LARGE SCALE MACHINE LEARNING
- MAPREDUCE -

LARGE SCALE MACHINE LEARNING

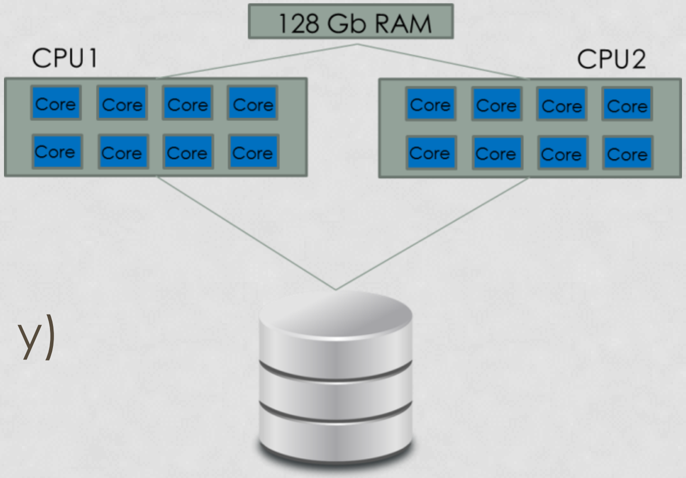
- Increasing computational needs:
  - Very large datasets (instances, attributes)
  - Complex algorithms / large models: large ensembles, computationally intensive optimization processes (deep learning, …)
  - Computationally intensive tasks: Crossvalidation, Hyper-parameter tuning, algorithm selection (try knn, decision trees, …)
- Increasing computational power:
  - Multicore (for example: i7 Intel computers have 4 real cores)
  - Large computer networked clusters
  - Alternative hardware: FPGAs (Field Programmable Gate Array), GPUs (Graphics processing unit)

PARALELLISM

- Every year we have faster and faster computers, but speed is becoming increasingly difficult. The alternative is doing many things in parallel:
  - Task parallelism: Different tasks running on the same data
  - Data parallelism: The same task run on different data in parallel.
  - Pipeline parallelism: Output of one task is input for another task

TASK PARALELLISM

- Different processes run on the same data
- Embarrassing paralallelism:
  - Crossvalidation:
    - cross_val(model, X, y, **n_jobs=4**, cv=3)
  - Hyper-parameter tuning (grid search)
    - GridSearchCV(model, **n_jobs=4**, cv=3).fit(X, y)
  - Ensembles:
    - RandomForestClassifier(**n_jobs=4**).fit(X, y)



- Check Olivier Grisel's tutorial ("Strategies & Tools for Parallel Machine Learning in Python)
  - http://es.slideshare.net/ogrisel/strategies-and-tools-for-parallel-machine-learning-in-python

PARALLELIZATION OF GRID SEARCH

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2	(2,2)	(2,4)	(2,6)	(2,8)
4	(4,2)	(4,4)	(4,6)	(4,8)
6	(6,2)	(6,4)	(6,6)	(6,8)

Grid search means: try all possible combinations of values for the hyper-parameters. Given that each combination is independent of the others, they can be carried out in parallel.

PARALLELIZATION OF CROSSVALIDATION

- For i in [1, 2, …, k]
  - Learn model with all partitions but i
  - Test model with partition i
- k independent iterations => they can be carried out in parallel

PARALLELIZATION OF ENSEMBLES

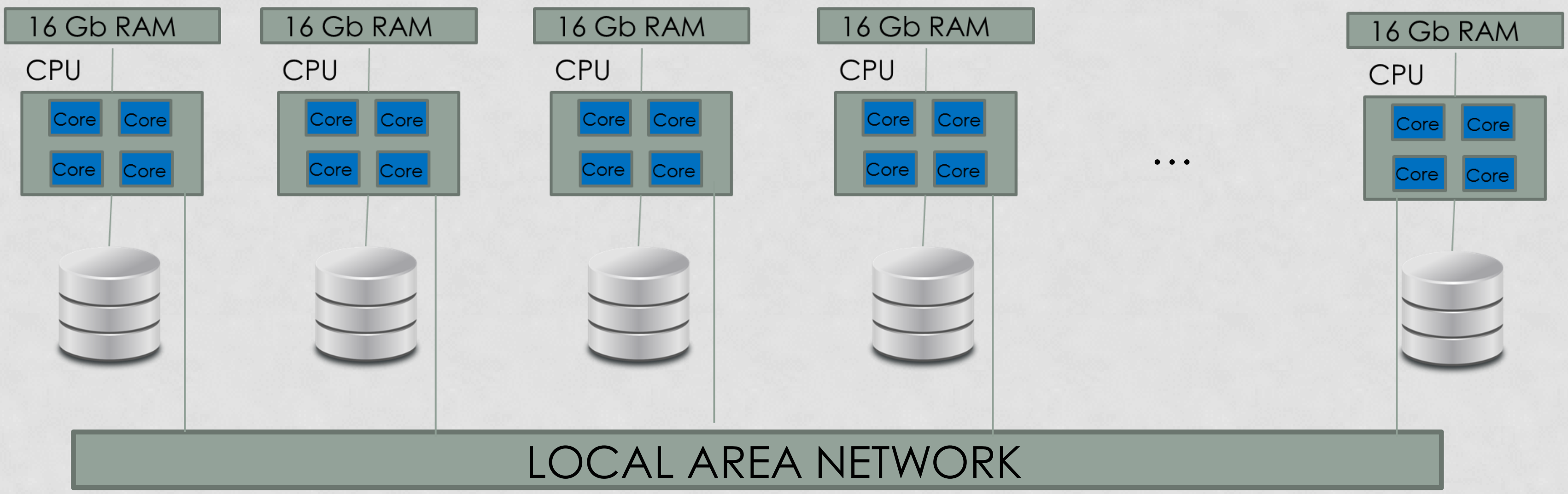
- We will talk about ensembles in future lectures
- It involves building not one, but hundreds or thousands of classifiers
- In one of the cases (Bagging and Random Forests), the models are independent of each other, and can be built in parallel.

“NON EMBARRASINGLY” PARALLELISM

- Not all algorithms are embarrasingly parallel
- For instance, it is not so easy to task-parallelize the decision tree learning algorithm (i.e. it is not so easy to decompose DT learning into subprocesses that can be run in parallel)
- But, crossvalidation, grid-search, and ensembles are processes that you are going to run, and probably that’s all the task-parallelism (embarrasingly so) that you will ever need

DATA PARALLELISM

- The same task running on different data, in parallel



BIG DATA

- Currently, Big Data means data parallelism
- Either:
  - Data does not fit on a single computer
  - or it takes too long to process on a single computer
- Three V's:
  - **Volume**: up to petabytes
  - Velocity: streaming
  - Variety: structured / unstructured (text, sensor data, audio, video, click streams, log files, …)
- It takes advantage of commodity hardware farms
- Current programming models: Mapreduce (Yahoo), Apache Spark, Dryad (Microsoft), Vowpal Wabbit (Microsoft)

MOTIVATION

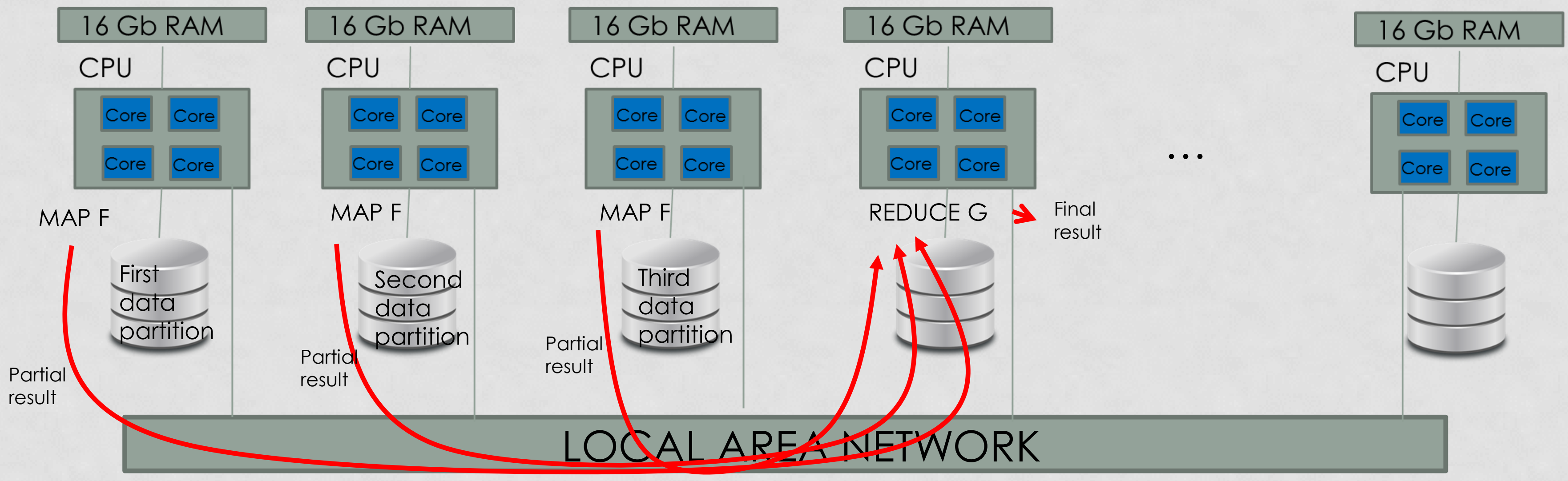
- *Using available comodity* hardware: basically, thousands of standard PCs organized in racks and with local hard disks

MAP REDUCE

- Programming model for data parallelism / distributed computation
- Based on two operations:
  - **Map**: executed in parallel in different computers
  - **Reduce**: combines results produced by the maps
- The aim of the model is that **heavy processing happens locally (map processes)**, where the data is stored.
  - Do not use the network, or use it as little as possible.
  - Results produced by **Map** are much smaller in size, and can be combined (**reduced**) in other computers.
- Origins: Google 2004 (page indices, etc. Several petabytes daily)
- Used in Facebook, LinkedIn, Tuenti, ebay, Yahoo, …
- Amazon AWS, Microsoft Azure, Google, … provide Map-Reduce platforms (not for free)

MAP REDUCE DATA PARALLELISM

- **Map** processes do the heavy processing locally, where data resides
- **Map** results (very small in size), are **partial results**, that travel across the network and are combined by the **reducer** into a **final result**.



MAPREDUCE PROGRAMMING MODEL

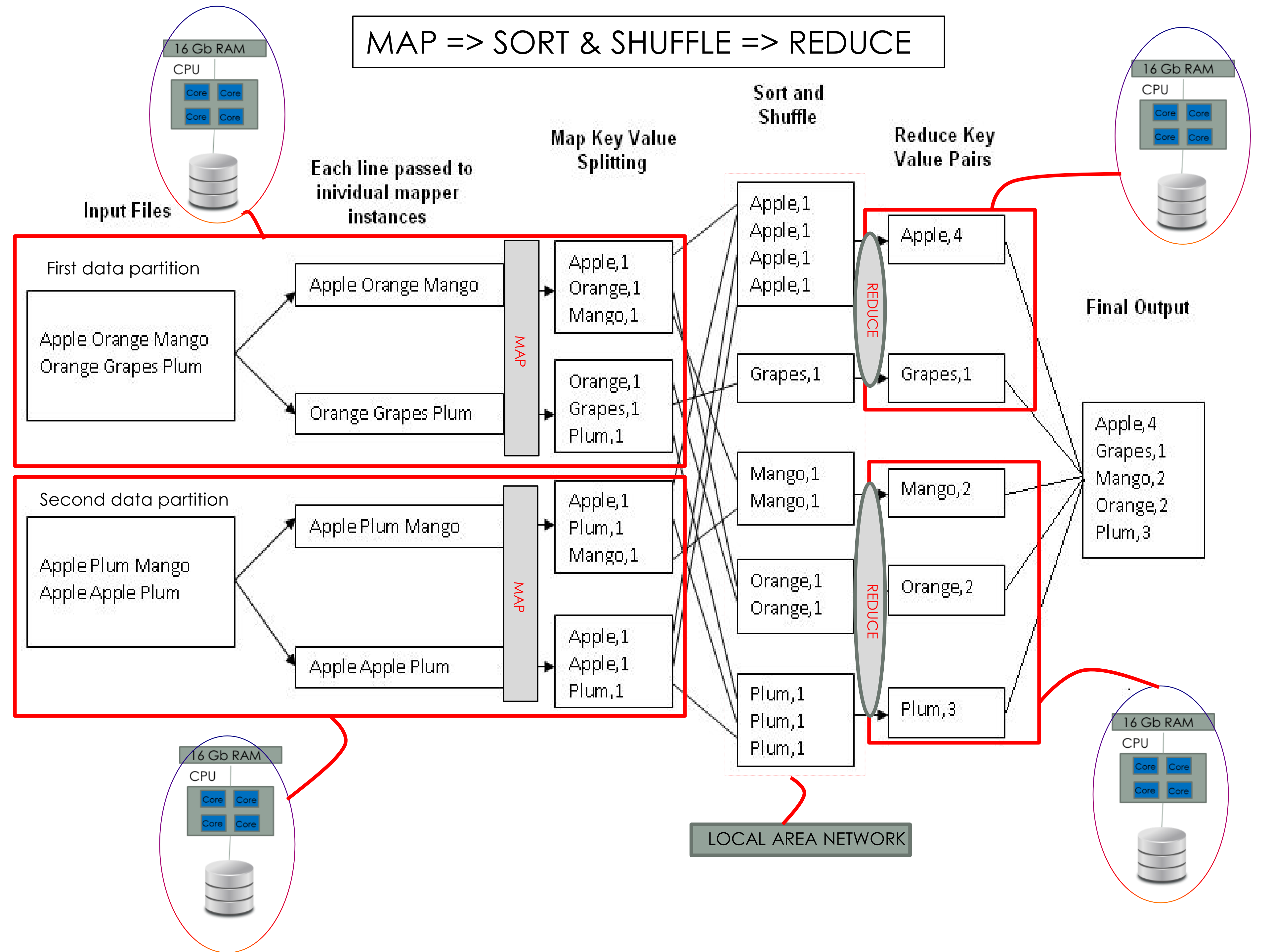
- Inspired in functional programming: map and reduce
- For instance, In Python:

```
In [1]: def f(x):
            return(x**2)

In [2]: map(f, [1,2,3])
Out[2]: [1, 4, 9]

In [6]: def g(a,b):
            """Add a plus b"""
            return(a+b)

In [8]: #1 + 2 + 3 + 4
        reduce(g,[1,2,3,4])
Out[8]: 10
```

COUNTING WORDS IN MAPREDUCE

- Let's suppose we have a huge dataset with text (like the *news* datasets we have already seen)
- Our aim is to count how many times each word appears in the dataset:

1. The huge dataset is split into different partitions (as many partitions as hard disks)
2. Function **map** counts words in a text
   - Note: each CPU / computer may be able to run several map functions in parallel (multicore)
3. Sort & shuffle: partial results from **maps** are grouped by key and delivered to **reduce** functions in other computers via the network, depending on keys. This is done automatically by the mapreduce system
   - Note: output of map can be grouped by **hashfunction**(key) rather than key. The user is responsible for defining the hashfunction
4. Function **reduce** adds occurrences of the same word


MAP => SORT & SHUFFLE => REDUCE
16 Gb RAM
CPU
Core
Input Files
Each line passed to inividual mapper instances
Map Key Value Splitting
Sort and Shuffle
Reduce Key Value Pairs
Final Output
First data partition
Apple Orange Mango
Orange Grapes Plum
Apple Orange Mango
Orange Grapes Plum
MAP
Apple,1
Orange,1
Mango,1
Orange,1
Grapes,1
Plum,1
Second data partition
Apple Plum Mango
Apple Apple Plum
Apple Plum Mango
Apple Apple Plum
Apple,1
Plum,1
Mango,1
Apple,1
Apple,1
Plum,1
Apple,1
Apple,1
Apple,1
Apple,1
Grapes,1
Mango,1
Mango,1
Orange,1
Orange,1
Plum,1
Plum,1
Plum,1
REDUCE
Apple,4
Grapes,1
Mango,2
Orange,2
Plum,3
Apple,4
Grapes,1
Mango,2
Orange,2
Plum,3
LOCAL AREA NETWORK

MAP AND REDUCE FUNCTIONS

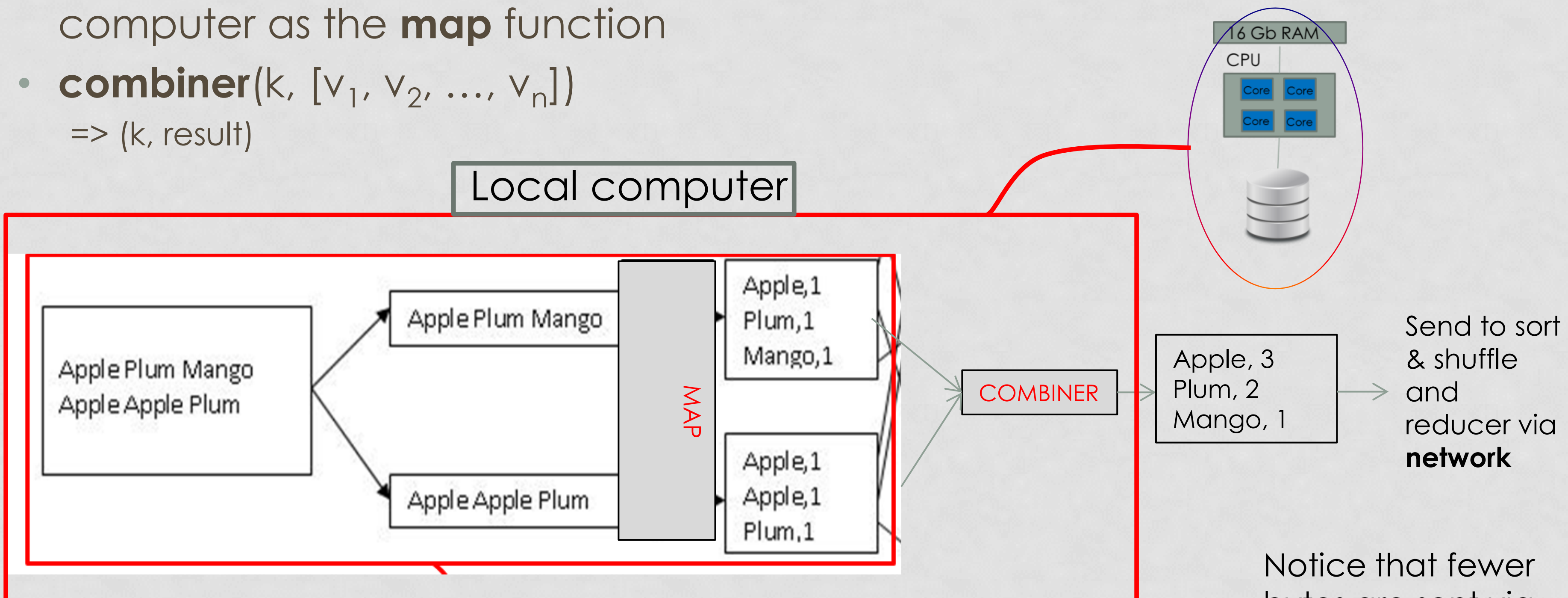
- The programmer has to program two functions: map and reduce. “Sort & Shuffle” is carried out automatically
- **map**(key, value)
  => [(key_1, $value_1$), (key_2, $value_2$), …, (key_n, $value_n$)]
- **Sort and shuffle**: (k_1, v_1), (k_1, v_2), …, (k_1, v_n), (k_2, w_1), …, (k_2, w_m), …
  => (k_1, [v_1, v_2, …, v_n]), (k_2, [w_1, w_2, …, w_m]), …
- **reduce**(k, [v_1, v_2, …, v_n])
  => result

COUNTING WORDS IN MAPREDUCE. EXAMPLE IN PYTHON

In [23]:
```
def mapper((key,value)):
    # key: document identifier
    # value: document contents
    words = value.split()
    for w in words:
      mr.emit_intermediate(w, 1)

def reducer(key, list_of_values):
    # key: word
    # value: list of occurrence counts
    total = 0
    for v in list_of_values:
      total += v
    mr.emit((key, total))
```

COMBINER FUNCTIONS

- There are additional operations that could be **reduced** in the local computer, instead of being sent to a remote reducer.
- Example: (apple, 1), (apple, 1) and (apple, 1) can be added locally, instead of being sent to the reducer via the network
- A **combiner** function is like a reducer, but it is executed in the same computer as the **map** function
- **combiner**(k, [v_1, v_2, …, v_n])
  => (k, result)



Notice that fewer bytes are sent via the network

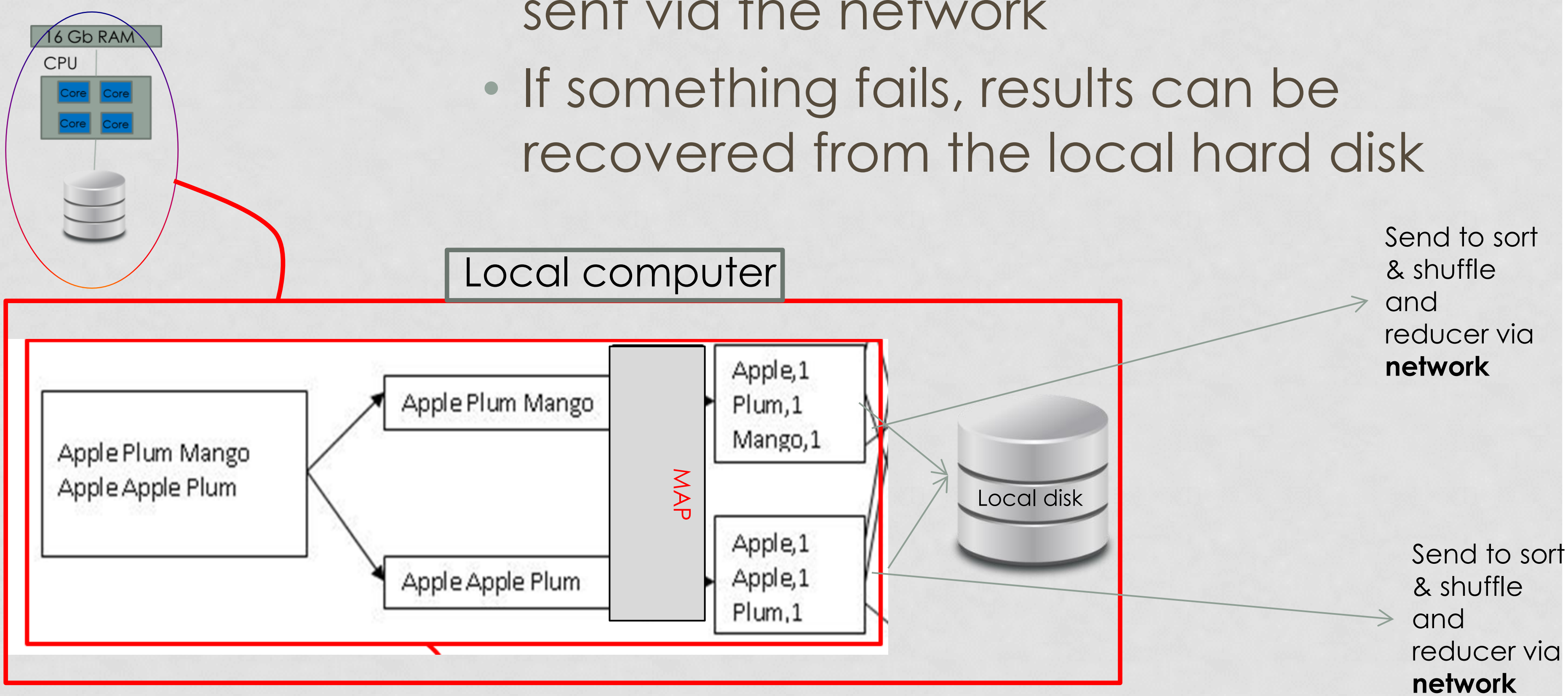
COUNTING WORDS IN MAPREDUCE. EXAMPLE IN PYTHON

- In the counting words problem, the **combiner** is just like the **reducer**

```
def mapper((key,value)):
    # key: document identifier
    # value: document contents
    words = value.split()
    for w in words:
      mr.emit_intermediate(w, 1)

def reducer(key, list_of_values):
    # key: word
    # value: list of occurrence counts
    total = 0
    for v in list_of_values:
      total += v
    mr.emit((key, total))

def combiner(key, list_of_values):
    # key: word
    # value: list of occurrence counts
    total = 0
    for v in list_of_values:
      total += v
    mr.emit((key, total))
```

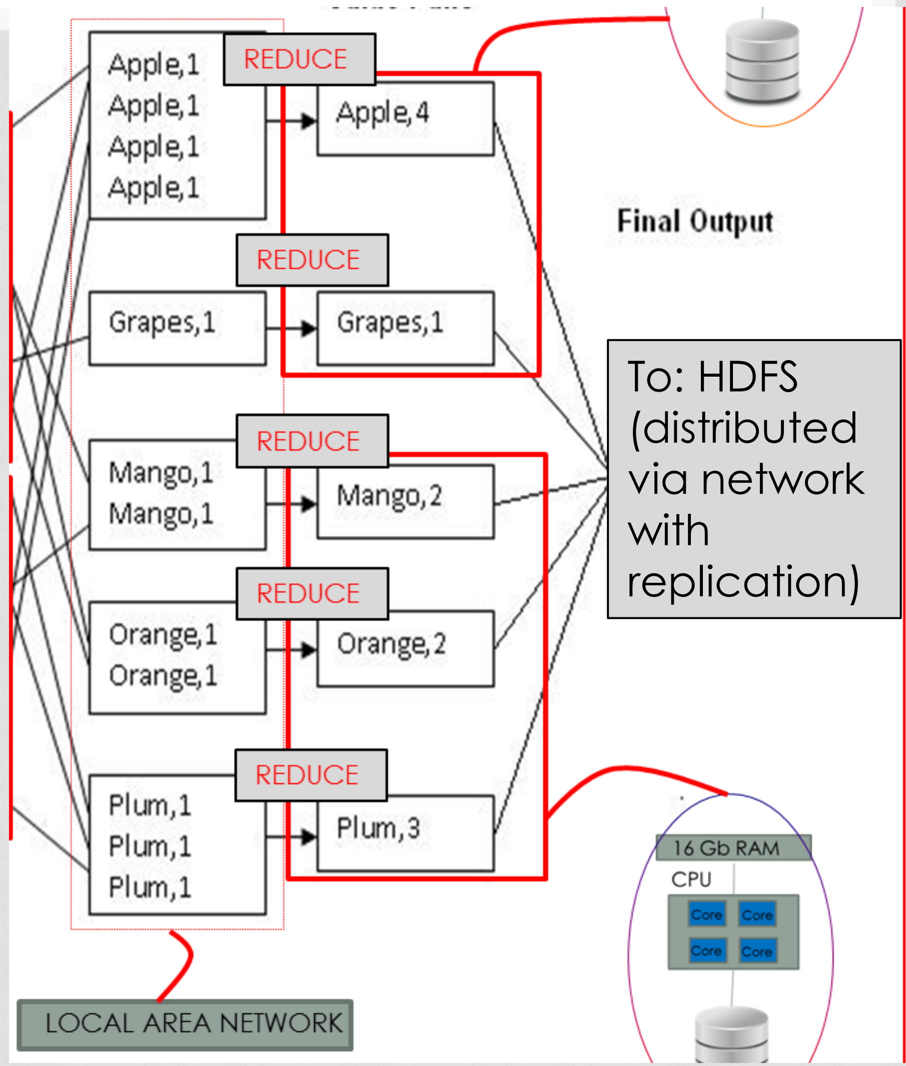
FAILURE RECOVERY

- The output of **maps** is written to the local hard disk, in addition to being sent via the network
- If something fails, results can be recovered from the local hard disk

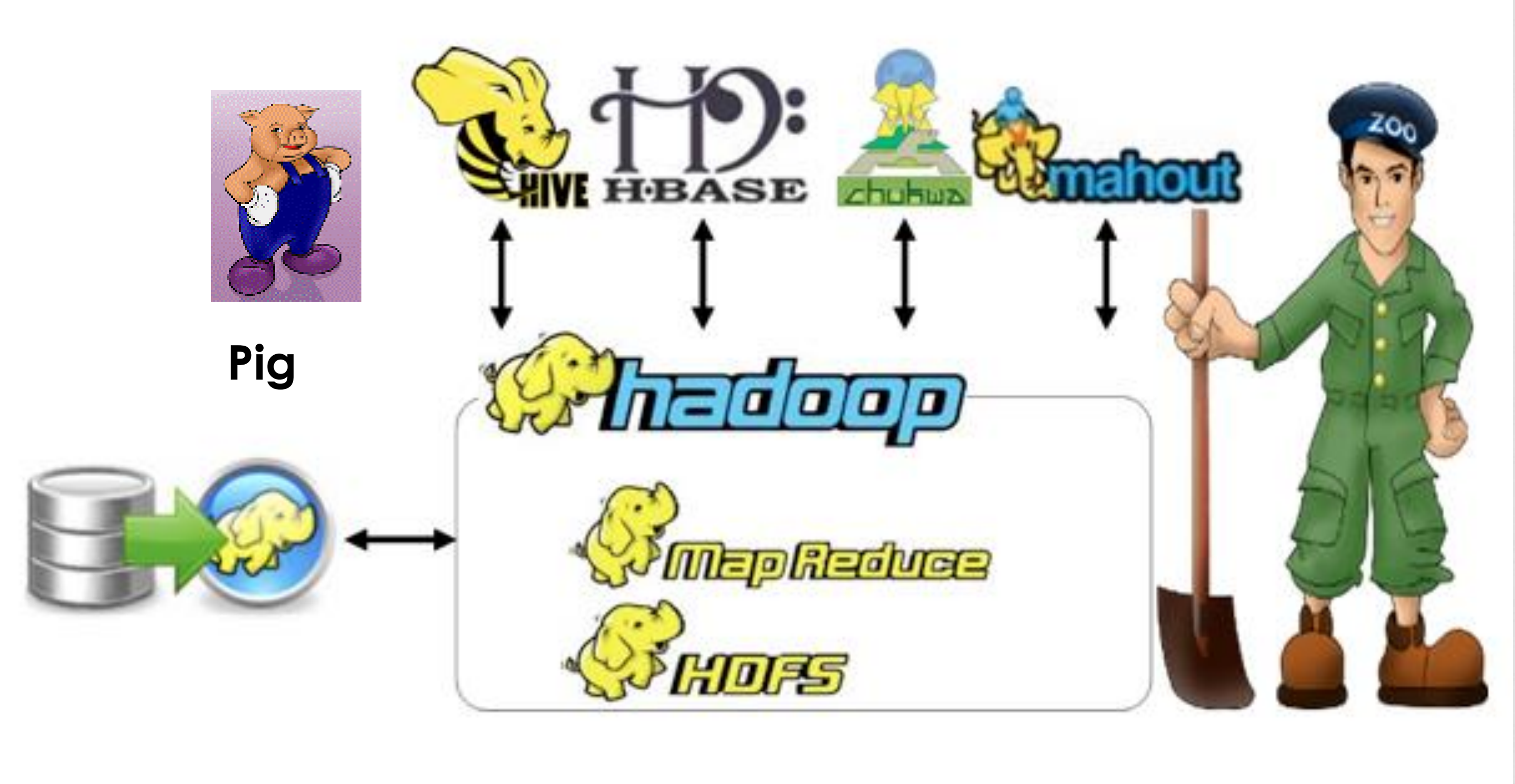


FAILURE RECOVERY

- The output of **reducers** (i.e. the final results) is written to the **distributed Hadoop File System (HDFS)** and made available to the user
- This is different than writing to local disks because it involves sending info via the network
- HDFS is a distributed file system: a unique file containing the results can be distributed across different hard disks in different computers in the network
- Additionally, the same file is **replicated** several times (usually three) for redundancy and recovery reasons
  - If a single computer can fail once every three years then, if the farm contains 1000 computers, 2.7 of them will fail every day!!



HADOOP ECOSYSTEM



HADOOP ECOSYSTEM

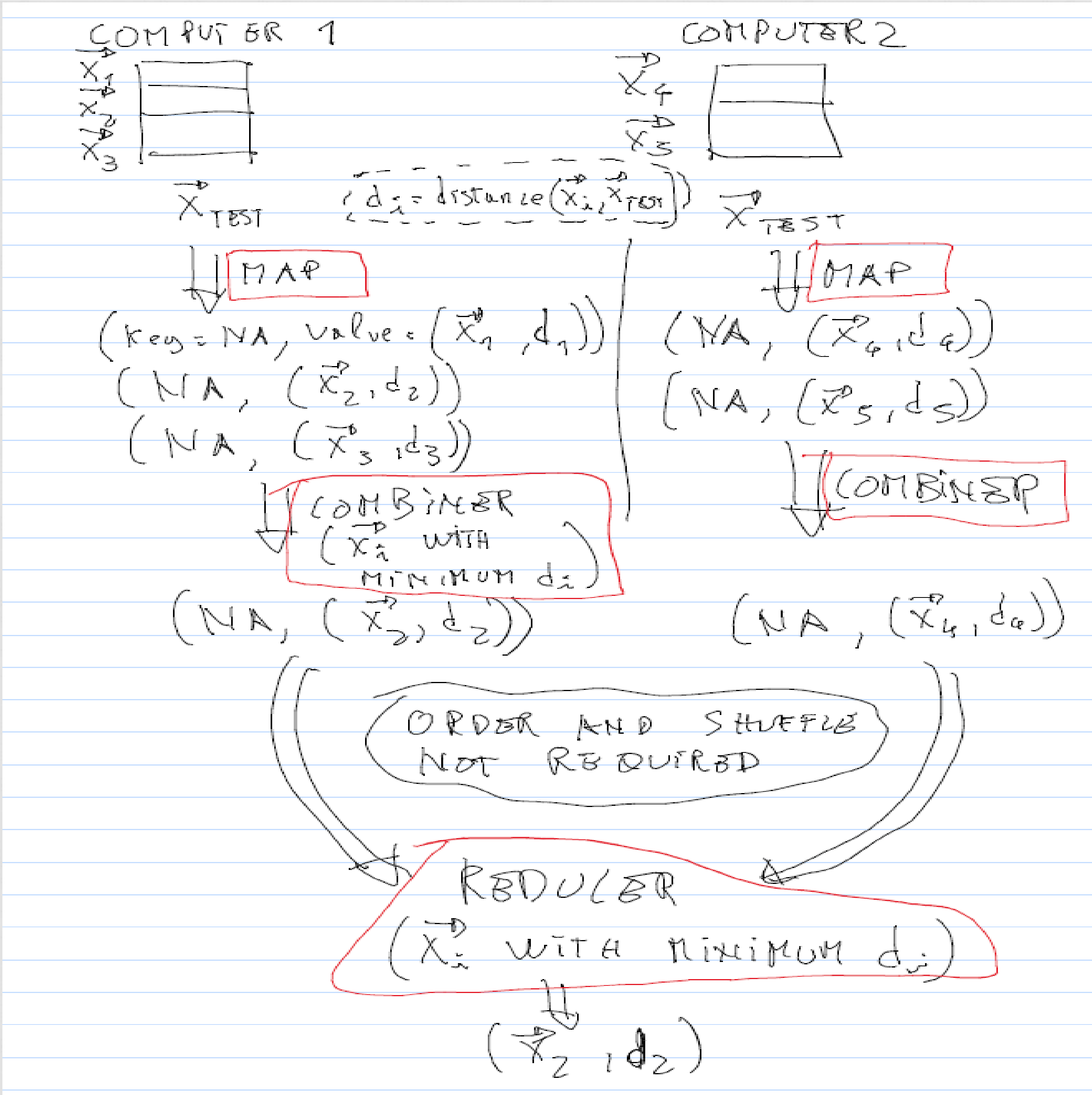
- Preferred programming language is Java (but it can be done with Python and R)
- Pig: data base platform. High level queries are translated to Mapreduce. Language: Pig-latin
- Hive: similar to Pig, but closer to SQL. Language: HiveQL
- Mahout: Mapreduce-based Machine Learning library
- Mapreduce is quickly being superceded by Apache Spark:

  "Apache Mahout, a machine learning library for Hadoop since 2009, is joining the exodus away from MapReduce. The project's community has decided to rework Mahout to support the increasingly popular Apache Spark in-memory data-processing framework, as well as the H2O engine for running machine learning and mathematical workloads at scale."
- But most ideas of Mapreduce are similar in Spark

KNN IN MAPREDUCE?

Anchalia, P. P., & Roy, K. The k-Nearest Neighbor Algorithm Using MapReduce Paradigm.

With k=1


COMPUTER 1
COMPUTER 2
$\vec{x}_1$
$\vec{x}_2$
$\vec{x}_3$
$\vec{x}_4$
$\vec{x}_5$
$\vec{x}_{TEST}$
$d_i = \text{distance}(\vec{x}_i, \vec{x}_{TEST})$
$\vec{x}_{TEST}$
MAP
MAP
(key = NA, value = ($\vec{x}_1$, d_1))
(NA, ($\vec{x}_2$, d_2))
(NA, ($\vec{x}_3$, d_3))
(NA, ($\vec{x}_4$, d_4))
(NA, ($\vec{x}_5$, d_5))
COMBINER
($\vec{x}_i$ WITH MINIMUM d_i)
COMBINER
(NA, ($\vec{x}_2$, d_2))
(NA, ($\vec{x}_4$, d_4))
ORDER AND SHUFFLE NOT REQUIRED
REDUCER
($\vec{x}_i$ WITH MINIMUM d_i)
($\vec{x}_2$, d_2)

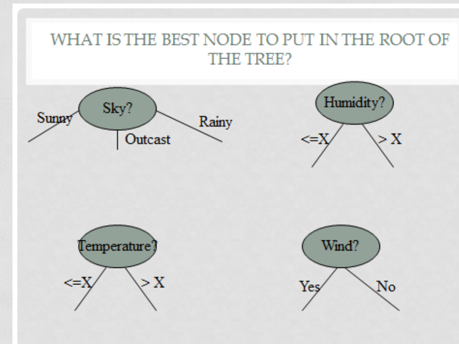
PLANET: MASSIVELY PARALLEL LEARNING OF TREE ENSEMBLES WITH MAPREDUCE

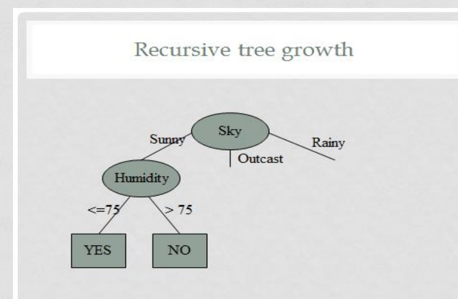
DECISION TREES WITH MAP REDUCE

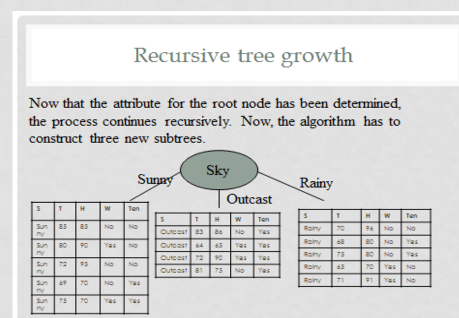
- **PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce**
- Biswanath Panda, Joshua S. Herbach, Sugato Basu, Roberto J. Bayardo
- 2009
- Google, Inc.

PARALLEL LEARNING OF A DECISION TREE

1. Learn different subtrees in different computers
   - Problem:
     - either the entire dataset is available to all computers (shared memory, or disk)
     - or the entire dataset is replicated in all computers (local disks or memory)
     - or the appropriate subsets of data are sent accross the network
2. Attribute selection: evaluate each attribute in a different computer:
   - Problem: similar to 1)
3. Evaluate different values of an attribute in different computers
   - Problem: similar to 1)







PARALLEL LEARNING OF A DECISION TREE

- Can we partition the dataset from the beginning into different computers and not move it around the network?
- Can we formulate the problem in Mapreduce terms?
- The computation of the impurity measure (e.g. entropy) can be distributed among processors

Entropy

$$\boxed{H(P) = -\sum_{Ci} p_{Ci} \log_2(p_{Ci})}$$

Average entropy (H) computation for Sky

Sky

Sunny

S	T	H	W	Ten
Sunny	85	85	No	No
Sunny	80	90	Yes	No
Sunny	72	95	No	No
Sunny	69	70	No	Yes
Sunny	75	70	Yes	Yes

"3 No, 2 Yes"

$$H = -\left((3/5)*\log_2(3/5) + (2/5)*\log_2(2/5) \right) = 0.97$$

Outcast

S	T	H	W	Ten
Outcast	83	86	No	Yes
Outcast	64	65	Yes	Yes
Outcast	72	90	Yes	Yes
Outcast	81	75	No	Yes

"0 No, 4 Yes"

$$H = -\left((0/4)*\log_2(0/4) + (4/4)*\log_2(4/4) \right) = 0$$

Rainy

S	T	H	W	Ten
Rainy	70	96	No	No
Rainy	68	80	No	Yes
Rainy	75	80	No	Yes
Rainy	65	70	Yes	No
Rainy	71	91	Yes	No

"3 No, 2 Yes"

$$H = -\left((3/5)*\log_2(3/5) + (2/5)*\log_2(2/5) \right) = 0.97$$

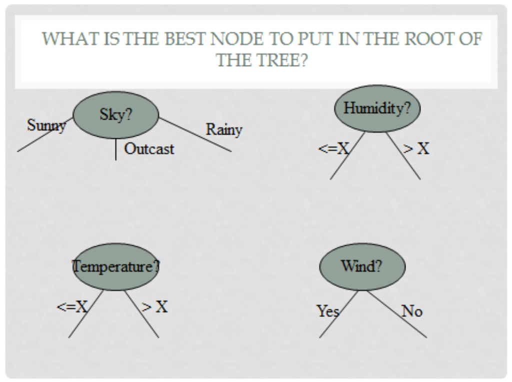
Weighted average entropy for Sky

- Weighted average entropy for Sky:
  - HP=(5/14)*0.97+(4/14)*0+(5/14)*0.97 = **0.69**
  - Note: there are14 instances in the data set

Discrete Tennis dataset

Let’s see an example for selecting the best attribute for the root node

Sky	Temperature	Humidity	Wind	Tennis
Sunny	Cold	Normal	No	Yes
Sunny	Moderate	Normal	Yes	Yes
Sunny	Hot	High	No	No
Overcast	Cold	Normal	Yes	Yes
Sunny	Moderate	High	No	No
Sunny	Hot	High	Yes	No
Overcast	Hot	High	No	Yes
Overcast	Moderate	High	Yes	Yes
Overcast	Hot	Normal	No	Yes
Rainy	Moderate	High	No	Yes
Rainy	Cold	Normal	Yes	No
Rainy	Cold	Normal	No	Yes
Rainy	Moderate	High	Yes	No
Rainy	Moderate	Normal	No	Yes



- In order to compute entropy for each attribute and attribute value, it is necessary to compute the following tables

Sky	Yes	No
Sunny	**2**	3
Overcast	4	0
Rainy	3	2

Temperature	Yes	No
Hot	2	2
Moderate	4	2
Cold	**3**	1

Humidity	Yes	No
High	**3**	4
Normal	6	1

Tennis	Yes	No
	9	5

Wind	Yes	No
Yes	**3**	3
No	6	2

Let's suppose we have **three** computers, with data distributed among them:

Sky	Temperature	Humidity	Wind	Tennis
Sunny	Cold	Normal	No	Yes
Sunny	Moderate	Normal	Yes	Yes
Sunny	Hot	High	No	No
Overcast	Cold	Normal	Yes	Yes
Sunny	Moderate	High	No	No

Sky	Temperature	Humidity	Wind	Tennis
Sunny	Hot	High	Yes	No
Overcast	Hot	High	No	Yes
Overcast	Moderate	High	Yes	Yes
Overcast	Hot	Normal	No	Yes
Rainy	Moderate	High	No	Yes

Sky	Temperature	Humidity	Wind	Tennis
Rainy	Cold	Normal	Yes	No
Rainy	Cold	Normal	No	Yes
Rainy	Moderate	High	Yes	No
Rainy	Moderate	Normal	No	Yes

FIRST PARTITION (MAP)

Sky	Temperature	Humidity	Wind	Tennis
Sunny	Cold	Normal	No	Yes
Sunny	Moderate	Normal	Yes	Yes
Sunny	Hot	High	No	No
Overcast	Cold	Normal	Yes	Yes
Sunny	Moderate	High	No	No

MAP / COMBINER

Sky	Yes	No
Sunny	**2**	2
Overcast	2	0
Rainy	0	0

Temperature	Yes	No
Hot	0	1
Moderate	1	1
Cold	**2**	0

Humidity	Yes	No
High	**0**	2
Normal	3	0

Wind	Yes	No
Yes	**2**	0
No	1	2

Tennis	Yes	No
	3	2

SECOND PARTITION (MAP)

Sky	Temperature	Humidity	Wind	Tennis
Sunny	Hot	High	Yes	No
Overcast	Hot	High	No	Yes
Overcast	Moderate	High	Yes	Yes
Overcast	Hot	Normal	No	Yes
Rainy	Moderate	High	No	Yes

MAP / COMBINER

Sky	Yes	No
Sunny	**0**	1
Overcast	3	0
Rainy	1	0

Temperature	Yes	No
Hot	2	1
Moderate	2	0
Cold	**0**	0

Humidity	Yes	No
High	**3**	1
Normal	1	0

Tennis	Yes	No
	4	1

Wind	Yes	No
Yes	**1**	1
No	3	0

THIRD PARTITION (MAP)

Sky	Temperature	Humidity	Wind	Tennis
Rainy	Cold	Normal	Yes	No
Rainy	Cold	Normal	No	Yes
Rainy	Moderate	High	Yes	No
Rainy	Moderate	Normal	No	Yes

MAP / COMBINER

Sky	Yes	No
Sunny	**0**	0
Overcast	0	0
Rainy	2	2

Temperature	Yes	No
Hot	0	0
Moderate	1	1
Cold	**1**	1

Humidity	Yes	No
High	**0**	1
Normal	2	1

Wind	Yes	No
Yes	**0**	2
No	2	0

Tennis	Yes	No
	2	2

MAP/COMBINER

Sky	Yes	No
Sunny	**2**	2
Overcast	2	0
Rainy	0	0

Temp	Yes	No
Hot	0	1
Moderate	1	1
Cold	**2**	0

Humidity	Yes	No
High	**0**	2
Normal	3	0

Wind	Yes	No
Yes	**2**	0
No	1	2

Tennis	Yes	No
	3	2

Sky	Yes	No
Sunny	**0**	1
Overcast	3	0
Rainy	1	0

Temp	Yes	No
Hot	2	1
Moderate	2	0
Cold	**0**	0

Humidity	Yes	No
High	**3**	1
Normal	1	0

Wind	Yes	No
Yes	**1**	1
No	3	0

Tennis	Yes	No
	4	1

Sky	Yes	No
Sunny	**0**	0
Overcast	0	0
Rainy	2	2

Temp	Yes	No
Hot	0	0
Moderate	1	1
Cold	**1**	1

Humidity	Yes	No
High	**0**	1
Normal	2	1

Wind	Yes	No
Yes	**0**	2
No	2	0

Tennis	Yes	No
	2	2

Sky	Yes	No
Sunny	**2**	3
Overcast	4	0
Rainy	3	2

Temperature	Yes	No
Hot	2	2
Moderate	4	2
Cold	**3**	1

Humidity	Yes	No
High	**3**	4
Normal	6	1

Wind	Yes	No
Yes	**3**	3
No	6	2

Tennis	Yes	No
	9	5

REDUCE

+

MAP & REDUCE

```
def mapper(key = (attribute, atr_value, class), value=NA)
   # Example: mapper((“Sky”, “Sunny”, “Yes”), NA)
   #    => result = ((“Sky”, “Sunny”, “Yes”), 1)
    emit(key=(atribute, atr_value, class), value = 1)
```

```
def reducer(key=(attribute, atr_value, class), value)
  # Example: reducer((“Humidity”, “High”, “No”), [2, 1, 1])
  #   => result = ((“Humidity”, “High”, “No”), 4)
  emit(key=(atribute, atr_value, class), sum(value))
```

MAP & COMBINER & REDUCE

```
def mapper(key = (attribute, atr_value, class), value=NA)
  # Example: mapper((“Sky”, “Sunny”, “Yes”), NA)
  #    => result = ((“Sky”, “Sunny”, “Yes”), 1)
   emit(key=(atribute, atr_value, class), value = 1)
```

```
def combiner(key=(attribute, atr_value, class), value)
  # Example: reducer((“Humidity”, “High”, “No”), [1, 1])
  #   => result = ((“Humidity”, “High”, “No”), 2)
  emit(key=(atribute, atr_value, class), sum(value))
```

```
def reducer(key=(attribute, atr_value, class), value)
  # Example: reducer((“Humidity”, “High”, “No”), [2, 1])
  #   => result = ((“Humidity”, “High”, “No”), 4)
```

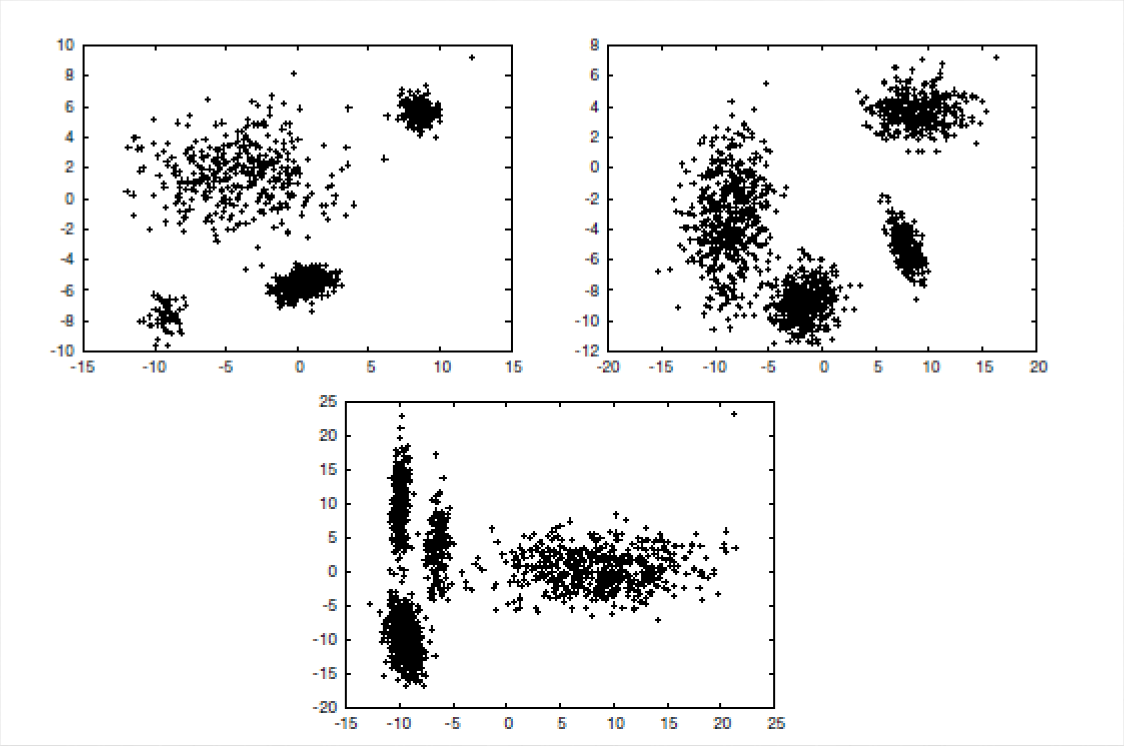
K-MEANS IN MAPREDUCE

Clustering

- Unsupervised Machine Learning (no label attribute)
- Find the grouping structure in data by locating “clusters”:
  - High similarity between instances in the cluster
  - Low similarity between instances of different clusters

Partitional clustering

- Distribute data into K clusters. K is a parameter
- Ill-defined problem: are clusters defined by closeness or by “contact”?



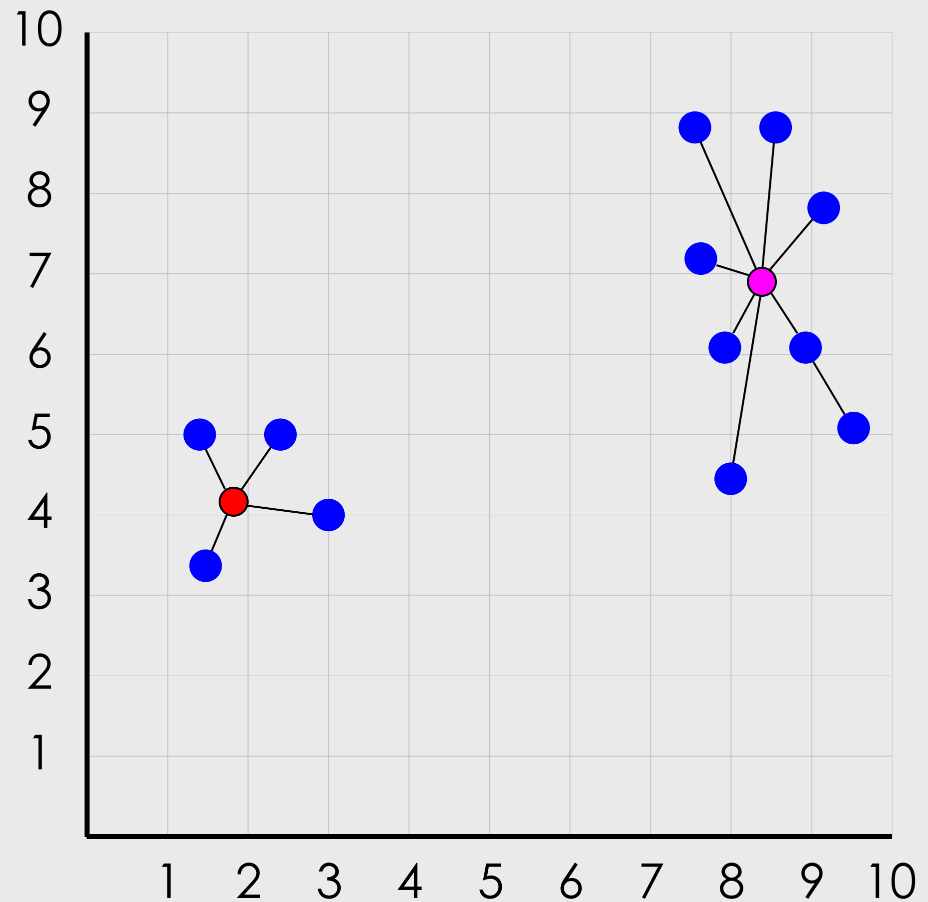
Quadratic error

It can be formulated as a minimization problem: locate k prototypes so that a loss function is minimized

$$se_{K_i} = \sum_{j=1}^{m} \|t_{ij} - C_k\|^2$$

$$se_K = \sum_{j=1}^{k} se_{K_j}$$

Loss or error function



Algorithm *k-means (k)*

1. Initialize the location of the k prototypes k_j

   (usually, randomly)

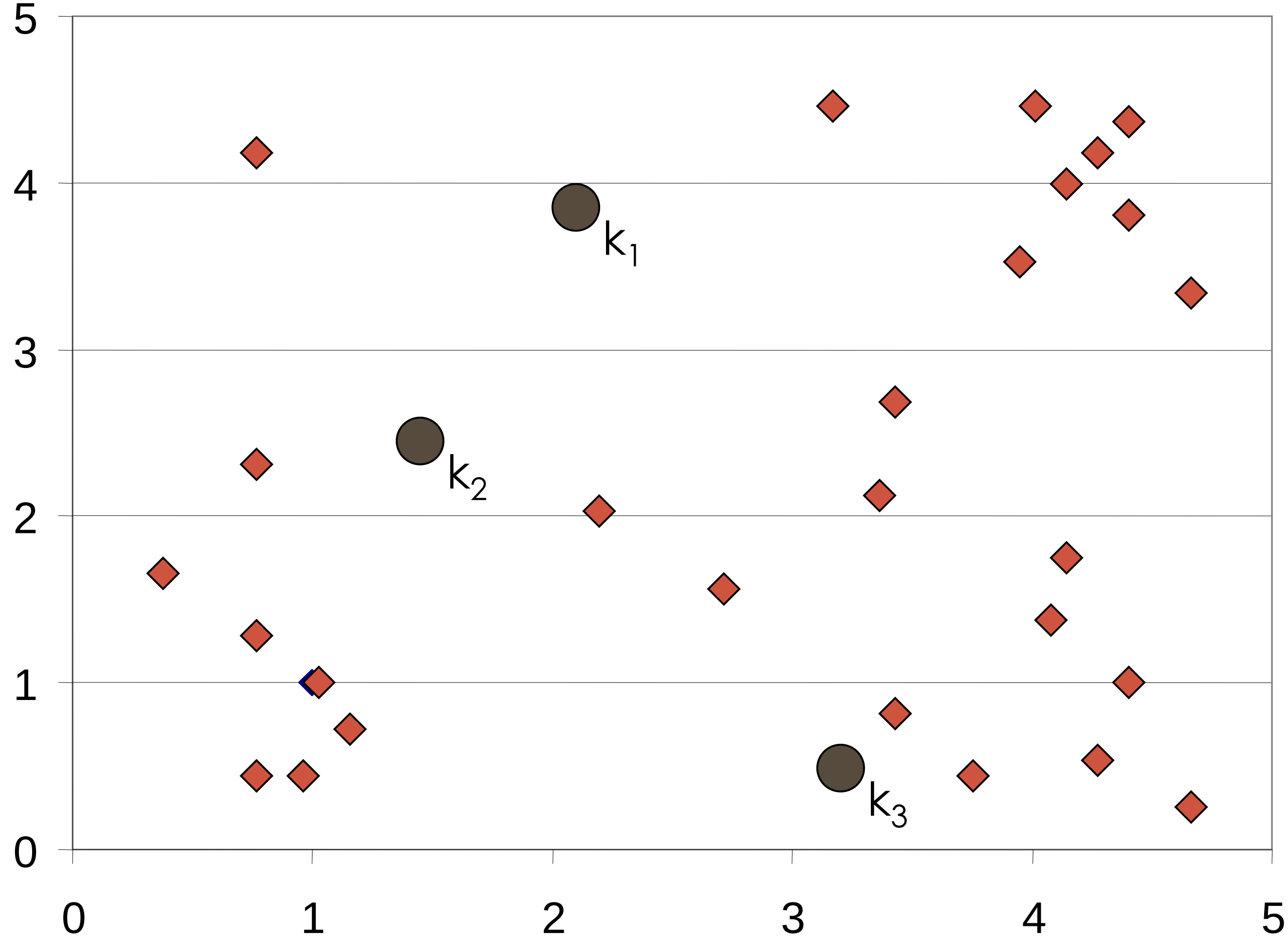
2. Assign each instance x_i to its closest prototype

   (usually, closeness = Euclidean distance).

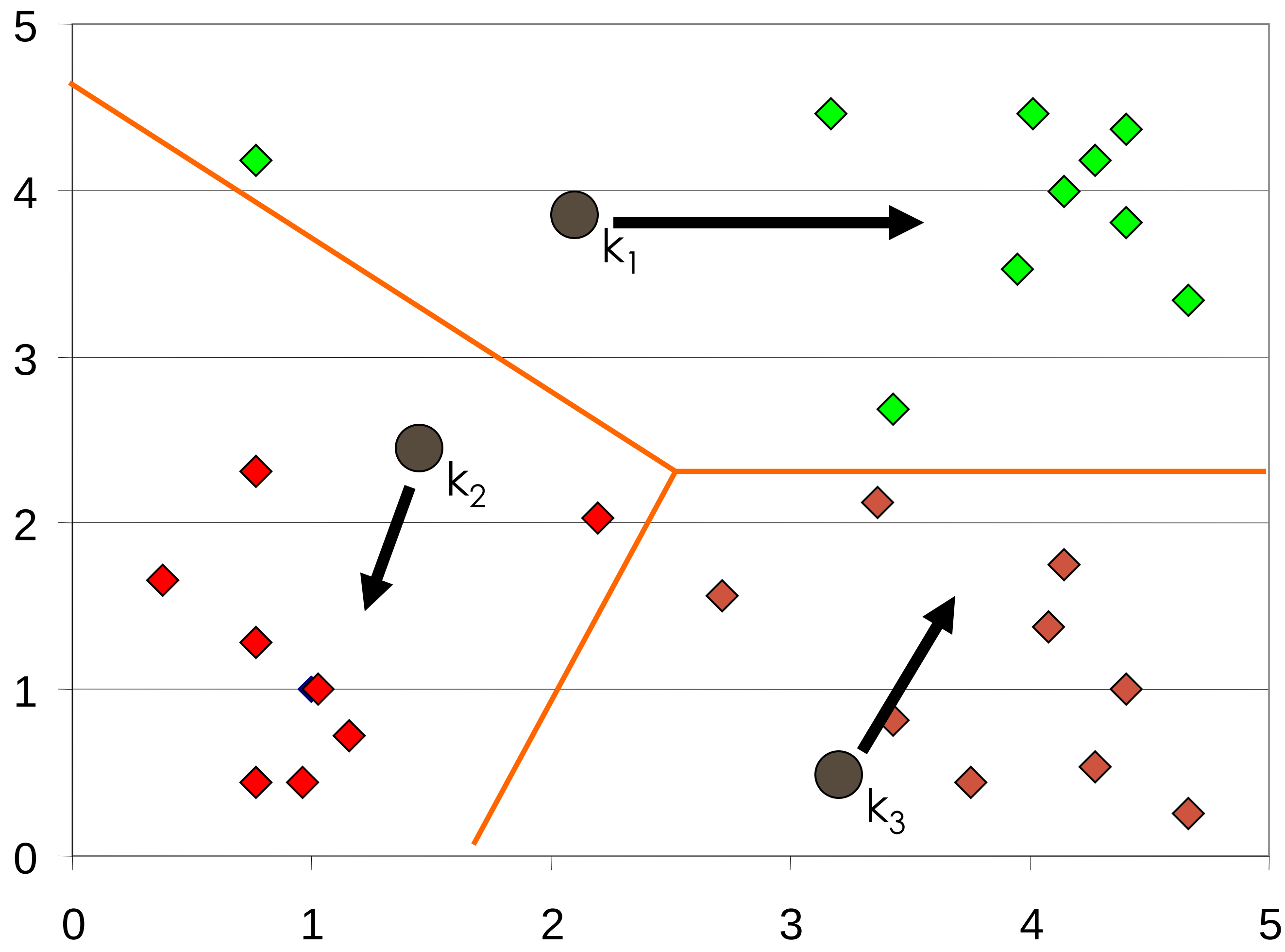
3. Update the location of prototypes k_j as the average of the instances x_i assigned to each cluster.

4. Go to 2, until clusters do not change

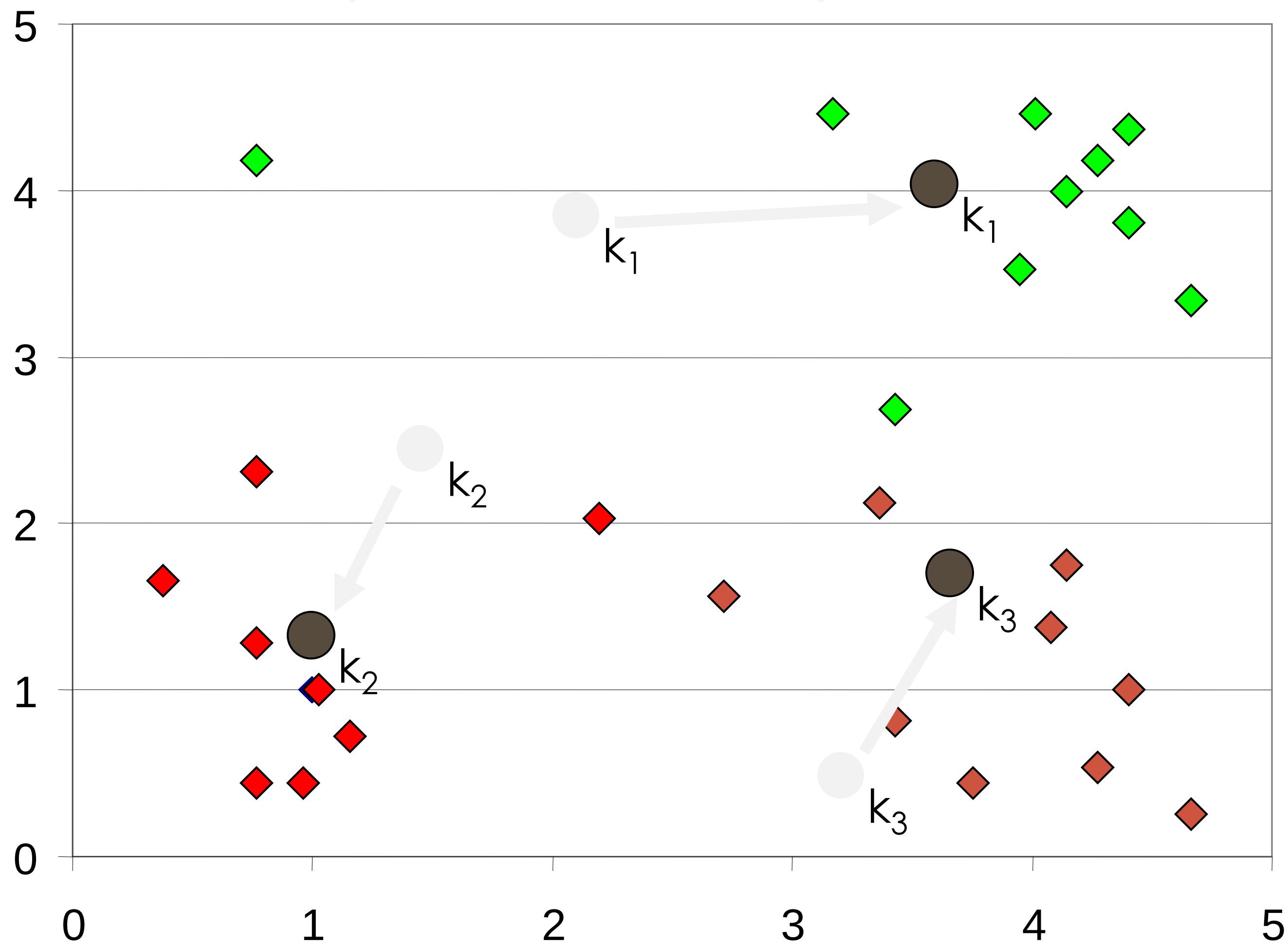
RANDOM INITIALIZATION OF PROTOTYPES



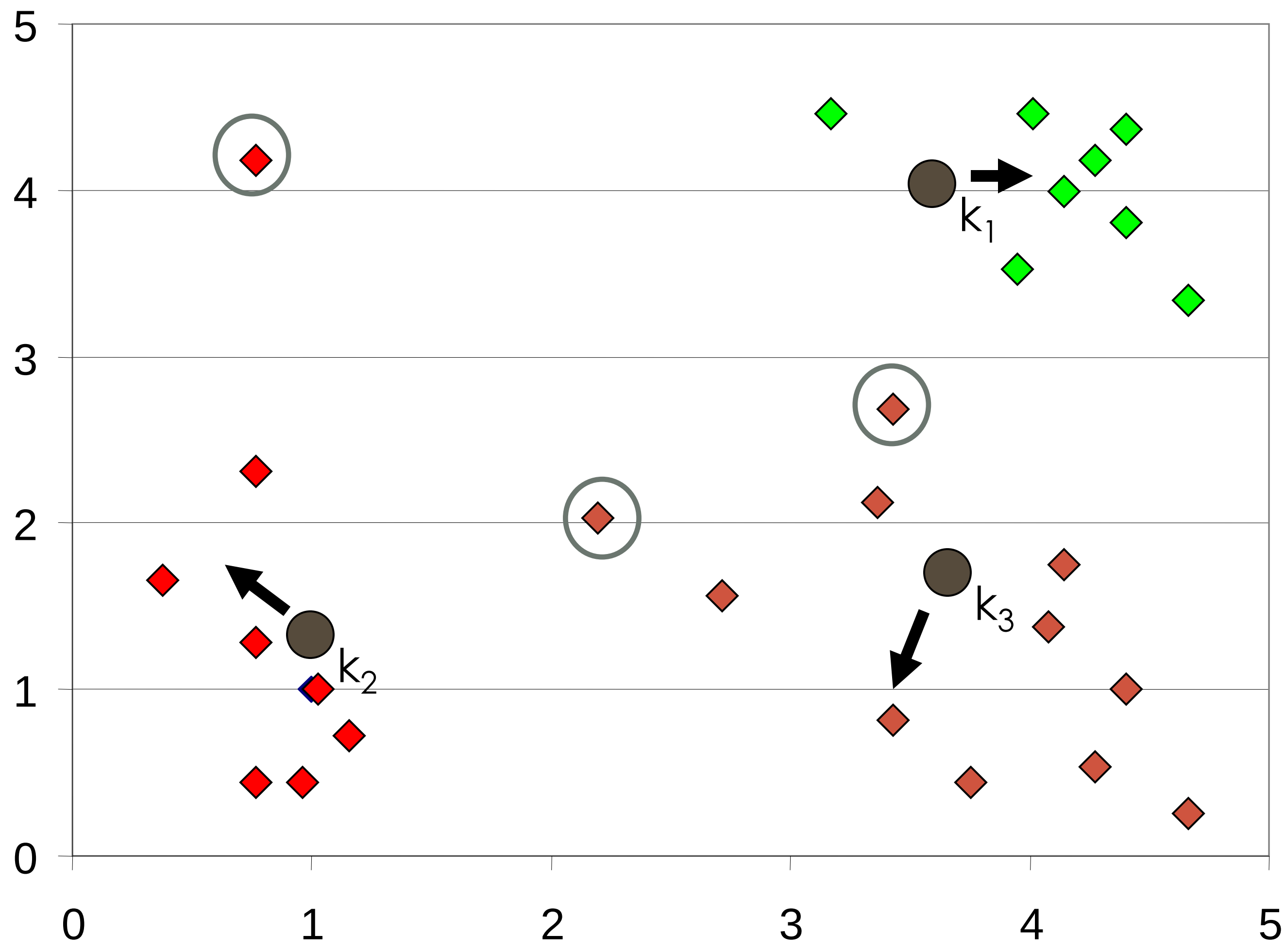
ASSIGNING INSTANCES TO CLOSEST PROTOTYPE



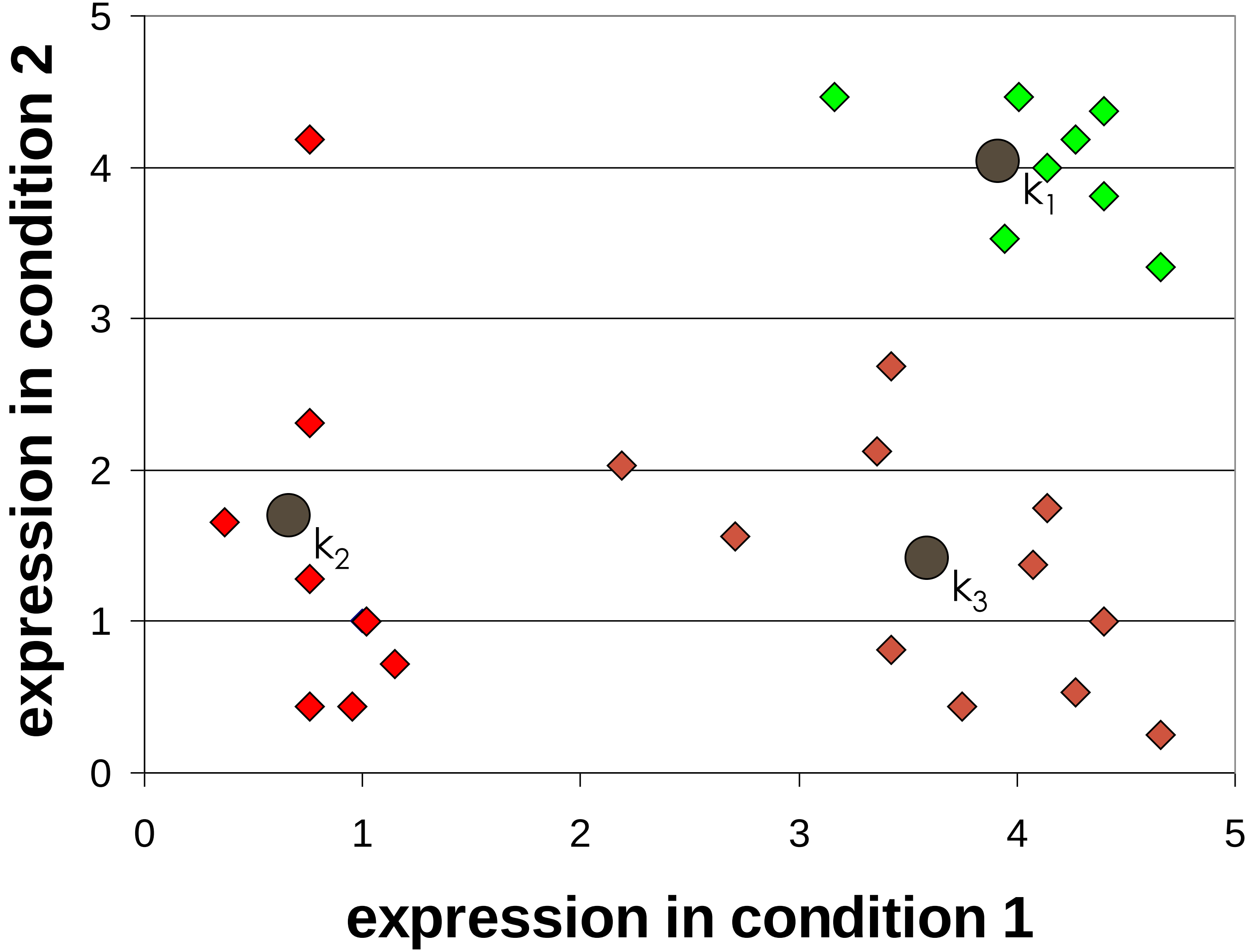
UPDATE PROTOTYPES (AVERAGE)



ASSIGNING INSTANCES TO CLOSEST PROTOTYPE

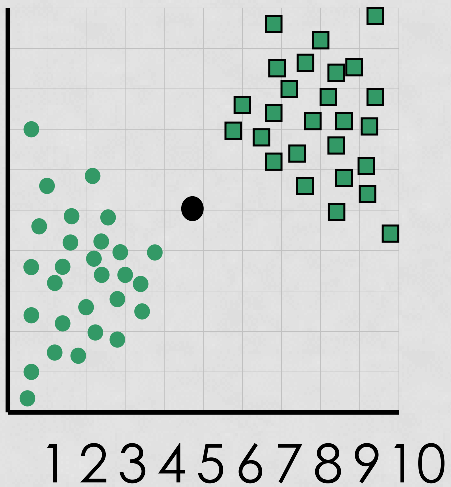


UPDATE PROTOTYPES (AVERAGE)

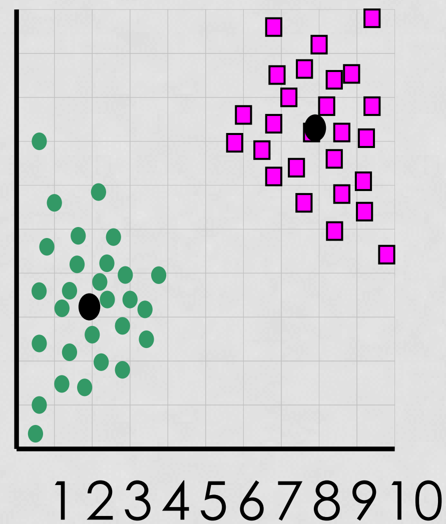


¿K?

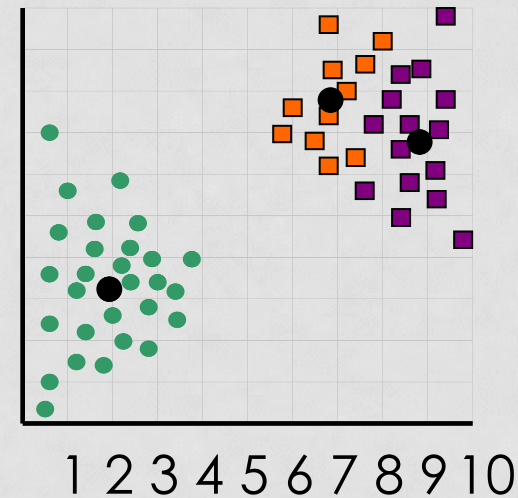
k = 1 Error= 873.0

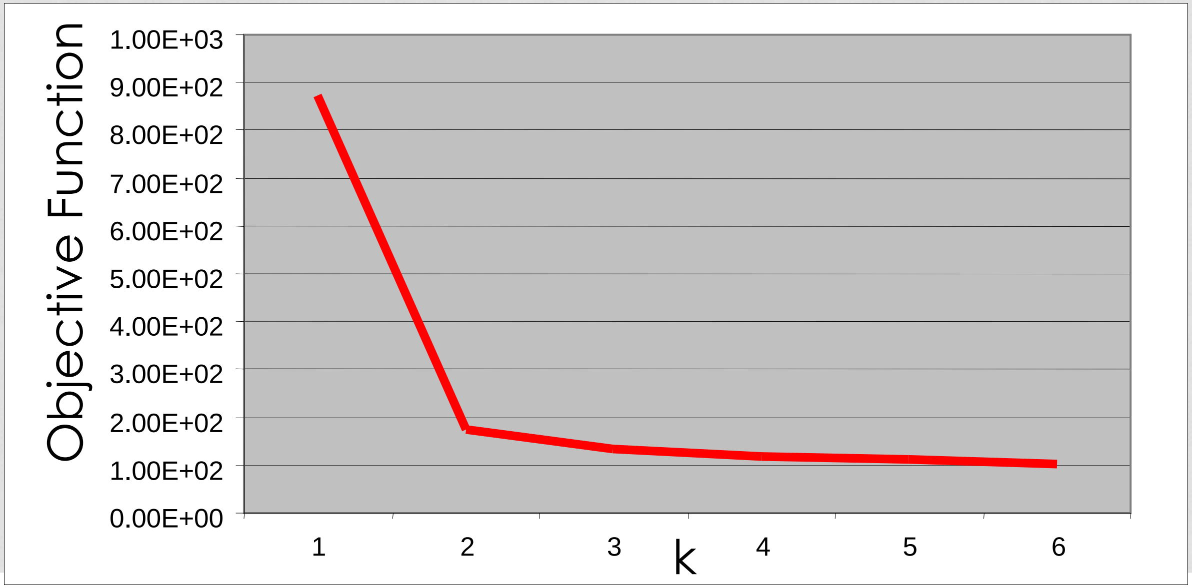


k = 2, Error = 173.1



k = 3, Error = 133.6





- How to program k-means in mapreduce?
- Remember that the goal is that instances remain in their initial location.

Algorithm *k-means (k)*

1. Initialize the location of the k prototypes k_j
2. Assign each instance x_i to its closest prototype
3. Update the location of prototypes k_j as the average of the instances x_i assigned to each cluster.
4. Go to 2, until clusters do not change

- Step 2 can be done for each instance independently of other instances. We assume that prototypes are few and can be sent to each computer through the network very fast.

Algorithm *k-means (k)*

1. Initialize the location of the k prototypes k_j

2. MAP = Assign each instance x_i to its closest prototype

3. Update the location of prototypes k_j as the average of the instances x_i assigned to each cluster.

4. Go to 2, until clusters do not change

- Step 4 updates prototypes by computing the average of their instances

Algorithm *k-means (k)*

1. Initialize the location of the k prototypes k_j
2. Assign each instance x_i to its closest prototype
3. REDUCE = Update the location of prototypes k_j as the average of the instances x_i assigned to each cluster.
4. Go to 2, until clusters do not change

MAPREDUCE FOR K-MEANS

def mapper(key, value) = > (key, list of values)

    # key = instance number (irrelevant)
    # value = instance **xi**
    key' = num. prototype
    value' = instance **xi**
    emit(key', value')

def reducer(key, list of values) => result

    # key = instance number
    # value = instance **xi**
    result = average of **xi**

EFFICIENCY?

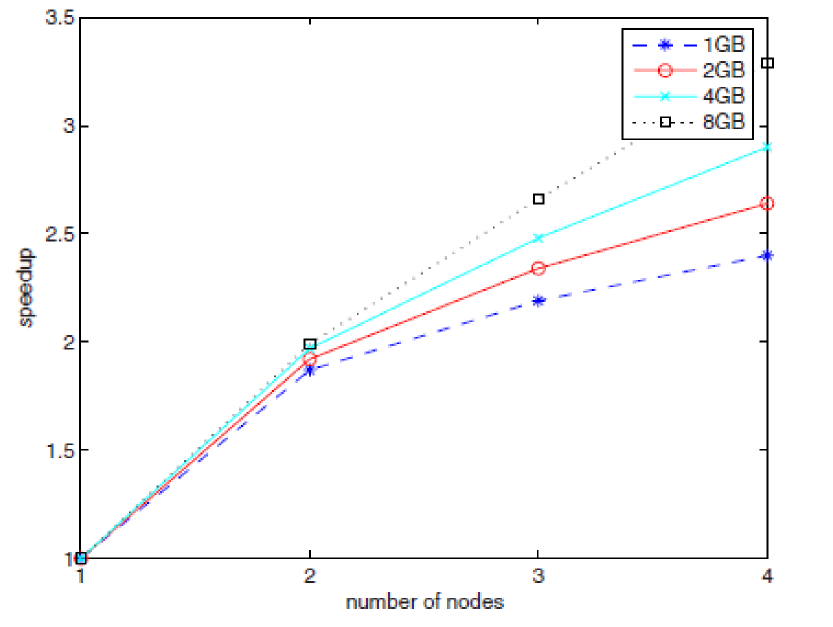
- If **map** output is (num. prototype, xi), processing of instances is not actually local, because all data must travel from **map** computers to reduce computers.
- Solution: use **combiner** functions, that perform a reduce locally: **map** outputs are grouped by key and the sum of instances is computed. **Reduce** functions are sent the sum of (local) instances and the number of (local) instances: (num. Prototype, sum of instances, num. of instances)
- **Reduce** functions just add the partial sums of instances and divide by the total number of instances

MAPREDUCE FOR K-MEANS

```
def mapper(key, value) = > (key, list of values)
    # key = instance number (irrelevant)
    # value = instance xi
    key' = num. prototype
    value' = instance xi
    Emit(key', value')

def combiner(key, list of values) => (key, value)
    # key = instance number
    # list fo values = instances xi
    value  = [sum of list-of-values, length of list-of-values]

def reducer(key, list of (sum, length) ) => result
    # key = num of  prototype
    # value = [centroide parcial, num.de.valores usados para calcular el centroide parcial]
    result = sum of list-of-values / sum of length
```

REFERENCE

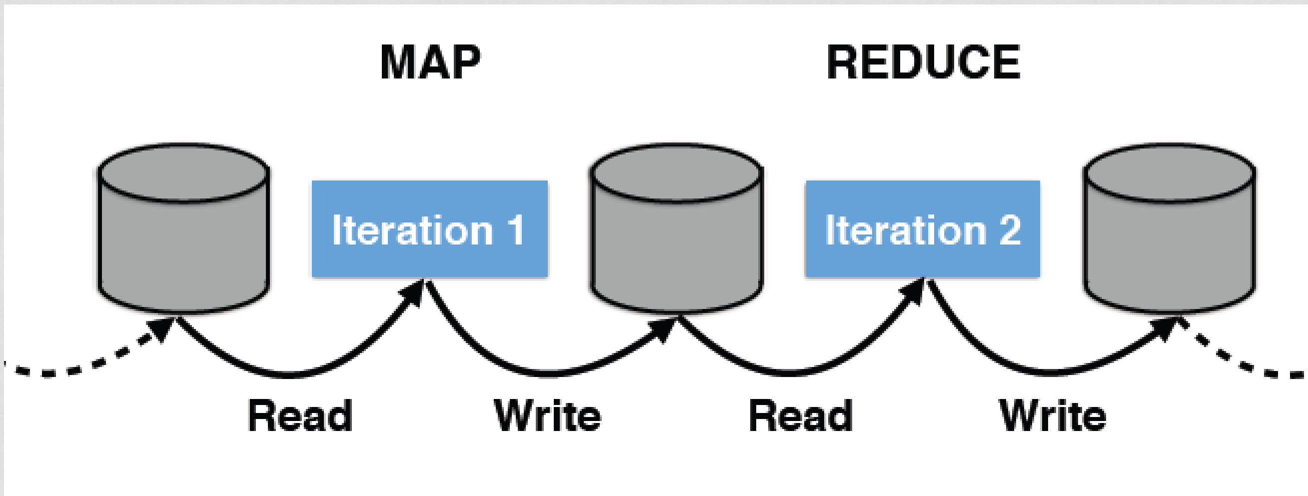
- Weizhong Zhao1, Huifang Ma1, and Qing He. **Parallel *K*-Means Clustering Based on MapReduce. *Cloud Computing*. 2009.**

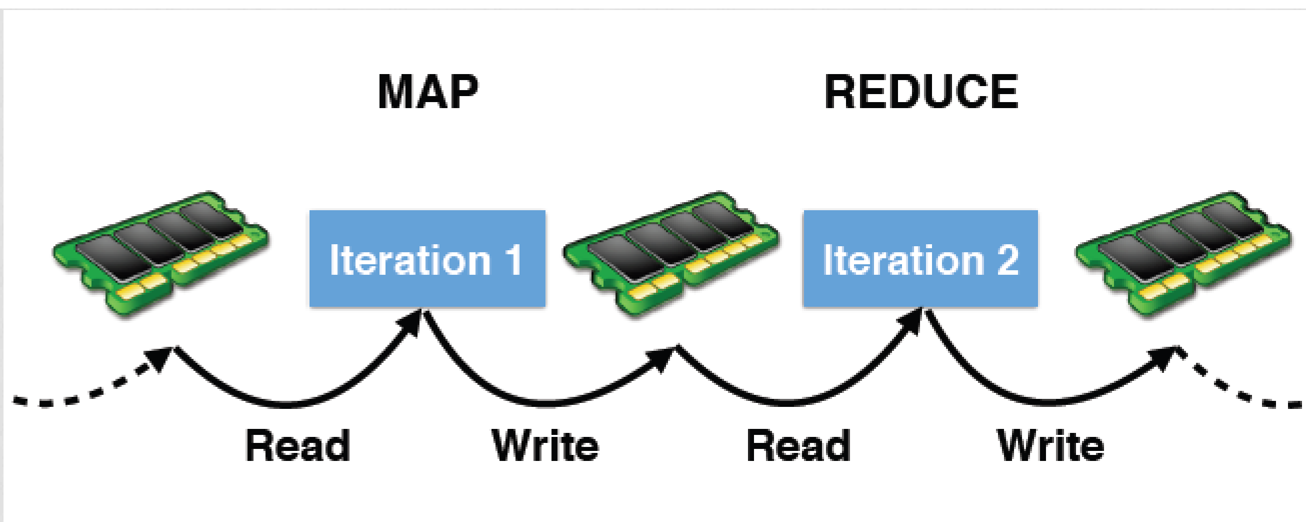


(a) Speedup

SPARK

HADOOP LIMITATIONS





SPARK ECOSYSTEM