Master Executive di II Livello

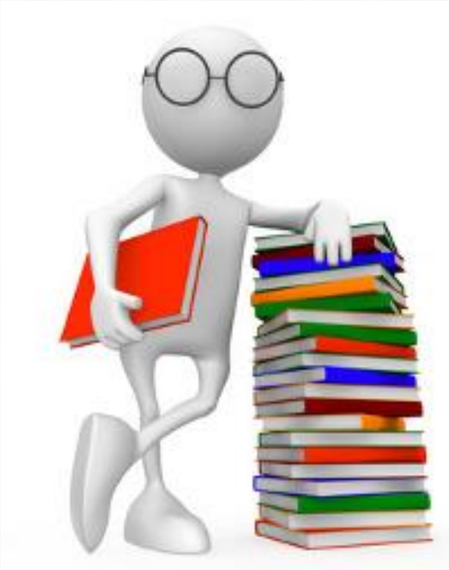
BIG DATA ANALYSIS AND BUSINESS INTELLIGENCE

Vamsi Krishna Varma Gunturi
Data science intern at ISTAT
vamsivarmagunturi@gmail.com

Introduction to MapReduce

fondazione INUIT TORVERGATA

Topics

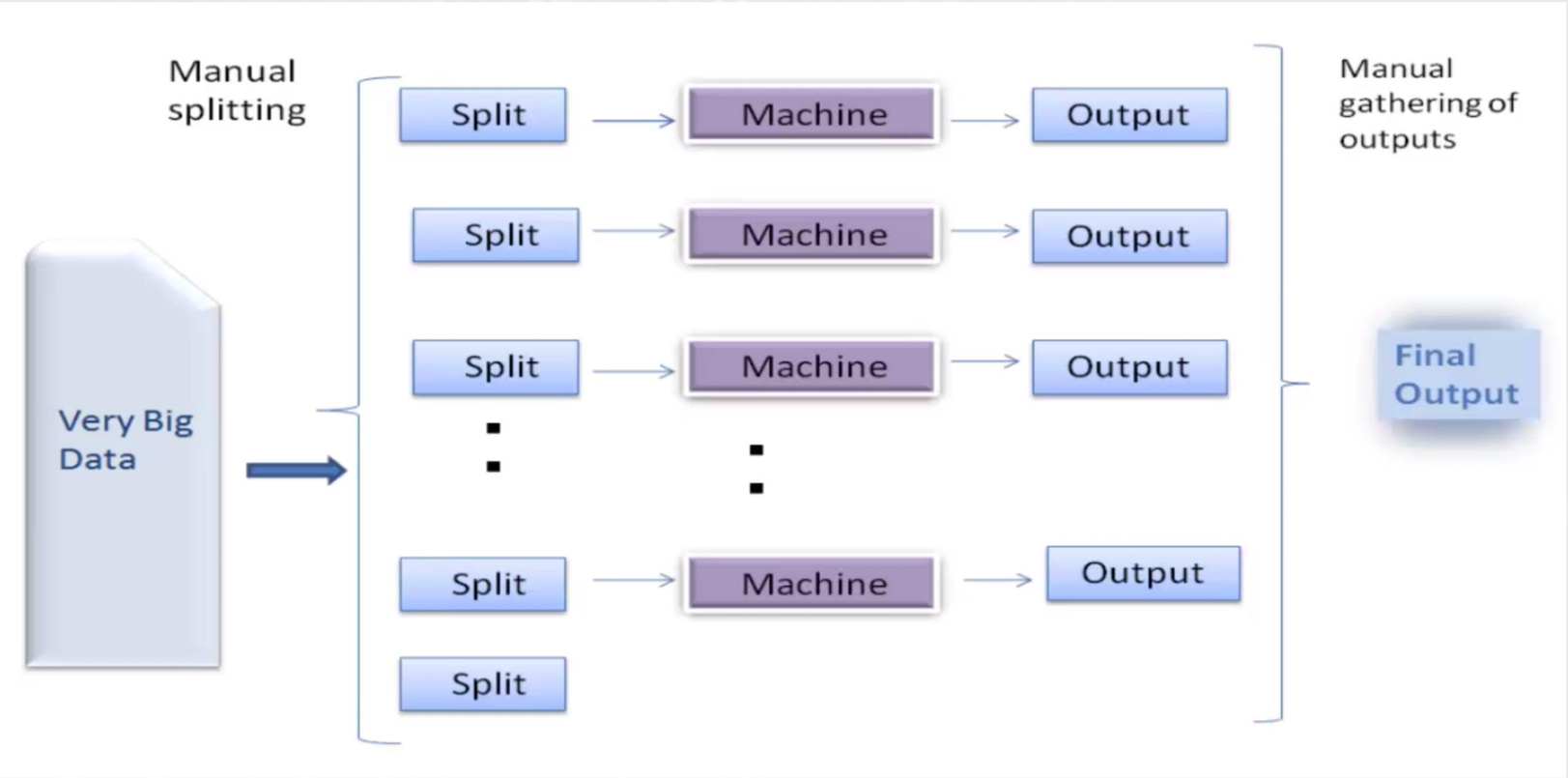


- What is MapReduce ?
- Traditional approach vs Hadoop approach
- Understanding how MR scales - Distributed computing
- MR components
- Example: Ratings Counter
- Basic flow of a MR program
- Types of file input formats in MR
- Chaining in Map reduce
- Combiners in Map reduce
- Use case: Most similar movie
- Hadoop streaming
- Amazon EMR and monitoring jobs
- Distributed computing

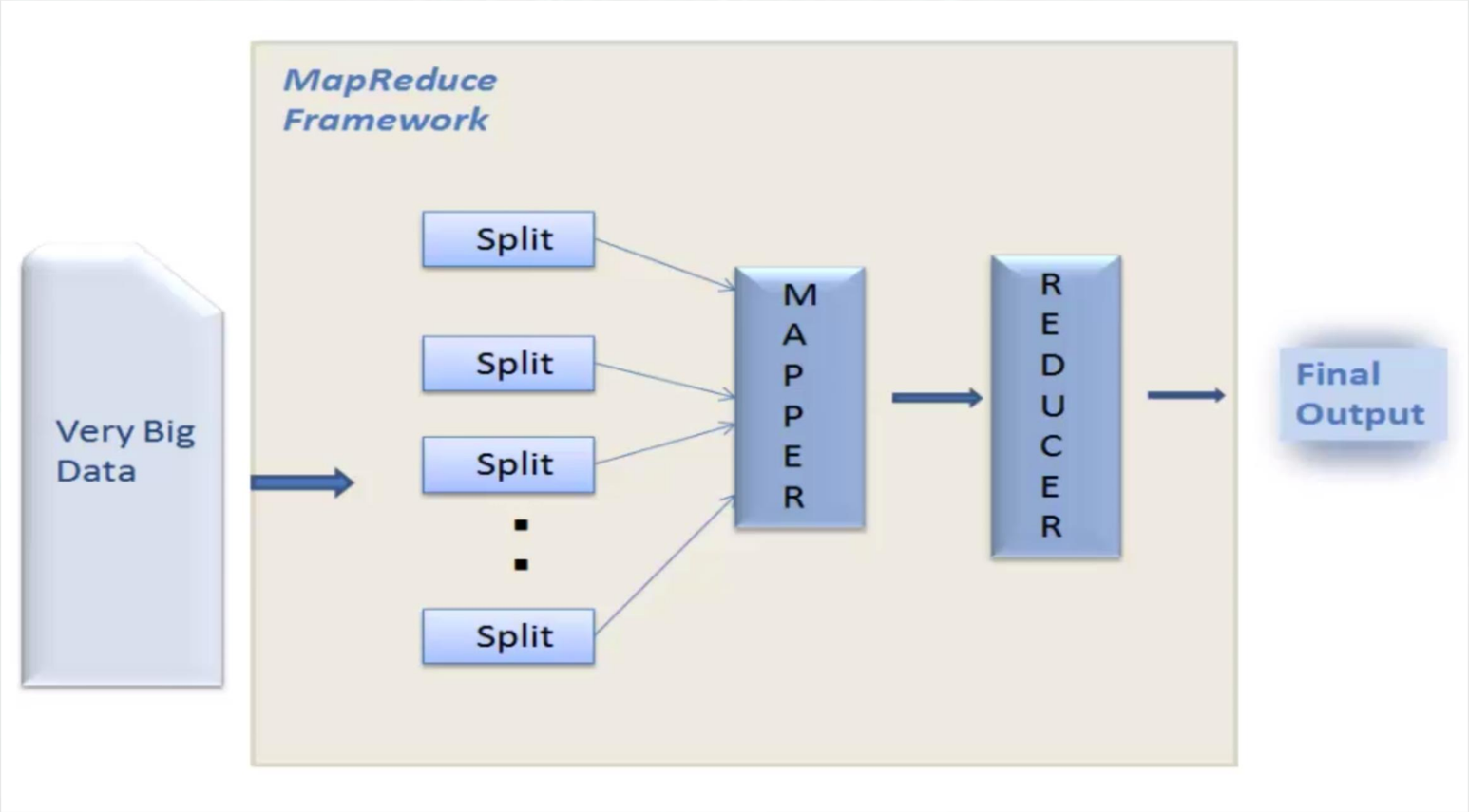
What is MapReduce ?

- Programming model, which is independent of the programming language or platform.
- Introduced by Google for web page indexing.
- Allows to process your data across an entire cluster.
- Enables parallel execution of data processing programs.
- Contains mappers and reducers.
- Mappers have the ability to transform your data in parallel across your entire computing cluster in a very efficient manner.
- Reducers are what aggregate that data together.
- In a nutshell: HDFS places the data on the cluster and MapReduce does the processing work.

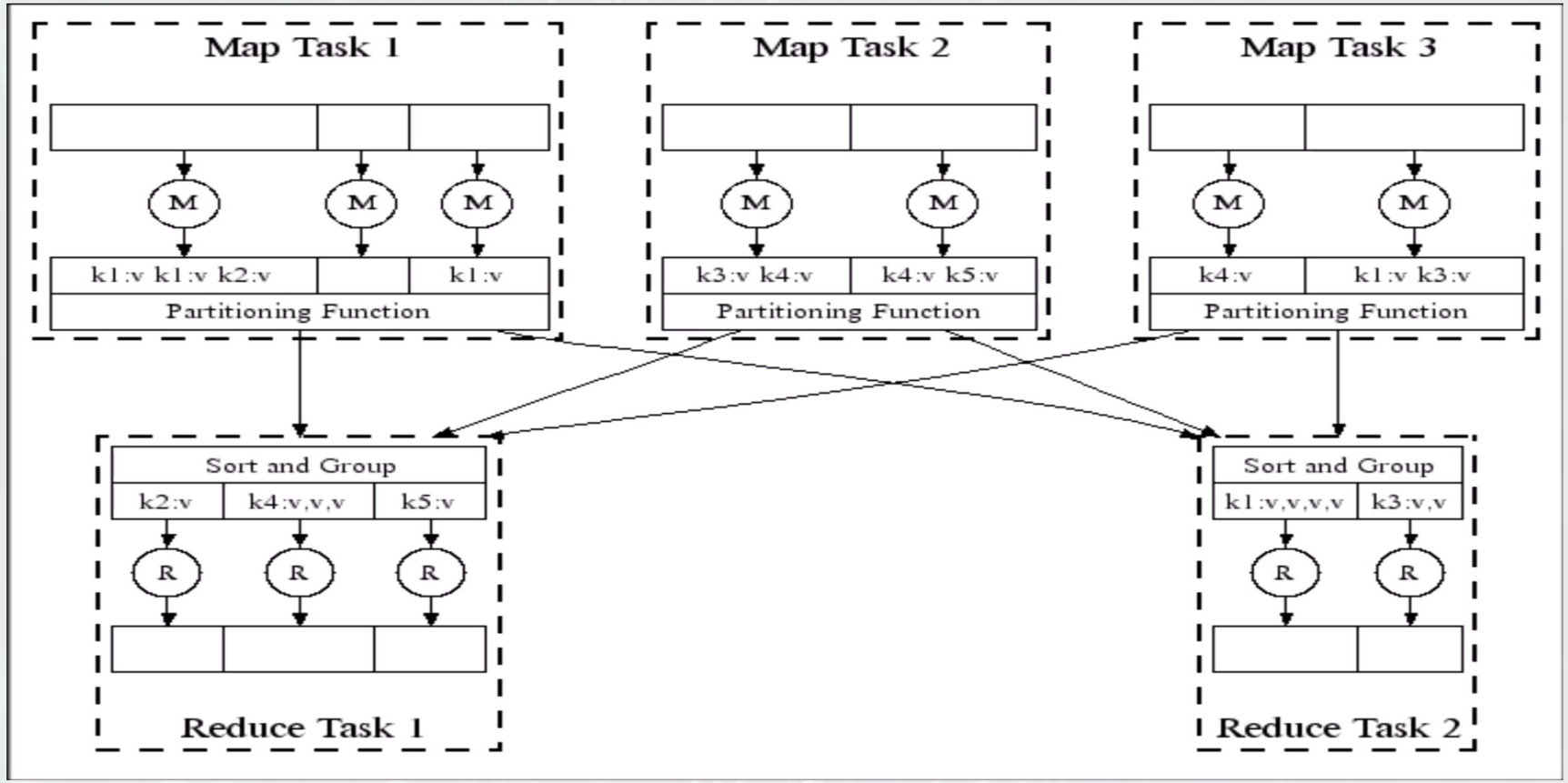
Traditional approach



Hadoop Approach



Understanding how MR scales



Understanding how MR scales

- Divide and conquer a large dataset.
- The real power is that map reduce and Hadoop can partition this job and break up the source input data into multiple mappers that run in parallel on different computers.
- So imagine you have an entire cluster of mappers and these are each responsible for parsing out individual chunks of that input data and outputting the key value pairs for that chunk of input data.
- Similarly that can be sorted and grouped in a distributed manner in our reducers can also run on more than one machine. Outputting the final result from multiple machines and then combining it all together at the end.
- Remember: Your code may be running on more than one computer, all at the same time.

MR components

MapReduce is managed with master-slave architecture included with the following components:

- Job Tracker:

  => master node of the MapReduce system.
  => manages the jobs and resources in the cluster (Task Trackers).
  => schedule each map as close to the actual data being processed on the Task Tracker.

- Task Tracker:

  => slaves that are deployed on each machine.
  => responsible for running the map and reducing tasks as instructed by the Job Tracker.

Ratings Counter

RatingCounter.py

```
1  from mrjob.job import MRJob
2
3  class MRRatingCounter(MRJob):
4      def mapper(self, key, line):
5          (userID, movieID, rating, timestamp) = line.split('\t')
6          yield rating, 1
7
8      def reducer(self, rating, occurences):
9          yield rating, sum(occurences)
10
11 if __name__ == '__main__':
12     MRRatingCounter.run()
13
```

The Mapper

```
196 242 3   881250949
186 302 3   891717742
22  377 1   878887116
244 51  2   880606923
166 346 1   886397596
298 474 4   884182806
115 265 2   881171488
```

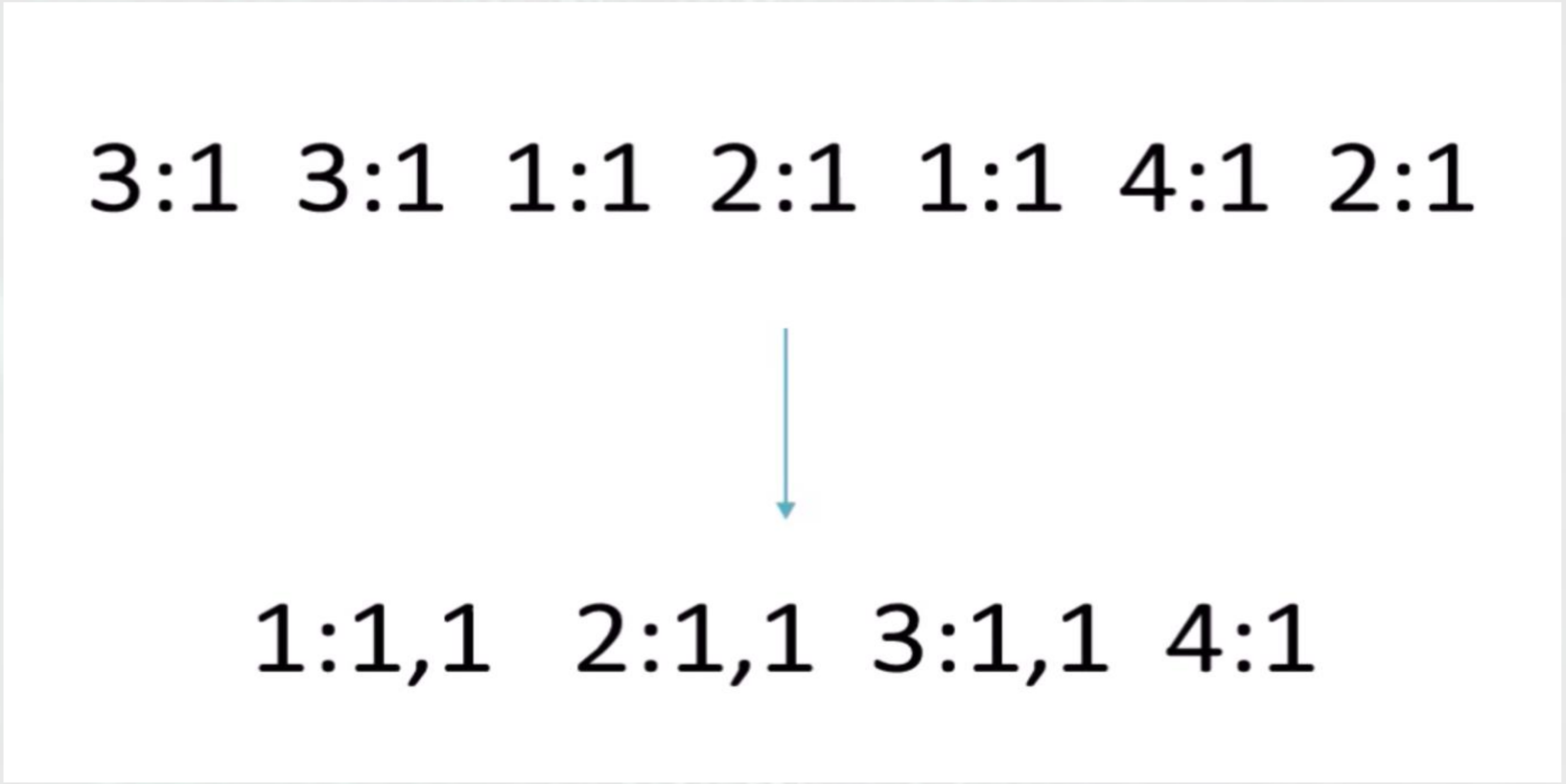


```
4     def mapper(self, key, line):
5         (userID, movieID, rating, timestamp) = line.split('\t')
6         yield rating, 1
```

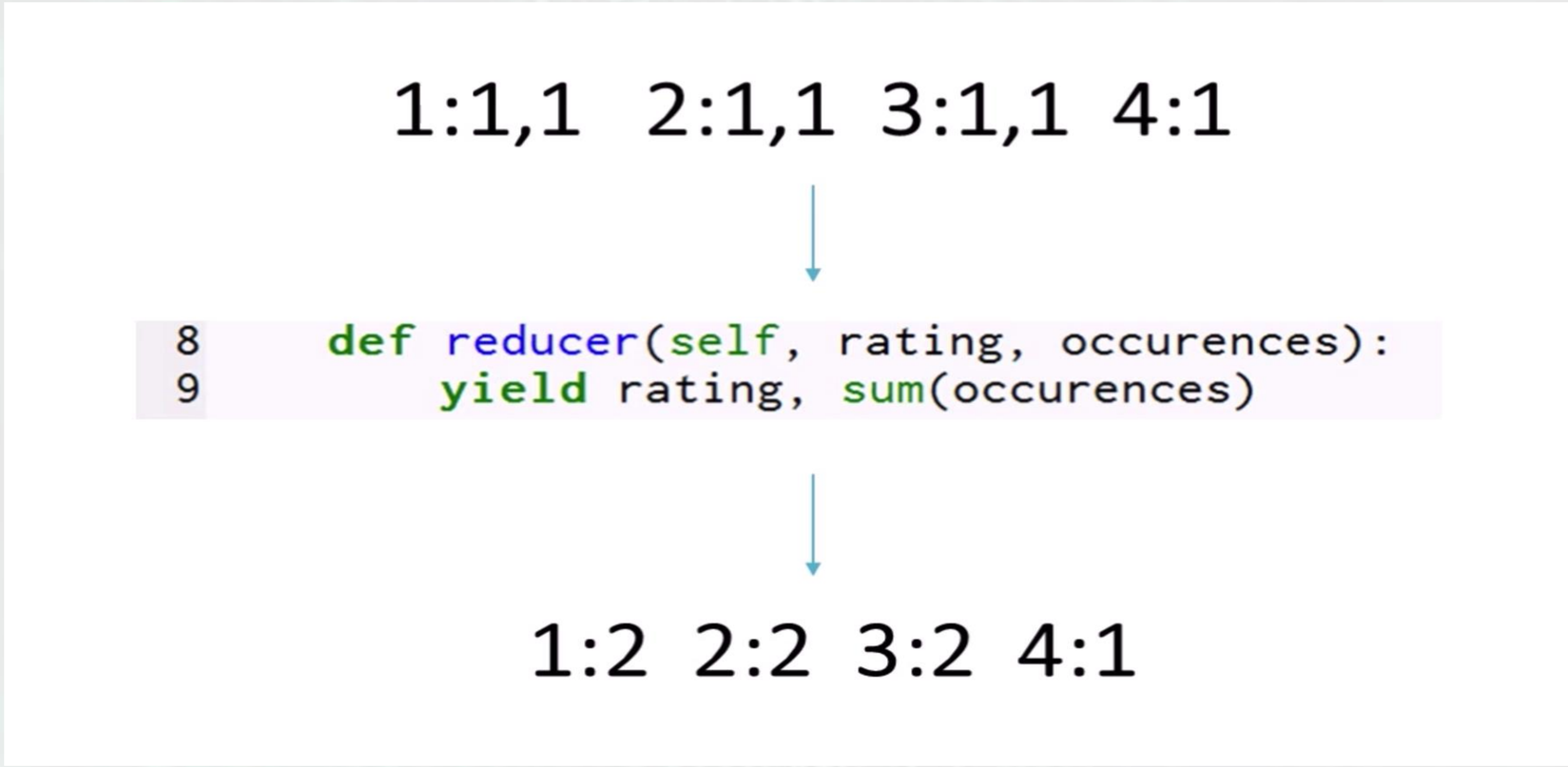


```
3:1 3:1 1:1 2:1 1:1 4:1 2:1
```

Sort and Group



The Reducer


1:1,1 2:1,1 3:1,1 4:1
8 def reducer(self, rating, occurences):
9 yield rating, sum(occurences)
1:2 2:2 3:2 4:1

Putting it all together

```
196  242  3    881250949
186  302  3    891717742
22   377  1    878887116
244  51   2    880606923
166  346  1    886397596
298  474  4    884182806
115  265  2    881171488
```

↓

```
4    def mapper(self, key, line):
5        (userID, movieID, rating, timestamp) = line.split('\t')
6        yield rating, 1
```

↓

```
3:1 3:1 1:1 2:1 1:1 4:1 2:1
```

↓

```
1:1,1  2:1,1 3:1,1 4:1
```

↓

```
8    def reducer(self, rating, occurences):
9        yield rating, sum(occurences)
```

↓

```
1:2 2:2 3:2 4:1
```

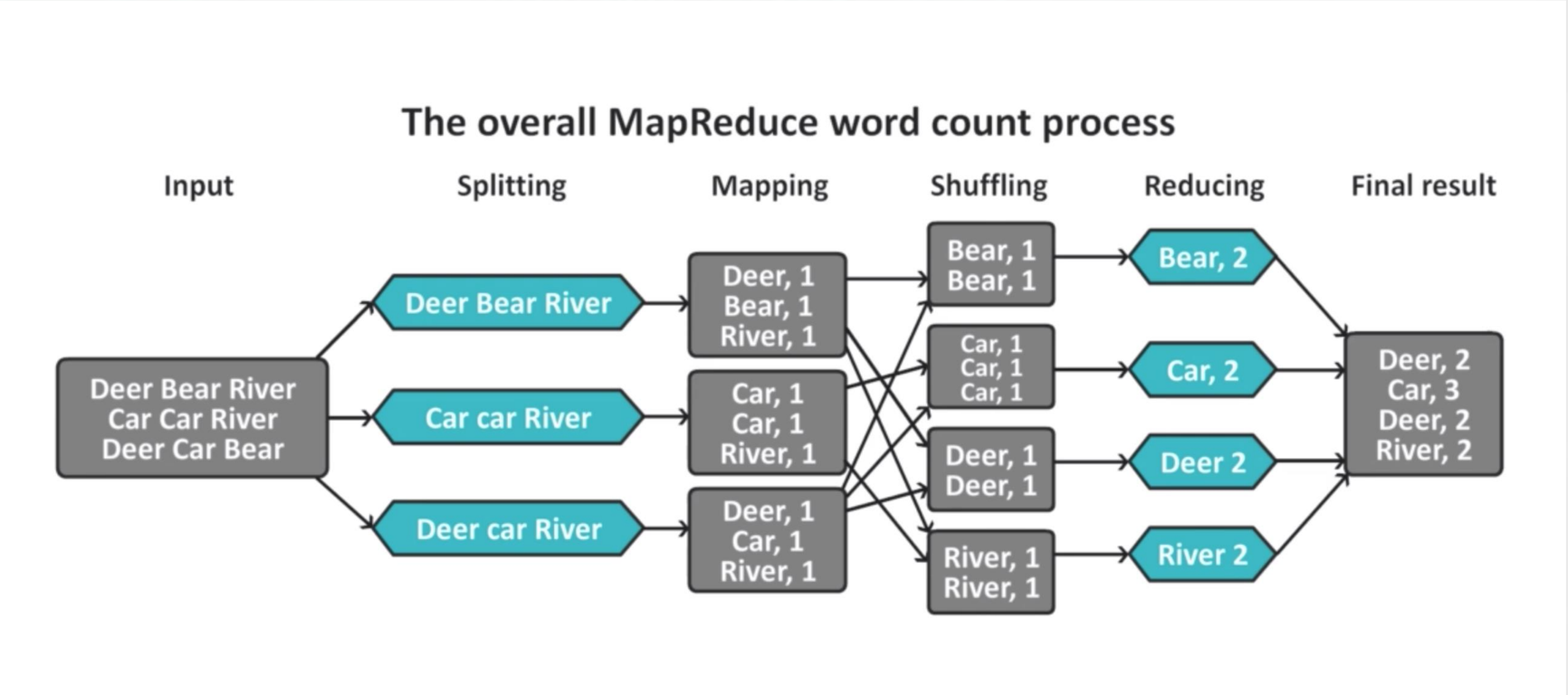
Summary notes

- The MAPPER converts raw source data in to key/value pairs. So for each row of data it extracts key and value. So it extracts and organizes what we care about from your source data.
- Duplicate keys are okay.
- Sort together all the keys and group together all the values seen for the key automatically.
- **Difficult part**: Framing problems as Map Reduce problems. Because it is important to understand the source data when you are trying to solve a map reduce problem.

Chained MapReduce Job

```
from mrjob.job import MRJob
from mrjob.step import MRStep
import re

WORD_REGEXP = re.compile(r"[\w']+")

class MRWordFrequencyCount(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   reducer=self.reducer_count_words),
            MRStep(mapper=self.mapper_make_counts_key,
                   reducer = self.reducer_output_words)
        ]

    def mapper_get_words(self, _, line):
        words = WORD_REGEXP.findall(line)
        for word in words:
            yield word.lower(), 1

    def reducer_count_words(self, word, values):
        yield word, sum(values)

    def mapper_make_counts_key(self, word, count):
        yield '%04d'%int(count), word

    def reducer_output_words(self, count, words):
        for word in words:
            yield count, word


if __name__ == '__main__':
    MRWordFrequencyCount.run()
```

Wordcount example



MapReduce Flow

- Mappers read file splits from HDFS and writes intermediates to disk. So mappers always write output to disk, they cannot stream the output directly to reducers.
- Reducers read the output of mappers (from the disk) and write the final output to HDFS.
- Configuration environmental variables:

```
export HADOOP_MAPRED_HOME = $HADOOP_HOME
export HADOOP_COMMON_HOME = $HADOOP_HOME
```

- YARN configuration – mapred-site.xml

```
<property>
        <name> mapreduce.framework.name </name>
        <value> yarn </value>
</property>
```

Combiners in MapReduce

- Reducer that runs in your mapper.
- For minimizing the amount of data that maps reduce these to shuffle around between nodes so if you are working on a large data set that's distributed amongst a large cluster.
- Running a combiner can be very important for getting the most performance out of your map reduce job.
- Basically your mapper can generate all these key value pairs. But do some of the reduction before it sends out that data to the reducer.
- By doing that you could save a lot of network overhead because you don't have to communicate as much information between mappers and reducers and it can also make things go a little bit faster.

Combiners in MapReduce

WordFrequencyWithCombiner.py

```
4  class MRWordFrequencyCount(MRJob):
5
6      def mapper(self, _, line):
7          words = line.split()
8          for word in words:
9              yield word.lower(), 1
10
11     def combiner(self, key, values):
12         yield key, sum(values)
13
14     def reducer(self, key, values):
15         yield key, sum(values)
16
17
```


Vamsi Krishna Varma Gunturi

Combiners in MapReduce

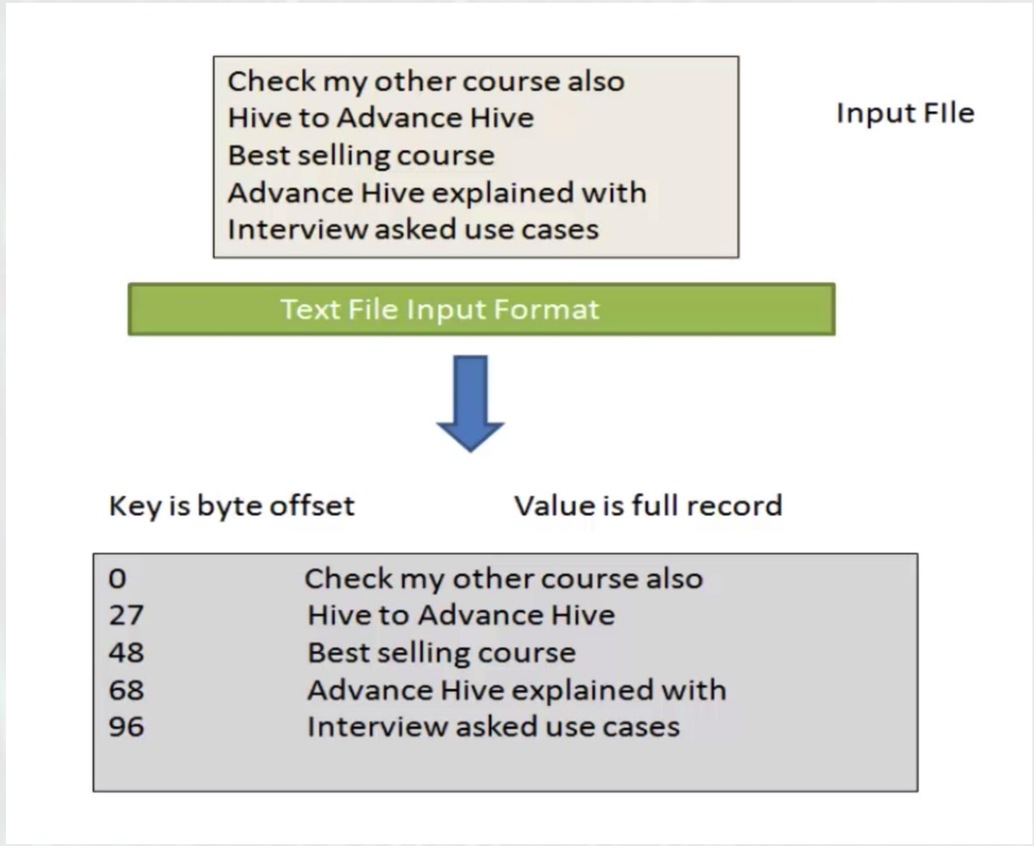
- Reducer that runs in your mapper.
- For minimizing the amount of data that maps reduce these to shuffle around between nodes so if you are working on a large data set that's distributed amongst a large cluster.
- Running a combiner can be very important for getting the most performance out of your map reduce job.
- Basically your mapper can generate all these key value pairs. But do some of the reduction before it sends out that data to the reducer.
- By doing that you could save a lot of network overhead because you don't have to communicate as much information between mappers and reducers and it can also make things go a little bit faster.

File Input Formats

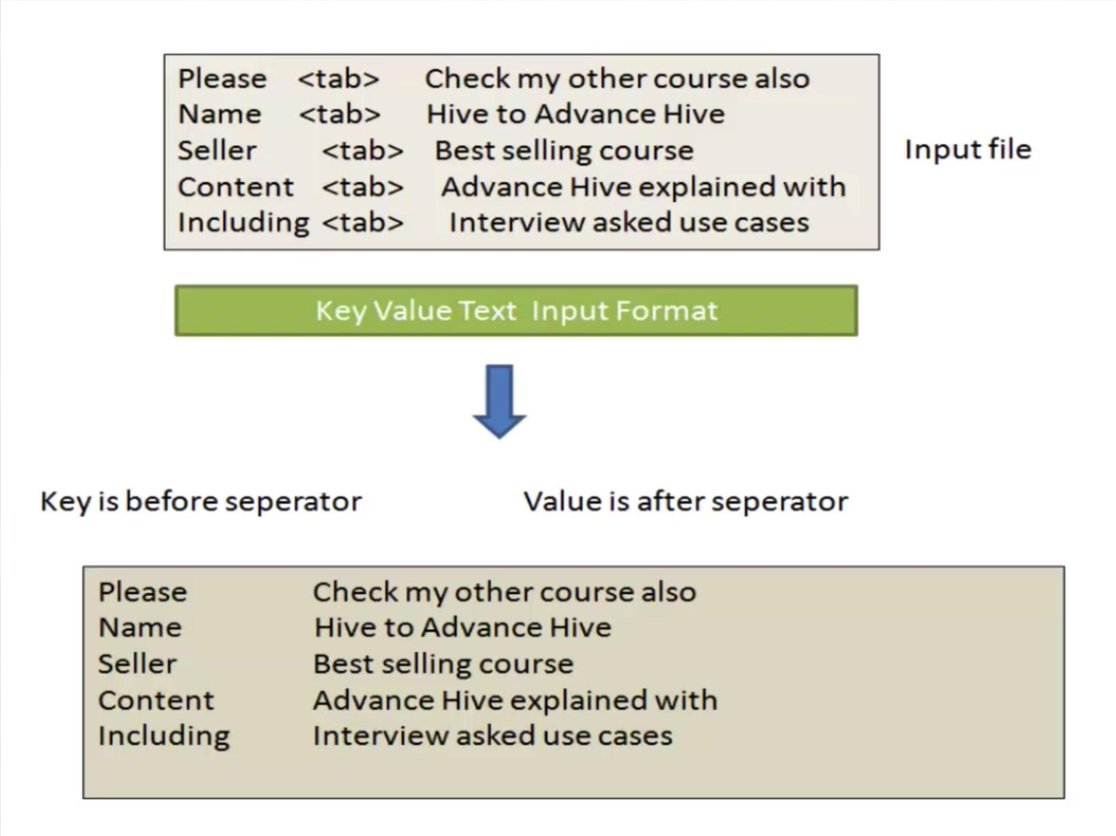
Primarily there are 4 types of input formats that we can use with map reduce -

- Text
- Key value text
- N-line
- Sequence file input format -
  - Sequence file as text
  - Sequence file as binary

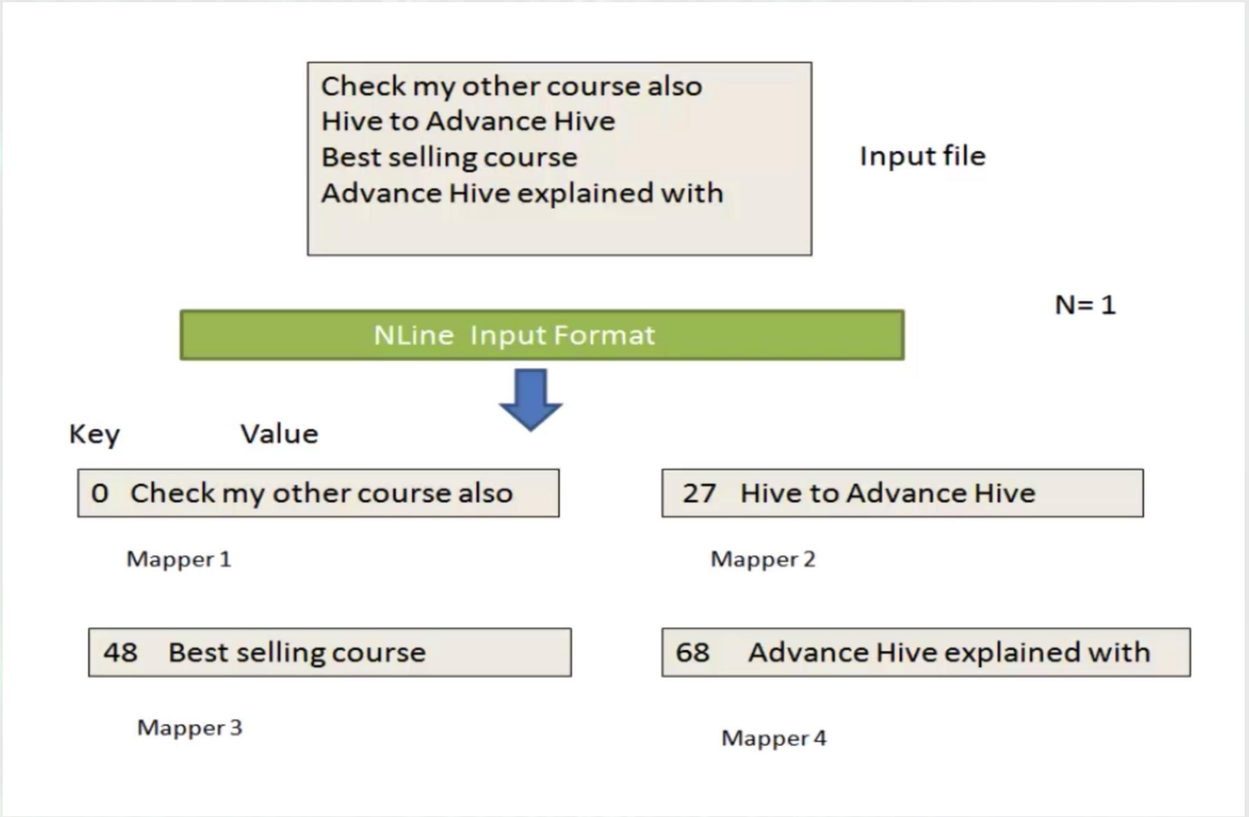
Text input format



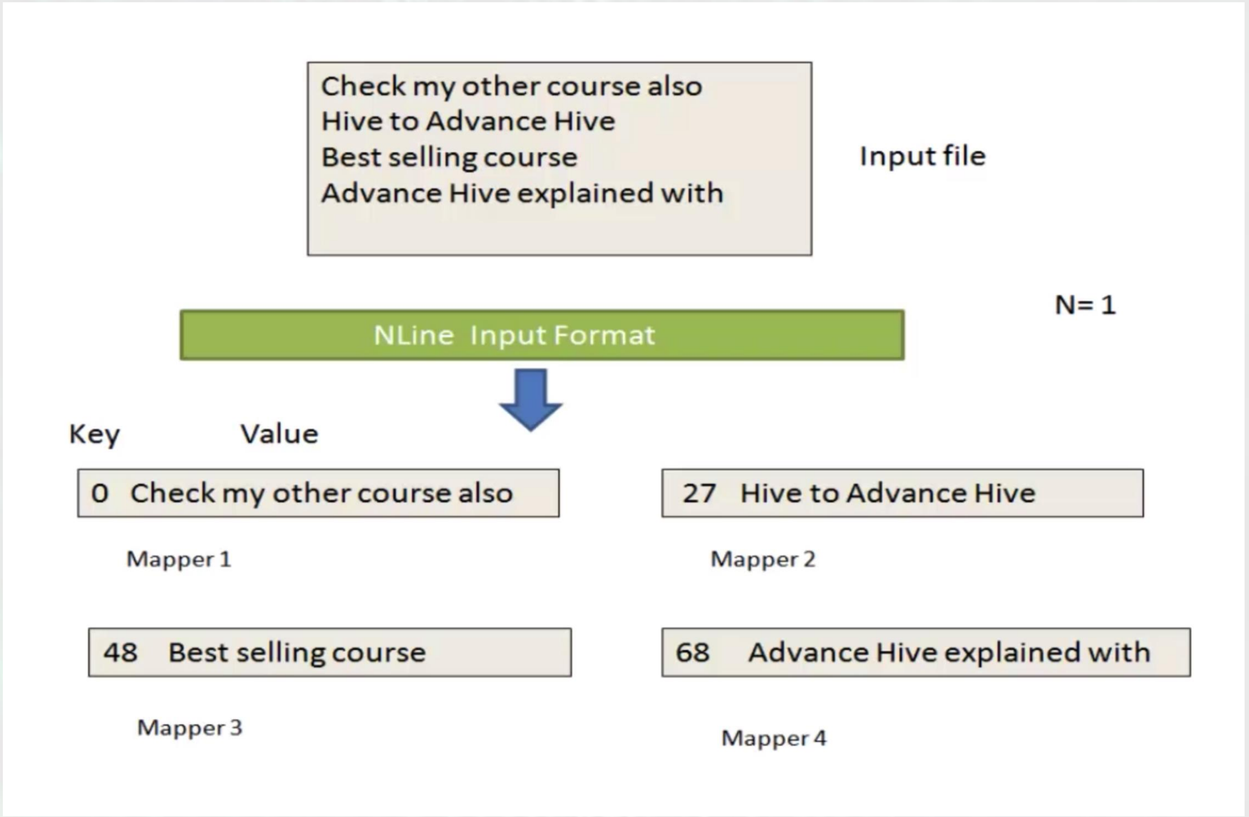
Key value text format



Fixed length format



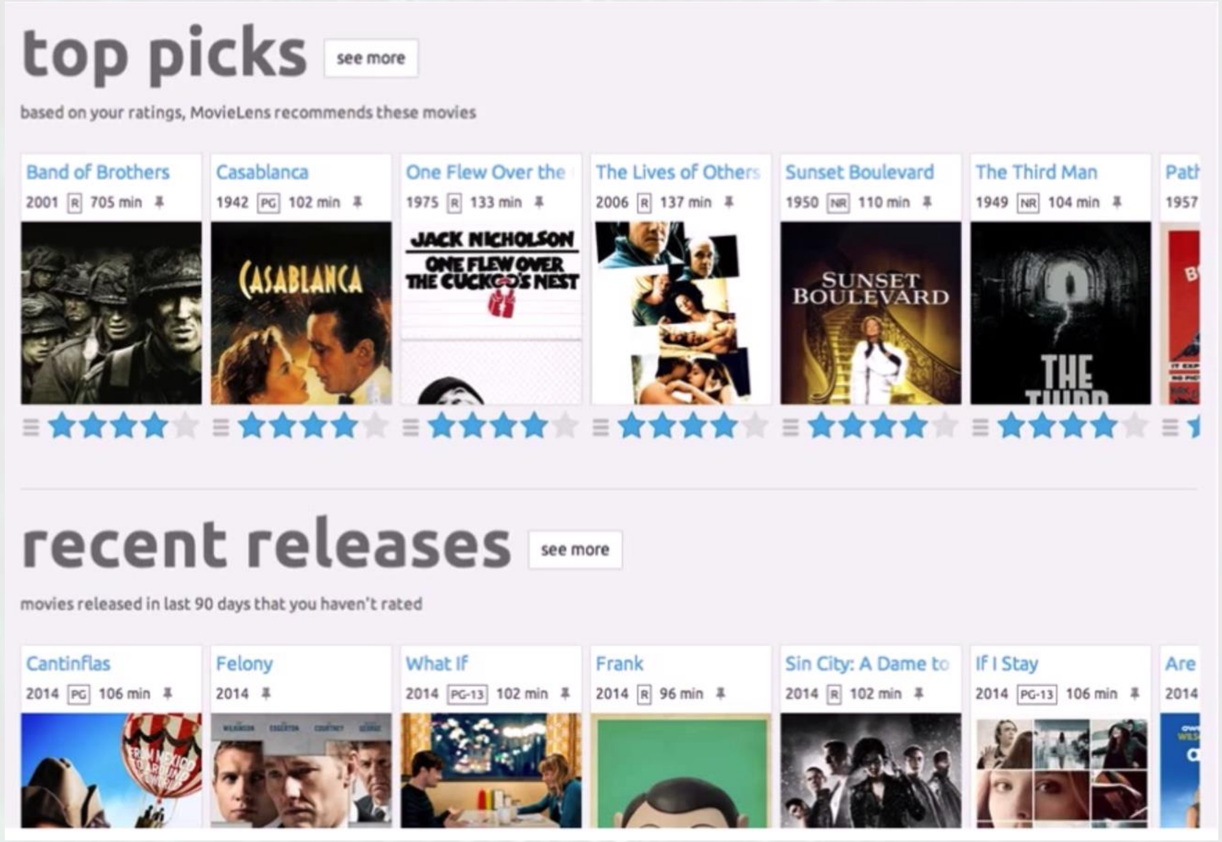
N-line format



Sequence Formats

- Used to read Sequence files.
- Reads the data from individual mapper files.
- Stores the data in form of binary key value pairs
- Keys and values are user defined
- Sequence file as text: converts key value pair to text objects by calling toString() method.
- Sequence file as binary: Retrieves the sequence file's key and values as binary objects.

Recommendations with MapReduce



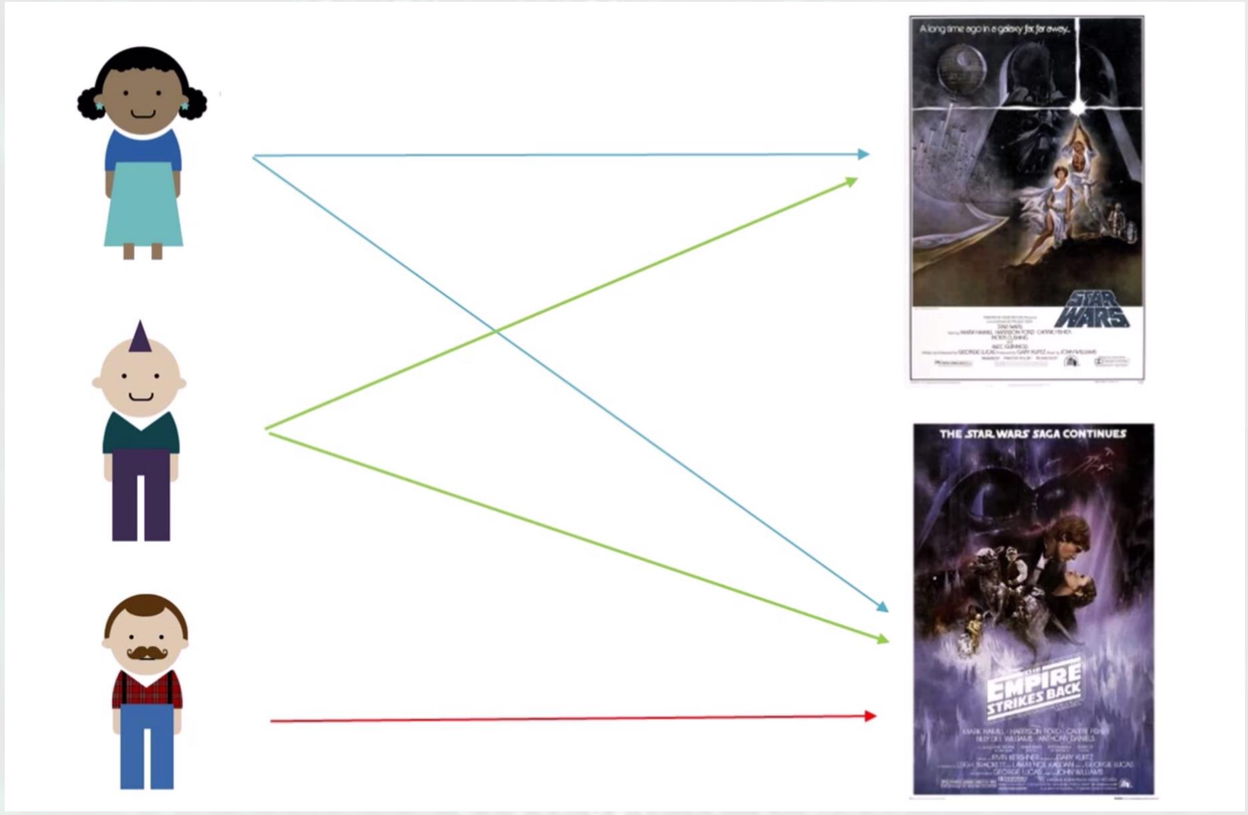
Vamsi Krishna Varma Gunturi

Item based collaborative filtering

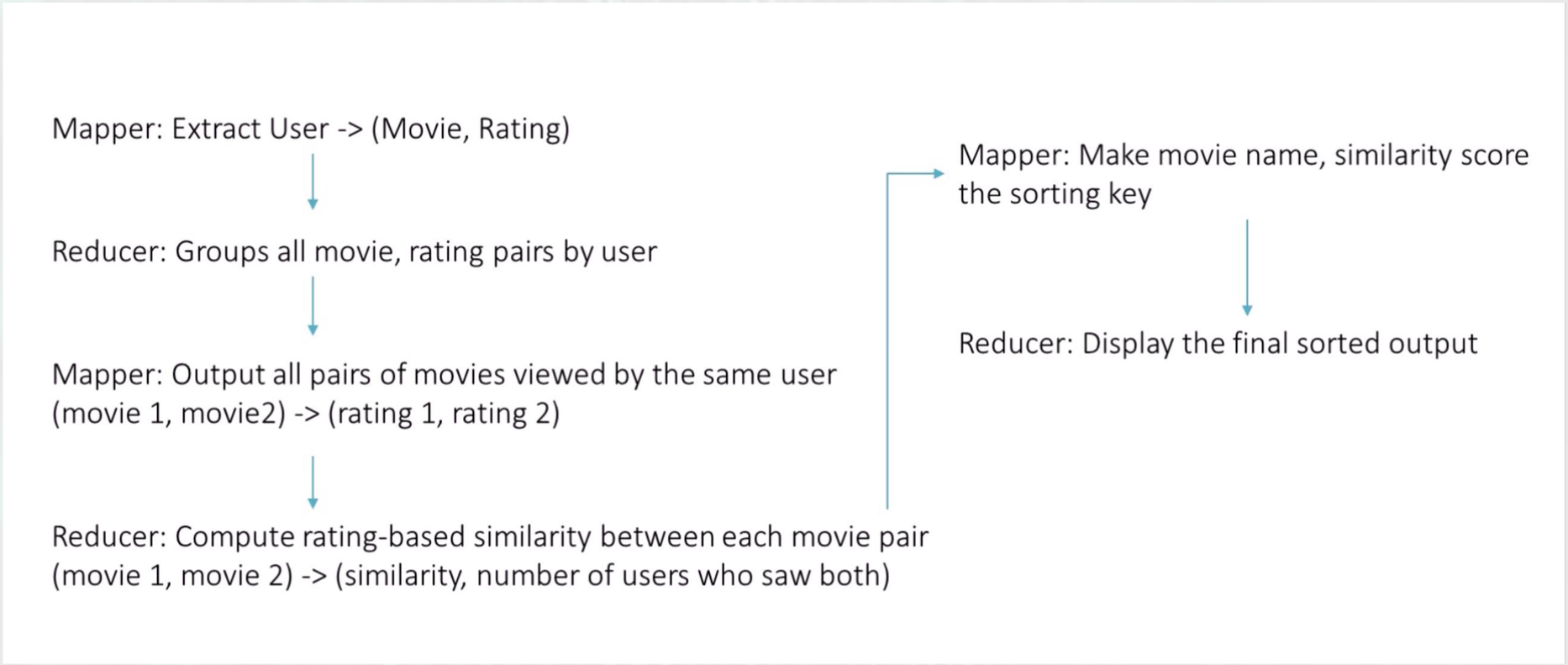
Find similar movies with MapReduce –

- We use MovieLens dataset.
- Find every pair of movies that were watched by the same person.
- Measure the similarity of their ratings across all users who watched both.
- Sort by movie, then by similarity strength.
- This is just one way to do it.

Item based collaborative filtering



Making it a MapReduce problem



Map reduce steps

- Movie,rating pairs by user.
- Compute the similarity between every possible movie combination.
- Sort and make the output easy to understand.

Off to code…

Step 1: Movie ,rating pairs by user

```
def mapper_parse_input(self, key, line):
    # Outputs userID => (movieID, rating)
    (userID, movieID, rating, timestamp) = line.split('\t')
    yield  userID, (movieID, float(rating))

def reducer_ratings_by_user(self, user_id, itemRatings):
    #Group (item, rating) pairs by userID

    ratings = []
    for movieID, rating in itemRatings:
        ratings.append((movieID, rating))

    yield user_id, ratings
```

Step 2: Compute the similarity between movie pairs

```
def mapper_create_item_pairs(self, user_id, itemRatings):
    # Find every pair of movies each user has seen, and emit
    # each pair with its associated ratings

    # "combinations" finds every possible pair from the list of movies
    # this user viewed.
    for itemRating1, itemRating2 in combinations(itemRatings, 2):
        movieID1 = itemRating1[0]
        rating1 = itemRating1[1]
        movieID2 = itemRating2[0]
        rating2 = itemRating2[1]

        # Produce both orders so sims are bi-directional
        yield (movieID1, movieID2), (rating1, rating2)
        yield (movieID2, movieID1), (rating2, rating1)

def reducer_compute_similarity(self, moviePair, ratingPairs):
    # Compute the similarity score between the ratings vectors
    # for each movie pair viewed by multiple people

    # Output movie pair => score, number of co-ratings

    score, numPairs = self.cosine_similarity(ratingPairs)

    # Enforce a minimum score and minimum number of co-ratings
    # to ensure quality
    if (numPairs > 10 and score > 0.95):
        yield moviePair, (score, numPairs)
```

Step 3: Sort and make the output easy to understand

```
def mapper_sort_similarities(self, moviePair, scores):
    # Shuffle things around so the key is (movie1, score)
    # so we have meaningfully sorted results.
    score, n = scores
    movie1, movie2 = moviePair

    yield (self.movieNames[int(movie1)], score), \
        (self.movieNames[int(movie2)], n)

def reducer_output_similarities(self, movieScore, similarN):
    # Output the results.
    # Movie => Similar Movie, score, number of co-ratings
    movie1, score = movieScore
    for movie2, n in similarN:
        yield movie1, (movie2, score, n)
```

Sample Output

```
"Star Wars (1977)"      ["Bound (1996)", 0.9527456995085753, 102]
"Star Wars (1977)"      ["Edge, The (1997)", 0.9527616417862327, 55]
"Star Wars (1977)"      ["Willy Wonka and the Chocolate Factory (1971)",
0.9527621394769873, 280]
"Star Wars (1977)"      ["House Arrest (1996)", 0.9527636751522675, 20]
"Star Wars (1977)"      ["Face/Off (1997)", 0.9527641980248506, 169]
"Star Wars (1977)"      ["Truth About Cats & Dogs, The (1996)", 0.9528130898769434,
216]
"Star Wars (1977)"      ["Die Hard: With a Vengeance (1995)", 0.9528592688483891,
140]
"Star Wars (1977)"      ["Forrest Gump (1994)", 0.9529364064474584, 284]
"Star Wars (1977)"      ["Star Trek III: The Search for Spock (1984)",
0.9531592424602998, 162]
"Star Wars (1977)"      ["Graduate, The (1967)", 0.9531868684781386, 202]
"Star Wars (1977)"      ["Phenomenon (1996)", 0.9534083756106774, 194]
"Star Wars (1977)"      ["Like Water For Chocolate (Como agua para chocolate)
(1992)", 0.953507457017634, 120]
"Star Wars (1977)"      ["Reservoir Dogs (1992)", 0.9538200610507257, 134]
"Star Wars (1977)"      ["Andre (1994)", 0.9538237591021411, 15]
"Star Wars (1977)"      ["Winnie the Pooh and the Blustery Day (1968)",
0.9539460113076144, 71]
"Star Wars (1977)"      ["Passion Fish (1992)", 0.9539665351851674, 26]
"Star Wars (1977)"      ["Three Musketeers, The (1993)", 0.9540537795695558, 81]
"Star Wars (1977)"      ["Chinatown (1974)", 0.9540777882082478, 126]
"Star Wars (1977)"      ["Ridicule (1996)", 0.9540853840007735, 30]
"Star Wars (1977)"      ["Fantasia (1940)", 0.9540945463937451, 163]
```

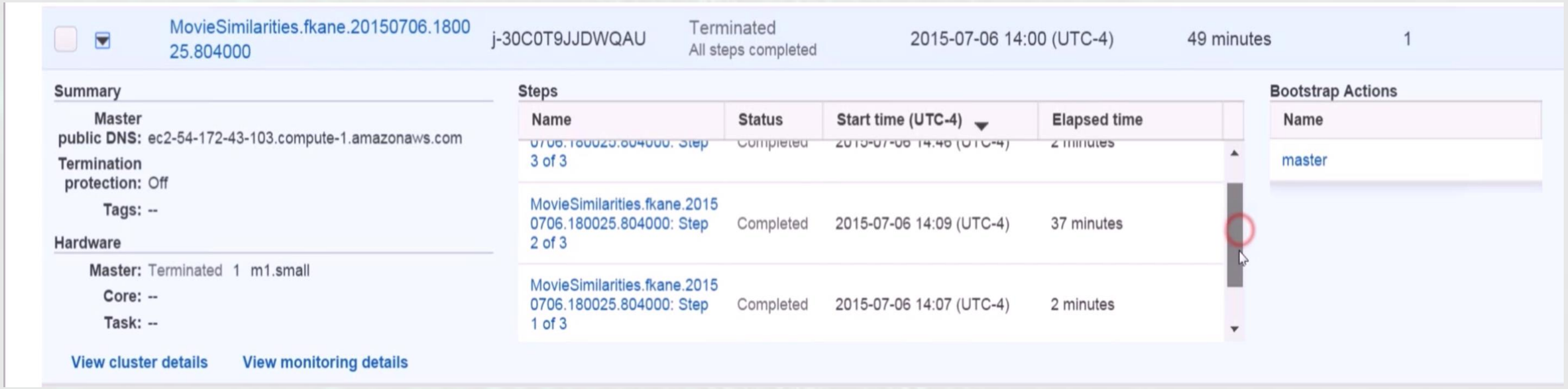
Hadoop Streaming

- Hadoop is written in Java for portability
- How do our Python MapReduce jobs run then ?
- Hadoop is told mappers and reducers are specific executables, that exchange data via text streams. That is Hadoop Streaming. So, Hadoop streaming provides a abstraction across the actual mappers and reducers themselves.
- So, as long as your Python mappers and reducers can generate input and output in a text format that is consistent, Hadoop can piece those together and figure out how to distribute those mappers and reducers
- So, that is the magic of Hadoop streaming and it's the core of how elastic map reduce works with your MapReduce code to actually distributed throughout a Hadoop cluster.

Amazon EMR

- EMR stands for Elastic map reduce
- Amazon EMR is a cloud based service to run map reduce jobs on a distributed environment
  Link: www.aws.amazon.com
- Create a new AWS account and open the AWS console.
- Now pick the access keys from Account => My Security Credentials => Access Keys
- Create a new access key and store the keys in your local environmental variables as AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY

Monitoring the MapReduce job on Amazon EMR



Distributed computing

- Running on 4 servers does not mean your job will run 4 times as fast! It will probably cost 4 times as much, though.

- To run on a single EMR node:
  > python MovieSimilarities.py -r emr --items=ml-100k/u.item ml-100k/u.data - It took **51 minutes** to execute

- To run on 4 EMR nodes:
  > python MovieSimilarities.py -r emr --num-ec2-instances=4 --items=ml-100k/u.item ml-100k/u.data – It took **25 minutes** to execute

Note: For the above commands to work we should be setting AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY prior to execution of the python script.

Master Executive di II Livello

BIG DATA ANALYSIS AND BUSINESS INTELLIGENCE

Vamsi Krishna Varma Gunturi
Data science intern at ISTAT
vamsivarmagunturi@gmail.com

Grazie

fondazione INUIT TORVERGATA