark Coding

- You can code in:
  - Scala
  - Java
  - Python
  - R
  - SQL
- We will be using Python in the class
- After you leave here you can use anything you like
  - Including “Not Spark”

Spark Key Objects

- RDD
  - Think of it like an array
  - You can do map/reduce operations on it
    - And others
  - But you can't assume everything is run on one machine
  - Unless you explicitly force that using forEach() or collect()
- DataFrame
  - Think of it like a DB resultSet
- You can convert from DF <-> RDD

Apache Spark RDD objects

- Typical operations include
  - map: apply a function to each line/element
  - flatMap: can return a sequence not just an element
  - filter: return element if func(element) is true
  - reduceByKey: reduces a set of [K,V] key/value pairs
  - reduce: apply a reducer function
  - collect: get all the results back to the master (driver) server in the cluster
  - foreach: apply a function across each element
- Operations on RDDs will happen across machines
  - Be careful!

Most common

- RDD.map(lambda x: …)
  - Applies the lambda function to each element in the RDD
- RDD.flatMap(lambda x: …)
  - The lambda produces a sequence of items that are then flattened into a single RDD
- RDD.reduce(lambda x,y: …)
  - Applies the function iteratively across all the elements in the RDD

reduceByKey

- Function (V,V) → V
- Takes pairs (K,V)
  - It will apply the function *within* the Key K
  - [(hello, 1), (hello, 1), (hello, 1), (world,1), (world, 1)]
    lambda x,y: x+y
- What is the result?

Getting results

- You often need to bring the results back to a single thread to display them:
  - collect()
- Alternatively you can save the results (which can happen in parallel)
  - RDD.saveAsTextFile()
  - DataFrame.save()

Other useful things

- first()
  - Returns the first member of an RDD
- take(10)
  - Returns the first 10 elements
- sample(..)/takeSample(..)
  - Samples the RDD
  - Very useful for reducing a massive dataset to something workable while you are testing
- count()
  - Counts the RDD
- countByKey()
  - Counts by key
  - Might have been useful in our word count example 😊
- forEach()
  - Allows you to do operations with side-effects (accumulators)

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
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.
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
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*) (Java and Scala)	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. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using `SparkContext.objectFile()`.
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. **Note**: modifying variables other than Accumulators outside of the `foreach()` may result in undefined behavior. See Understanding closures for more details.

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
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).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
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.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numTasks*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numTasks` argument to set a different number of tasks.

aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
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. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`.
cogroup(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is 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).
pipe(*command*, *[envVars]*)	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(*numPartitions*)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
repartition(*numPartitions*)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
repartitionAndSortWithinPartitions(*partitioner*)	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling `repartition` and then sorting within each partition because it can push the sorting down into the shuffle machinery.

Serialization

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

Lambda syntax

- Lambda's are unnamed functions
  - From Alonzo Church's 1930s work on the Lambda Calculus
  - Recently added to Java

Lambda syntax in Python

- Simply:

```
f = lambda x: x.split()
g = lambda x,y: x+y
```

Tuples
Clever pattern matching

A tuple in Python is just (x,y) or (x,y,z)

You can have tuples in tuples:

(x, (y,w), z)

What parameters do the following functions take and return?

lambda x,y: x+y
lambda (x,y): x+y
lambda (w,v),(x,y): ((w+x), (v+y))
lambda (x,(y,z)): (x,y+z)

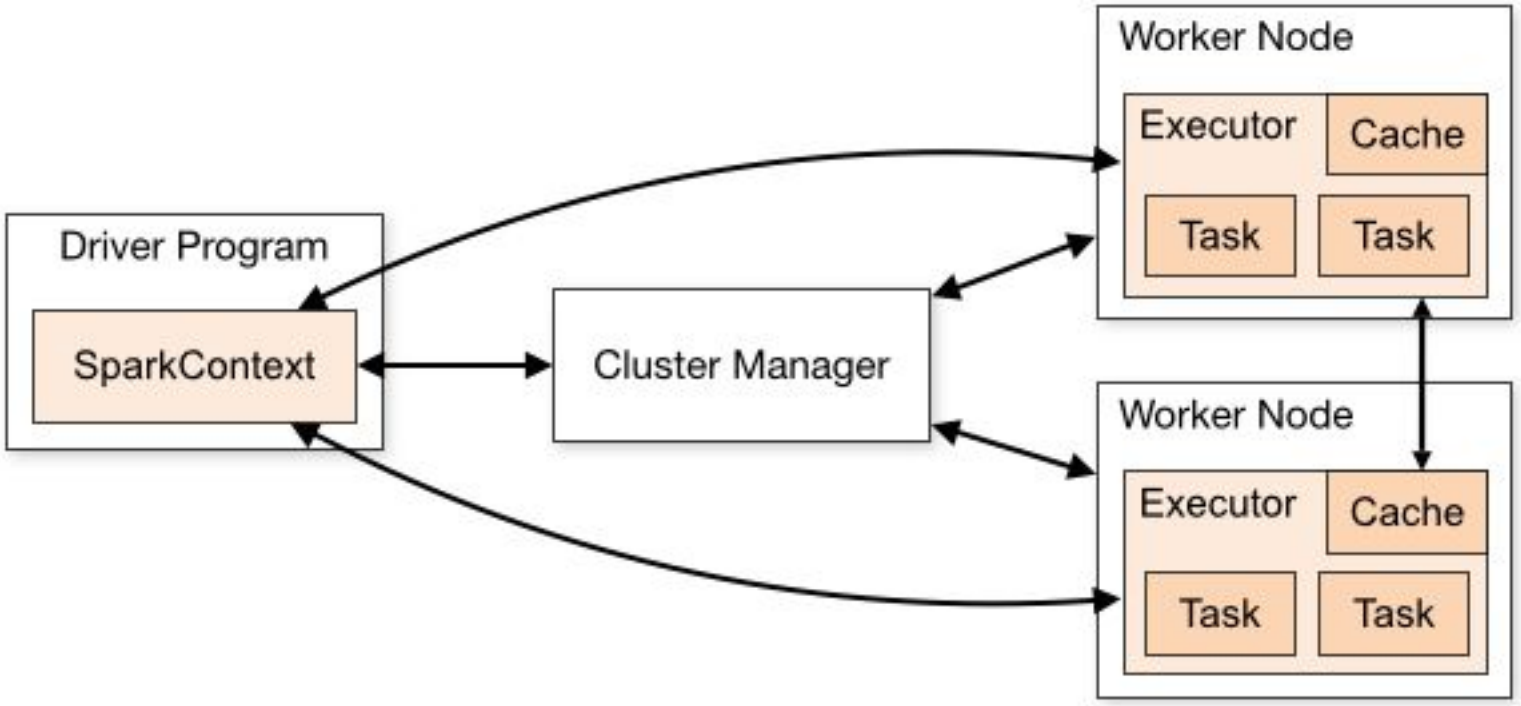
Example

```
sc = SparkContext()

books = sc.textFile("books/*")
split = books.flatMap(lambda line: line.split())
numbered = split.map(lambda word: (word, 1))
wordcount = numbered.reduceByKey(lambda a,b: a+b)

for k,v in wordcount.collect():
  print k,v

sc.stop()
```

What doesn't work in a cluster

```
counter = 0
rdd = sc.parallelize(data)

# Wrong: Don't do this!!
rdd.foreach(lambda x: counter += x)

print("Counter value: " + counter)
```

Apache Spark cluster model



How to count across a cluster?

- Accumulators

```
acc = sc.accumulator()
rdd = sc.parallelize(data)
rdd.foreach(lambda x: acc.add(x))
```

What also doesn't work

- rdd.forEach(println)
- Of course this *will* work when you test in local mode

SparkSQL

- Integrates into existing Spark programs
  - Mixes SQL with Python, Scala or Java
- Integrates data from CSV, Avro, Parquet, JDBC, ODBC, JSON, etc
  - Including joins across them
- Fully supports Apache Hive
  - *If you build it with Hive support*
- Fits into the resilient scalable model of Spark

Spark SQL example

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

lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: Row(name=p[0], age=int(p[1])))

schemaPeople = sqlContext.createDataFrame(people)
schemaPeople.registerTempTable("people")

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

teenNames = teenagers.map(lambda p: "Name: " + p.name)
for teenName in teenNames.collect():
  print(teenName)
```

DataFrame

Based on Python and R dataframes

- Column based object used by SQL
- Offers SQL like programming
- Supports algebraic optimisation and code gen
- E.g. in Scala:

```
means = users.where(users["age"] > 20)
              .groupBy("city")
              .avg("income")
```

And they run up to 2-5x faster than equivalent computations expressed via the functional API.

More SQL

```
df.
  select('postcode’,’id').
  withColumn('first_pc',
    split(df.postcode, '\s’[0]).
    where((col("first_pc") == 'SW11') or
                (col("first_pc") == 'OX1')).
     groupBy('first_pc').
     agg({"id": "count"}).show()
```

Spark packages

- A wide set of plugins
  - Currently 148 community donated plugins
- Data connectors
  - Cassandra, Couchbase, Mongo, CSV, etc
- Machine Learning, Neural networks
- Streaming
- etc

Using Spark Packages

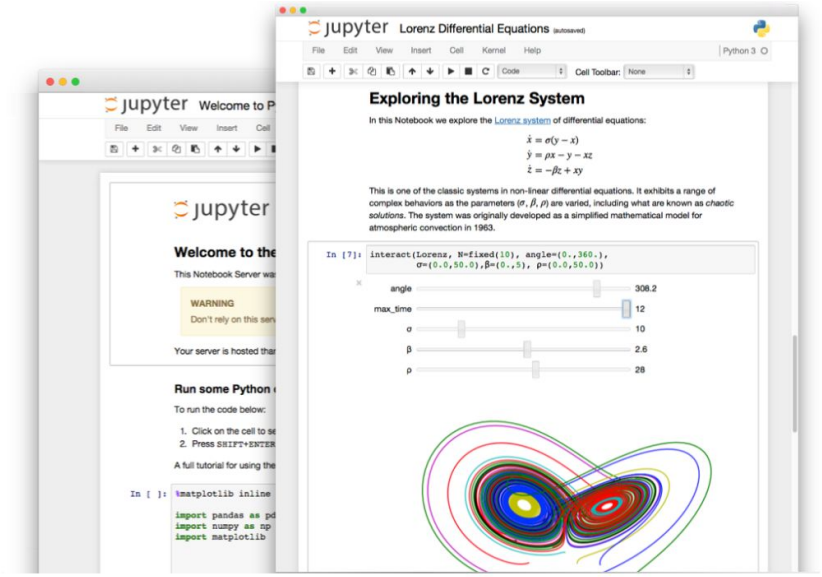
Automatic download from the web:

```
bin/spark-shell
    --packages com.databricks:spark-csv_2.11:1.2.0
```

Notebooks

- A way of creating and sharing big data analysis
- Combines code, comments, analysis and results
  - Jupyter
    - Previously IPython, now much extended for Big Data
    - Supports not just Python, but Spark Scala and others
  - Apache Zeppelin
    - A new project aimed at multiple big data analysis models

Jupyter



The Jupyter Notebook

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and explanatory text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, machine learning and much more.

Language of choice

The Notebook has support for over 40 programming languages, including those popular in Data Science such as Python, R, Julia and Scala.

Share notebooks

Notebooks can be shared with others using email, Dropbox, GitHub and the Jupyter Notebook Viewer.

Interactive widgets

Code can produce rich output such as images, videos, LaTeX, and JavaScript. Interactive widgets can be used to manipulate and visualize data in realtime.

Big data integration

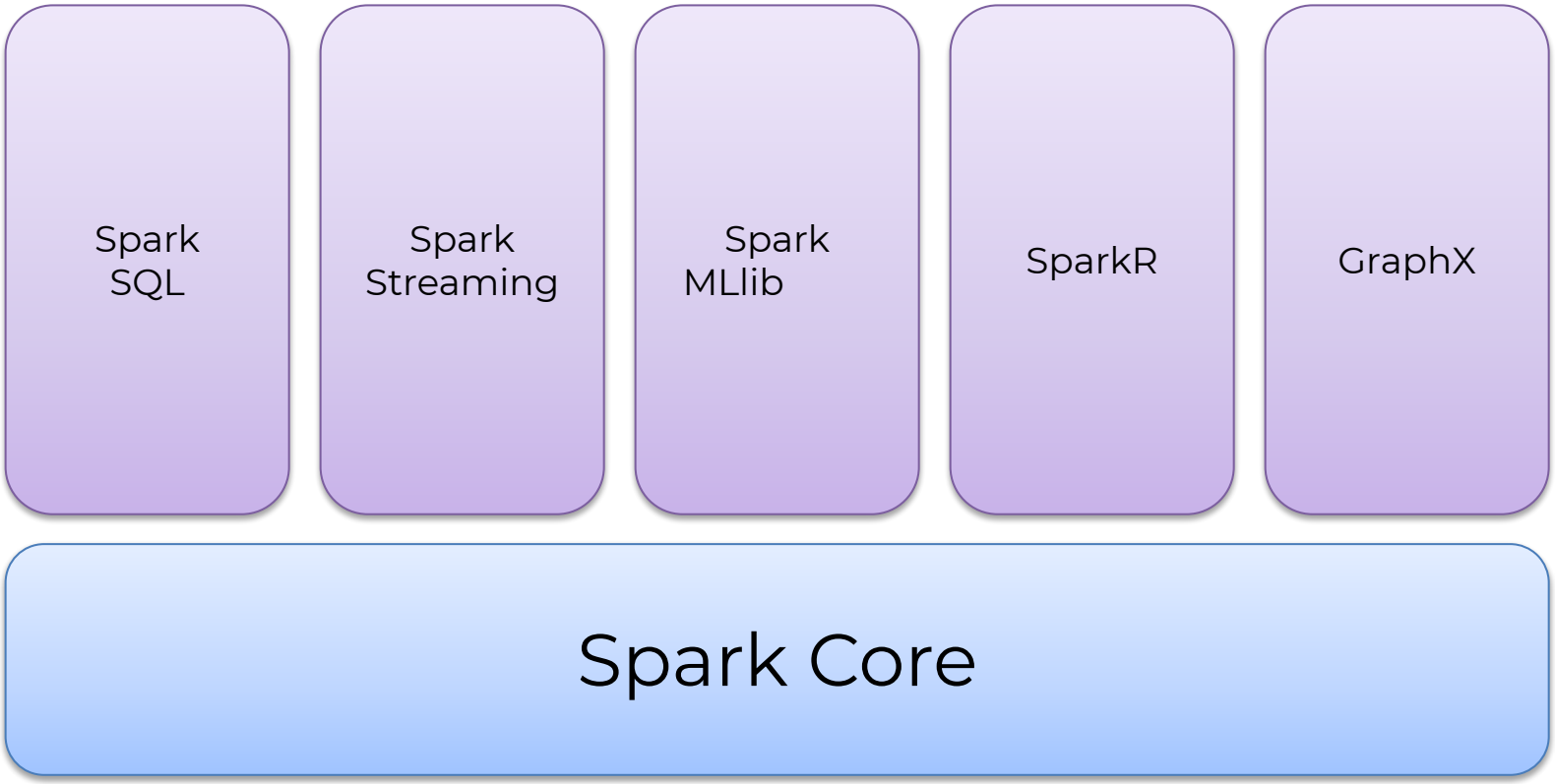
Leverage big data tools, such as Apache Spark, from Python, R and Scala. Explore that same data with pandas, scikit-learn, ggplot2, dplyr, etc.

Time for a lab!

Locality

- Spark understands the locality of data:
  - Already in memory
  - HDFS location
  - Cassandra location

Spark Extras



Spark Extras

- Spark Streaming
  - Realtime analysis in Spark
- Spark MLLib
  - Like Mahout – Machine learning in Spark
- GraphX
  - Graph processing in Spark
- SparkR
  - R statistical analysis on Spark

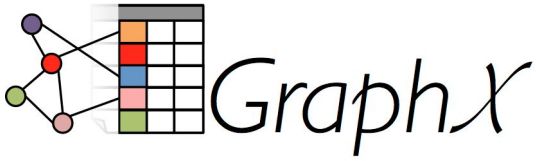
Spark MLlib

- Simple stats and correlation testing
- Classification and regression
- Collaborative Filtering
  - Alternating Least Squares
- Clustering
  - k-means, etc
- Frequent Pattern Mining
- Plus more

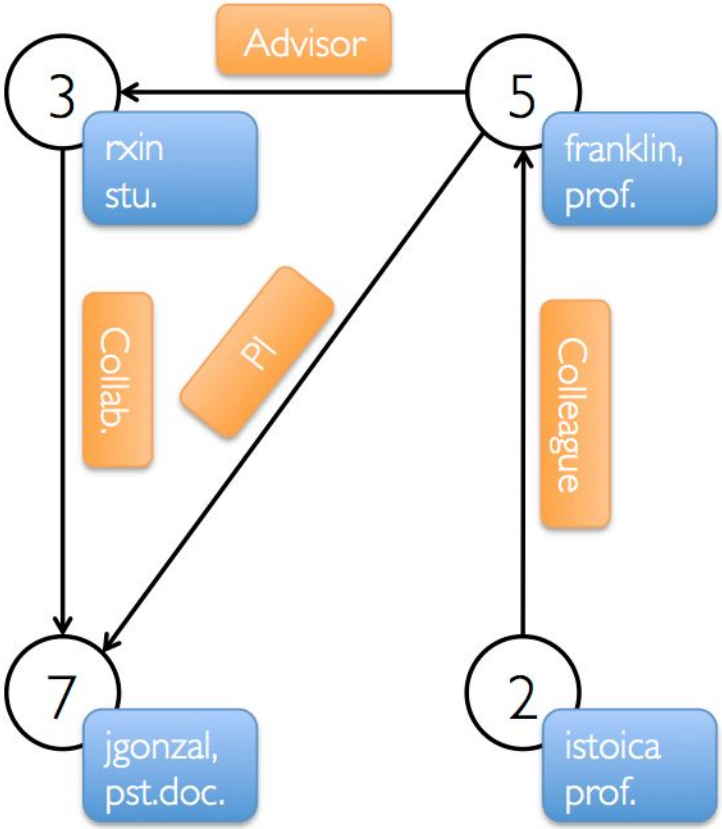
MLlib example

```
from pyspark.mllib.fpm import FPGrowth

data = sc.textFile("data/mllib/sample_fpgrowth.txt")

transactions = data.map(lambda line: line.strip().split(' '))

model = FPGrowth.train(transactions, minSupport=0.2,
numPartitions=10)

result = model.freqItemsets().collect()
for fi in result:
    print(fi)
```

GraphX



Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

R

- R is an open source system for statistics and graphics
  - Based on the S language from AT&T Bell Labs
- Supports a wide variety of statistical techniques and graphing tools
- An extensible set of packages that provide extra functions via CRAN
  - The Comprehensive R Archive Network

SparkR

- A lightweight approach to use Spark from within R
- Also works with MLlib for machine learning
- Allows complex statistical analysis to be done on a Spark cluster

Questions?