A file of size 8 PB (petabytes) needs to be stored in HDFS. Assuming block size=128 MB, find the number of blocks needed.

Answer = 8 x 2 ^50 / 128 x 2 ^20 = 2^26

**MapReduce**

• Design Considerations:

- process vast amounts of data (multi-terabyte data-sets)

- parallel processing

- large clusters (thousands of nodes) of commodity hardware

- reliable

- fault-tolerant

- should be able to increase processing power by adding more nodes

-> "scale-out" and not "scale-up".

- sharing data or processing between nodes is bad

-> ideally want "shared-nothing" architecture.

- want batch processing

-> process entire dataset and not random seeks

**Isolated Tasks**

• Map and Reduce tasks work in isolation from each other

• This is called task isolation.

• Saves bandwidth, no waiting for other nodes.

• Ideally, each node works on local data => Idea of moving computation to data

• There is a process called TaskTracker that runs on each DataNode.

• It monitors the tasks and communicates results with a JobTracker that runs on NameNode

**Job Tracker is the master node** (runs with the namenode)

• Receives the user’s job

• Decides on how many tasks will run (number of mappers)

• Decides on where to run each mapper (concept of locality)

This file has 5 Blocks => 5 map tasks

1 key => 1 reduce task

**stragglers**

MapReduce attempts to locate slow tasks (stragglers) and run redundant (speculative) tasks that will optimistically commit before the corresponding stragglers

§ This process is known as speculative execution

**combiner are optional optimizations!**

" May be run 0, 1, or multiple times!

**“Stripes” Analysis"**

! Advantages!

" Far less sorting and shuffling of key-value pairs!

" Can make better use of combiners!

! Disadvantages!

" More difficult to implement!

" Underlying object more heavyweight!

" Fundamental limitation in terms of size of event space!

**Projection in MapReduce**

• Easy!

– Map over tuples, emit new tupleswith appropriate

attributes

– No reducers, unless for regrouping or resorting tuples

– Alternatively: perform in reducer, after some other

processing

• Basically limited by HDFS streaming speeds

– Speed of encoding/decoding tuplesbecomes

important

– Relational databases take advantage of compression

– Semistructureddata? No problem!

• **In MapReduce:**

–Map over tuples, emit time, keyed by url

–Framework automatically groups values by keys

–Compute average in reducer

–Optimize with combiners

**Reduce-side Join**

• Basic idea: group by join key

– Map over both sets of tuples

– Emit tuple as value with join key as the intermediate key

– Execution framework brings together tuples sharing the same key

– Perform actual join in reducer

– Similar to a “sort-merge join” in database terminology

**Map-side Join: Parallel Scans**

• If datasets are sorted by join key, join can be

accomplished by a scan over both datasets

• How can we accomplish this in parallel?

– Partition and sort both datasets in the same manner

• **In MapReduce:**

– Map over one dataset, read from other corresponding partition

– No reducers necessary (unless to repartition or resort)

**In-Memory Join**

• Basic idea: load one dataset into memory, stream

over other dataset

– Works if R << S and R fits into memory

– Called a “hash join” in database terminology

• MapReduce implementation

– Distribute R to all nodes

– Map over S, each mapper loads R in memory, hashed by join key

– For every tuple in S, look up join key in R

– No reducers, unless for regrouping or resorting tuples

**In-Memory Join: Variants**

• Striped variant:

– R too big to fit into memory?

– Divide R into R1 , R2 , R3 , ... s.t. each Rn fits into memory

– Perform in-memory join: "n, Rn S

– Take the union of all join results

**• Memcachedjoin:**

– Load R into memcached

– Replace in-memory hash lookup with memcached

lookup

**Key thing to know about Spark**

• What is the entry point for using Spark

functionality?

- SparkContext(sc) object

- It represents the connection to a Spark cluster,

and can be used to create RDDs, accumulators

and broadcast variables on that cluster.

- Almost every line of code starts with scobject.

val distFile= sc.textFile("data.txt")

By default, Spark creates one partition for each

block of the file (blocks being 64MB by default in

HDFS).

val lines = sc.textFile("data.txt")

val lineLengths = lines.map(s => s.length)

val totalLength = lineLengths.reduce((a, b) => a + b)

ÞWhen is the dataset loaded in memory?

ÞThird line:

We run reduce, which is an action. At this point

Spark breaks the computation into tasks to run

on separate machines, and each machine runs

both its part of the map and a local reduction

**RDD Abstraction**

An RDD is a read-only , partitioned collection of records

Can only be created by :

(1) Data in stable storage

(2) Other RDDs (transformation , lineage)

An RDD has enough information about how it was

derived from other datasets(its lineage)

Users can control two aspects of RDDs:

1) Persistence (in RAM, reuse)

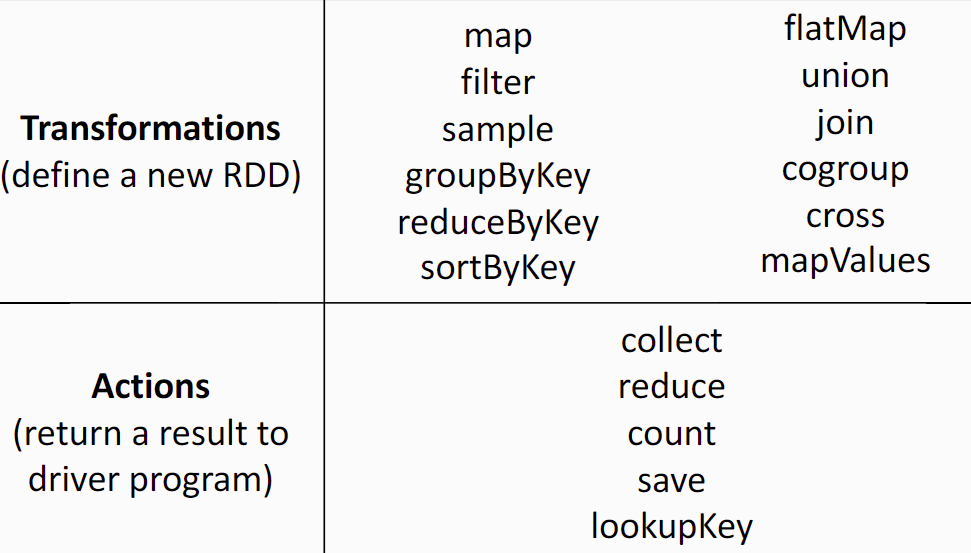
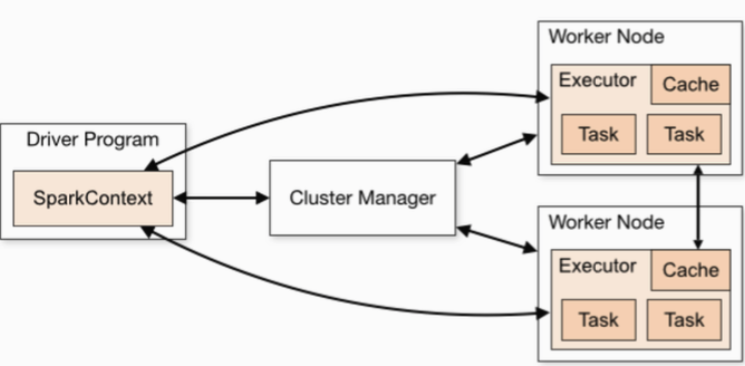
2) Partitioning (hash, range, [<k, v>])

RDD Types: parallelized collections

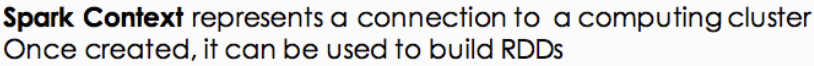
RDD Types: Hadoop Datasets

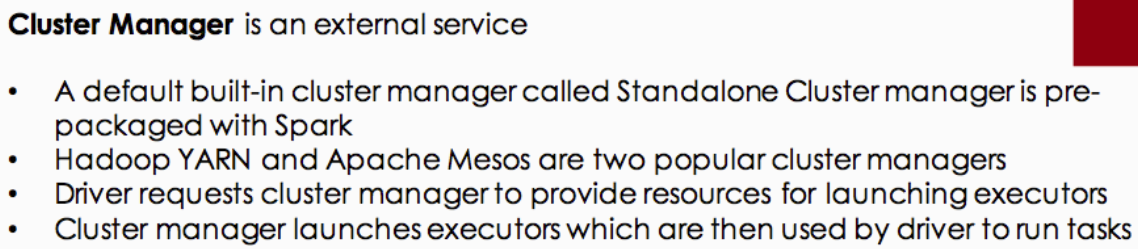
RDDs track the graph of transformations that

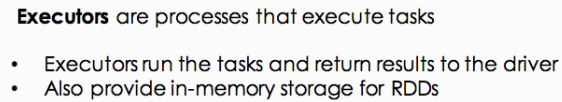
built them (their lineage) to rebuild lost data











Shortcomings of MapReduce

• Hard to reuse intermediate results across iterative computations

- Why?

Stable storage is needed which is slow and limiting

• Not suitable for interactive queries

• We want faster in-memory computing

• Need a newer model of fault tolerance

- Replication is not good for in-memory computing

(Imagine replicating an object 3x in-memory)

• We still want a distributed model of computation

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