Computational Frameworks

MapReduce

OUTLINE

1. MapReduce
2. Partitioning

MapReduce

Motivating scenario

At the dawn of the big-data era (early 2000's), the following scenario was rapidly emerging

- In a wide spectrum of domains there was an increasing need to analyze large amounts of data.
- Available tools and commonly used platforms could not practically handle very large datasets.
  - Algorithms with high polynomial complexities were unfeasible, and platforms had limited main memory.
  - In extreme cases, even *touching all data items* would prove quite time consuming. (E.g., $\simeq 50 \cdot 10^9$ *web pages of about 20KB each* $\rightarrow$ *1000TB of data*.)
- The use of powerful computing systems, which has been confined until then to a limited number of applications – typically from computational sciences (*physics, biology, weather forecast, simulations*) – was becoming necessary for a much wider array of applications.

Motivating scenario

Powerful computing systems, which feature multiple processors, multiple storage devices, and high-speed communication networks, pose several problems

- They are costly to buy and to maintain, and they become rapidly obsolete.
- Fault-tolerance becomes serious issue: a large number of components implies a low *Mean-Time Between Failures* (MTBF).
- Developing software that effectively benefits from parallelism requires *sophisticated programming skills*.

System with N components $C_1\ C_2\ \dots\ C_N$ that fail independently with probability p in one time unit

$p = \text{Prob}(C_i \text{ fails at time unit } t)$ for any i and t

$1-(1-p)^N = \text{Prob}(\exists C_i \text{ that fails at time unit } t)$ for any t

$X =$ # time units before the first/next failure

If failures occur independently in different time units

$\Rightarrow X$ has a geometric distribution

$$E[X] = \frac{1}{1-(1-p)^N} \xrightarrow[N \to +\infty]{} 1$$

MapReduce

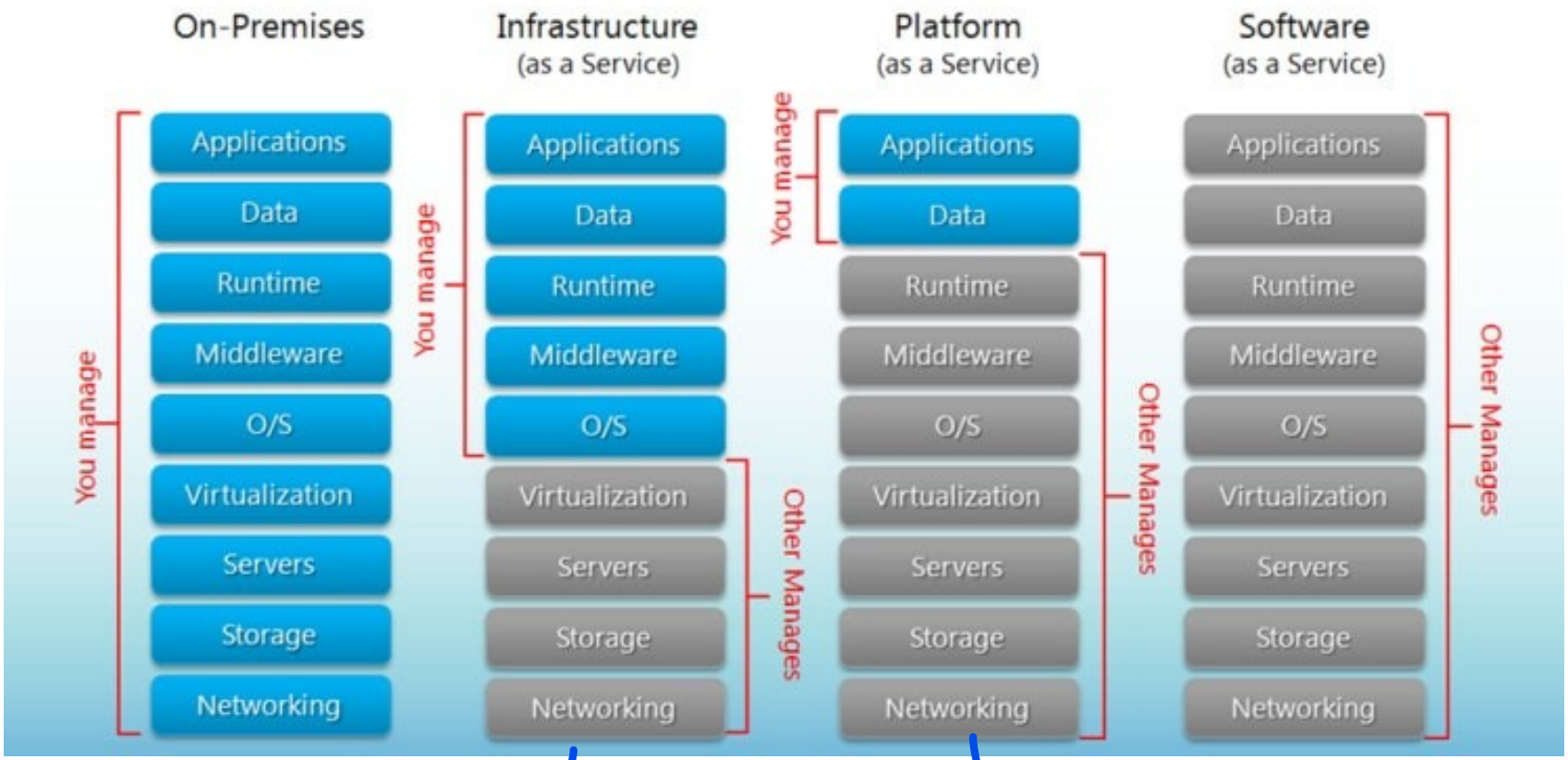
- Introduced by Google in 2004 as a programming framework for big data processing on distributed platforms.
- Its original formulation (see [DG08]) contained
  - A programming model
  - An implementation tailored towards a cluster-based computing environment.
- Since then, MapReduce implementations have been employed on clusters of commodity processors and cloud infrastructures for a wide array of big-data applications
- **Main features:**
  - Data centric view
  - Inspired by functional programming (map, reduce functions)
  - Ease of algorithm/program development. Messy details (e.g., task allocation; data distribution; fault-tolerance; load-balancing) are hidden to the programmer

Platforms

Typically, MapReduce applications run on

- Clusters of commodity processors (On-premises)
- Platforms from cloud providers (Amazon AWS, Microsoft Azure)
  - IaaS (*Infrastructure as a Service*): provides the users computing infrastructure and physical (or virtual) machines. This is the case of CloudVeneto which provides us a cluster of virtual machines.
  - PaaS (*Platform as a Service*): provides the users computing platforms with OS; execution environments, etc.

Platforms

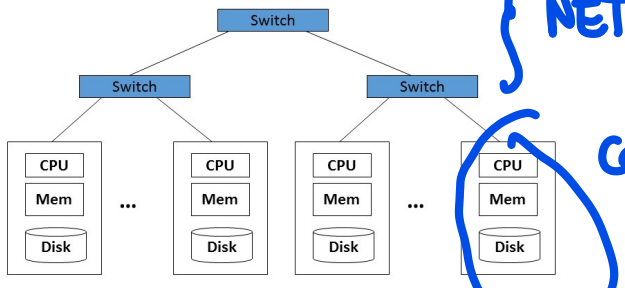


Our cluster on CloudVeneto

Hw3

Typical cluster architecture

- Racks of 16-64 compute nodes (commodity hardware), connected (within each rack and among racks) by fast switches (e.g., 10 Gbps Ethernet)



- Distributed File System (DFS)
  - Files divided into *chunks* (e.g., 64MB per chunk)
  - Each chunk replicated (e.g., 2x o 3x) with replicas in different nodes to ensure fault-tolerance.
  - Examples: Google File System (GFS); Hadoop Distributed File System (HDFS)

MapReduce-Hadoop-Spark

Several software frameworks have been proposed to support MapReduce programming. For example:

- Apache Hadoop: 

most popular MapReduce implementation (*often used as synonymous of MapReduce*) from which an entire ecosystem of alternatives has stemmed, aimed at improving its (initially) very poor performance. The Hadoop Distributed File System is still widely used.

- Apache Spark (*learned in this course*): one of the most popular and widely used frameworks for big-data applications. It supports the implementation of MapReduce computations, but provides a much richer programming environment. It uses the HDFS.

In this course we use

* MapReduce as a CONCEPTUAL MODEL for high-level algorithm design and analysis
* Spark as a PROGRAMMING FRAMEWORK to implement MapReduce algorithms

MapReduce computation

A MapReduce computation can be viewed as a sequence of rounds.

A round transforms a set of key-value pairs into another set of key-value pairs (*data centric view* !), through the following two phases

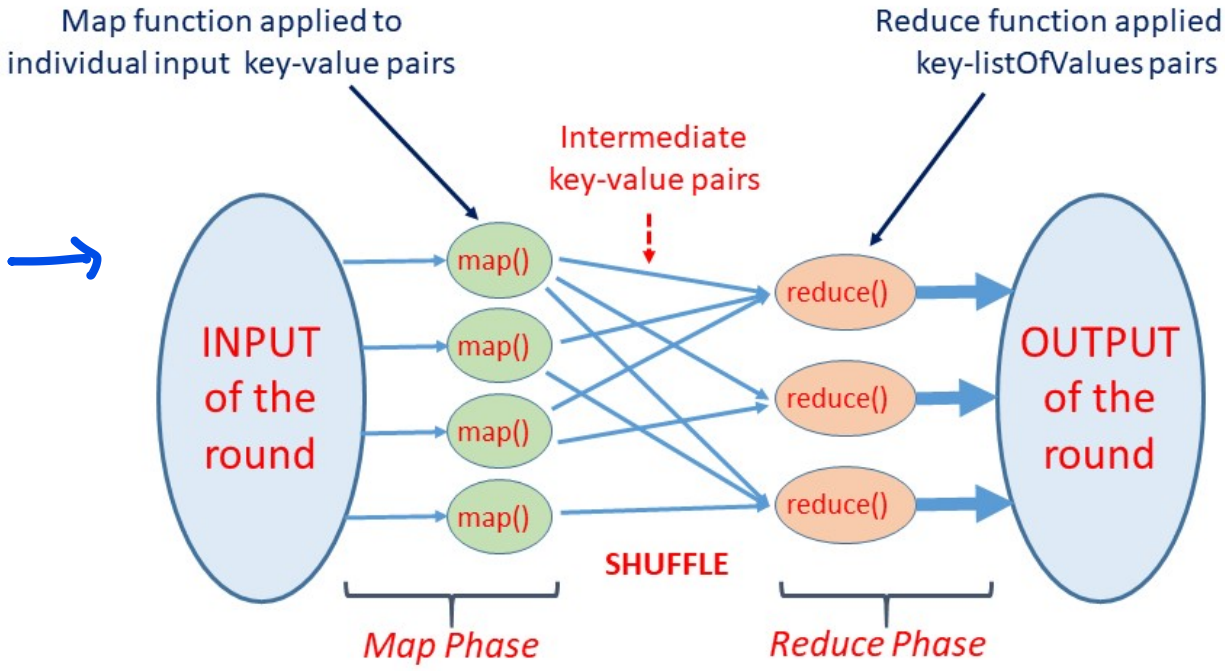
- Map phase: for each input key-value pair separately, a user-specified map function is applied to the pair and produces ≥ 0 other key-value pairs, sometimes called intermediate pairs.
- Reduce phase: for each key k separately, a user-specified reduce function is applied to (k, L_k), where L_k is the list of values of intermediate pairs with key k, and produces producing ≥ 0 key-value pairs, which is the output of the round.

Remark: between the two phases, a SHUFFLE is performed where the intermediate pairs are grouped by key.

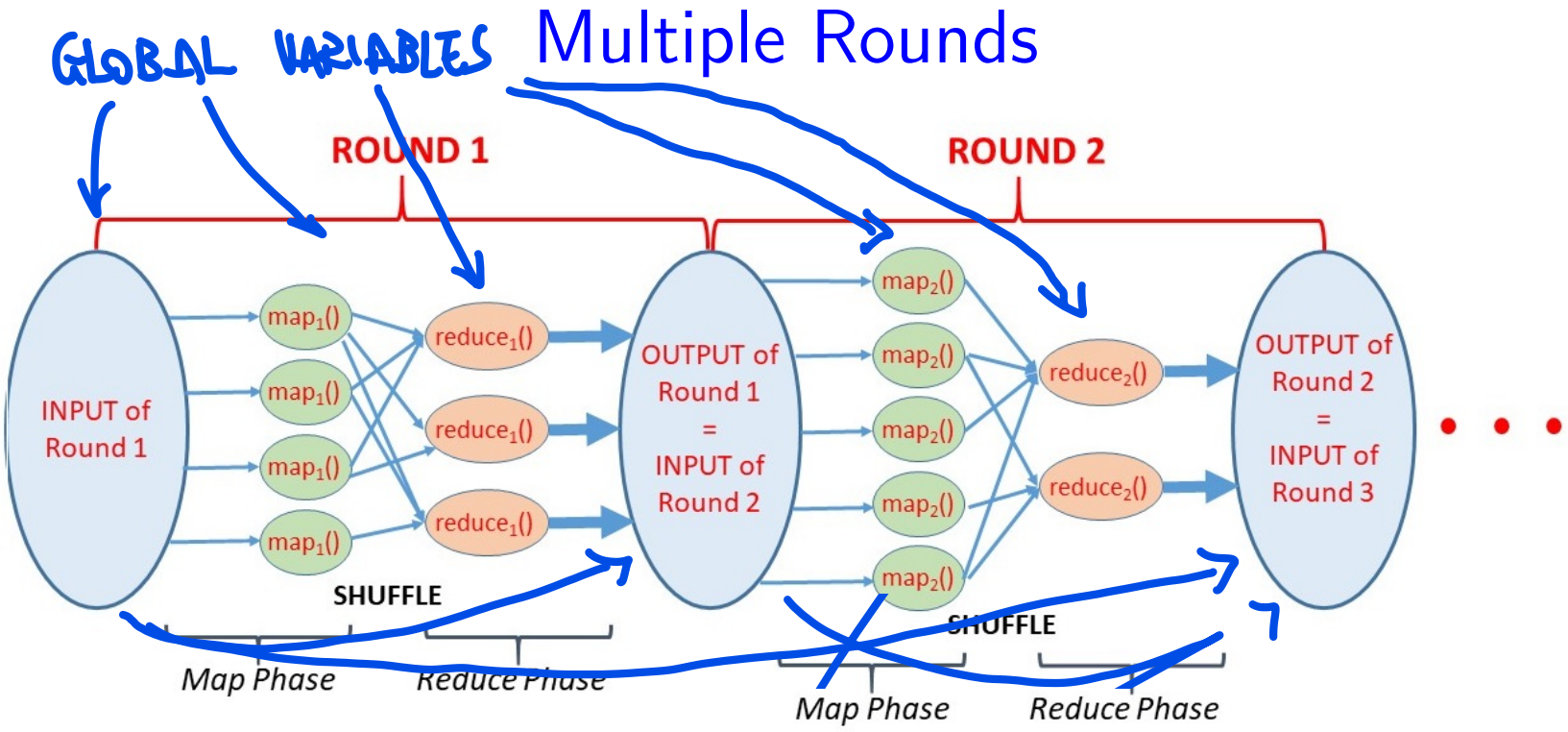
TERMINOLOGY : "reducer" = an application of the reduce function to a pair (k, L_k)

One Round

key-value pairs →



Multiple Rounds



In practice, a bit more flexibility is used:

- the input of a round may comprise a subset of the initial input data and/or subsets of the output of previous rounds, if any.
- some data can be used as global variables known to all processing elements that carry out the computation. Programming frameworks such as Spark make provisions in this sense.

Why key-value pairs?

The use of key-value pairs in MapReduce seems an unnecessary complication, and one might be tempted to regard individual data items simply as *objects* belonging to some domain, without imposing a distinction between keys and values.

Justification. MapReduce is data-centric and focuses on data trasformations which are independent of where data actually reside. The keys are needed as addresses to reach the objects, and as labels to define the groups in the reduce phases.

Practical considerations.

- One should choose the key-value representations in the most convenient way. The domains for key and values may change from one round to another.
- Programming frameworks such as Spark, while containing explicit provisions for handling key-value pairs, allow also to manage data without the use of keys. In this case, internal addresses/labels, not explicitly visible to the programmer, are used.

Specification of a MapReduce algorithm

A *MapReduce (MR) algorithm* should be specified so that

- The input and output of the algorithm is clearly defined
- The sequence of rounds executed for any given input instance is unambiguously implied by the specification
- For each round the following aspects are clear
  - input, intermediate and output sets of key-value pairs
  - functions applied in the map and reduce phases.
- Meaningful values (or asymptotic) bounds for the key performance indicators (defined later) can be derived.

Pseudocode style

To specify a MR algorithm with a fixed number of rounds R, we will use the following style:

Input: description of the input as set of key-value pairs
Output: description of the output as set of key-value pairs

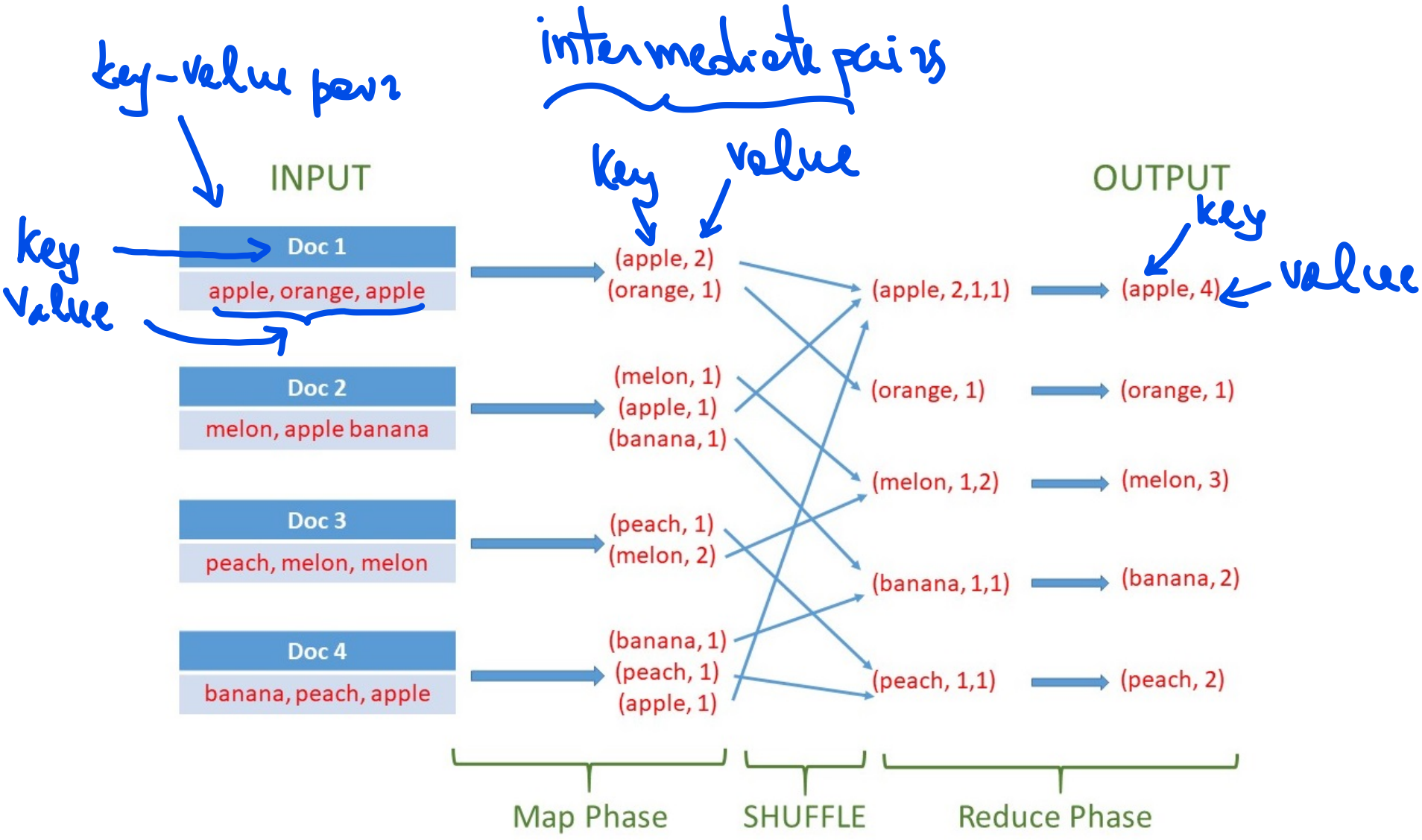
Round 1:

- *Map phase*: description of the function applied to each key-value pair
- *Reduce phase*: description of the function applied to each group of key-value pairs with the same key

Round 2, 3, ..., R: similarly

Observation: *for simplicity, we sometime provide a high-level description of a round, which, however, must enable a straightforward yet tedious derivation of the Map and Reduce phases.*

Example: Word count



Example: Word count

Input : k documents D_1 D_2 .. D_k

D_i = (name , list-of-words)

N = Total number of words (with repetitions) across all docs

Output : $\{(w, c(w)) : w \in D_1 \cup D_2 \cup \ldots \cup D_k$

$c(w)$ = # occurrences of w in $D_1, D_2, \ldots, D_k\}$

In the above example $N = 12$ $k = 4$

Example: Word count

Round 1

* Map Phase : for each $1 \le i \le K$ separately

$$D_i \xrightarrow{\text{map function}} \{(w, c_i(w)) : c_i(w) = \#\text{ occurences of } w \text{ in } D_i\}$$

* Reduce Phase : for each word w separately let L_w be the list $c_1(w), c_2(w), \ldots, c_K(w)$ and note that some $c_i(\,)$ may not exist

$$(w, L_w) \xrightarrow{\text{reduce function}} \left(w, \sum_{c_i(w) \in L_w} c_i(w)\right)$$

Analysis of a MapReduce algorithm

The analysis of an MR algorithm aims at estimating the following key performance indicators:

- Number of rounds R.
- Local space M_L: *maximum amount of main memory required by a single invocation of a map or reduce function*, for storing the input and any data structure needed by the invocation. *The maximum is taken over all rounds and all invocations in each round*.
- Aggregate space M_A: *maximum amount of (disk) space which is occupied by the stored data* at the beginning/end of a map or reduce phase. *The maximum is taken over all rounds*.

Observations

* R is a rough estimate of the running time (assuming that, in a round, time is dominated by the shuffle of the data, considered as a fixed cost)
* M_L represents the amount of MAIN MEMORY required at each executor (i.e., compute node)
* M_A represents the overall amount of DISK STORAGE required in the entire system (e.g. capacity of the HDFS)

Analysis of Word count

$R = 1$

M_L:

* N_i = # words in D_i, with $1 \leq i \leq K$
* $N_{max} = \max\{N_i : 1 \leq i \leq K\}$
* Map Phase : $O(N_{max})$
* Reduce Phase : $O(K)$

$\Rightarrow M_L = O(\max\{N_{max}, K\}) = O(N_{max} + K)$

Analysis of Word count

M_A:

* $|\text{input}| = O(N)$
* $|\text{intermediate pairs}| = O(N)$
* $|\text{output}| = O(N)$

$\Rightarrow M_A = O(N)$

Obs. M_L can grow very large (up to proportional to N) when the input contains very large docs or many small docs

Design goals for MapReduce algorithms

Here is a simple yet important observation.

Observation

For every problem solvable by a sequential algorithm in space S there exists a 1-round MapReduce algorithm with $M_L = M_A = \Theta(S)$: *run the sequential algorithm on the whole input with one reducer*.

Give the same key to all input elements and use the sequential algorithm as reduce function

Remark: the trivial solution implied by the observation is impractical for very large inputs for the following reasons:

- A platform with very large main memory is needed.
- No parallelism is exploited.

Design goals for MapReduce algorithms

and, in general, for good distributed algorithms

Design Goals for MapReduce Algorithms

1. Few rounds (e.g., $R = O(1)$);
2. Sublinear local space (e.g., $M_L = O(|\text{input}|^{\epsilon})$, with $\epsilon < 1$);
3. Linear aggregate space (i.e., $M_A = O(|\text{input}|)$), or only slightly superlinear;
4. Low complexity of each map or reduce function.

although their impact on performance will be kept under control if they are run on small subsets of data

Design goals for MapReduce algorithms

1) and 4) AIM at TIME EFFICIENCY

* 1) keeps shuffle and synchronization costs low
* 4) keeps computation costs (work) low

2) and 3) AIM at SPACE EFFICIENCY

* 2) ensures feasibility of the computation even if compute nodes have limited main memory
* 3) avoids (excessive) data replication

Design goals for MapReduce algorithms

To achieve the goals

* Make sure that map and reduce functions are applied to small subsets of data (especially reduce f.)
  - to attain goals 2) and 4)
  - to exploit parallelism of the underlying platform (recall that map and reduce functions can be applied in parallel in the respective phases
* Seek suitable tradeoffs between R, on one side, and M_L and M_A, on the other side

Pros and cons of MapReduce

Why is MapReduce suitable for big data processing?

- Data-centric view. Algorithm design focuses on data transformations targeting the above design goals.
- Usability. The programmer is lifted from performing crucial but difficult activities such as allocation of tasks to workers, data management, and handling of failures, which are instead dealt with transparently by the supporting framework.
- Portability/Adaptability. Applications can run on different platforms and the supporting framework will do best effort to exploit parallelism and minimize data movement costs.
- Cost. MapReduce applications can be run on moderately expensive platforms, and many popular cloud providers support their execution.

Pros and cons of MapReduce

What are the main drawbacks of MapReduce?

- Number of rounds provide only a coarse evaluation of time performance. It ignores the runtimes of map and reduce functions and the actual volume of data shuffled. More sophisticated (yet, less usable) performance measures exist.
- Curse of the last reducer. In some cases, one or a few reducers may be much slower than the others, thus delaying the end of the round. Algorithm designers should aim at balancing the load (if at all possible) among reducers.
- MapReduce is not suitable for applications that require very high performance and a tight coupling with the underlying architecture (e.g., from computational sciences).

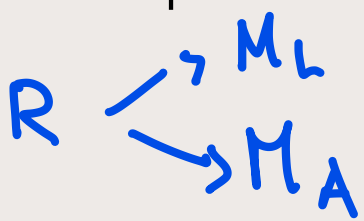
Partitioning

Partitioning

One of the main goals in the design of efficient MapReduce algorithms is to avoid that large amounts of data are given in input to individual reduce functions (e.g., sum of partial word counts from many documents). To achieve the goal, we must

- Subdivide the relevant data into small *partitions* either at random or deterministically.
- Make reduce functions work separately in individual partitions.

Observation. Tyically, partitioning induces the following tradeoff between performance indicators:



The smaller the amount of data that you can process at once, the larger the number of rounds that you may need to examine the relevant relations among the data

RANDOM PARTITIONING

Improved word count

Consider the word count problem, k documents and N word occurrences overall, in a realistic scenario where k and N are huge (e.g., *huge number of small documents*).

In the 1-round algorithm, M_L is at least k in the worst case

We now see how to reduce the local space requirements at the expense of an extra round.

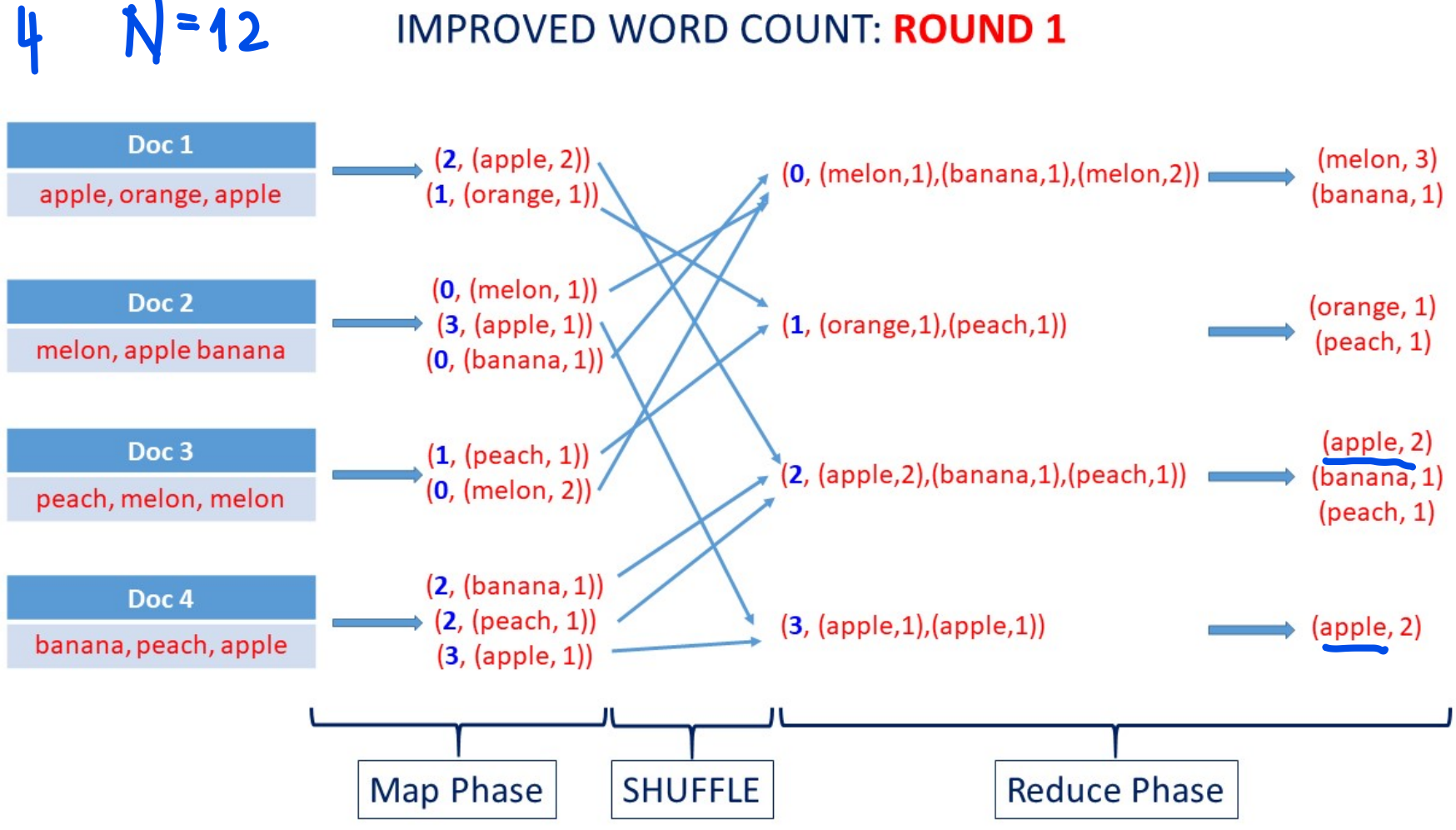
Idea:

- **Round 1**: partition intermediate pairs into $\ell = o(N)$ groups using random keys in $[0, \ell)$ and compute local word counts within each group;
- **Round 2**: For each word, aggregate local counts to produce the global count.

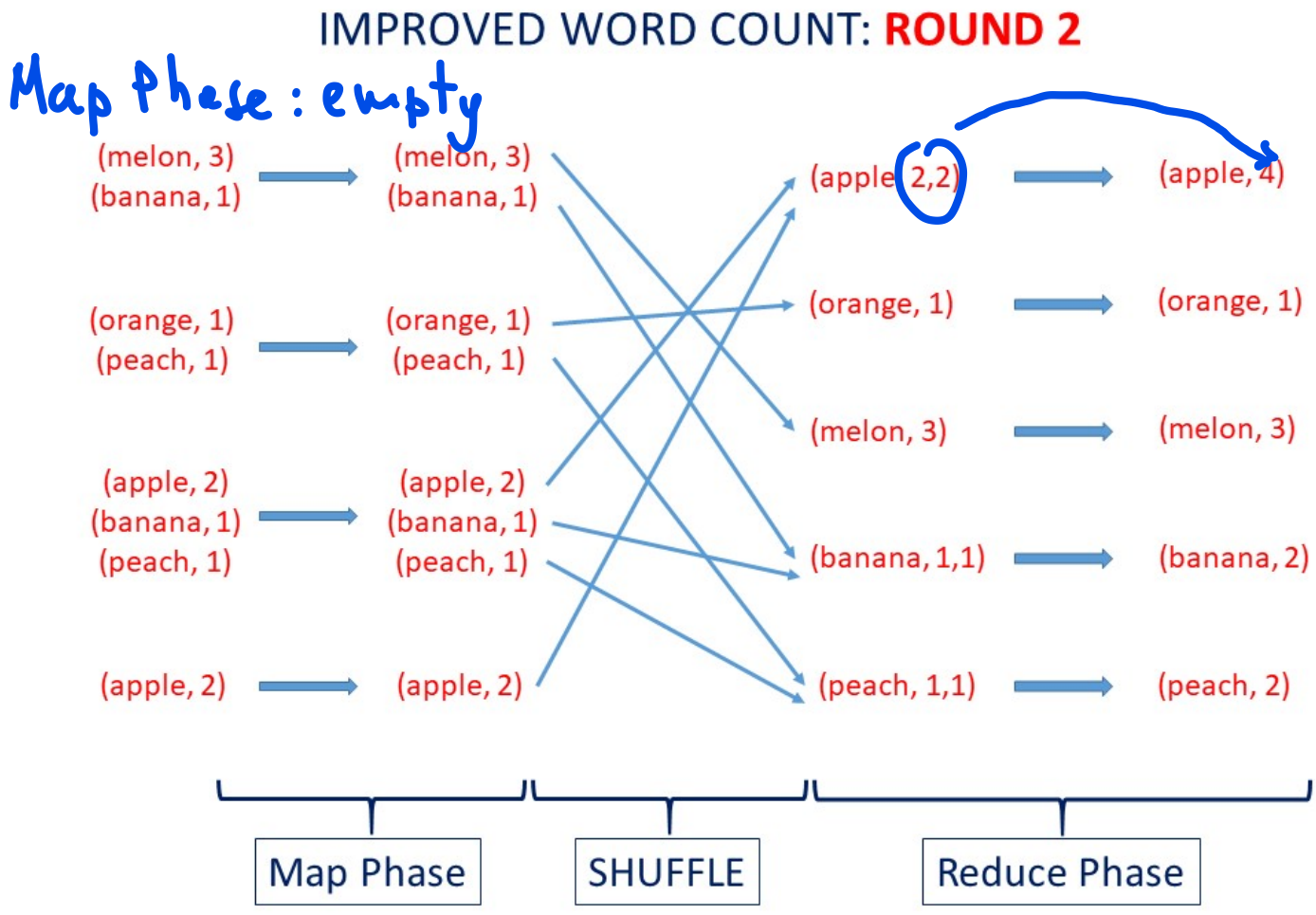
Improved word count: example

$$\ell = 4$$

$k=4 \quad N=12$



Improved word count: example



Improved word count: pseudocode

Input/output : as before

Round 1

Map Phase: for each $1 \le i \le K$ separately

$D_i \to \{(x, (w, c_i(w))) : w \in D_i, x \in [0, l)$ random$\}$

Reduce Phase: for each $x \in [0, l)$ separately, let

L_x the list of $(w, c_i(w))$'s that were assigned random key x

$(x, L_x) \to \{(w, c(x,w)) :$ w occurs in L_x and

$$c(x,w) = \sum_{(w, c_i(w)) \in L_x} c_i(w) \}$$

Improved word count: pseudocode

Round 2

Map Phase: empty

Reduce Phase: for each word w seperately, let L_w be the list of $c(x,w)$ with $x \in [0,\ell)$ (some of them may not exist)

$$(w, L_w) \to \left(w, c(w) = \sum_{c(x,w) \in L_w} c(x,w)\right)$$

Improved word count: analysis

Let:

- N_i = be the number of words in D_i.
- m_x = number of intermediate pairs with random key x.
- $m = \max\{m_x : 0 \leq x < \ell\}$.

and $N_{max} = \max\{N_i : 1 \leq i \leq k\}$

Obs. m_x and m depend on ℓ

We have

- $R = 2$
- $M_L = O\left(\max\{N_i : 1 \leq i \leq k\} + m + \ell\right)$. ($\max\{N_i : 1 \leq i \leq k\} = N_{max}$)
- $M_A = O(N)$

Improved word count: analysis

$R = 2$

M_L :

R1
- * Map Phase = $O(N_{max})$
- * Reduce Phase = $O(m)$

R2
- * Map Phase : $\emptyset$
- * Reduce Phase = $O(\ell)$

Obs. If each doc has distinct words, then $m \geq N/\ell$

$\Rightarrow M_L \in O(N_{max} + m + \ell)$

Improved word count: analysis

M_A :

* The input and output of each round requires $O(N)$ space
* In both R1 and R2, $O(N)$ intermediate pairs are generated

$\Rightarrow M_A = O(N)$

We now assume the N_{max} is "small" and focus the attention on l and m

How large can m be?

Theorem

Fix $\ell = \sqrt{N}$ and suppose that in Round 1 the keys assigned to intermediate pairs independently and with uniform probability from $[0, \sqrt{N})$. Then, with probability at least $1 - 1/N^5$

$$m = O\left(\sqrt{N}\right).$$

Therefore, for $\ell = \sqrt{N}$, the preceding analysis shows that

$$M_L = O\left(\max\{N_i : 1 \leq i \leq k\} + \sqrt{N}\right) = O(\sqrt{N}) \text{ if } N_{max} = O(\sqrt{N})$$

with probability at least $1 - 1/N^5$ (*very close to 1*, for large N).

N.B.: This is an example of probabilistic analysis, as opposed to more traditional worst-case analysis.

Useful probabilistic tools

Union bound

Given a countable set of events $E_1, E_2, \ldots, E_r$, we have:

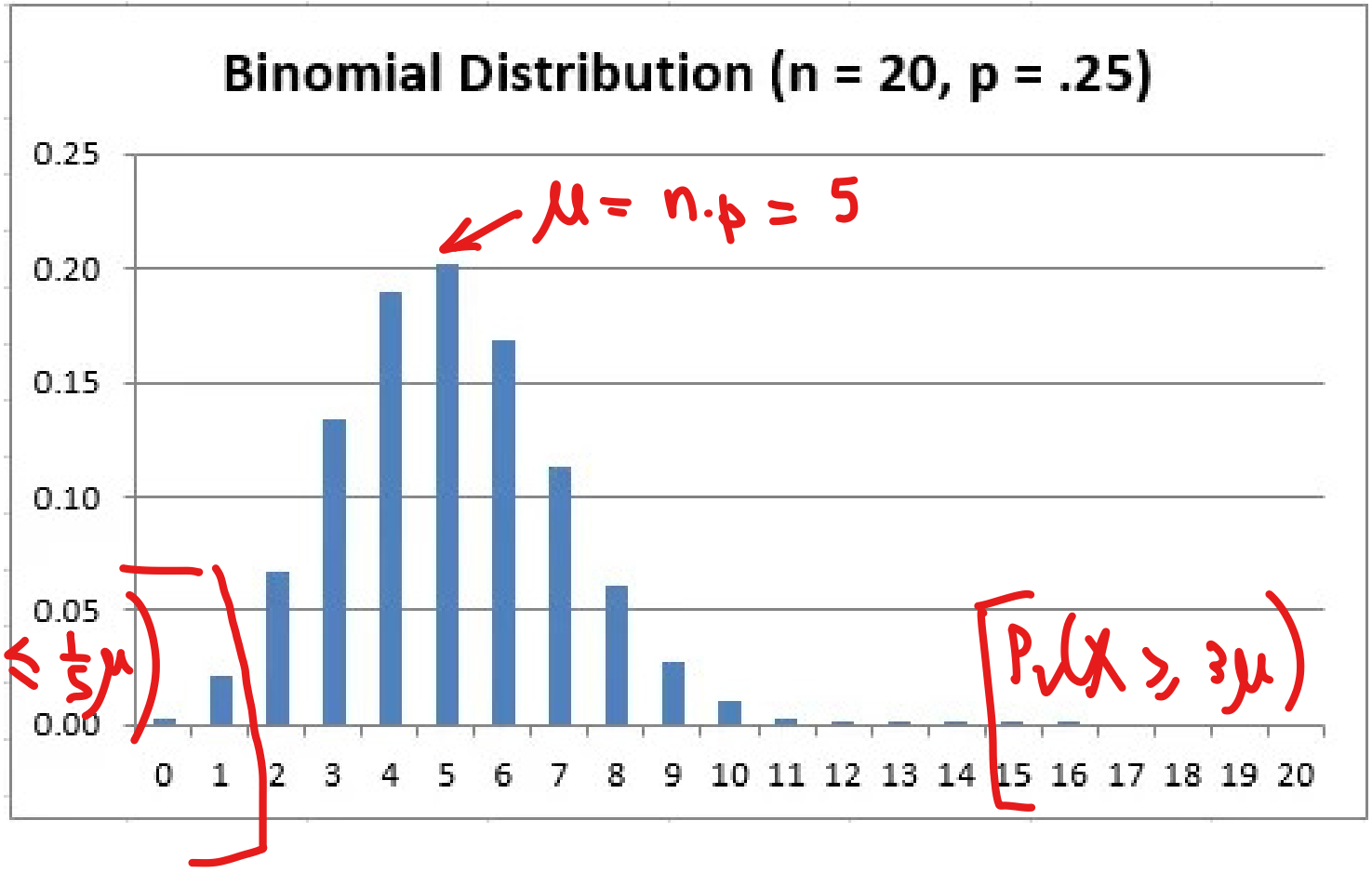
$$\Pr\left(\bigcup_{i=1}^{r} E_i\right) \leq \sum_{i=1}^{r} \Pr(E_i).$$

Chernoff bound (e.g., see [MU05]) CONCENTRATION BOUND

Let $X_1, X_2, \ldots, X_n$ be n i.i.d. Bernoulli random variables, with $\Pr(X_i = 1) = p$, for each $1 \leq i \leq n$. Thus, $X = \sum_{i=1}^{n} X_i$ is a Binomial(n, p) random variable. Let $\mu = E[X] = n \cdot p$. For every $\delta_1 \geq 6$ and $\delta_2 \in (0, 1)$ we have that

(a)
(b)

$$\begin{aligned} \Pr(X \geq \delta_1 \mu) &\leq 2^{-\delta_1 \mu} \\ \Pr(X \leq (1 - \delta_2)\mu) &\leq 2^{-\delta_2^2 \mu / 2} \end{aligned}$$


Binomial Distribution (n = 20, p = .25)
0.25
0.20
0.15
0.10
0.05
0.00
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
μ = n·p = 5
Pr(X ≤ ⅓μ)
Pr(X ≥ 3μ)

Proof of theorem

$\ell = \sqrt{N}$

RECALL

m_x = # intermediate pairs in R1 with random key x

$m = \max\{m_x : x \in [0, \sqrt{N})\}$

Let N' = # intermediate pairs generated in R1 $\Rightarrow N' \leq N$

$\Rightarrow$ for any arbitrary $x \in [0, \sqrt{N})$ we have

$m_x \sim \text{Binomial}(N', 1/\sqrt{N})$

$\mu = E[m_x] = N' \frac{1}{\sqrt{N}} \leq \sqrt{N}$

Proof of theorem

By Chernoff Bound (a)

$$P_r(m_x \geq 6\sqrt{N}) \leq 2^{-6\sqrt{N}}$$

since $6\sqrt{N} = \delta_1 \mu$ $\delta_1 \geq 6$

For $N \geq 1$ $\quad \sqrt{N} \geq \lg_2 N$

$$\Rightarrow 2^{-6\sqrt{N}} \leq 2^{-6\lg_2 N} = \left(2^{-(\lg_2 N)}\right)^6 = (1/N)^6$$

$$\Rightarrow P_r(m_x \geq 6\sqrt{N}) \leq (1/N)^6$$

Now $P_r\left(m = \max_{x \in [0,\sqrt{N})} m_x \geq 6\sqrt{N}\right)$

$$= P_r\left(\exists\, x \in [0,\sqrt{N}) : m_x \geq 6\sqrt{N}\right)$$

Proof of theorem

By Union Bound

$$P_1\left(\exists x \in [0, \sqrt{N}) : m_x \geq 6\sqrt{N}\right) \leq$$

$$\leq \sum_{x \in [0,\sqrt{N})} P_2\left(m_x \geq 6\sqrt{N}\right) \leq \sqrt{N} \cdot \frac{1}{N^c} \leq \frac{1}{N^5}$$

$$\Rightarrow P_2\left(m < 6\sqrt{N}\right) = 1 - P_2\left(m \geq 6\sqrt{N}\right)$$

$$\geq 1 - 1/N^5$$

$\Rightarrow$ With probability at least $1 - 1/N^5$, $m < 6\sqrt{N}$

hence, $m = O(\sqrt{N})$

Observations

- The choice of subdividing the intermediate pairs into $\sqrt{N}$ partitions provides an example of a simple tradeoff between local space and number of rounds.

- While we will often make this choice in the high-level algorithm design, in practice the number of partitions is usually fixed as a small multiple of the number of available workers. as long as the workers have sufficient main memory to run the map and reduce functions
- If stringent bounds on the local space are imposed one can force the algorithm to attain them, trading rounds for space. One of the exercises will explore this issue.

DETERMINISTIC PARTITIONING

Class count

Problem: given a set S of N objects with class labels, count how many objects belong to each class.

More precisely:

Input: Set S of N objects represented by pairs $(i, (o_i, \gamma_i))$, for $0 \leq i < N$, where o_i is the i-th object, and γ_i its class.
Output: The set of pairs $(\gamma, c(\gamma))$ where γ is a class labeling some object of S and $c(\gamma)$ is the number of objects of S labeled with γ.

ID

Obs. Integer keys may seem somewhat artificial However, in practice it is conceivable that each object contains a unique ID that can play the role of index i

Class count in 1 round (straightforward)

Round 1

Map Phase: $\forall 0 \le i < N$ separately $(i, (o_i, r_i)) \rightarrow (r_i, 1)$

Reduce Phase: for each class r separately, let

L_r : list of 1's from intermediate pairs $(r, 1)$

$(r, L_r) \rightarrow \underbrace{(r, c(r) = |L_r|)}_{\text{output pair}}$

Obs. Initial keys (i's) are not exploited

Class count in 1 round (straightforward)

M_L :

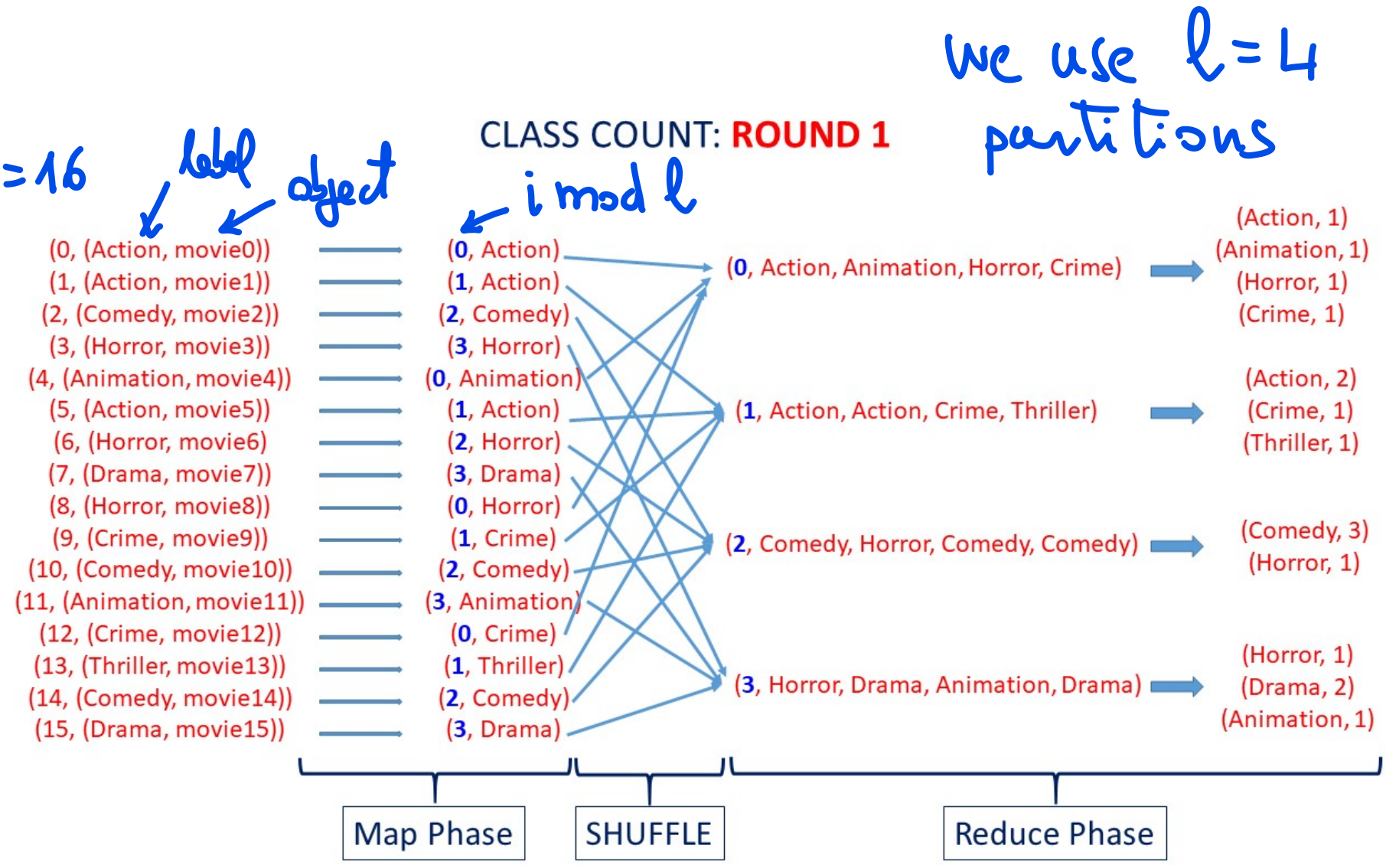
Map Phase = $O(1)$ assuming that each object takes constant space

Reduce Phase = $O(N)$ in the (extreme) case where a large fraction of objects have the same class label

$M_A = O(N)$

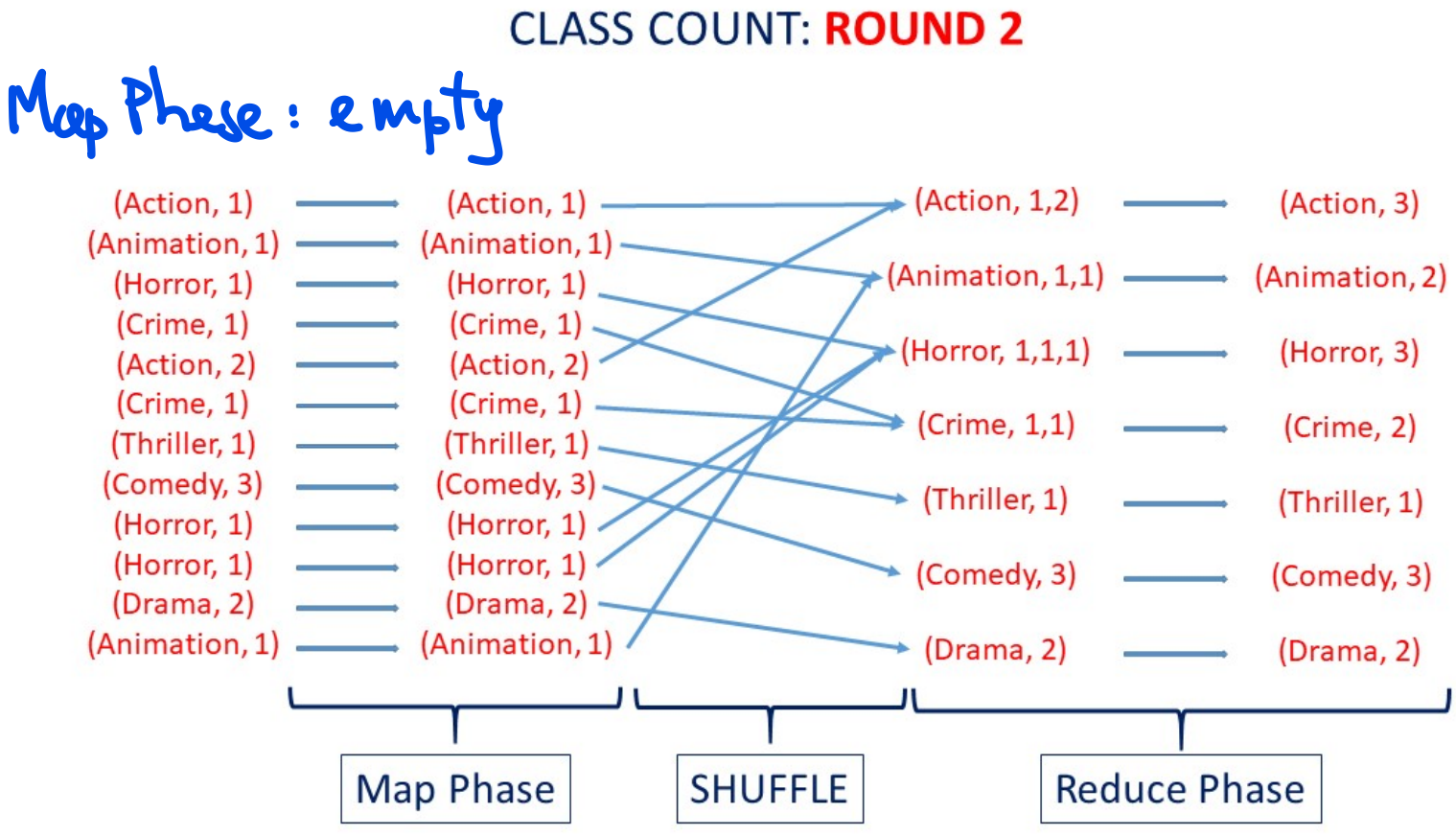
⇒ The 1 round algorithm does not meet the design goals for MR-algorithms

Class count in 2-rounds



Notation: $i \bmod l \equiv$ reminder of division i/l

Class count in 2-rounds



Class count in 2-rounds

R1

Map Phase: $\forall\ 0 \le i < N$ separately, $(i, (o_i, \gamma_i)) \to (i \bmod \ell, \gamma_i)$

Reduce Phase: for each $j \in [0, \ell)$ separately, let

L_j = list of class labels γ in intermediate pairs (j, γ)

$(j, L_j) \to \{(\gamma, c(j,\gamma) : \gamma \in L_j$ and $c(j,\gamma) = \#$ occurrences of γ in $L_j\}$

Class count in 2 rounds

Round 2

Map Phase : empty

Reduce Phase: for each γ seperately, let

L_γ = list of $c(j,\gamma)$'s from pairs produced by R1, with with $j \in [0, \ell)$. (some $c(i,\gamma)$'s may be missing)

$$(\gamma, L_\gamma) \longrightarrow \left(\gamma, c(\gamma) = \sum_{c(j,\gamma) \in L_\gamma} c(j,\gamma)\right)$$

R1 ↓ R2 ↓

$$M_L = O(N/\ell + \ell) \xRightarrow{\ell = \sqrt{N}} M_L = O(\sqrt{N})$$

$$M_A = O(N)$$

Exercise (See ex. 2.3.1.(b) of [LRU14])

Let S be a set of N integers represented by pairs (i, x_i), with $0 \leq i < N$, where i is the key of the pair and x_i is an arbitrary integer.

1. Design a 2-round MR algorithm to compute the arithmetic mean of the integers in S, using $O\left(\sqrt{N}\right)$ local space and $O(N)$ aggregate space. Carefully specify the input and output pairs of each round. You may assume that N is a global variable known to the algorithm.
2. Show how to modify the algorithm of the previous point to reduce the local space bound to $O\left(N^{1/4}\right)$, at the expense of an increased number of rounds.
3. Can you generalize the previous point to attain $O(N^{\epsilon})$, for any $\epsilon \in (0, 1/2)$?

Exercise (See ex. 2.3.1.(d) of [LRU14])

Let S be a set of N integers represented by pairs (i, x_i), with $0 \leq i < N$, where i is the key of the pair and x_i is an arbitrary integer. We want to design an efficient MR algorithm to compute the number D of distinct integers in S.

1. Design a simple 2-round MR algorithm to compute D and analyze its local and aggregate space requirements. Under what circumstances is the local space proportional to N, thus not meeting the design goals of MR algorithms?
2. Design a better MR algorithm to compute D in $O(1)$ rounds, using $O\left(\sqrt{N}\right)$ local space and $O(N)$ aggregate space.

Exercise

Consider the Class count problem with the input consisting only of object-class pairs (o, γ), without integer keys in $[0, N)$ (now the objects act as keys). Devise an efficient MR algorithm for the problem.

Hint: use random keys and target probabilistic guarantees.

Exercise

Design an $O(1)$-round MR algorithm for computing a matrix-vector product $W = A \cdot V$, where A is an $m \times n$ matrix, V is an n-vector, and $m \leq \sqrt{n}$. Your algorithm must use $o(n)$ local space and linear (i.e., $O(mn)$) aggregate space.

How would your solution change if m were larger?

Summary

- MapReduce.
  - Motivating scenario.
  - Main Features, platforms and software frameworks.
  - MapReduce computation: high-level structure, algorithm specification, analysis, key performance indicators.
  - Algorithm design goals.
- Partitioning:
  - Randomized partitititioning: improved word count.
  - Probabilistic tools: Chernoff and union bounds.
  - Deterministic partitioning: class count.

References

- **[LRU14]** J. Leskovec, A. Rajaraman and J. Ullman. Mining Massive Datasets. Cambridge University Press, 2014. Chapter 2 and Section 6.4
- **[DG08]** J. Dean and A. Ghemawat. MapReduce: simplified data processing on large clusters. OSDI'04 and CACM 51,1:107113, 2008
- **[MU05]** M. Mitzenmacher and E. Upfal. Proability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, 2005. (Chernoff bounds: Theorems 4.4 and 4.5)
- **[P+12]** A. Pietracaprina, G. Pucci, M. Riondato, F. Silvestri, E. Upfal: Space-round tradeoffs for MapReduce computations. ACM ICS'112.