

Computational Frameworks

MapReduce

OUTLINE

- ① MapReduce
- ② Partitioning

MapReduce

Motivating scenario

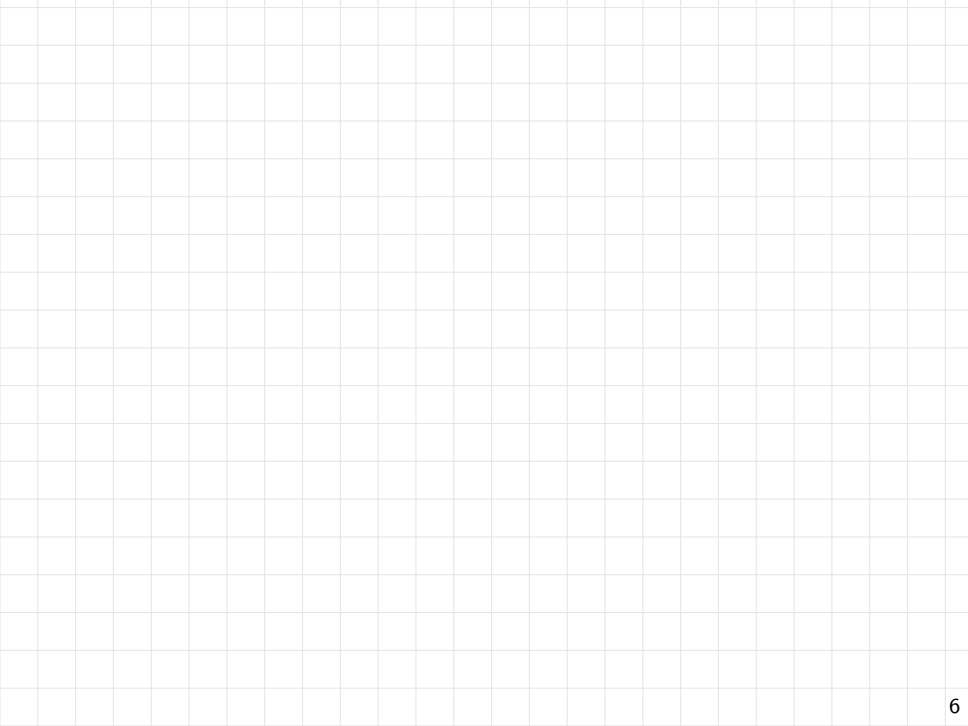
At the dawn of the big-data era (early 2000's), the following scenario was rapidly emerging

- 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*.



MapReduce

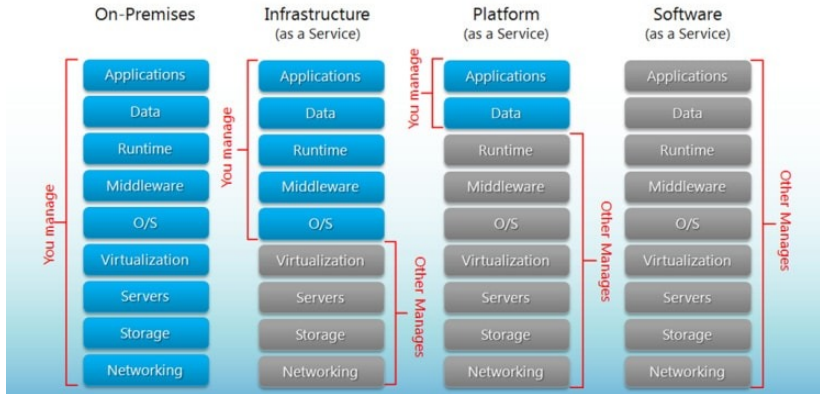
- Introduced by Google in 2004 as a **programming framework for big data processing on distributed platforms**.
- Its original formulation (see [DG08]) contained
 - A **programming model**
 - An **implementation** tailored towards a cluster-based computing environment.
- Since then, MapReduce implementations 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

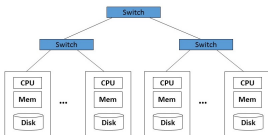
- Clusters of commodity processors (On-premises)
- Platforms from cloud providers (Amazon AWS, Microsoft Azure)
 - IaaS (*Infrastructure as a Service*): provides the users computing infrastructure and physical (or virtual) machines. This is the case of CloudVeneto which provides us a cluster of virtual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

Platforms



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)

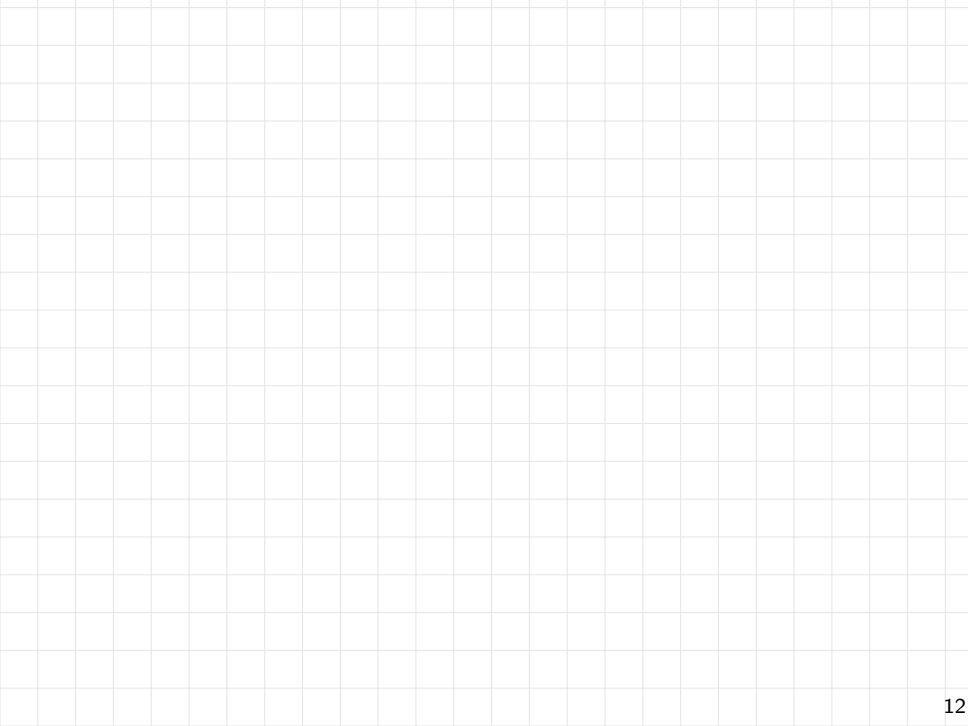


- Distributed File System (DFS)
 - Files divided into *chunks* (e.g., 64MB per chunk)
 - Each chunk replicated (e.g., 2x or 3x) with replicas in different nodes to ensure **fault-tolerance**.
 - Examples: Google File System (GFS); **Hadoop Distributed File System (HDFS)**

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.



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

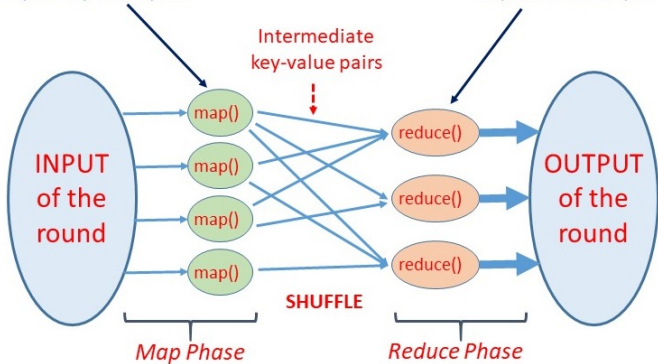
- **Map phase**: for each input key-value pair separately, a user-specified **map function** is applied to the pair and produces ≥ 0 other key-value pairs, sometimes called **intermediate pairs**.
- **SHUFFLE**: intermediate pairs are grouped by key, by creating, for each key k , a **list L_k of values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

Terminology: the application of the reduce function to a pair (k, L_k) is referred to as **reducer**.

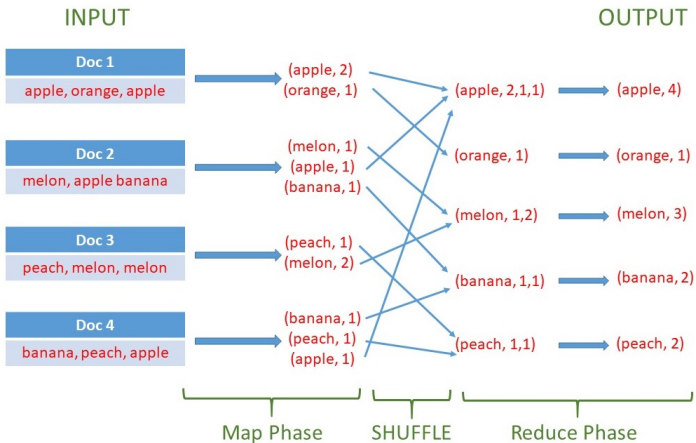
One Round

Map function applied to individual input key-value pairs

Reduce function applied to key-listOfValues pairs

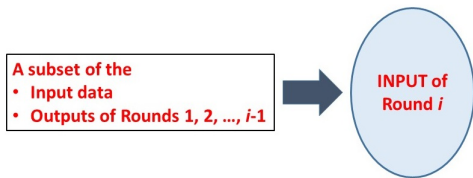


Example: Word count



Multiple Rounds and Shared Variables

- A MapReduce computation may require **multiple rounds**. If so, the **input of a round** comprises input data and/or data from the outputs of the previous rounds. (Typically, for $i > 1$, the input of Round i is the output of Round $i - 1$.)



- It is often convenient to define **shared variables**, available to all executors, which maintain global information, possibly across different rounds.

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 transformations 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

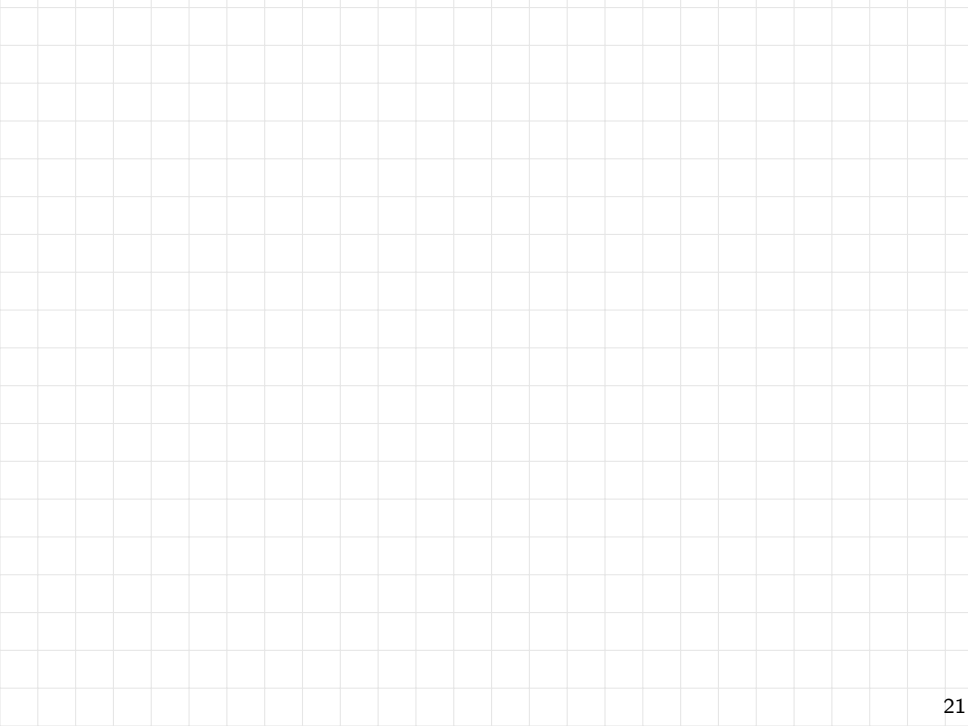
Round 1:

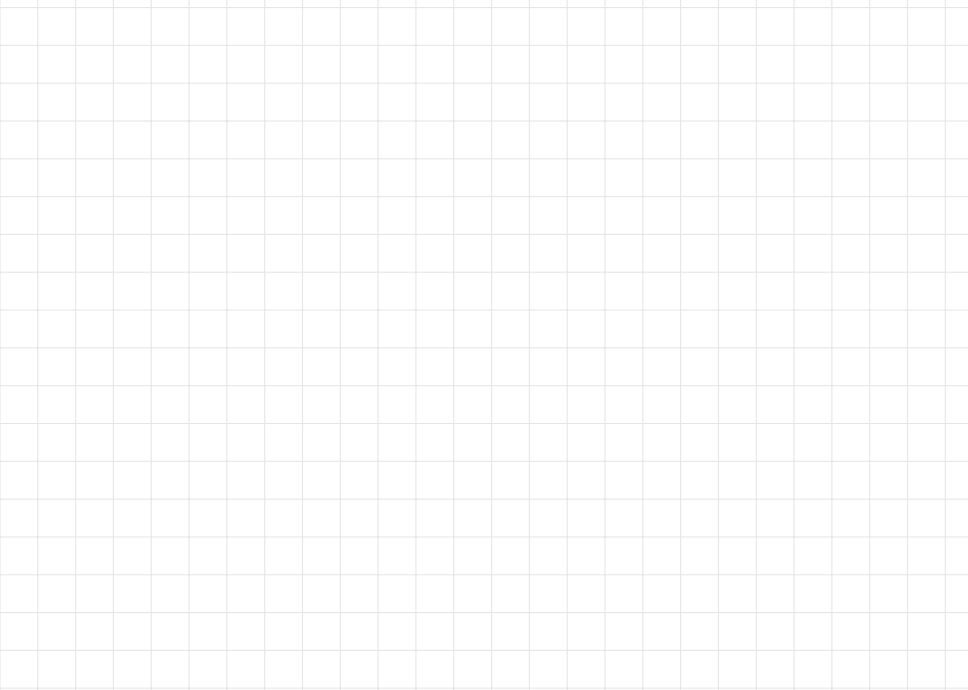
- *Map phase*: description of the function applied to each key-value pair
- *Reduce phase*: description of the function applied to each group of key-value pairs with the same key

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





Execution of a round on a distributed platform

- The user program is forked into a **master process** and several **executor processes**. The master is in charge of assigning map and reduce tasks to the various executors, and to monitor their status (idle, in-progress, completed).
- **Input and output files reside on a Distributed File System, while intermediate data are stored on the executors' local memories/disks.**
- **Map phase:** each executor is assigned a subset of input pairs and applies the map function to them sequentially, one at a time.
- **Shuffle:** the system collects the intermediate pairs from the executors' local spaces and groups them by key.
- **Reduce phase:** each executor is assigned a subset of (k, L_k) pairs, and applies the reduce function to them sequentially, one at a time.

Performance 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 (including possible shared variables). 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

Analysis of Word count

Analysis of Word count

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 both M_L and M_A proportional to S .

Remark: the trivial solution implied by the observation is not practical 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

Design Goals for MapReduce Algorithms

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

Design goals for MapReduce algorithms

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?

- **Running time is only coarsely captured by R .** More sophisticated (yet, less usable) time-performance measures should be used.
- **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

A typical approach to attain the stated goal on local space is to

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

Observation. Partitioning may increase the number of rounds and, more rarely, the aggregate space requirements. Hence, suitable tradeoffs must be sought.

DETERMINISTIC PARTITIONING

Class count

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

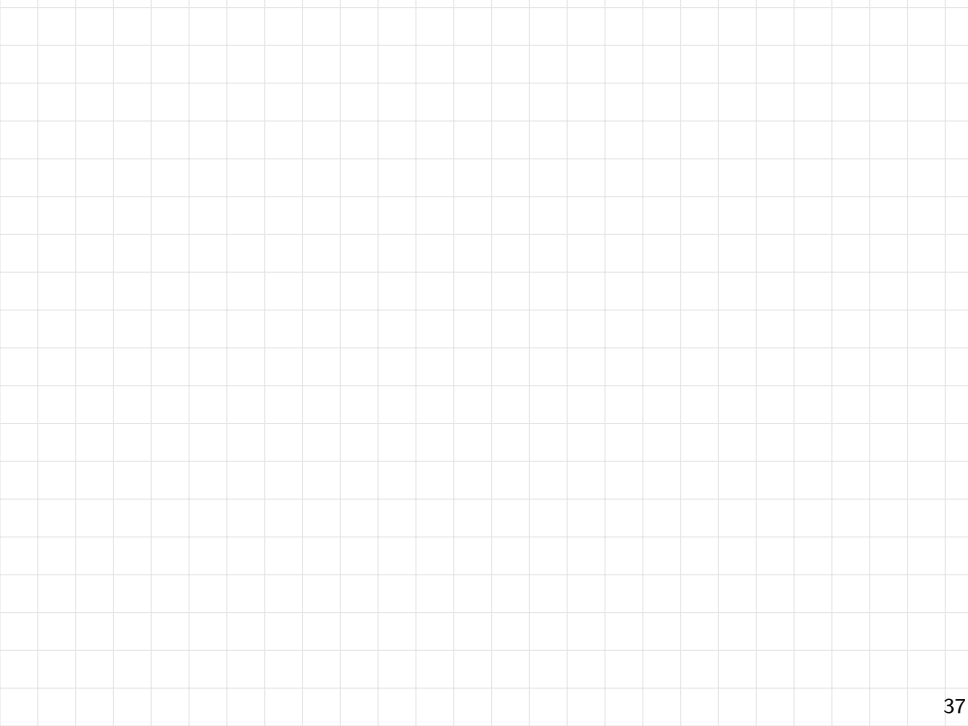
More precisely:

Input: Set S of N objects represented by pairs $(i, (\gamma_i, o_i))$, for $0 \leq i < N$, where γ_i is the class of the i -th object (o_i).

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 .

Remark: we will always assume that N (the number of input pairs) is a known value, i.e., a shared variable.

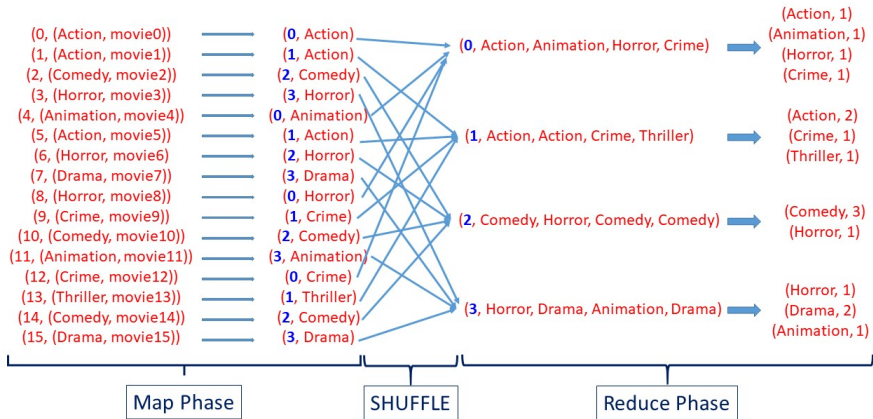
Class count in 1 round (naive - no partitioning)



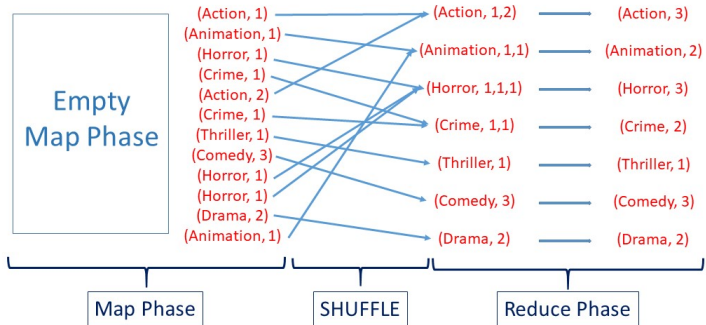
Class count in 2 rounds with deterministic partitions

(example: $\ell = 4$ partitions)

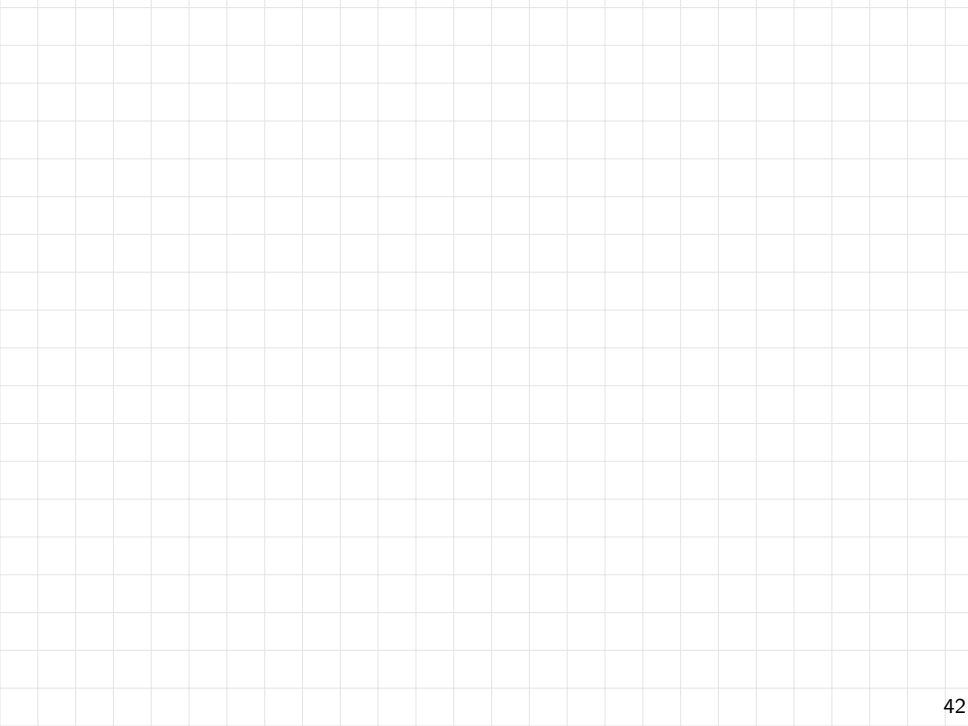
ROUND 1



ROUND 2



Class count in 2 rounds with deterministic partitions



Observations

- The choice of $\ell = \sqrt{N}$ partitions (which we will often make) provides an example of a simple tradeoff between local space and number of rounds.
- If more stringent bounds on the local space are imposed, different partitionings may be needed, yielding different round-space tradoffs. One of the exercises will explore this.
- In practice, the number of partitions is fixed as a multiple of the number of available workers, as long as local space is not an issue.

RANDOM PARTITIONING

Consider the class count problem, but assume that the input pairs are provided **without the integer keys in $[0, N)$** .

Namely:

Input: Set S of N objects represented by pairs (γ_i, o_i) , for $0 \leq i < N$, where γ_i is the class of the i -th object (o_i).

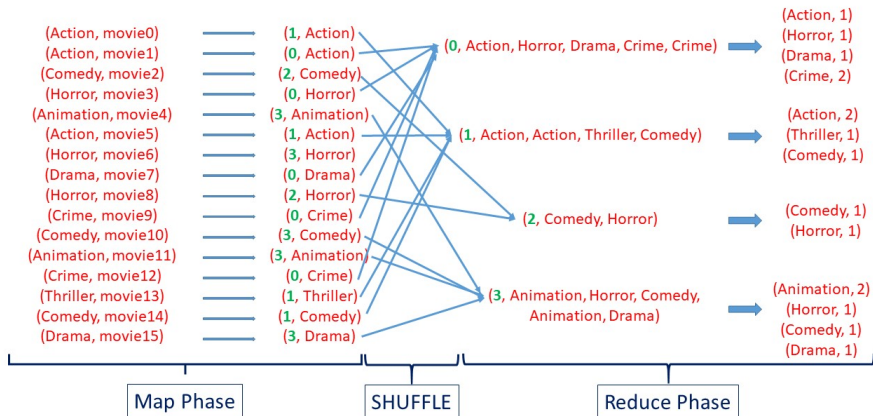
Output: The set of pairs $(\gamma, c(\gamma))$ where γ is a class labeling some object of S and $c(\gamma)$ is the number of objects of S labeled with γ .

We can employ the **2-round approach** seen before, but the partitioning in Round 1 will be achieved now by assigning a **random key in $[0, \ell)$ to each of the intermediate pair**.

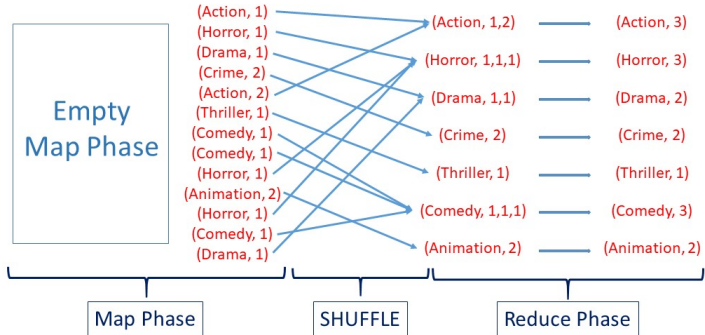
Class count in 2 rounds with random partitions

(example: $\ell = 4$ partitions)

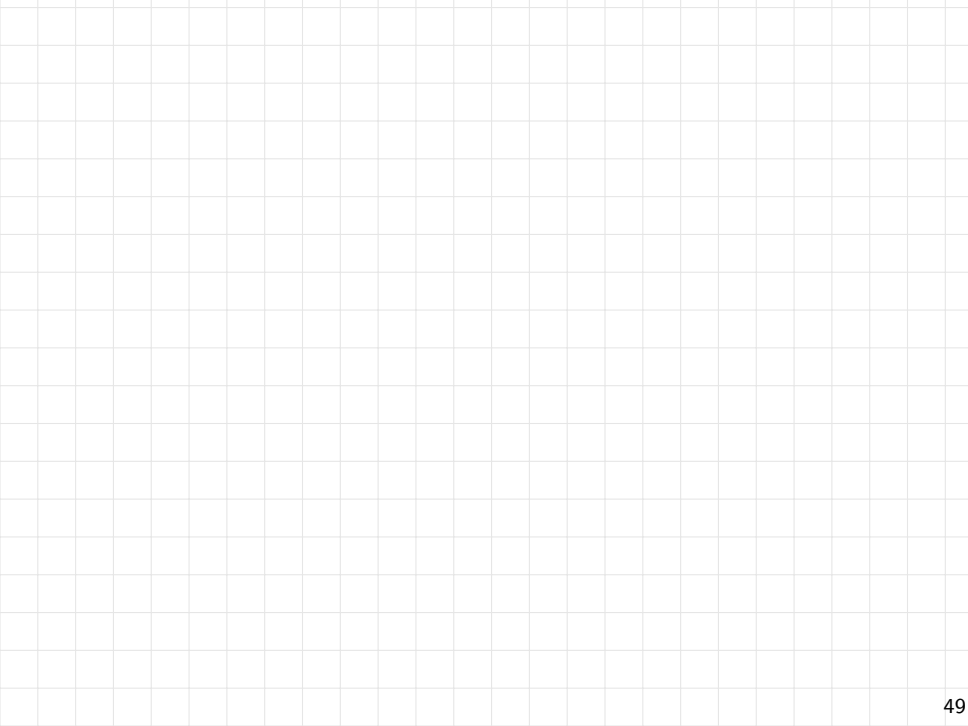
ROUND 1



ROUND 2



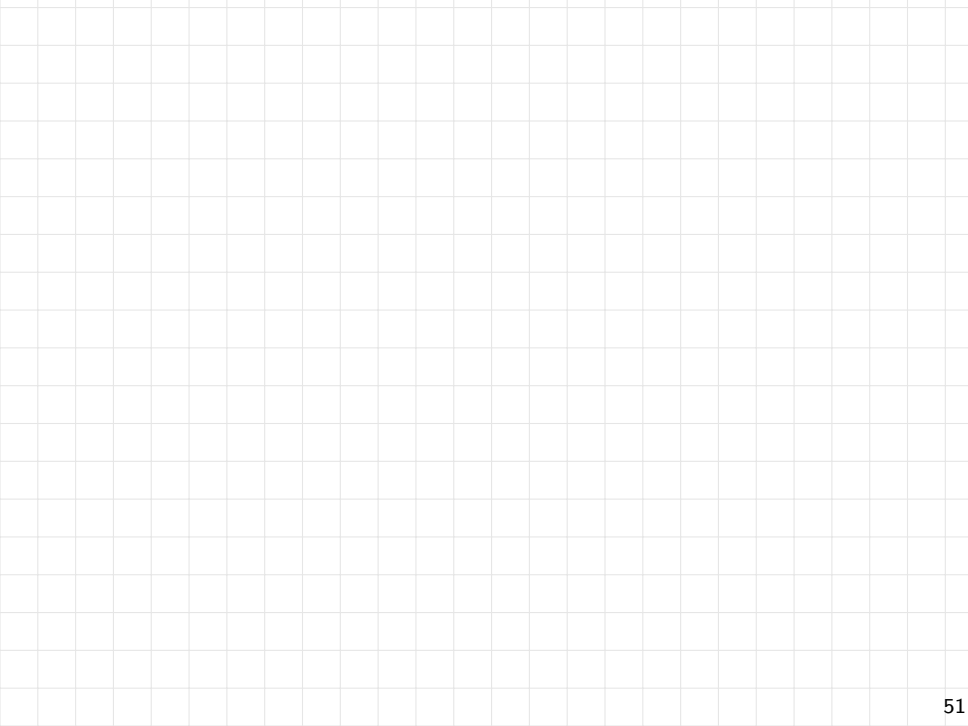
Class count in 2 rounds with random partitions (pseudocode)

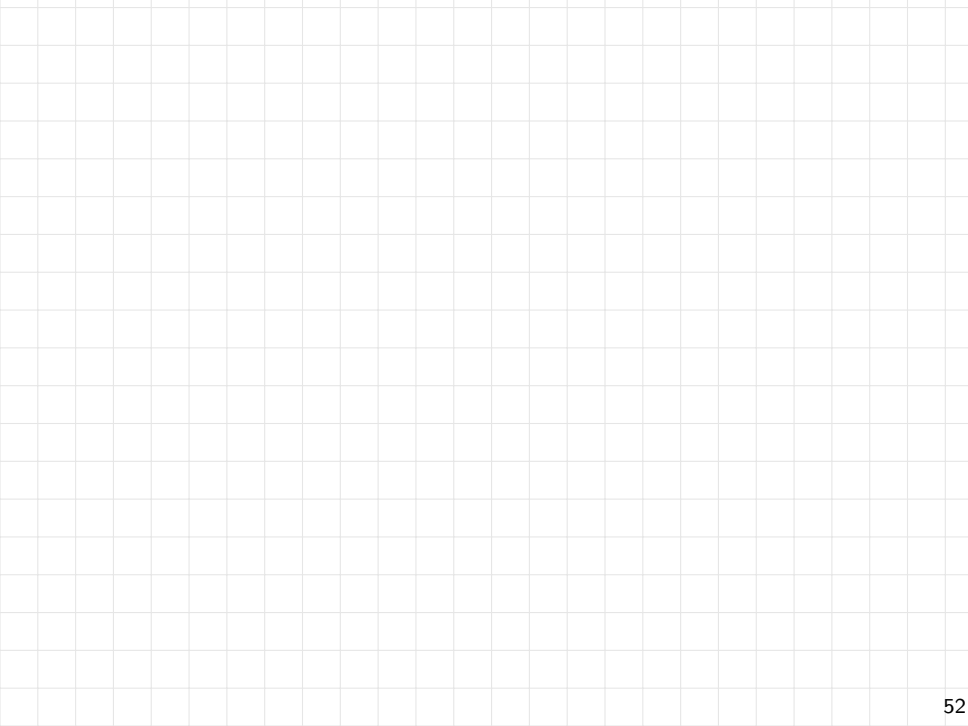


Class count in 2 rounds with random partitions (analysis)

Let:

- m_x = number of intermediate pairs with random key x .
- $m = \max\{m_x : 0 \leq x < \ell\}$.





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 $1 - 1/N^5$

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

Based on the theorem (which we will prove shortly) and on the previous analysis, by setting $\ell = \sqrt{N}$ we get

$$M_L = O(\sqrt{N}),$$

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

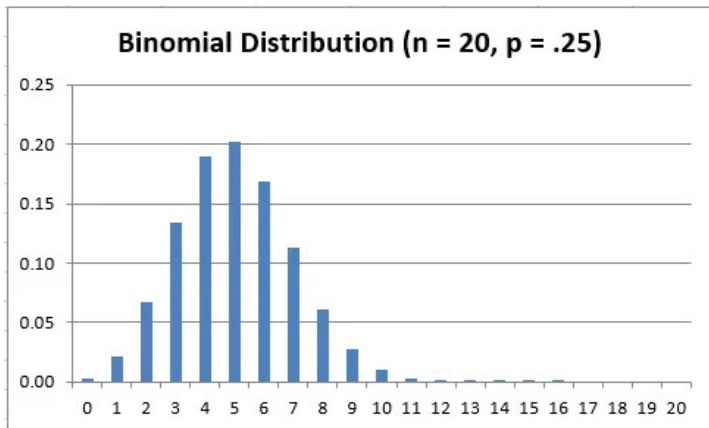
Given a countable set of events E_1, E_2, \dots, E_r , we have:

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

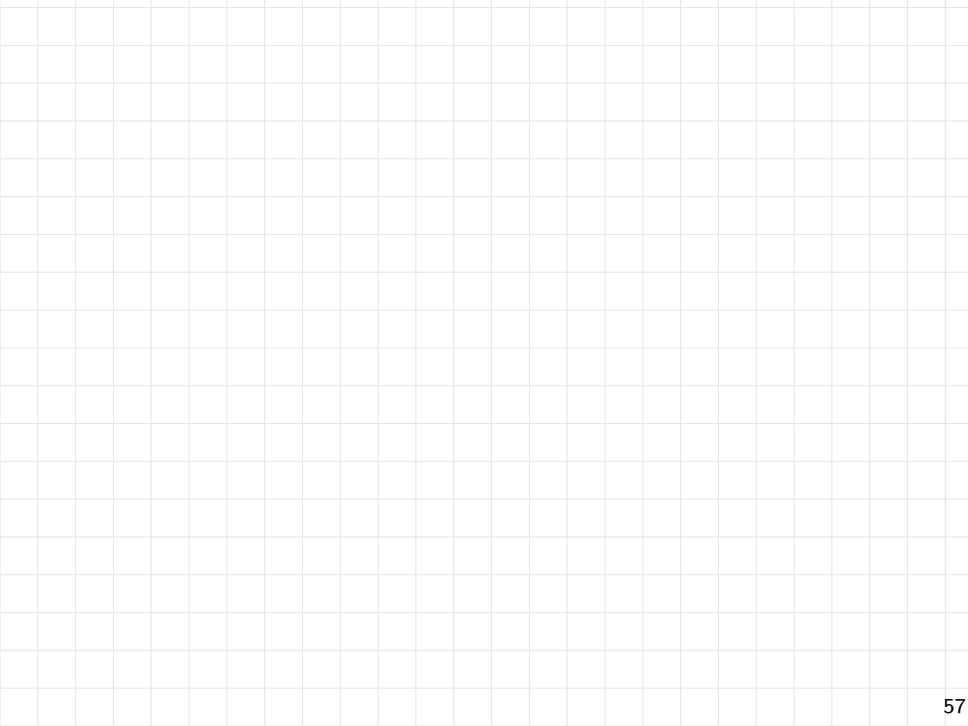
Chernoff bound (e.g., see [MU05])

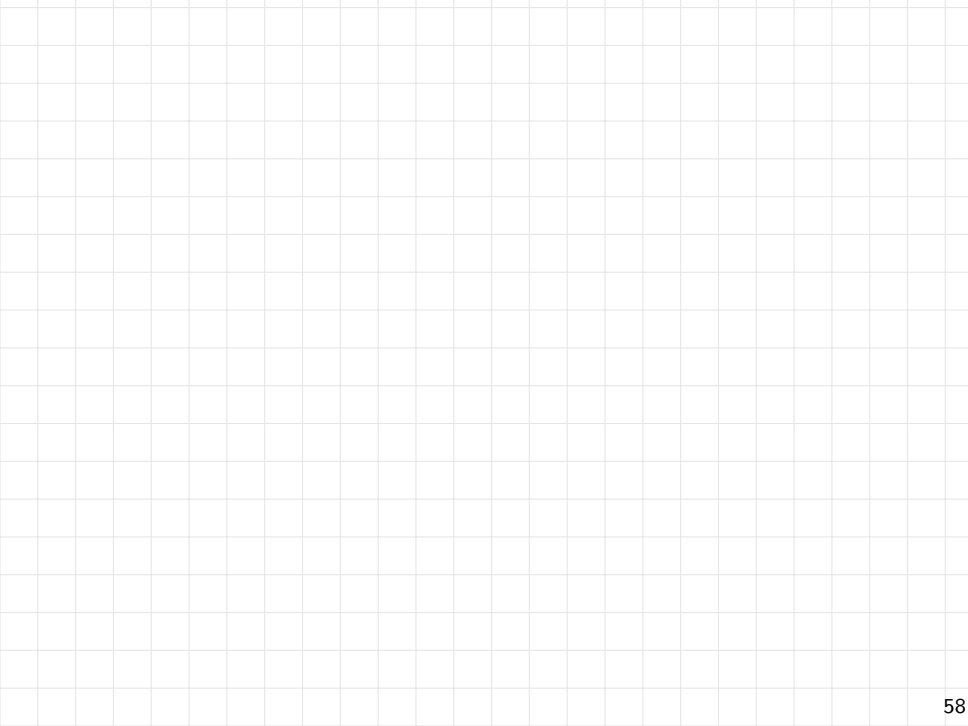
Let X_1, X_2, \dots, X_n be n i.i.d. Bernoulli random variables, with $\Pr(X_i = 1) = p$, for each $1 \leq i \leq n$. Thus, $X = \sum_{i=1}^n X_i$ is a $\text{Binomial}(n, p)$ random variable. Let $\mu = E[X] = n \cdot p$. For every $\delta_1 \geq 0$ and $\delta_2 \in (0, 1)$ we have that

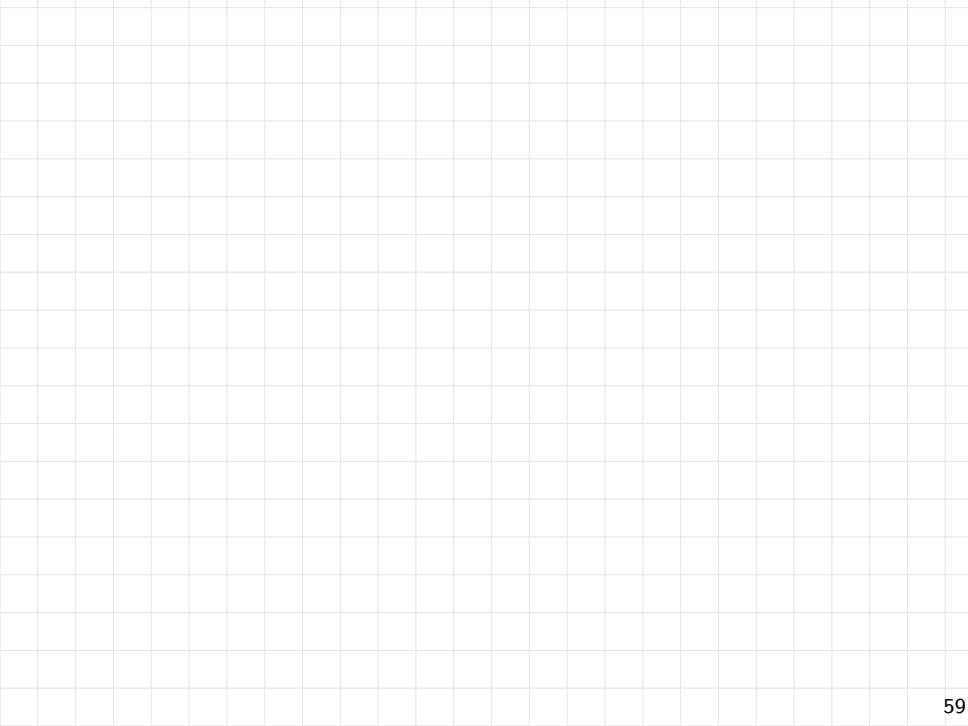
$$\begin{aligned} \text{(a)} \quad \Pr(X \geq (1 + \delta_1)\mu) &\leq 2^{-\delta_1\mu} \\ \text{(b)} \quad \Pr(X \leq (1 - \delta_2)\mu) &\leq 2^{-\delta_2^2\mu/2} \end{aligned}$$



Proof of theorem







Observations on partitioning

- **Deterministic partitioning:** applicable whenever input pairs contain pieces of data which can be mapped into partition indices that ensure a sufficiently even partitioning of the pairs.
- **Random partitioning:** always applicable, but to ensure a sufficiently even partitioning, the expected size of each partition should not be too small, and a good random generator must be used.

Theory question from Written Exam (22/23)

A MapReduce algorithm receives in input N web documents, each represented by a pair (url, D) , where D is the document and url is its address. Assume that each pair occupies $O(1)$ space. Describe a map phase that splits the input into \sqrt{N} small partitions, and state the theoretical guarantees of your partitioning.

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 $O(\sqrt{N})$ local space and $O(N)$ aggregate space.
- 2 Show how to modify the algorithm of the previous point to reduce the local space bound to $O(N^{1/4})$, at the expense of an increased number of rounds.
- 3 Can you attain $M_L = O(M)$, for any $M < \sqrt{N}$?

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 $O(\sqrt{N})$ local space and $O(N)$ aggregate space.

Exercise

Design a 2-round MR algorithm for the word count problem, using random partitioning, and analyze its local and aggregate space requirements.

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:
 - Deterministic partitioning
 - Randomized partitioning
 - Probabilistic tools: Chernoff and union bounds.

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. Probability 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'12.