# lecture 5:spark

wbg231

January 2023

## 1 review

- name node manages meta data, coordinates data access and ensures data integrity
- the data note stores each block of data and does data processing could be where computation is done in map reduce
- think of it as worker and head node

### why spark

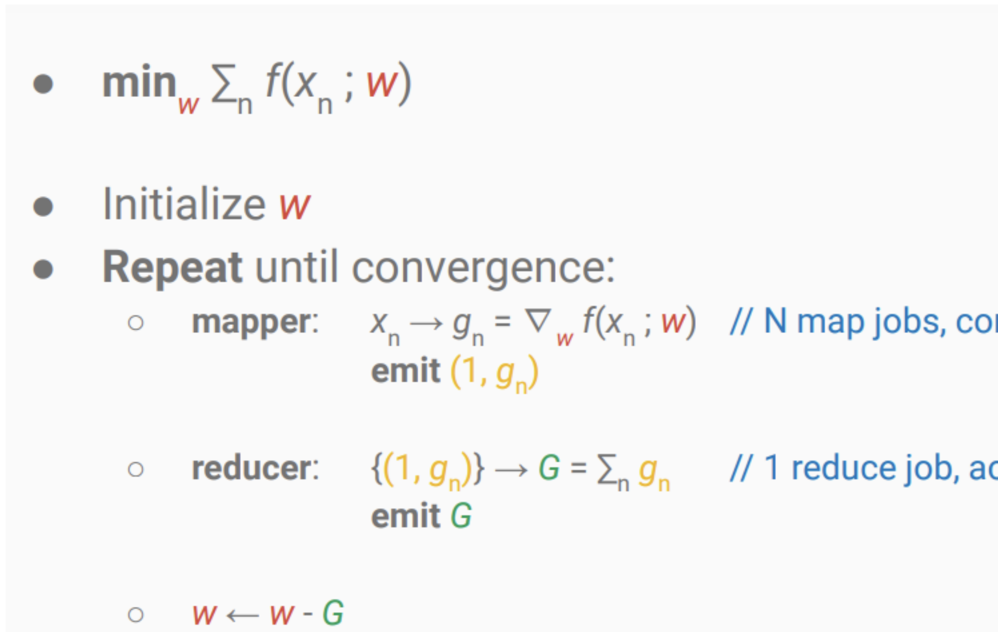
- what was good about map reduce
  - it is scalable on hadoop and allows for parallel processing of big data
  - it has fault tolerance
  - it works well with off the shelf hardware
  - takes care of most of the really gritty stuff of parallel computing like job scheduling
- what was bad about map reduce
  - it is build on an acyclic data flow models
  - it is to low level

### why is it to low level

- map reduce is great for one time jobs with simple dependencies on big data
- but it can not work with iterative jobs, complex queries exc

### gradient descent example

- imagine trying to do gradient descent in map reduce




- as you can see here
- you would have to at every stage in the loop map n jobs to compute the gradient
- and then reduce those into one value by taking a sum adn use that to take a step
- so each step of a gradient descent algorithm requires a full map reduce
- and we do note about the previous iteration once it is done (so they could be parallelized)
- further notice that the reducer can not start it's job until all the mappers are done with theres so there is high latency
- more generally some computations have complex pipelines that are ill suited for map reduce
- like for instance in cases where we want to do many iterations quickly

### Resilient distributed records

- Resilient distributed records or RDD's are one solution to this

## reusing data

- complex computations usually have intermediate steps
- map reduce only likes to compute something save an intermediate result and move onto the next step
- tht can be wasteful and awkward ot use for some problems

## RDDs

- an RDD has
  - a data source
  - a lineage graph of transformations to apply to the data
  - interfaces for data partitioning and iteration
- think of this as deferred computation
  - nothing is computed until you ask for for it
  - nothing is saved until you say so
  - this makes optimization easier
- we are going to say RDD[T] is an RDD with type t

## RDD example: log processing

- suppose we have a long document and we want to find all the lines in that document that start with the word error

Spark code

```
lines = spark.textFile("hdfs://...")

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

errors.filter(_.contains("MySQL"))
      .map(_.split('\t')(3))
      .collect()
```

- 
- so first off note what the colors mean rdd data transformations action

- so the first thing we do is read the file into an rdd
- then we transform the data using two filters and a map
- then we take an action and collect the data
- note that no computation happens until the action (in this case collect)

## transformations

- transformations turn one or more rdds into a new rdd
- transformations are cheap to construct because they do not actually do the computation until and action is taken
- building an rdd is like writing a map reduce script or sql query (nothing goes until you click enter)
- example transformations are map, filter, union

## actions

- actions are what execute the computation defined by an RDD
- results of actions are not RDDs
- example actions: count, collect, reduce, save

## work backwards then forwards

- notice that every step depends on what happened before it
- and previously computed Rdd's can be cached and reused
- any lost/corrupted RDD can be rebuilt from a linear graph

### lineage graph

- a linear graph does not have to be a line there can be any rdd can have multiple parents
- once a parent rdd is computer it can be cached and used by many descents
- lineage can be pipelined
- we do not need to wait for all lines to finish to build errors
- there is no need for intermediate storage like in map reduce

- more or less until an action is taken we have the linage graph so any worker node can do all transformations in it's lineage graph as soon as it finishes one with out weighting for the others to finish
- compare this to non pipelined implementations that have to finish all of one transformation then move to the next.

### slido

- RDDs can depend on each other only in a feed forward one to one fashion
- i think false, there can be multiple parents so it does not have to be one to one

### the rdd interface

- partitions() returns a list of partitions kinda like splits in map reduced
- preferred locations (p) ie the HDFS node where partition p can be found
- dependencies() the the dependencies for this RDD
- iterator(p, parentitters) get elements of partition p given parents partitions
- partitioner get meta data about how the rdd is partitioned

### narrow and wide dependencies

- narrow dependencies all the partitions of one RDD go to one child rdd
- this is good we have low computation, data stays localized, it is easy to pipeline, it is easy to recover from computer failure
- a wide dependencies is when the partitions of parent RDD goes to multiple child RDD partitions
- high computation
- high latency
- hard to pipeline
- hard to recover data if a computer fails

### slido

- much like in map reduce spark ahs to wait for a step in a lineage to complete
- false that is the whole point

## actually using spark in 2023

### V0 in 2009-2012

- a cluster computing frame work for using RDDs
- integrates with hadoop ecosystem
- written in scala with API in other languages like R java and python

### architecture: session and driver

- the driver is the process that you run on the head or login node
- the session object connects your code to the cluster and compute nodes.

### why is spark written in scala

- RDD design fits well with functional programming
- scala compiles to the java virtual machine, which is compatible with python

### closure

- closure are a functional programming concept that combines a function with its environment
- scala is well suited for this

### gradient descents in spark

- here is the psudo code
- as we can see the outer loop still runs in series
- but the inner loop can run in paralell

- so there is a lot less waiting than if the inner loop had to wait for every othe rpoint evert time
- then the grad is a shared accumulator which is a write only data structure that always is added to

### beyond scala

- spark can work in a lot of languages now
- beware r and python spark still may not be as fast as scala spark
- it is expensive to use operations that are just in python since but spark tries to limit this by seralizing all data
- so we do not write raw rdd code in python, but we do use existing packages written in scala with python bindings

## spark data frames

- rdds are very good but can be cumbersome, for ad hoc computations
- data frames are common representions in many languages
- spark 2 has a data frame api as a primary interface

### rdd are more than columns

- RDDs can be derived from other Rdds through transformations
- RDD also have partition information which influinces how they are stored in HDFS
- one or more RDDs can be used to form a data frame
- could have an RDD with compound types but RDDs are more convient

### data frames and rdds

- data frames in spark are like relations in RDBMS
- they have well defined shcmes with types over columns
- each row is pretty much a tuple
- data frame operations are translated into RDD transformations

- RDD transfomration can be executed within theJVM, that means while using a data frame we can use python or other language operations with out seralization
- when using RDBMS we often think of our data in rows
- data frames are implemented as a collection rdd with 1 column = 1 rdd
- that is data frames are column oriented
- this does not change how we interact with them much as a programer but does change how we think about storage of data

### spark sql

- we cab also use sql queries in spakr (or us a data frame object oriented method)
- queries are executable against data frames
- data frames are secretly RDDs not RDBMS
- queries can be optimized by analyzing the lineage graph of the RDD that is being worked with

### reparitioning

- before running an action on a dataframe run teh explain method
- this will tell you an execution plan, and might help identify bugs
- be carefull with .collect() it can be a bad includegraphics

### map reduce requires both map and reduce to be deterministic, is this the same for map reduce

- true for map reduce and for spark

### determinism in spark

- transfomrations in spark need have to be deterministic
- with out this reconstructin from a lineage graph would not make sense
- what problems could this present when would ramdomization be helpful

## slido

- do any of the stonebreaker critisms of map reduce apply to spark?
- it is higher level so that is good
- it has query optimization but that does not replace indexing
- it is missing RDMS features, but partitioning is kinda liek that, we do have shcemas transactions matter les due to read only data
- RDMS compatable speark i not a RDMS ut it is better integrated with data frames

## wrap up on spark

- RDD frame work is more flexible than map reduce
- chacing can make interactive job faster
- spark sql dta frames make devlopment faster