TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

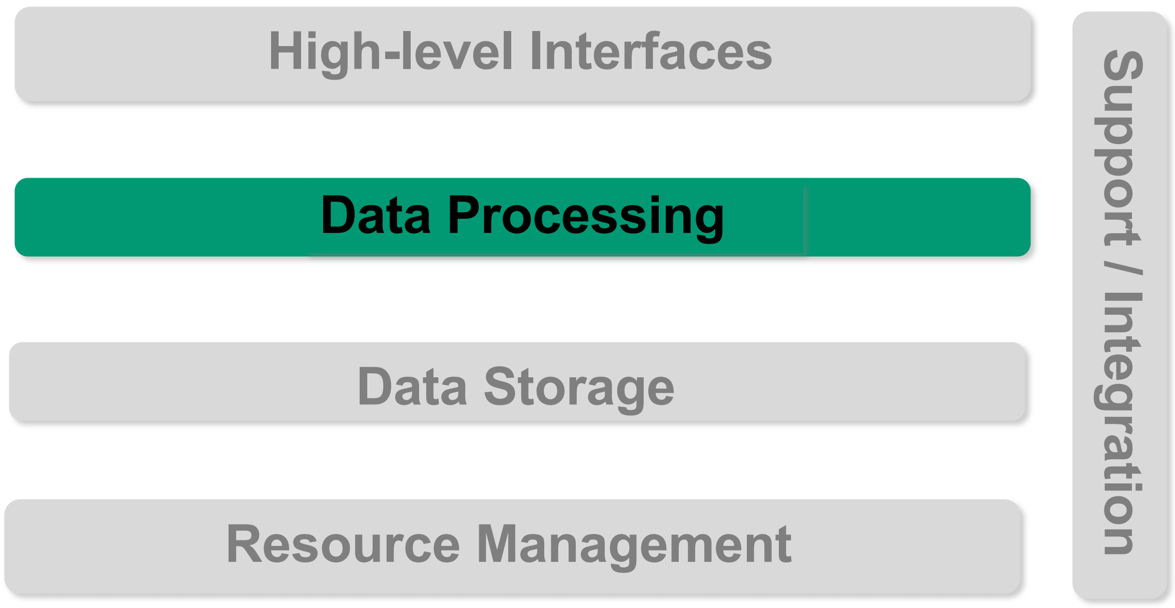
Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

MapReduce and Hadoop

Corso di Sistemi e Architetture per Big Data
A.A. 2021/22
Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

The reference Big Data stack



MapReduce

MapReduce

- Programming model for processing huge amounts of data sets over thousands of servers
  - Originally proposed by Google in 2004: MapReduce: simplified data processing on large clusters
  - Based on a shared nothing approach
- Also an associated **implementation** (framework) of the distributed system that runs the corresponding programs
- Some examples of applications for Google:
  - Web indexing
  - Reverse Web-link graph
  - Distributed sort
  - Web access statistics

MapReduce: programmer view

- MapReduce hides system-level details
  - Key idea: separate the *what* from the *how*
  - MapReduce abstracts away the “distributed” part of the system
  - Such details are handled by the framework
- Programmers get simple API
  - Don’t have to worry about handling
    - Parallelization
    - Data distribution
    - Load balancing
    - Fault tolerance

Typical Big Data problem

Map

- Iterate over a large number of records
- Extract something of interest from each record
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

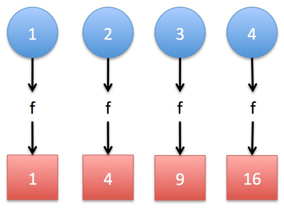
Reduce

Key idea: provide a functional abstraction of the two Map and Reduce operations

Your first MapReduce example (in Lisp)

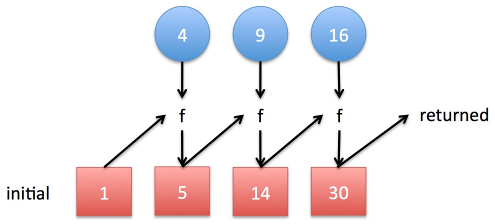
- *Example*: sum-of-squares (sum the square of numbers from 1 to n) in MapReduce fashion
- Map function:

  map square [1,2,3,4]
  returns [1,4,9,16]



- Reduce function:

  reduce [1,4,9,16]
  returns 30 (sum of square elements)



MapReduce: model

- Processing occurs in two phases: **Map** and **Reduce**
  - Functional programming roots (e.g., Lisp)
- Map and Reduce are defined by the programmer
- Input and output: sets of key-value pairs
- Programmers specify two functions: map and reduce
- **map**$(k_1, v_1) \rightarrow [(k_2, v_2)]$
- **reduce**$(k_2, [v_2]) \rightarrow [(k_3, v_3)]$
  - (k, v) denotes a (key, value) pair
  - […] denotes a list
  - Keys do not have to be unique: different pairs can have the same key
  - Normally the keys k_1 of input elements are not relevant

Map

- Execute a function on a set of key-value pairs (input shard) to create a new list of key-value pairs

  map (in_key, in_value) →
  list(out_key, intermediate_value)

- Map calls are distributed across machines by automatically partitioning input data into *shards*
  - Parallelism is achieved as keys can be simultaneously processed by different machines
- MapReduce library groups together all intermediate values associated with the same intermediate key and passes them to the Reduce function

Reduce

- Combine values in sets to create a new value

  reduce (out_key, list(intermediate_value)) →
  list(out_key, out_value)

  - The key in output is often identical to the key in the input
  - Parallelism is achieved as reducers operating on different keys can be executed simultaneously

MapReduce program

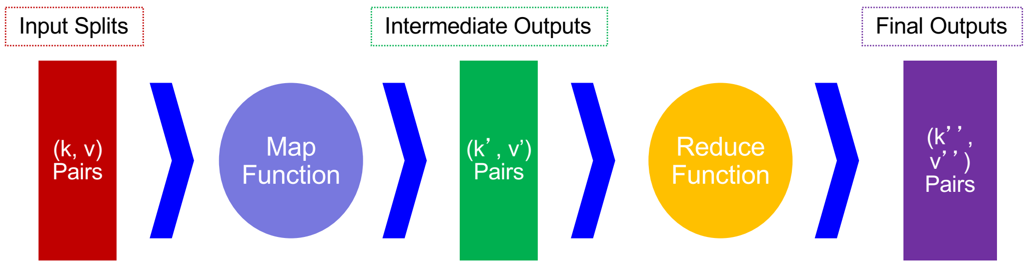
- A MapReduce program, referred to as a job, consists of:
  - Code for Map and Reduce packaged together
  - Configuration parameters (where the input lies, where the output should be stored)
  - Input data set, stored on the underlying distributed file system
    - The input will not fit on a single computer's disk
- Each MapReduce job is divided by the system into smaller units called tasks
  - Map tasks or mappers
  - Reduce tasks or reducers
  - All mappers need to finish before reducers can begin
- The output of MapReduce job is also stored on the underlying distributed file system
- A MapReduce program may consist of many rounds of different map and reduce functions

MapReduce computation

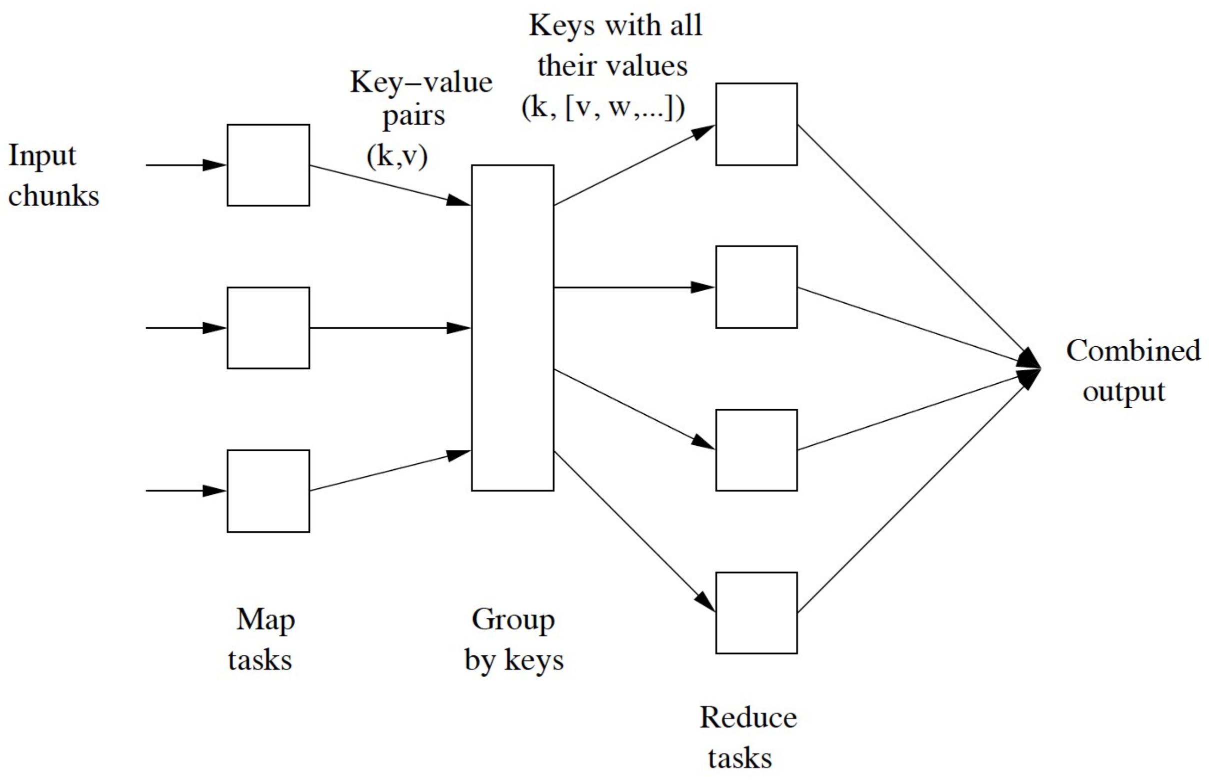
1. Some number of Map tasks each are given one or more chunks of data from a distributed file system
2. These Map tasks turn the chunk into a sequence of key-value pairs
   - The way key-value pairs are produced from the input data is determined by the code written by the user for the Map function
3. The key-value pairs from each Map task are collected by a master controller and sorted by key
4. The keys are divided among all the Reduce tasks, so all key-value pairs with the same key wind up at the same Reduce task
5. The Reduce tasks work on one key at a time, and combine all the values associated with that key in some way
   - The manner of combination of values is determined by the code written by the user for the Reduce function
6. Output key-value pairs from each reducer are written persistently back onto the distributed file system
7. The output ends up in r files, where r is the number of reducers
   - Such output may be the input to a subsequent MapReduce phase

Where the magic happens

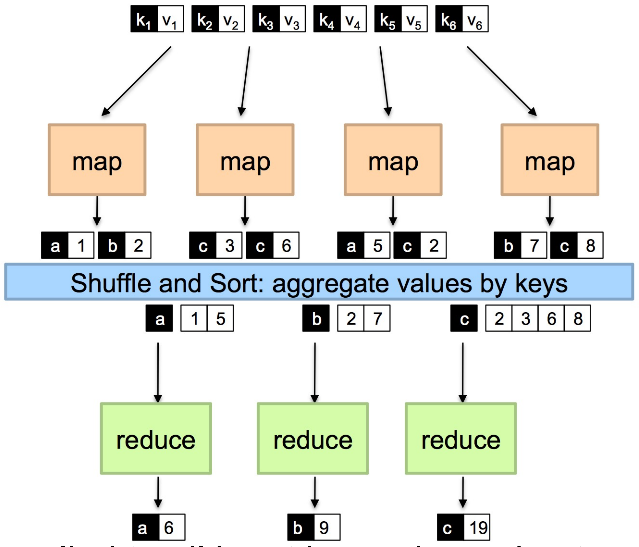
- Implicit between the map and reduce phases is a distributed "group by" operation on intermediate keys, called **shuffle and sort**
  - Transfer data from mappers to reducers, merging and sorting mappers' intermediate output
  - Intermediate data arrives at every reducer sorted by key
- Intermediate keys are transient
  - Not stored on the distributed file system
  - But "spilled" to the local disk of each node



MapReduce computation: the complete picture



A simplified view of MapReduce: example



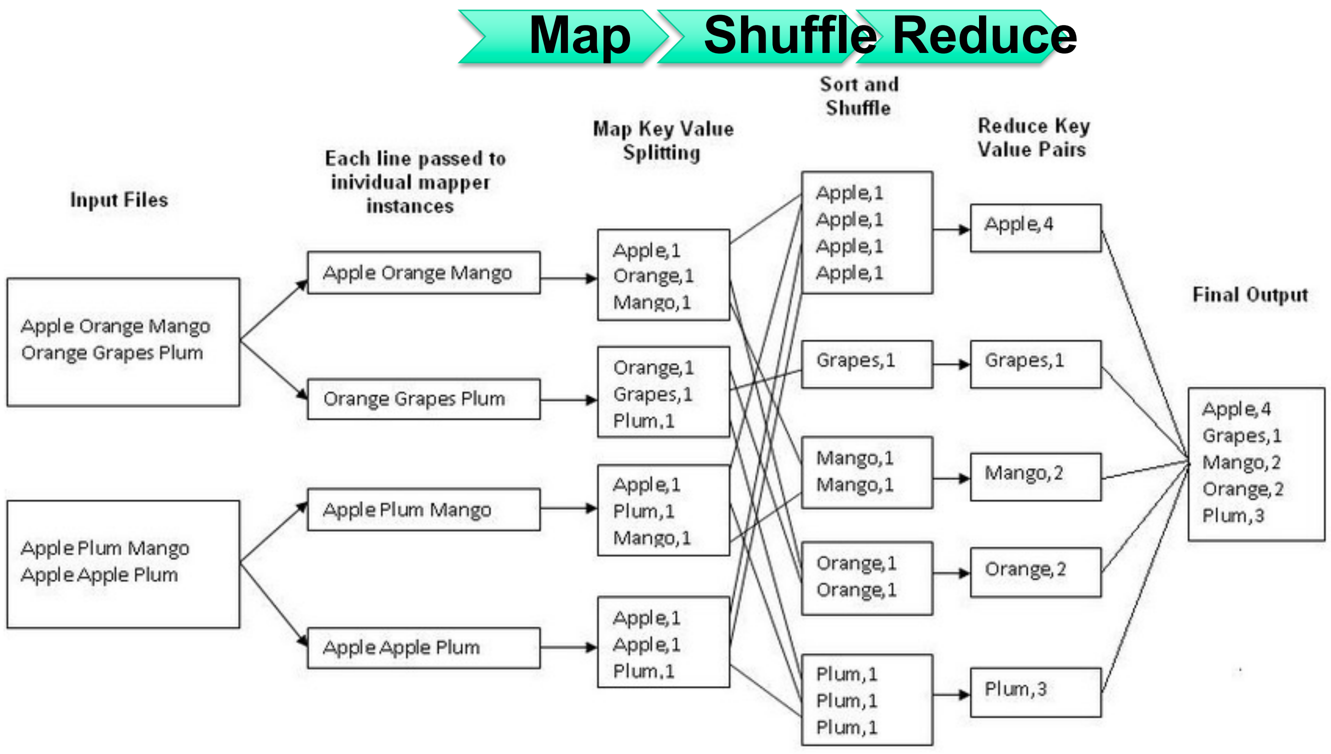
- Mappers are applied to all input key-value pairs, to generate an arbitrary number of intermediate pairs
- Reducers are applied to all intermediate values associated with the same intermediate key
- Between the map and reduce phase lies a barrier that involves a large distributed sort and group by

“Hello World” in MapReduce: WordCount

- **Problem**: count the number of occurrences for each word in a large collection of documents
- **Input**: repository of documents, each document is an element
- **Map**: read a document and emit a sequence of key-value pairs where:
  - Keys are words of the documents and values are equal to 1:

    $(w_1, 1), (w_2, 1), \dots , (w_n, 1)$
- **Shuffle and sort**: group by key and generate pairs of the form $(w_1, [1, 1, \dots , 1]) , \dots , (w_n, [1, 1, \dots , 1])$
- **Reduce**: add up all the values and emit $(w_1, k) ,\dots, (w_n, l)$
- **Output**: (w, m) pairs where:
  - w is a word that appears at least once among all the input documents and m is the total number of occurrences of w among all those documents

WordCount in practice



WordCount: Map

- Map emits each word in the document with an associated value equal to “1”

```
Map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
          EmitIntermediate(w, “1” );
```

WordCount: Reduce

- Reduce adds up all the "1" emitted for a given word

```
Reduce(String key, Iterator values):
// key: a word
// values: a list of counts
int result=0
for each v in values:
    result += ParseInt(v)
Emit(AsString(result))
```

- This is pseudo-code; see MapReduce paper for code

Example: WordLengthCount

- **Problem**: count how many words of certain lengths exist in a collection of documents
- **Input**: a repository of documents, each document is an element
- **Map**: read a document and emit a sequence of key-value pairs where the key is the length of a word and the value is the word itself:

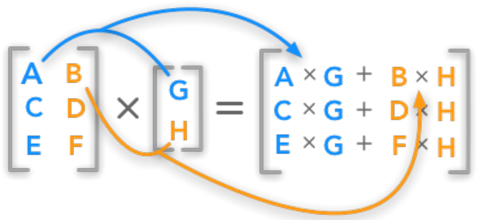
  (i, w1), … , (j, wn)
- **Shuffle and sort**: group by key and generate pairs of the form

  (1, [w1, … , wk]) , … , (n, [wr, … , ws])
- **Reduce**: count the number of words in each list and emit:

  (1, l) , … , (p, m)
- **Output**: (l,n) pairs, where l is a length and n is the total number of words of length l in the input documents

Example: matrix-vector multiplication

- Sparse matrix $A = [a_{ij}]$ size n x n
- Vector $x = [x_j]$ size n x 1
- **Problem**: matrix-vector multiplication $y = Ax$ where $y_i = \Sigma_{j=1\ldots n}\, a_{ij}x_j$
  - Used in many algorithms, e.g., PageRank



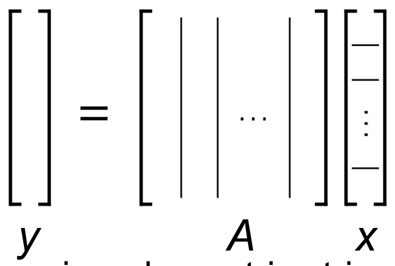
$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \times \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 1\times 2 + 2\times 4 \\ 3\times 2 + 4\times 4 \\ 5\times 2 + 6\times 4 \end{bmatrix} = \begin{bmatrix} 10 \\ 22 \\ 34 \end{bmatrix}$$

Example: matrix-vector multiplication

- To simplify, let us assume that x can fit into the memory of each mapper
- **Map**: apply to $((i, j), a_{ij})$ and produce key-value pair $(i, a_{ij}x_j)$
- **Reduce**: receive $(i, [a_{i1}x_1, \ldots, a_{in}x_n])$ as input and sum all values of the list associated with a given key i, i.e., $y_i = \Sigma_{j=1\ldots n}\, a_{ij}x_j$. The result will be a pair (i, y_i)

Example: matrix-vector multiplication

- What if vector *x* does not fit in mapper's memory?
- *Solution*:
  - Split *x* in horizontal stripes that fit in memory
  - Split *A* accordingly in vertical stripes, stripes of *A* do not need to fit in memory

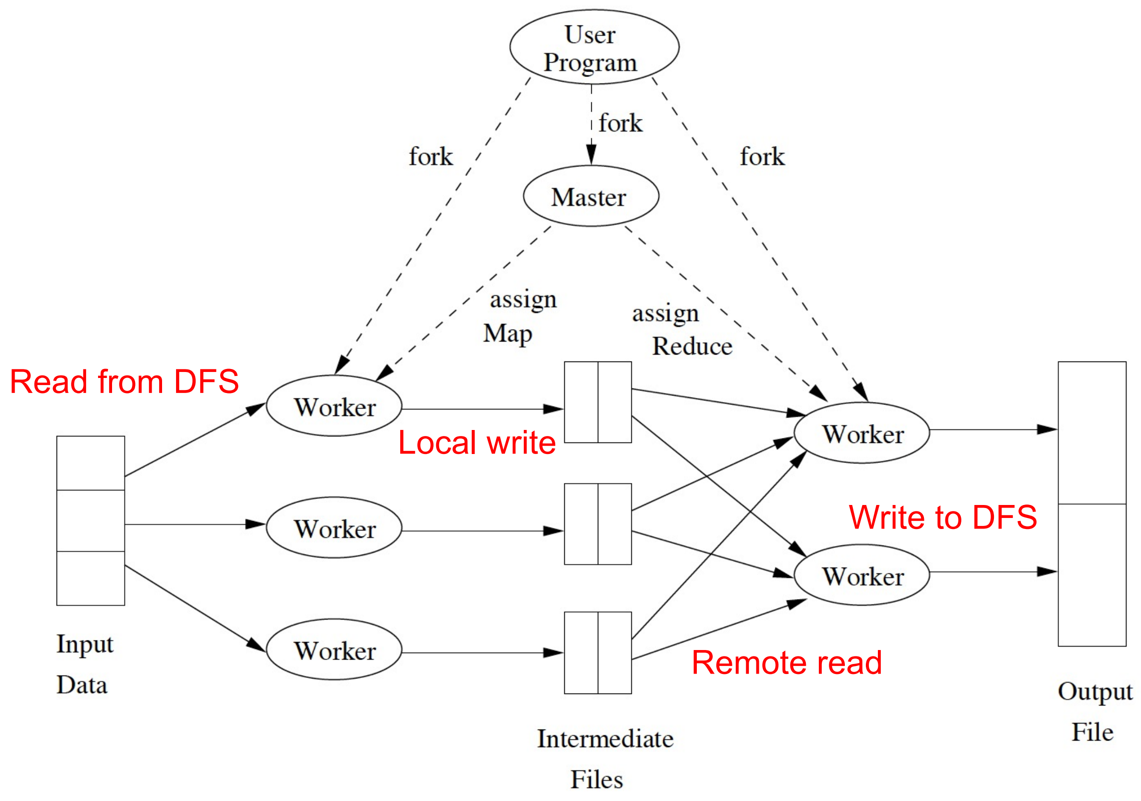


  - Each mapper is assigned a matrix stripe; it also gets the corresponding vector stripe
  - Map and Reduce functions are as before
- More efficient solution by partitioning A into square blocks, rather than stripes

MapReduce: execution overview

- Master-worker architecture
- The master coordinates the map and reduce tasks controlling the flow of the MapReduce job
  - Assigns (i.e., schedules) the job component tasks on the workers, monitoring them and re-executing the failed tasks
- The workers execute the map and reduce tasks

MapReduce: execution overview



Coping with failures

- Compute node hosting the master fails
  - The entire MapReduce job must be restarted
  - The worst scenario
- Worker node hosting a mapper fails
  - All the map tasks that were assigned to this node will have to be redone on another node, even if they had completed, because the disk(s) of the failed node is inaccessible
- Worker node hosting a reducer fails
  - Reschedule reducer on another worker node

Optimization: combining

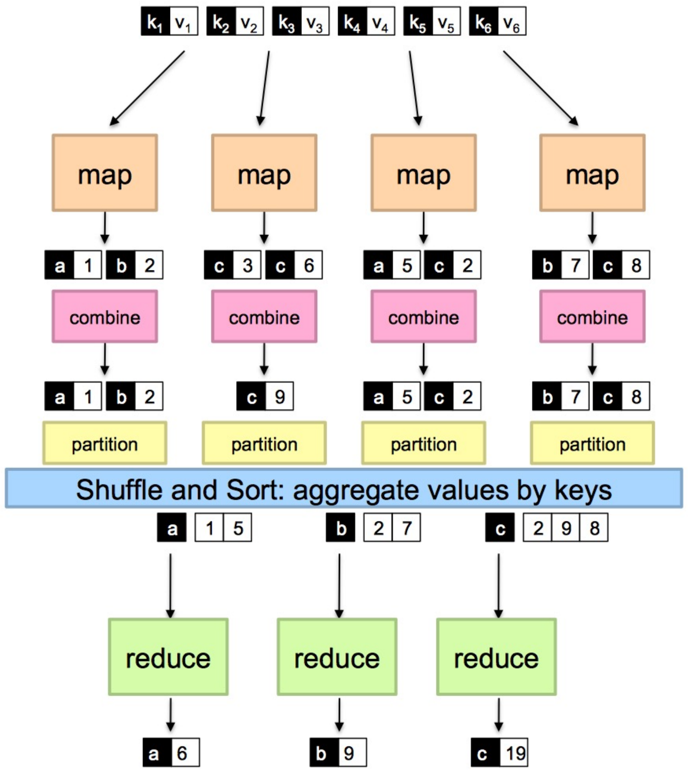
- How to improve performance?
- Run a mini reduce phase on local map output, thus pushing some of what the reducers do to the preceding mappers
- Let's apply a **combiner** to local Map output

  combine $(k_2, [v_2]) \rightarrow [(k_3, v_3)]$
- But Reduce function needs to be associative and commutative
  - Values to be combined can be combined in any order, with the same result
  - E.g., addition in WordCount's Reduce

Optimization: combining

- In many cases the same function can be used for combining as the final reduction
- But shuffle and sort is still necessary!
- Pros:
  - Reduce amount of intermediate data
  - Reduce network traffic

WordCount with combiners



WordCount with combiners

- **Problem**: count the number of occurrences for each word in a large collection of documents
- **Input**: repository of documents, each document is an element
- **Map**: read a document and emit a sequence of key-value pairs where:
  - Keys are words of the documents and values are equal to 1:
    $(w_1, 1), (w_2, 1), \dots, (w_n, 1)$
- **Combiner**: group by key, add up all the values and emit:
  $(w_1, i), \dots, (w_n, j)$
- **Shuffle and sort**: group by key and generate pairs of the form $(w_1, [p, \dots, q]), \dots, (w_n, [r, \dots, s])$
- **Reduce**: add up all the values and emit $(w_1, k), \dots, (w_n, l)$
- **Output**: (w,m) pairs where:
  - w is a word that appears at least once among all the input documents and m is the total number of occurrences of w among all those documents

Shuffle and sort

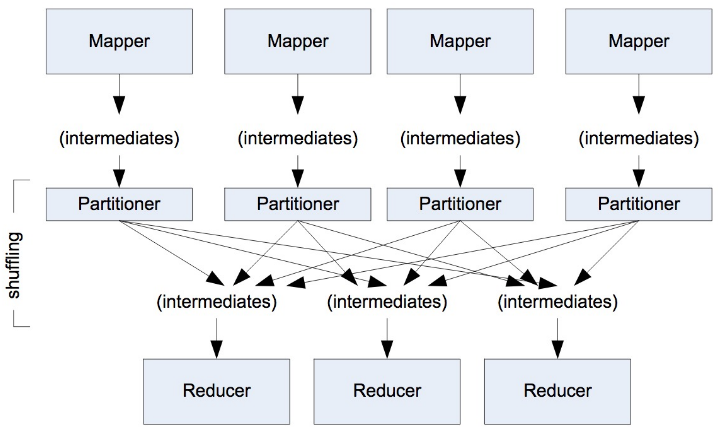
- Between Map(+combine) and Reduce phases
  - Data is shuffled: parallel-sorted and exchanged
  - Data is moved to the correct reducer
- Parallel sort: on mapper side
  - Key-value pairs must be sorted but dataset too large to be sorted on one machine
  - Solution: perform sorting in stages
  - Each mapper partitions its output by reducer, based on the hash of the key, and writes each partition to a sorted file on its local disk

Shuffle and sort

- Move data from mappers to reducers
  - Whenever a mapper finishes writing its sorted output files, it notifies the master; each reducer periodically asks the master until it has retrieved them all
  - Each reducer connects to its necessary mapper and get the files of sorted key-value pairs
- Merge: on reducer side
  - Each reducer merges the files from mappers together, preserving their sort ordering
  - Then it starts reducing on the merged input

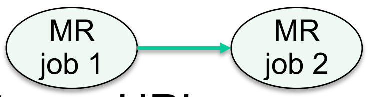
Optimization: partitioning

- How to divide the intermediate key space in a custom way?
- Through a **partitioner**
  - Assigns intermediate key-value pairs to reducers



MapReduce workflows

- Few problems can be solved using a single MapReduce job
- Example: to find the most visited URLs in a logfile we need 2 MapReduce jobs chained together



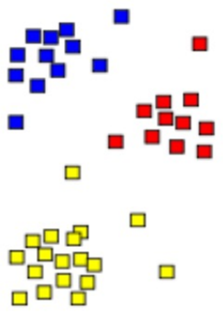
  - 1st job to count the number of visits per URL
    - Like WordCount: mappers emit (URL, 1) key-value pairs; reducers aggregate URL counts
  - 2nd job to sort them
    - Mappers of 2nd job swap keys and values, making counter as key and URL as value
    - Reducers of 2nd job run the identity function, because they get in input URLs already sorted by frequency
    - We still need the reducers, why?

MapReduce workflows

- MapReduce jobs can be chained together into workflows
  - Output of one job becomes input to next job
- Not only a linear chain of jobs but also more complex directed acyclic graph (DAG) of jobs
- Job chaining generates intermediate files on DFS (written to and read from)
  - Performance drops
- Apache Oozie: tool to manage Hadoop jobs as DAGs

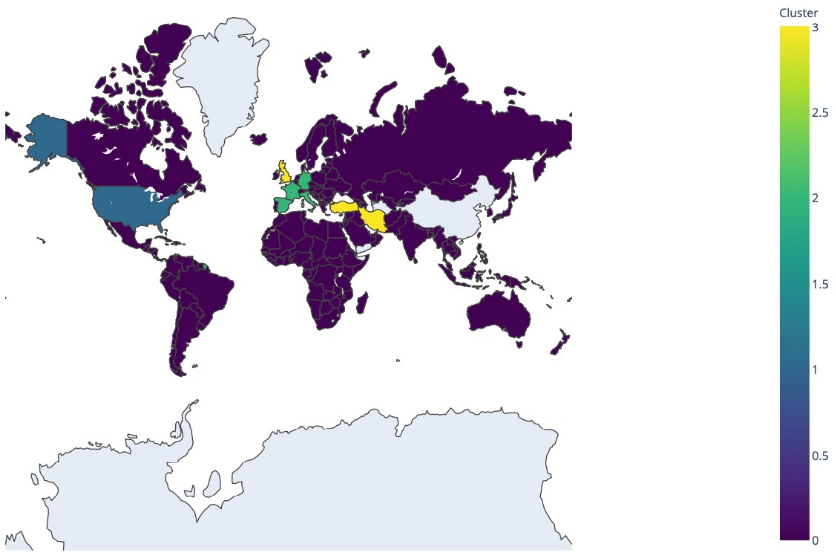
Example: *k*-means in MapReduce

- Clustering is the process of examining a collection of “points,” and grouping the points into “clusters” according to some distance measure
- Examples of cluster analysis
  - Customer segmentation: look for similarity between groups of customers
  - Stock market clustering: group stock based on performances
  - Reduce dimensionality of a dataset by grouping observations with similar values



Clustering: example

- Confirmed cases of Covid-19 (up to April 9, 2020)



Distance between points

- We first need to define distance between two data points
- Most popular is Euclidean distance
  - Distance between points *p* and *q* is given by:

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2}$$

where *n* is the number of independent variables in the space

- Another popular one is Manhattan distance
  - Sum of absolute values instead of squares
  - Based on gridlike street geography of Manhattan

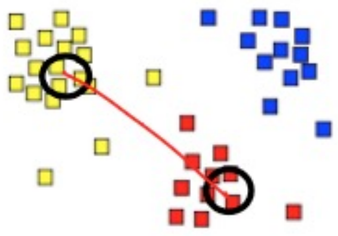
$$d_{Manhattan}(p,q) = \sum_{i=1}^{n} |p_i - q_i|$$

Distance between clusters

- How to define the distance between two clusters?
- Which are the most representative data points for the clusters between which we can calculate the distance?

Distance between clusters

- Centroid distance
  - Distance between centroids of clusters



- Centroid is the point that has the mean position of all data points in each coordinate
  - Example: for points (-1, 10, 3), (0, 5, 2), and (1, 20, 10), the centroid is located at

    ((-1+0+1)/3, (10+5+20)/3, (3+2+10)/3) = (0, 35/3, 5)
- Note: the centroid does not have to be - and rarely is - one of the original data points

k-means clustering

- *k*-means is a well-known clustering algorithm belonging to point assignment class of clustering algorithms
  - Points are considered in some order, and each one is assigned to the cluster into which it best fits
  - *k*-means assumes Euclidian space

k-means clustering

- A variety of heuristic algorithms for *k*-means exist
- We consider Lloyd's algorithm, the first and simplest
  - It tries to minimize the within-cluster sum of squares

```
Specify desired number of clusters k;
Initially choose k data points that are likely to be in different clusters;
Make these data points the centroids of their clusters;
Repeat
    For each remaining data point p do
        Find the centroid to which p is nearest;
        Add p to the cluster of that centroid;
    Re-compute cluster centroids;
Until no improvement is made;
```

MapReducing 1 iteration of *k*-means

- **Classify**: assign each point to nearest cluster centroid

$$z_i \leftarrow \arg\min_j \|\boldsymbol{\mu}_j - \boldsymbol{x_i}\|_2^2$$

$\boldsymbol{x_i}$: data point
$\boldsymbol{\mu}_j$: centroid for cluster j
z_i: cluster i label

- **Re-center**: update cluster centroids as mean of assigned points

$$\boldsymbol{\mu}_j = \frac{1}{n_j} \sum_{i:z_i=j} \boldsymbol{x_i}$$

n_j: number of points in cluster j

MapReducing 1 iteration of *k*-means

- **Classify**: assign each point to nearest cluster centroid

$$z_i \leftarrow \arg\min_j \|\boldsymbol{\mu}_j - \boldsymbol{x_i}\|_2^2$$

Map: given ({$\boldsymbol{\mu}_j$}, $\boldsymbol{x}_i$), for each point $\boldsymbol{x}_i$ emit (z_i, $\boldsymbol{x}_i$)

Parallel over data points

- **Re-center**: update cluster centroids as mean of assigned points

$$\boldsymbol{\mu}_j = \frac{1}{n_j} \sum_{i:z_i=j} \boldsymbol{x_i}$$

Reduce: average over all points in cluster j (z_i=j)

Parallel over cluster centroids

Classification step as Map

- Classify: assign each point to nearest cluster centroid

$$z_i \leftarrow \arg\min_j \|\boldsymbol{\mu}_j - \boldsymbol{x}_i\|_2^2$$

```
map([$\mu_1$, $\mu_2$, …, $\mu_k$], $x_i$)
        $z_i \leftarrow \arg\min_j \|\boldsymbol{\mu}_j - \boldsymbol{x}_i\|_2^2$
        emit ($z_i$, $x_i$)
```

Map on data point and cluster centroids

Emit z_i (the cluster label) as key and data point x_i as value

Re-center step as Reduce

- Re-center: update cluster centroids as mean of assigned points

$$\boldsymbol{\mu}_j = \frac{1}{n_j} \sum_{i:z_i=j} \boldsymbol{x}_i$$

```
reduce(j, x_in_clusterj: [$x_i$, …])
        sum = 0
        count = 0
        for x in x_in_clusterj
                sum += x
                count += 1
        emit (j, sum/count)
```

Reduce on data points assigned to cluster j (having the cluster label j as key)

Emit cluster label j as key and new centroid for cluster j as value

Multiple iterations for *k*-means

- *k*-means is an iterative algorithm: needs an iterative version of MapReduce
- Our implementation so far: each mapper gets a data point and all cluster centroids
  - ✗ Too many mappers!
- Better implementation: each mapper gets many data points
  - Anyway, at each iteration we must broadcast the new centroids across the MapReduce cluster and repeat multiple phases of Map and Reduce until convergence (or maximum number of steps)
- Any other optimization?

Optimizing *k*-means for MapReduce

- Combiners can be used to optimize the distributed algorithm
  - Compute for each centroid local sums of points
  - Send to reducer: <centroid, partial sums>
- Can use a single reducer
  - ✓ Data to reducers is small
  - ✓ Single reducer can tell immediately if the computation has converged
  - ✓ Creation of a single output file

Example: PageRank in MapReduce

- Various methods to compute PageRank, but we are interested in MapReduce
- Let’s first consider the basic update rule

$$R_{i+1}(u) = \sum_{v \in B_u} \frac{R_i(v)}{N_v}$$

- Idea:
  – Start by initializing each node with same PageRank
  – "Pass around" PageRank fluid from nodes to their out-neighbours

PageRank: Map

```
class Mapper
    method Map(nid $n$, node $N$)
        $p \leftarrow N.\text{PageRank}/|N.\text{AdjacencyList}|$
        Emit(nid $n, N$)                          ▷ Pass along graph structure
        for all nodeid $m \in N.\text{AdjacencyList}$ do
            Emit(nid $m, p$)                      ▷ Pass PageRank mass to neighbors
```

- Represent graph as adjacency list
  – Adjacency matrix may be very sparse, so better to use adjacency list to represent only nonzero elements
- Evenly divide up PageRank of node with id *n* and pass each piece *p* along its out-links, keyed by outbound node (node with id *m*)
- Graph structure itself (node *N*) must be passed from iteration to iteration

PageRank: Reduce

```
class REDUCER
    method REDUCE(nid $m$, $[p_1, p_2, \ldots]$)
        $M \leftarrow \emptyset$
        for all $p \in$ counts $[p_1, p_2, \ldots]$ do
            if ISNODE($p$) then
                $M \leftarrow p$                    ▷ Recover graph structure
            else
                $s \leftarrow s + p$        ▷ Sum incoming PageRank contributions
        $M$.PAGERANK $\leftarrow s$
        EMIT(nid $m$, node $M$)
```

- PageRank contributions from in-links are summed up at each destination node *m*

MapReduce iteratively

- Full algorithm is iterative
  - Each MapReduce job corresponds to one iteration of the algorithm
- Initialize nodes' PageRank to uniform distribution
- Run Map and Reduce tasks iteratively
- Until convergence (no change)

PageRank: a little more complex

- Model assumes that surfer does not always follow a link, but sometimes randomly picks links to follow, i.e., jumps to a another page
- Scaled PageRank

$$R_{i+1}(u) = c \sum_{v \in B_u} \frac{R_i(v)}{N_v} + (1-c)E(u)$$

Still an issue to solve: sinks

- Sink: node having no out-links to other nodes
- Sinks can create problems
  - Total PageRank mass will not be conserved since in Map no key-value pairs will be emitted when a sink is encountered
- Solution: if any PageRank is lost due to sinks, redistribute that PageRank uniformly throughout the graph for next iteration

PageRank in MapReduce

- Basic version of PageRank in MapReduce has two issues to solve
  1. Sinks
  2. Random jump factor

- In MapReduce algorithm

1. Keep track of any lost PageRank
   - E.g., by using a special reserved intermediate key or using a counter to keep track of sink's PageRank value
2. Take into account that surfer can jump to a random page

PageRank in MapReduce

- Basic version of PageRank in MapReduce has two issues to solve
  1. Sinks
  2. Random jump factor

- Solution: after basic Map and Reduce, do a cleanup pass
- Deal both with missing mass due to sinks and random jump factor

Cleanup pass

- Adjust the PageRank *p* of each node to be

$$p' = c\left(p + \frac{m}{N}\right) + (1-c)\frac{1}{N}$$

  - *p:* current PageRank
  - *m*: mass lost due to sinks
  - *N*: number of nodes in the graph
  - Assume *E*(*u*) is uniform distribution (i.e., 1/*N*)
- This can be done using a Map job (no Reduce)
- So one iteration of PageRank requires two MapReduce jobs
  - The first to distribute PageRank along links
  - The second to take care of sinks and random jump factor

Full PageRank in MapReduce

- Initialize nodes’ PageRank with uniform distribution
- Run iteratively the two MapReduce jobs described before
- Until convergence (no change)

Apache Hadoop

What is Apache Hadoop?

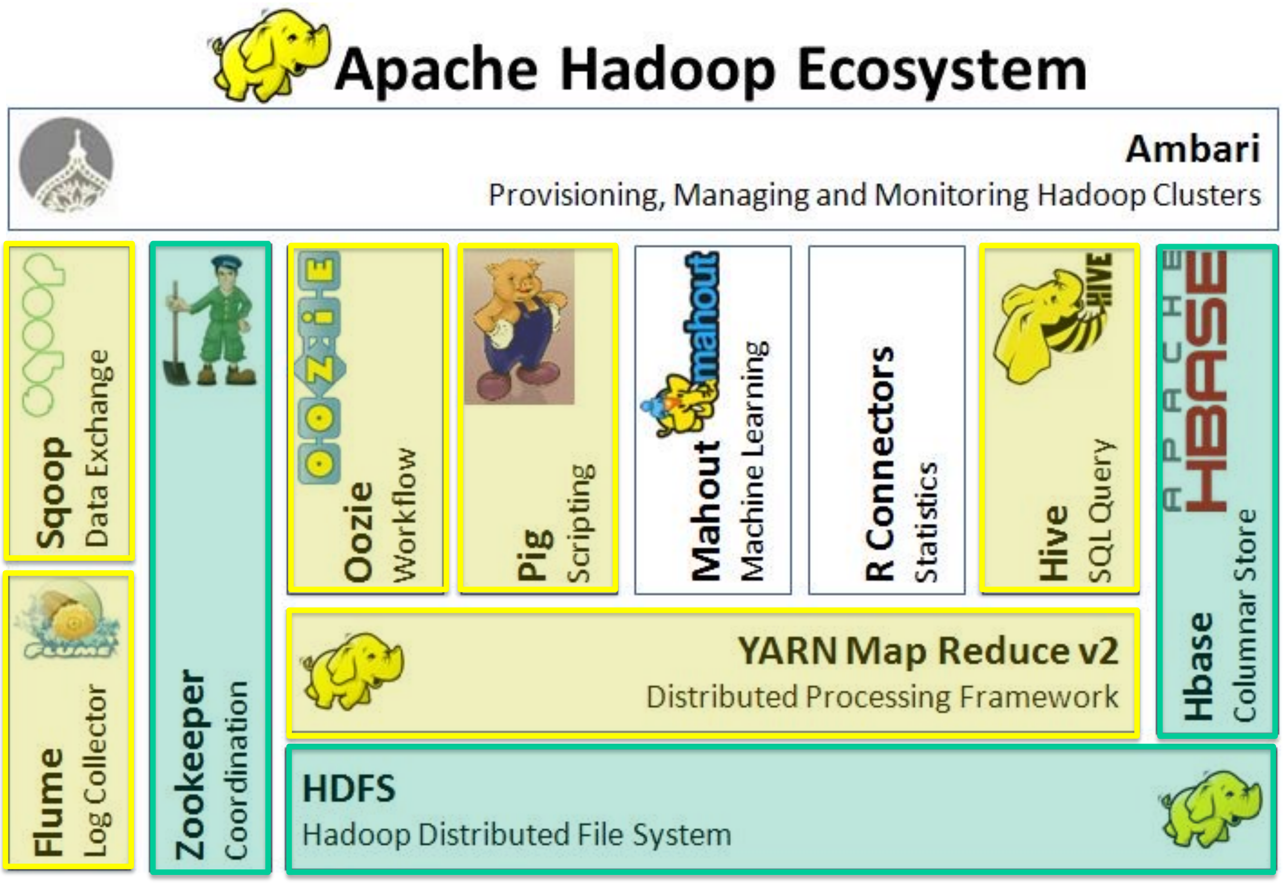
- Open-source software framework for reliable, scalable, distributed data-intensive computing
  - Originally developed by Yahoo!
- Goal: storage and processing of data-sets at massive scale
- Infrastructure: cluster of commodity hardware
- Core components:
  - **HDFS**: Hadoop Distributed File System
  - **Hadoop YARN**
  - **Hadoop MapReduce**
- Plus many related projects
  - Apache Pig, Apache Hive, Apache Hbase, …

Hadoop runs on clusters

- Compute nodes are stored on racks
  - 8-64 compute nodes on a rack
- Cluster composed by many racks of compute nodes
- How to assign tasks to compute nodes?
  - Take into account interconnection topology
    - Nodes on same rack are typically connected by 10 Gbit/s Ethernet
    - Racks are interconnected by another level of network or a switch
    - Bandwidth of intra-rack communication is greater than that of inter-rack communication
- How to deal with failures of compute nodes?
  - Files are stored redundantly
  - Computation is divided into tasks

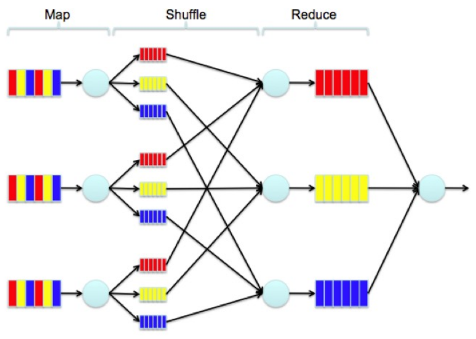
Hadoop core

- **HDFS** ✓
  - Distributed file system
  - Data is replicated across the cluster
  - Fault-tolerant
- **Hadoop YARN**
  - Cluster resource management
- **Hadoop MapReduce**
  - Distributed framework to run applications which process large data sets in parallel on large clusters (> 1000 nodes) of commodity hardware in a reliable, fault-tolerant manner



Hadoop core: MapReduce

- We have already examined the MapReduce programming paradigm

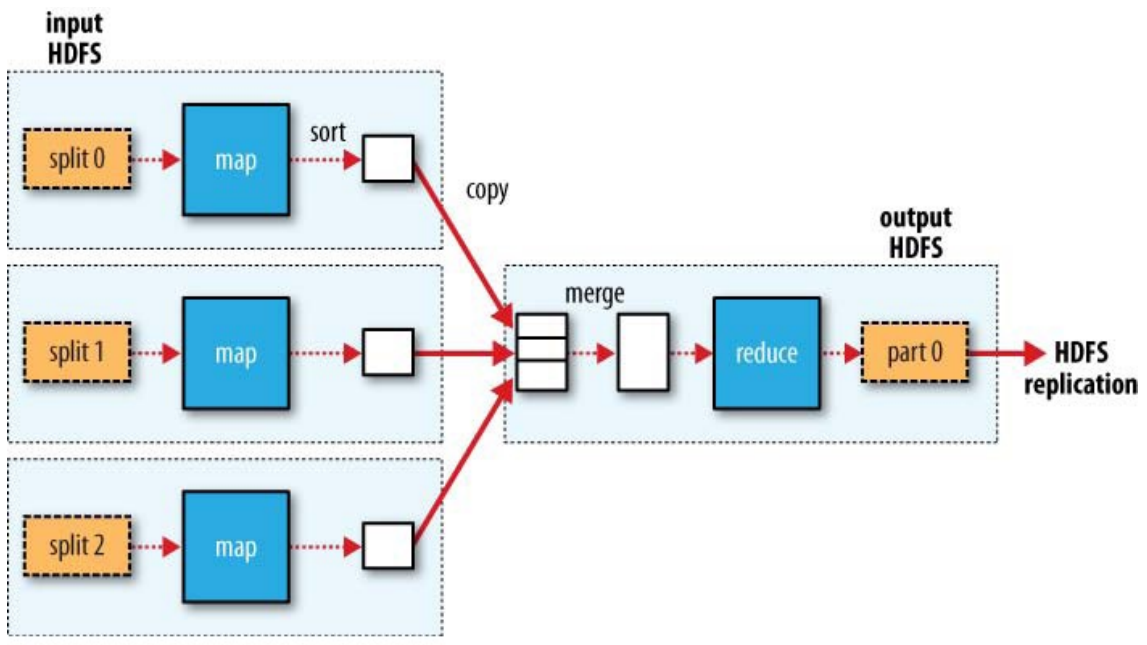


- Also used in other MPP environments and NoSQL databases (e.g., Vertica and MongoDB)

Basic flow of how to use Hadoop

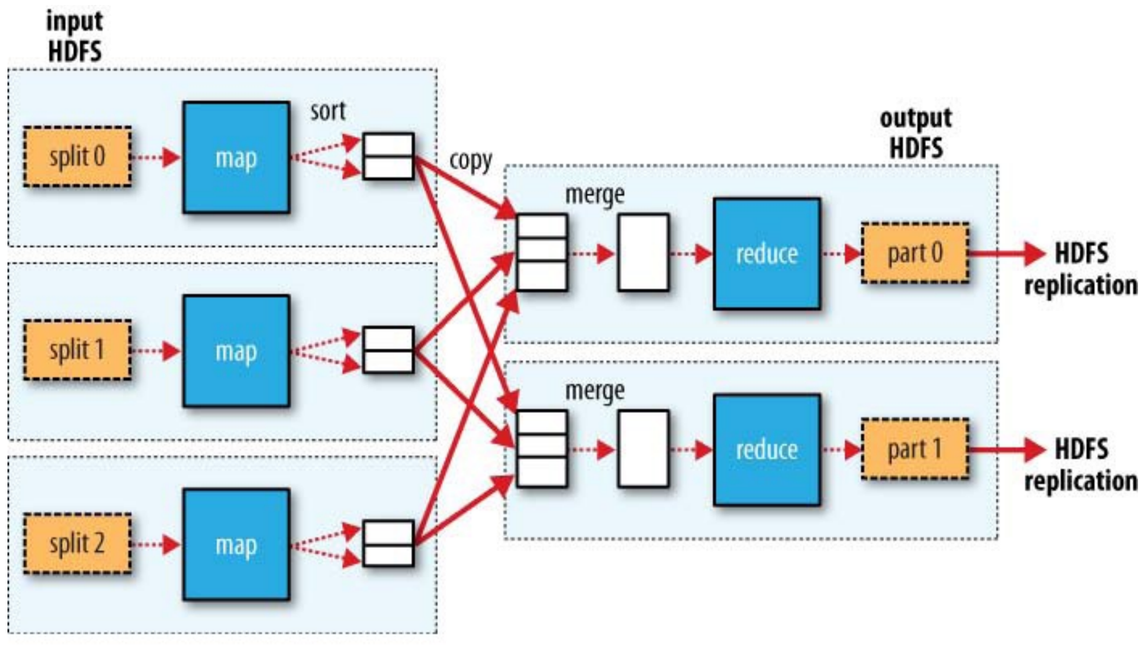
- Load data in HDFS
- Use MapReduce to analyze data
- Store results in HDFS
- Read results from HDFS

MapReduce data flow: single Reduce task

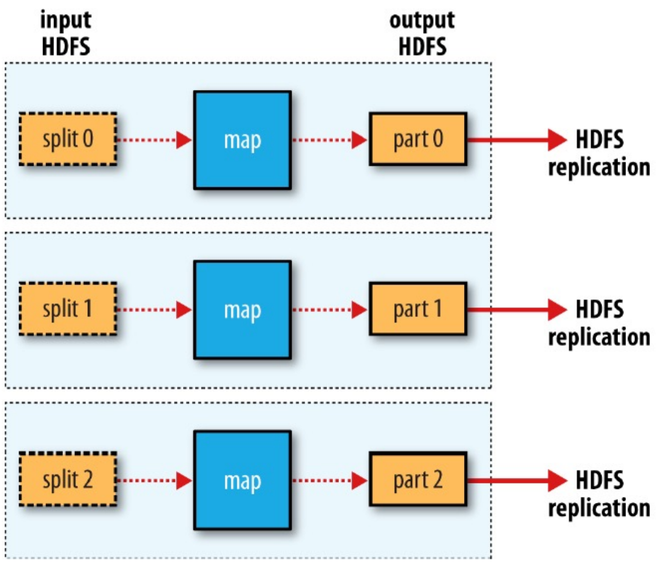


MapReduce data flow: multiple Reduce tasks

- When there are multiple reducers, map tasks partition their output



MapReduce data flow: no Reduce task



Optional Combiner

- Many MapReduce jobs are limited by cluster bandwidth
  - How to minimize amount of data transferred between map and reduce tasks?
- Use Combine task
  - Combiner: optional localized reducer that applies a user-provided method to combine mapper output
  - Performs data aggregation on intermediate data of the same key for the Map task's output before transmitting the result to the Reduce task
    - Takes each key-value pair from the Map task, processes it, and produces the output as key-value collection pairs
- Reduce task is still needed to process records with the same key from different Map tasks

Optional Partitioner

- When there are multiple reducers, map tasks partition their output
  - One partition for each Reduce task
- Goal: to determine which reducer will receive which intermediate keys and values
  - Records for any given key are all in a single partition
- Default partitioner uses a hash function on the key to determine which bucket (i.e., reducer)
- Partitioning can be also controlled by a user-defined function
  - Requires to implement a custom partitioner

Programming languages for Hadoop

- Default programming language: Java
- Java program with at least 3 parts:
  1. A Main method which configures the job, and launches it
     - Set number of reducers
     - Set mapper and reducer classes
     - Set optional partitioner
     - Set other Hadoop configurations
  2. A Mapper class
     - Takes (k,v) inputs, writes (k,v) outputs
  3. A Reducer class
     - Takes k, Iterator[v] inputs, and writes (k,v) outputs

WordCount in Java

- Let's analyze WordCount code in Java
  http://bit.ly/1m9xHym

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

WordCount in Java

map method processes one line at a time, splits the line into tokens separated by white spaces, via StringTokenizer, and emits a key-value pair <word, 1>

```
public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }
```

WordCount in Java

reduce method sums up the values, which are the occurrence counts for each key

```
public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }
```

WordCount in Java

main method specifies various facets of the job

```
public static void main(String[] args) throws Exception {
   Configuration conf = new Configuration();
   Job job = Job.getInstance(conf, "word count");
   job.setJarByClass(WordCount.class);
   job.setMapperClass(TokenizerMapper.class);
   job.setCombinerClass(IntSumReducer.class);
   job.setReducerClass(IntSumReducer.class);
   job.setOutputKeyClass(Text.class);
   job.setOutputValueClass(IntWritable.class);
   FileInputFormat.addInputPath(job, new Path(args[0]));
   FileOutputFormat.setOutputPath(job, new Path(args[1]));
   System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
}
```

Output of each map is passed through local combiner (same as Reducer) for local aggregation, after being sorted on keys

WordCount in Java: example

- Input files

```
Hello World Bye World
```

```
Hello Hadoop Goodbye Hadoop
```

- Output file

```
Bye 1
Goodbye 1
Hadoop 2
Hello 2
World 2
```

WordCount in Java: example

- 1st mapper output

```
< Hello, 1>
< World, 1>
< Bye, 1>
< World, 1>
```

- After local combiner

```
< Bye, 1>
< Hello, 1>
< World, 2>
```

WordCount in Java: example

- 2nd mapper output

```
< Hello, 1>
< Hadoop, 1>
< Goodbye, 1>
< Hadoop, 1>
```

- After local combiner

```
< Goodbye, 1>
< Hadoop, 2>
< Hello, 1>
```

- Reducer output

```
< Bye, 1>
< Goodbye, 1>
< Hadoop, 2>
< Hello, 2>
< World, 2>
```

WordCount on Hadoop

- Let’s analyze WordCount code
- Code in Java http://bit.ly/1m9xHym

- Same example in Python
  - Use Hadoop Streaming for passing data between Map and Reduce code via stdin and stdout
  - See Michael Noll’s tutorial http://bit.ly/19YutB1

Other programming languages

- Use Hadoop Streaming to write Map and Reduce functions in programming languages other than Java https://hadoop.apache.org/docs/r1.2.1/streaming.html
  - Uses Unix standard streams as interface between the mapper/reducer and MapReduce framework
- Allows to use any language (e.g., Python) that can read standard input (*stdin*) and write to standard output (*stdout*) to write mapper and reducer tasks
  - See example in Python: use Hadoop Streaming for passing data between Map and Reduce code via stdin and stdout
  - Observe that reducer interface is different from Java: instead of receiving reduce(k, Iterator[v]), your script is sent one line per value, including the key
  - As alternative, in Python you can use mrjob https://mrjob.readthedocs.io/

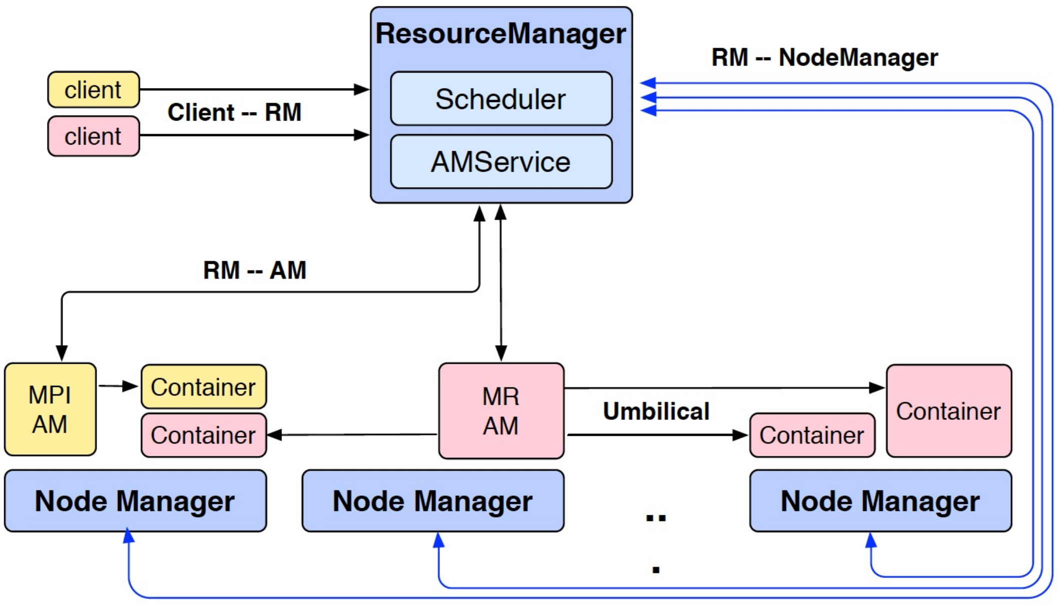
YARN

- **YARN**: Yet Another Resource Negotiator
  - Framework for job scheduling and cluster resource management
- Turns out Hadoop into an analytics platform in which resource management functions are separated from programming model
  - Can support not only MapReduce, but also other frameworks (e.g., Spark)

We will study it later

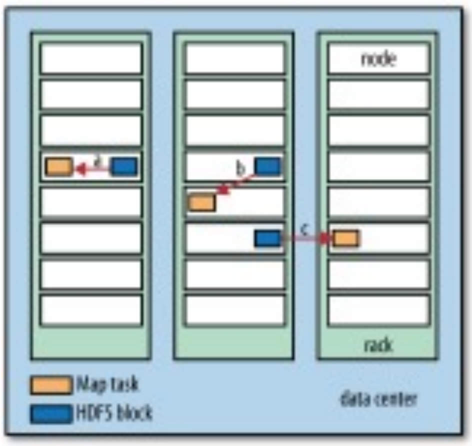
YARN: architecture

- Global ResourceManager (RM)
- A set of per-application ApplicationMasters (AMs)
- A set of NodeManagers (NMs)



YARN: data locality optimization

- Scheduling the job, YARN tries to run the mapper on a data-local node (*data locality* optimization)
  - So to not use cluster bandwidth
  - Otherwise rack-local
  - Off-rack as last choice



Hadoop installation

- Some straightforward alternatives:

1. Our Docker environment
2. Single-node installation https://bit.ly/3c92ZPZ
3. Cloudera CDH and MapR distributions
   - Integrated Hadoop-based stack containing all the components needed for production, tested and packaged to work together
   - Include at least Hadoop, HDFS (not in MapR), HBase, Hive, Pig, Sqoop, Flume, Kafka, Spark
   - Include also security frameworks (e.g., Sentri) to handle access control, security policies, …
   - MapR distribution: own distributed Posix file system (MapR-FS) rather than HDFS

Hadoop configuration

- Tuning Hadoop clusters for good performance is somehow magic
  - Disk I/O is usually the performance bottleneck
- Tuning hw and sw parameters, e.g.:
  - Find optimal number of disks so to maximize I/O bandwidth
  - Increase open file limit
  - Find optimal HDFS block size (number of mappers)
  - JVM settings for Java heap usage and garbage collection
- There are also Hadoop-specific parameters that can be tuned for performance
  - Number of mappers
  - Number of reducers
  - Plus other map-side and reduce-side tuning parameters

How many mappers

- Number of mappers
  - Driven by number of blocks in input files
  - You can adjust the HDFS block size to adjust the number of mappers
  - Good level of parallelism: 10-100 mappers per-node, but up to 300 mappers for very CPU-light map tasks
  - Task setup takes a while, so it is best if mappers take at least a minute to execute

How many reducers

- Number of reducers
  - Can be user-defined (default is 1)
  - Use Job.setNumReduceTasks(int)
  - The right number of reduces seems to be 0.95 or 1.75 multiplied by (<*no. of nodes*> * <*no. of maximum containers per node*>)
    - 0.95: all of the reduces can launch immediately and start transferring map outputs as the maps finish
    - 1.75: the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing
  - Can be set to *zero* if no reduction is desired
    - No sorting of map-outputs before writing them to output file

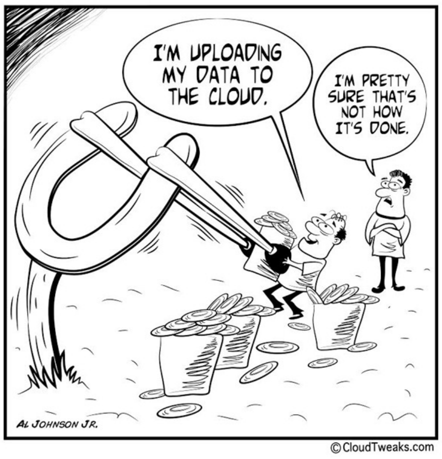
See http://bit.ly/2oK0D5A

Some tips for performance

- To handle massive I/O and save bandwidth
  - Compress input data, map output data, reduce output data
- To address massive I/O in partition and sort phases
  - Each mapper has a circular buffer memory to which it writes the output; when the buffer is full, its content is written ("spilled") to disk. Avoid that records are spilled more than once
  - How? Adjust spill records and sorting buffer
- To address massive network traffic caused by large map output
  - Compress map output
  - Implement a combiner to reduce the amount of data passing through the shuffle

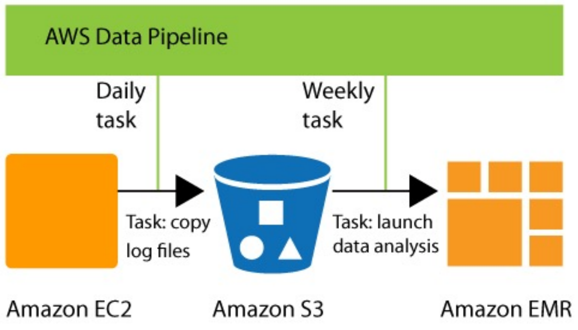
Hadoop in the Cloud

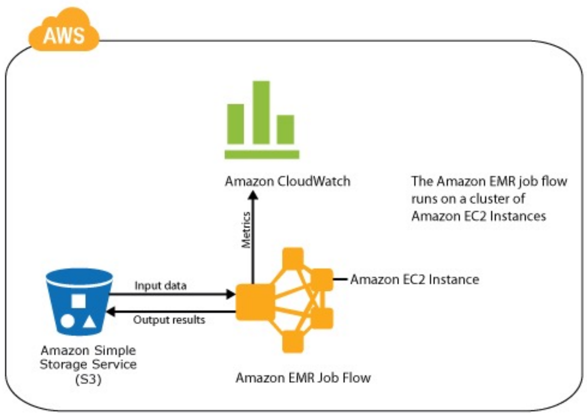
- Pros:
  - Gain Cloud scalability and elasticity
  - Do not need to manage and provision the infrastructure and the platform
- Main challenges:
  - Move data to the Cloud
    - Latency is not zero (because of speed of light)!
    - Minor issue: network bandwidth
  - Data security and privacy


I'M UPLOADING MY DATA TO THE CLOUD.
I'M PRETTY SURE THAT'S NOT HOW IT'S DONE.
AL JOHNSON JR.
©CloudTweaks.com

Amazon Elastic MapReduce (EMR)

- Distribute the computational work across a cluster of virtual servers running in AWS cloud (EC2 instances)
- Not only Hadoop: also Hive, Pig, Hbase, Spark, Flink and Presto
  - Usually not the latest released version
- Input and output: Amazon S3, HDFS, DynamoDB …
- Access through AWS Console, command line




AWS
Amazon CloudWatch
The Amazon EMR job flow runs on a cluster of Amazon EC2 Instances
Metrics
Input data
Output results
Amazon EC2 Instance
Amazon Simple Storage Service (S3)
Amazon EMR Job Flow

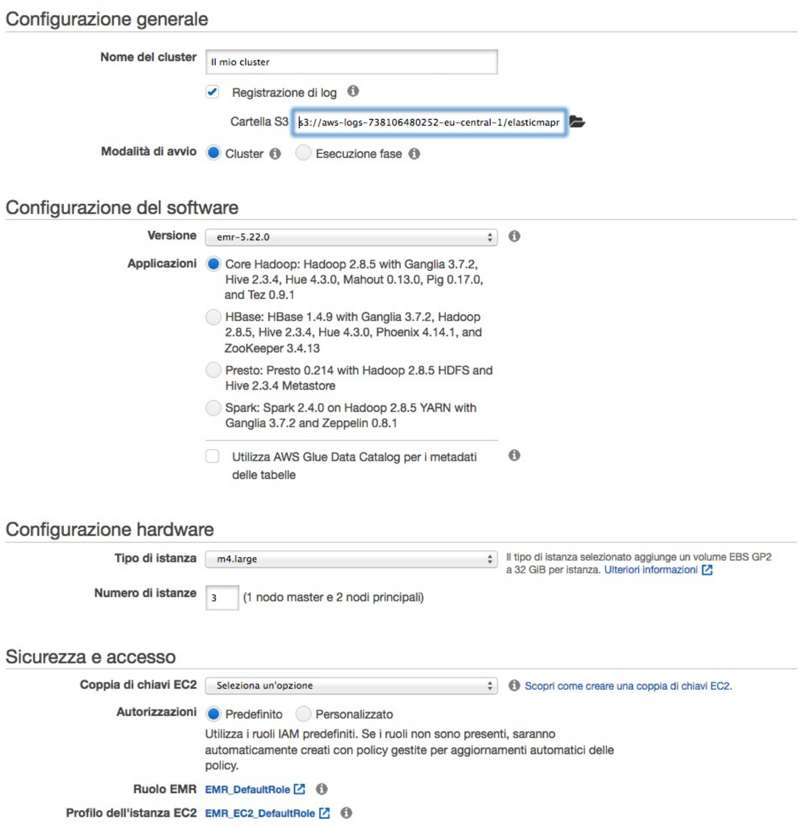
Create EMR cluster

Steps:

1. Provide cluster name
2. Select EMR release and applications to install
3. Select instance type (including spot instances)
4. Select number of instances
5. Select EC2 key-pair to connect securely to the cluster

See http://amzn.to/2nnujp5

Running cluster can be automatically scaled or manually resized



Configurazione generale

Nome del cluster: Il mio cluster

[✓] Registrazione di log

Cartella S3: s3://aws-logs-738106480252-eu-central-1/elasticmapr

Modalità di avvio: [✓ Cluster] Esecuzione fase

Configurazione del software

Versione: emr-5.22.0

Applicazioni:
- [✓ Core Hadoop: Hadoop 2.8.5 with Ganglia 3.7.2, Hive 2.3.4, Hue 4.3.0, Mahout 0.13.0, Pig 0.17.0, and Tez 0.9.1]
- HBase: HBase 1.4.9 with Ganglia 3.7.2, Hadoop 2.8.5, Hive 2.3.4, Hue 4.3.0, Phoenix 4.14.1, and ZooKeeper 3.4.13
- Presto: Presto 0.214 with Hadoop 2.8.5 HDFS and Hive 2.3.4 Metastore
- Spark: Spark 2.4.0 on Hadoop 2.8.5 YARN with Ganglia 3.7.2 and Zeppelin 0.8.1

[ ] Utilizza AWS Glue Data Catalog per i metadati delle tabelle

Configurazione hardware

Tipo di istanza: m4.large — Il tipo di istanza selezionato aggiunge un volume EBS GP2 a 32 GiB per istanza. Ulteriori informazioni

Numero di istanze: 3 (1 nodo master e 2 nodi principali)

Sicurezza e accesso

Coppia di chiavi EC2: Seleziona un'opzione — Scopri come creare una coppia di chiavi EC2.

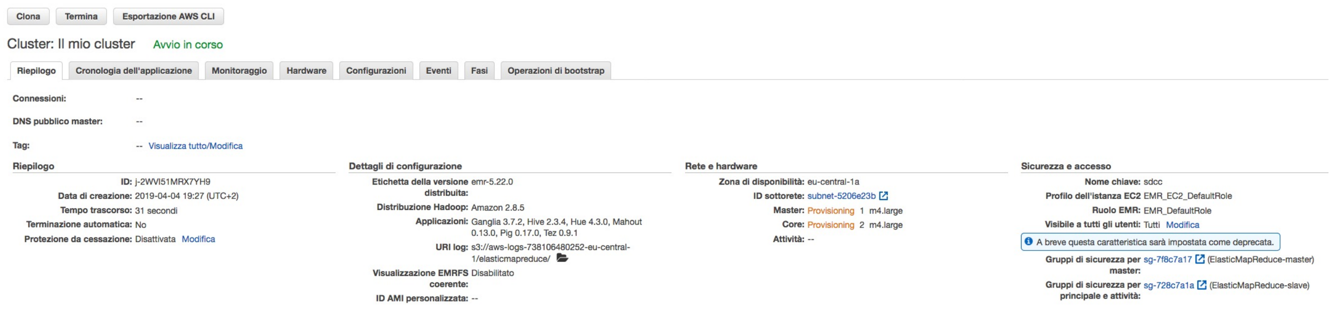
Autorizzazioni: [✓ Predefinito] Personalizzato

Utilizza i ruoli IAM predefiniti. Se i ruoli non sono presenti, saranno automaticamente creati con policy gestite per aggiornamenti automatici delle policy.

Ruolo EMR: EMR_DefaultRole

Profilo dell'istanza EC2: EMR_EC2_DefaultRole

EMR cluster details



Cluster: Il mio cluster Avvio in corso

Connessioni: --

DNS pubblico master: --

Tag: -- Visualizza tutto/Modifica

Riepilogo

ID: j-2WVI51MRX7YH9
Data di creazione: 2019-04-04 19:27 (UTC+2)
Tempo trascorso: 31 secondi
Terminazione automatica: No
Protezione da cessazione: Disattivata Modifica

Dettagli di configurazione

Etichetta della versione distribuita: emr-5.22.0
Distribuzione Hadoop: Amazon 2.8.5
Applicazioni: Ganglia 3.7.2, Hive 2.3.4, Hue 4.3.0, Mahout 0.13.0, Pig 0.17.0, Tez 0.9.1
URI log: s3://aws-logs-738106480252-eu-central-1/elasticmapreduce/
Visualizzazione EMRFS coerente: Disabilitato
ID AMI personalizzata: --

Rete e hardware

Zona di disponibilità: eu-central-1a
ID sottorete: subnet-5206e23b
Master: Provisioning 1 m4.large
Core: Provisioning 2 m4.large
Attività: --

Sicurezza e accesso

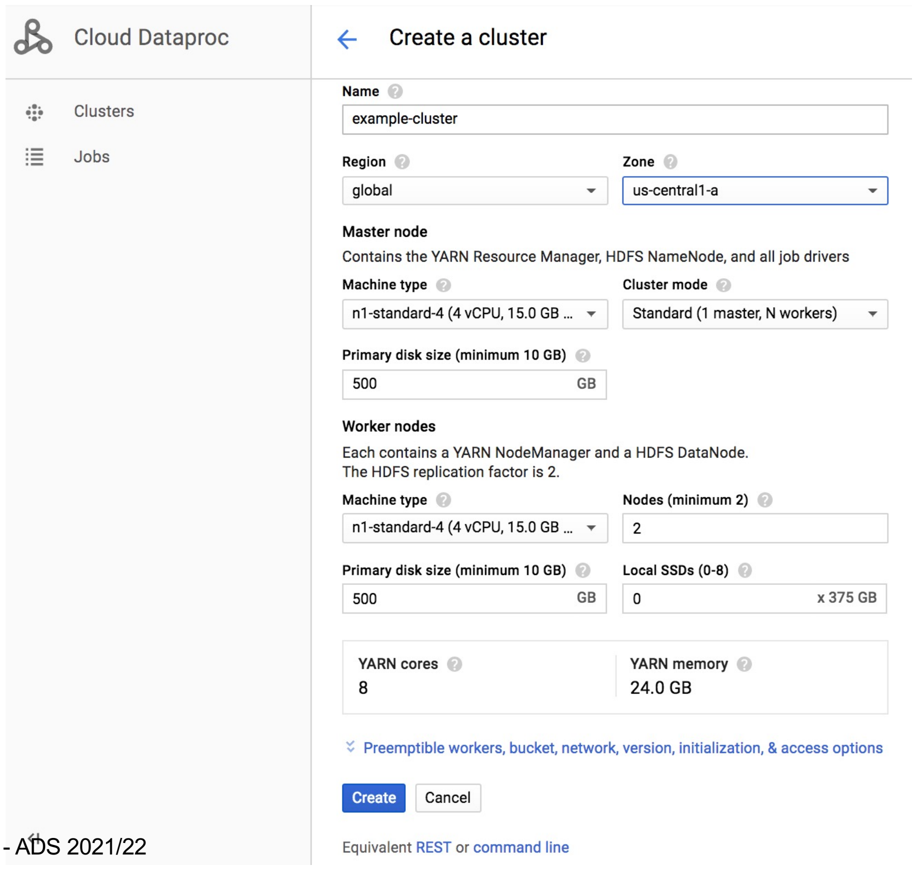
Nome chiave: sdcc
Profilo dell'istanza EC2: EMR_EC2_DefaultRole
Ruolo EMR: EMR_DefaultRole
Visibile a tutti gli utenti: Tutti Modifica
A breve questa caratteristica sarà impostata come deprecata.
Gruppi di sicurezza per master: sg-7f8c7a17 (ElasticMapReduce-master)
Gruppi di sicurezza per principale e attività: sg-728c7a1a (ElasticMapReduce-slave)

- You can still tune some parameters for performance
  - Some EC2 parameters (heap size used by Hadoop and Yarn)
  - Hadoop parameters (e.g., memory for map and reduce JVMs)

Google Cloud Dataproc

- Distribute the computational work across a cluster of virtual servers running in Google Cloud Platform
- Not only Hadoop (plus Hive and Pig), also Spark
- Input and output from other Google services: Cloud Storage, Bigtable
- Access through REST API, Cloud SDK, and Cloud Dataproc UI
- Fine-grain pay-per-use (seconds)

Create Cloud Datproc cluster



Create a cluster

Name: example-cluster

Region: global

Zone: us-central1-a

Master node
Contains the YARN Resource Manager, HDFS NameNode, and all job drivers

Machine type: n1-standard-4 (4 vCPU, 15.0 GB ...

Cluster mode: Standard (1 master, N workers)

Primary disk size (minimum 10 GB): 500 GB

Worker nodes
Each contains a YARN NodeManager and a HDFS DataNode.
The HDFS replication factor is 2.

Machine type: n1-standard-4 (4 vCPU, 15.0 GB ...

Nodes (minimum 2): 2

Primary disk size (minimum 10 GB): 500 GB

Local SSDs (0-8): 0 x 375 GB

YARN cores: 8

YARN memory: 24.0 GB

Preemptible workers, bucket, network, version, initialization, & access options

Create Cancel

Equivalent REST or command line

References

- Dean and Ghemawat, MapReduce: simplified data processing on large clusters, *OSDI 2004.*
- Leskovec, Rajaraman, and Ullman, Mining of Massive Datasets, 3rd edition, chapter 2, 2020.
  - See also chapter 7 on clustering and chapter 5 on PageRank
- White, “Hadoop: The Definitive Guide, 4th edition”, O’Reilly, 2015.
- Miner and Shook, “MapReduce Design Patterns”, O’Reilly, 2012.