Big Data Engineering

Apache Spark



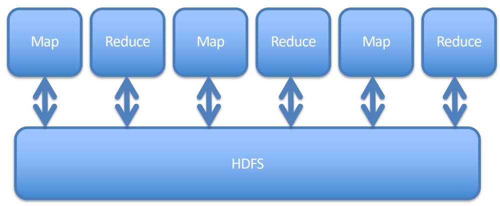
## Contents



## 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

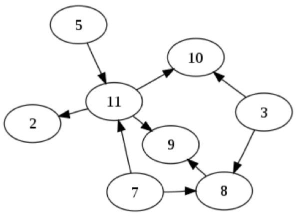
## Improved Approach

- A new model based on memory
  - Based on Directed Acyclic Graphs (DAGs)
  - 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



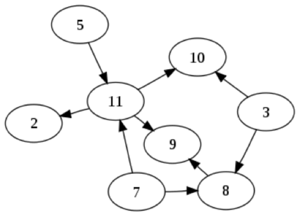
## Resilient Distributed Datasets

- 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



## RDD Lineage

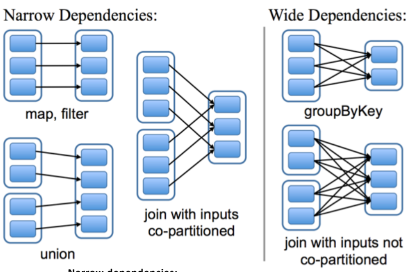
(aka RDD operator graph and RDD dependency graph)




- Which RDDs depend on which other RDDs?
- Why is it important that it is a DAG?



## Narrow and Wide dependencies

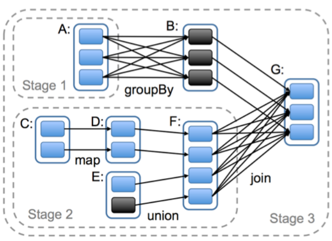



**Narrow dependencies:**
Each partition of the parent is used by one child partition
**Wide Dependencies:**
multiple child dependencies depend upon it

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

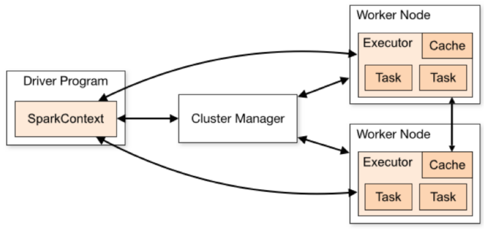
## Hadoop vs Spark sorting

| | 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




© Paul Fremantle 2015. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License
See http://creativecommons.org/licenses/by-nc-sa/4.0/

## Spark Coding

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

© Paul Fremantle 2015. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License
See http://creativecommons.org/licenses/by-nc-sa/4.0/

## 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
  - Just like a Pandas DataFrame except distributed across machines and threads
- You can convert from DF <-> RDD

© Paul Fremantle 2015. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License
See http://creativecommons.org/licenses/by-nc-sa/4.0/

## 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!

© Paul Fremantle 2015. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License
See http://creativecommons.org/licenses/by-nc-sa/4.0/

## 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

© Paul Fremantle 2015. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License
See http://creativecommons.org/licenses/by-nc-sa/4.0/

## 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. |

## Transformations and Actions

- **Transformations**: create a new RDD from an existing one
- **Actions**: return a value to the driver program after carrying out a computation
- What are map(), reduce() and reduceByKey()?
- All transformations in Spark are **lazy**: only carried out when needed by an action



## Serialization and Deserialization

- **serialization** = converting an object into a sequence of bytes which can be persisted to a disk or database or can be sent through streams.
- **deserialization** = creating object from sequence of bytes.
- may be necessary for persistence of data
- but avoid unnecessary serializations and deserializations (e.g., RDD → DF → RDD)
- why?



## Lambda syntax

- Lambda's are unnamed functions
  - From Alonzo Church's 1930s work on the Lambda Calculus
- 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()
```



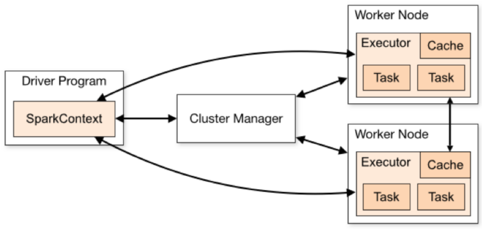
## What doesn't work
## in a cluster

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

# Wrong: Don't do this!!
rdd.foreach(lambda x: counter += 1)

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

## Apache Spark cluster model






## How to count across a cluster?

- Accumulators

```
acc = sc.accumulator(0)
rdd = sc.parallelize(data)
rdd.foreach(lambda x: acc.add(1))
```

© Paul Fremantle 2015. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License
See http://creativecommons.org/licenses/by-nc-sa/4.0/

## What also doesn't work

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

© Paul Fremantle 2015. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License
See http://creativecommons.org/licenses/by-nc-sa/4.0/

## Questions?

© Paul Fremantle 2015. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License
See http://creativecommons.org/licenses/by-nc-sa/4.0/