# BIG DATA COMPUTING

**a cura di:**
**MARCO ZANETTI**

# Indice

# 1 Big Data

Big data phenomenon is the result of:

- Technology progress (as storage, bandwidth and computing capacity);
- Reduction of ICT costs;
- Pervasiveness of digital technologies.

In particular, large amount of data arise in fields like retailing, finance, science, medicine. The term *Big data* relates to two different issues:

- What to do with data (Machine learning);
- Processing challenges that derive from massive dataset sizes. A dataset is **large** if the analysis of the dataset with state-of-the-art tools and algorithm is infeasible.

We'll focus on the second issue.

## 1.1 The four V's of Big Data

The computing challenges that arise when dealing with large datasets are summarized by the so-called *four V's representation*:

- **Volume:** This refers to the size of the data which poses several computational challenges and requires a data-centre perspective;
- **Veracity:** Data generated by real world applications often contains noise and uncertainty, hence accuracy of solutions must be reconsidered;
- **Velocity**: Sometimes data arrive at such high rate that they cannot be stored and processed off-line. Hence stream processing is needed;
- **Variety**: large datasets arise in very different scenarios. More effective processing is achieved by adapting to the actual characteristics of the data.

# 2 MapReduce

Introduced by Google in 2004 as a programming framework for big data processing on distributed platforms. MapReduce uses a data-centric view and is inspired by functional programming (like maps and reduce functions). Messy details like task allocation and data distribution are hidden to the programmer while the environment deal with them itself. mapReduce applications run on either clusters of commodity processors or platforms from cloud providers. In particular, a cloud provider provide:

- **IAAS (Infrastructure As A Service)**: provides the users computing infrastructures and physical/virtual machines;
- **PAAS (Platform As A Service)**: provides the users computing platform with OS.

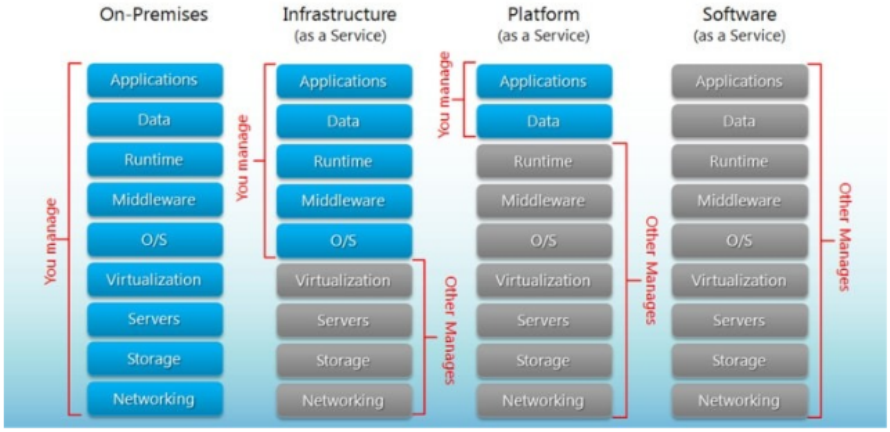


Figura 1: Platforms

**Typical cluster architecture** A cluster may consists in racks of 16 64 compute nodes connected by fast switches. One important component of a cluster is the distributed file system, which is implemented using the storage component. This is relevant when using MapReduce since data are frequently read and written in this file system, so it needs to provide fast access and reliability. On the DFS, the file are divided into chunks. Each chunk gets replicated in different nodes (and possibly racks) to ensure reliability. The distribution of the chunks of a file is represented into a master node file which is also replicated: a directory records where all the master nodes are.

**Hadoop-Spark Frameworks** Apache Hadoop was the most popular MapReduce implementation from which an entire ecosystem has stemmed. The Hadoop distributed file system is still widely used. Apache Spark is the most used framework for Big Data applications. It supports the implementation of MapReduce computations while providing a much richer programming environment.

## 2.1 MapReduce Computation

MapReduce computation is viewed as a sequence of rounds. Each round transforms a set of data, represented as a set of key-value pairs, into another set of key-value pairs trough the following two phases:

- **Map Phase:** A user specified map function is applied separately to each input key-value pair and produce $\geq 0$ other key-value pairs, referred to as intermediate key-value pairs.
- **Reduce Phase:** The intermediate key-value pairs are grouped by key and a user specified reduce function is applied separately to each group of key-value pairs with the same key, producing $\geq 0$ other key-value pairs, which is the output of the round.

The output of a round is the input of the next round. The use of key-value pairs is justified by the fact that MapReduce is data-centric and thus focuses on data transformation which are independent from where the data actually reside. The keys are needed as addresses to reach the

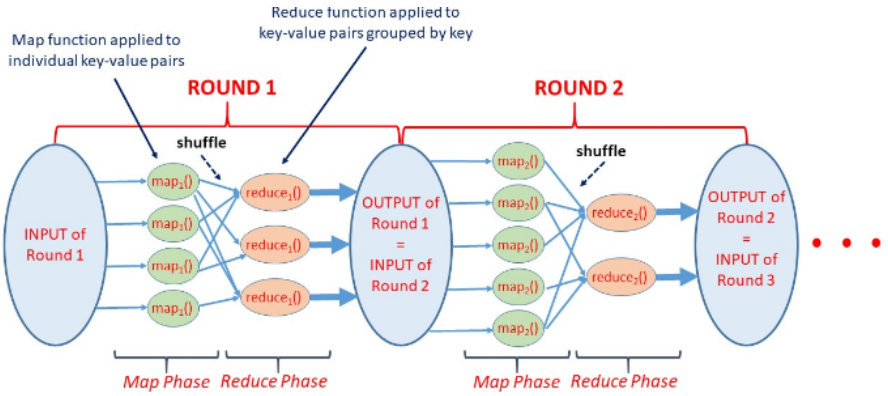


Figura 2: MapReduce round

objects, and as labels to define the groups in the reduce phase. In particular, 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.

**Implementation of a Round** First, the input file is split into $X$ **chunks**, where each chunk forms the input of a **map task**. Each map task is assigned to a worker (a compute node) which applies the map function to each key-value pair of the corresponding chunk, buffering the intermediate key-value pairs it produce in its local disk. Its intermediate key-values pairs, while residing in the local disks, are partitioned into $Y$ buckets through an hash function $h$:

$$(k, v) \longrightarrow Bucket\ i = h(x) mod Y$$

Each bucket forms the input of a different reduce task which is assigned to a worker. Note that we need to hash the key-value pairs in the buckets to make sure each bucket will have the same amount (approximately) of key-value pairs.
The worker applies the reduce function to each group of key-value pairs with the same key, writing the output on the Distributed File System. The application of the reduce function is called **reducer**. The user program is forked into a master process and several worker processes. The master process is in charge of assuming map and reduce tasks to the various workers, and to monitor their status (idle, in progress or completed). The input and output key-value pairs of a round reside on a distributed file system, while intermediate data are stored on the worker's local disks. In each round the data are shuffled, for moving the intermediate key-value pairs from the compute nodes where they were produced (by map tasks) to the compute nodes where they must be processed (by reduce tasks). It's important to note that the shuffle is a very expensive operation of the round.

The values $X$ and $Y$ are design parameters:

- $X$ is the number of map tasks;
- $Y$ is the number of buckets/reduce tasks.

This parameters should be set by the user to maximize performance.

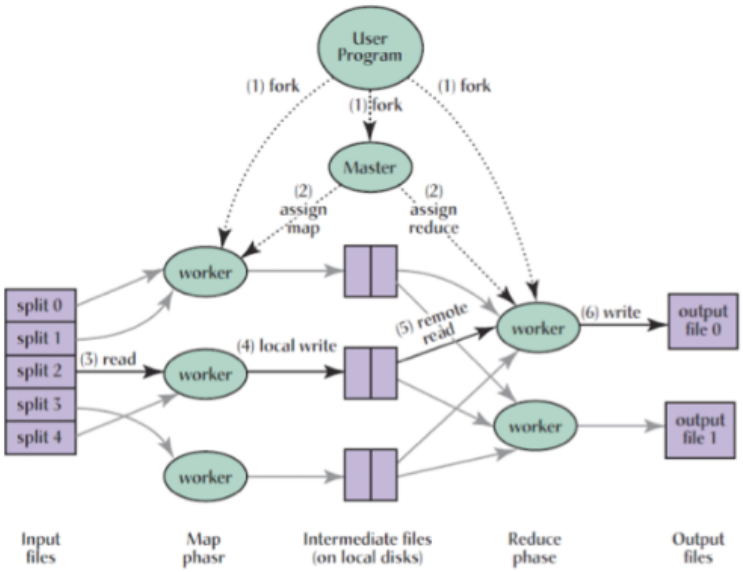


Figura 3: Implementation of a MapReduce round

**Dealing with faults** The distributed file system is fault-tolerant: if some kind of fault occurs, the data are preserved or might be recovered. If a fault occurs, the master program is "in charge" of detecting it, by pinging periodically to detect failures. If a round fail, all the map tasks (completed or in-progress) are reset to idle and will be rescheduled. Even if a map task is completed, the failure of the worker makes its output unavailable to reduce tasks, hence it must be rescheduled. Reduce task's in-progress at failed worker gets reset to idle and will be rescheduled. If the master fails, then the whole MapReduce task is aborted and must be restarted.

**Specification of a MapReduce algorithm** A MapReduce algorithm should be specified so that

- The input and output are 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 and the functions applied in the map and reduce phase.

Also, the specification of the algorithm should make possible its analysis (correctness and performances). That is, meaningful values bounds for the key performance indicators can be derived. Let's consider a MapReduce algorithm with a fixed number of rounds $R$. Then, we can use the following style to represent it:

- **Input:** Description of input as set of key-value pairs;
- **Output:** description of output as set of key-value pairs

- **Round 1:**
  - *Map Phase:* description of 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:** ...
- **Round R:** ...

For simplicity, we sometimes provide a high-level description of a round, which, however, must enable a straight forward yet tedious derivation of the Map and Reduce phases.

**Analysis of a MapReduce algorithm** The analysis of a MapReduce algorithm aims at estimating the following key performance indicators:

- *Running time* (number of rounds $R$);
- *Local Space* $M_L$: this is the maximum amount of memory required, in any round, by a single invocation of the map or reduce function used in that round, for storing the input and any data structure needed by the invocation.
- *Aggregate Space* $M_A$: maximum amount of (disk) space which is occupied by the stored data at the beginning/end of the map or reduce phase of any round.

In particular, those indicators are usually estimated trough asymptotic analysis as function of the instance size. $M_L$ bounds the amount of main memory required at each worker, while $M_A$ bounds the overall amount of (disk) space that the executing platform must provide.

## 2.2 Word Count

- **Input:** collection of text documents $D_1, D_2, \ldots, D_k$ containing $N$ words occurrences (counting repetitions). Each document is a key-value pair,whose key is the document's name and the value is its content.
- **Output:** The set of pairs $(w, c(w))$ where $w$ is a word occurring in the documents, and $c(w)$ is the number of occurrences of $w$ in the documents.
- **Round 1:**
  - *Map Phase:* for each document $D_i$, produce the set of intermediate pairs $(w, c_i(w))$, one for each word $w \in D_i$, where $c_i(w)$ is the number of occurrences of $w$ in $D_i$.
  - *Reduce Phase:* For each word $w$, gather all intermediate pairs $(w, c_i(w))$ and return the output pair $(w, c(w))$ where $c(w)$ is the sum of the $c_i(w)$'s.

The output, in this case, is the set of pairs $(w, c(w))$ produced by the various reducers, hence one pair for each word.

**Analysis of Word count** Let $N_i$ be the number of words in $D_i$ counting repetitions (hence, $N = \sum_{i=1}^{k} N_i$), and assume that each word takes constant space. Let also $N_{max} = max_{i=1,k} N_i$. We have:

- $R = 1$;
- $M_L = O(max\{N_{max}, k\}) = O(N_{max} + k)$. This bound is justified as follows. In the map phase, the worker processing $D_i$ needs $O(N_i)$ space. In the reduce phase, the worker in charge of word $w$ adds up the contribution of at most $k$ pairs $(w, c_i(w))$.
- $M_A = O(N)$, since the input size is $O(N)$ and $O(N)$ additional pairs are produced by the algorithm, overall.

## 2.3 Design goals for MapReduce algorithms

Here is a simple yet important observation:

**Theorem 1.** *For every computational problem solvable by a sequential algorithm in space $S(|input|)$ there exists a 1-round MapReduce algorithm with $M_L = M_A = \Theta(S(|input|))$.*

*Dimostrazione.* Run sequential algorithm on whole input with one reducer. □

Note that the trivial solution implied by the above theorem is impractical for very large inputs for the following reasons: A platform with very large main memory is needed, and no parallelism is exploited.
In general, to process efficiently very large input instances, an algorithm should aim at:breaking the computation into a (hopefully small) number of rounds that execute several tasks in parallel, each task working efficiently on a (hopefully small) subset of the data. In MapReduce terms, the design goals are the following:

- Few rounds;
- Sublinear local space (e.g $M_L = O(|input|^{\varepsilon})$ for some constant $\varepsilon \in (0, 1)$);
- Linear aggregate space (e.g $M_A = O(|input|)$) or only slightly superlinear;
- Polynomial complexity of each map or reduce function.

Algorithms usually exhibit trade-offs between performance indicators. Often, small $M_L$ enables high parallelism but may incur large $R$.

## 2.4 Pros and Cons of MapReduce

MapReduce is suitable for big data processing due to its:

- **Data-centric view:** Algorithm design can focus on data transformations, targeting the aforementioned design goals. Moreover, programming frameworks supporting MapReduce (e.g., Spark) usually make allocation of tasks to workers, data management, handling of failures, totally transparent to the programmer.
- **Portability/Adaptability:** Applications can run on different platforms and the underlying implementation of the 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.

On the other hand, the main drawbacks of MapReduce are:

- **Weakness of the number of rounds as performance measure:** 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 maybe much slower than the other ones, thus delaying the end of the round. When designing MapReduce algorithms one should try to ensure load balancing (if at all possible) among reducers.

## 2.5 Chernoff bound

Chernoff bound is an important tool we'll use n many proofs about Mapreduce and many other topics.

**Definition 1**

**Chernoff 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 $\mu = \mathbb{E}[X] = n \cdot p$. For every $\delta_1 \geq 5$ and $\delta_2 \in (0, 1)$ we have that

$$Pr(X \geq (1 + \delta_1)\mu) \leq 2^{-(1+\delta_1)\mu}$$

$$Pr(X \leq (1 - \delta_2)\mu) \leq 2^{-\mu\delta_2^2/2}$$

## 2.6 Partitioning technique

When some aggregation functions may potentially receive large inputs or skewed ones, it is advisable to partition the input, either deterministically or randomly, and perform the aggregation in stages. Some examples are an improved version of word count and a class count primitive.

**Improved Word count** Consider the word count problem with $k$ documents and $N$ word occurrences overall, in a realistic scenario where $k = \Theta(N)$ and $N$ is huge. The following algorithm reduces local space requirements at the expense of an extra round:

- **Idea:** Partition intermediate pairs randomly in $o(N)$ groups and compute the word count in two stages.
- **Round 1:**
  - *Map phase:* for each document $D_i$, produce the intermediate pairs $(x, (w, c_i(w)))$, one for every word $w \in D_i$. where $x$ (the key of the pair) is a random integer in $[0, \sqrt{N})$ and $c_i(w)$ is the number of occurrences of $w$ in $D_i$;
  - *Reduce phase:* For each key $x$ gather all pairs $(x, (w, c_i(w)))$, and for each word $w$ occuring in these pairs produce the pair $(w, c(x, w))$ where $c(x, w) = \sum_{(x,(w,c_i(w)))} c_i(w)$. Now, $w$ is the key for $(w, c(x, w))$.
- **Round 2:**

- *Map phase:* empty;
- *Reduce phase:* for each word $w$ we gather the at most $\sqrt{N}$ pairs $(w, c(x,w))$ resulting at the end of the previous round, and return the output pair $(w, \sum_x c(x,w))$.

- **Analysis:** Let $m_x$ be the number of intermediate pairs with key $x$ produced by the Map phase of Round 1, and let $m = max_x(m_x)$. As before, let $N_i$ be the number of words in $D_i$. We have
  - A number of rounds $R = 2$;
  - Local memory equals to $M_L = O(max_{i=1,k} N_i + m + \sqrt{N})$;
  - The total memory we need is $M_a = O(N)$.

To define how big $m$ is we need to use the Chernoff bound.

**Bound on m for improved Word count**

**Theorem 2.** *Suppose that the keys assigned to the intermediate pairs in Round 1 are i.i.d random variables with uniform distribution in* $[0, \sqrt{N})$*. Then, with probability at least* $1 - 1/N^5$*:*

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

Therefore, from the therorem and the precedent analysis we get

$$M_L = O(max_{i=1,k} N_i + \sqrt{N})$$

with probability at least $1 - 1/N^5$. In fact, for large $N$ the probability becomes very close to 1.

*Dimostrazione.* Let $N^{'} \leq N$ be the number of intermediate pairs produced by the Map phase of Round 1, and consider an arbitrary key $x \in [0, \sqrt{N})$. In particular, $m_x$ is a $Binomial(N^{'}, 1/\sqrt{N})$ random variable with expectation $\mu = \frac{N'}{\sqrt{N}} \leq \sqrt{N}$. By the Chernoff bound we have

$$Pr(m_x \geq 6\sqrt{N}) \leq \frac{1}{2^{6\sqrt{N}}} \leq \frac{1}{N^6}$$

Now, by the union bound we have that

$$Pr(m \geq 6\sqrt{N} \leq \sum_{x \in [0,\sqrt{N})} Pr(m_x \geq 6\sqrt{N} \leq \sqrt{N} Pr(m_x \geq 6\sqrt{N}) \leq \frac{1}{N^5}$$

Therefore, with probability at least $1 - \frac{1}{N^5}$ we have $m \leq 6\sqrt{N}$, i.e $m = O\sqrt{N}$. □

**Observation** The choice of partitioning the intermediate pairs into groups of size $O(\sqrt{N})$, in the first round of Improved word count, is somewhat arbitrary, but it provides an example of a simple trade-off between local space and number of rounds. In the rest of the course we'll often make similar choices. However, it is important to understand that the approach can be generalized to attain a given target on $M_L$ (e.g $M_L = O(N^{\varepsilon})$ for some $\varepsilon \in (0,1)$) exercising a finer trade-off between $M_L$ and number of rounds.

**Class Count** Suppose that we are given a set $S$ of $N$ objects, each labelled with a class from a given domain, and we want to count how many objects belong to each class. More precisely:

- **Input:** Set $S$ of $N$ objects represented by the pairs $(i, (o_i, \gamma_i))$, for $0 \leq i \leq N$, where $o_i$ is the $i$-th object, and $\gamma_i$ its class;
- **Output:** The set of pairs $(\gamma, c(\gamma))$ where $\gamma$ is a class labelling some object of $S$ and $c(\gamma)$ isthe number of objects of $S$ labelled with $\gamma$.
- **Observation:** The straightforward 1-round algorithm may require $O(N)$ local space, in case a large fraction of objects belong to the same class. The much-more efficient algorithm which uses the class partitioning technique is the following:
- **Round 1:**
  - *Map phase:* map each pair $(i, (o_i, \gamma_i))$ into the intermediate pair $(imod\sqrt{N}, (o_i, \gamma_i))$, where *mod* is the reminder of the integer division;
  - *Reduce phase:* for each key $j \in [0, \sqrt{N})$ gather the set (say $S^{(j)}$) of all intermediate pairs with key $j$ and, for each class $\gamma$ labelling some object in $S^{(j)}$, produce the pair $(\gamma, c_j(\gamma))$, where $c_j(\gamma)$ is the number of objects of $S^{(j)}$ labelled with $\gamma$.
- **Round 2:**
  - *Map phase:* empty;
  - *Reduce phase:* for each class $\gamma$, gather the at most $\sqrt{N}$ pairs $(\gamma, c(\gamma))$ resulting at the end of the previous round, and return the output pair $(\gamma, \sum_j c(\gamma))$.

While here we use a partition technique which is deterministic (the *mod* function), the one we've used in the word count was randomized. We can't use a deterministic technique for the word count since we can't know at priori how many pairs can be generated.

**accuracy for efficiency** There are problems for which exact Map Reduce algorithms may be too costly, namely they may require a large number of rounds, or large (i.e close to linear) local space, or large aggregate space. These algorithms become impractical for very large inputs. In these scenarios, giving up exact solutions (if acceptable for the application) may greatly improve efficiency.

## 2.7 Maximum Pairwise Distance

Suppose that we are given a set $S$ of $N$ points from some metric space and we want to determine the maximum distance between two points $x$ and $y$, for a given distance function $d(x, y)$. More precisely, we can define

- **Input:** Set $S$ of $N$ points represented by pairs $(i, x_i)$, for $0 \leq i < N$, where $X_i$ is the $i$-th point.
- **Output:** the pair $(0, max_{0 \leq i,j < N} d(x_i, x_j))$.

We can substantially reduce the aggregate space requirements if we tolerate a factor 2 error in the estimate of the maximum pairwise distance $d_{max}$. For an arbitrary point $x_i$ define

$$d_{max}(i) = max_{0 \leq j < N} d(x_i, x_j)$$

Then this lemma follows

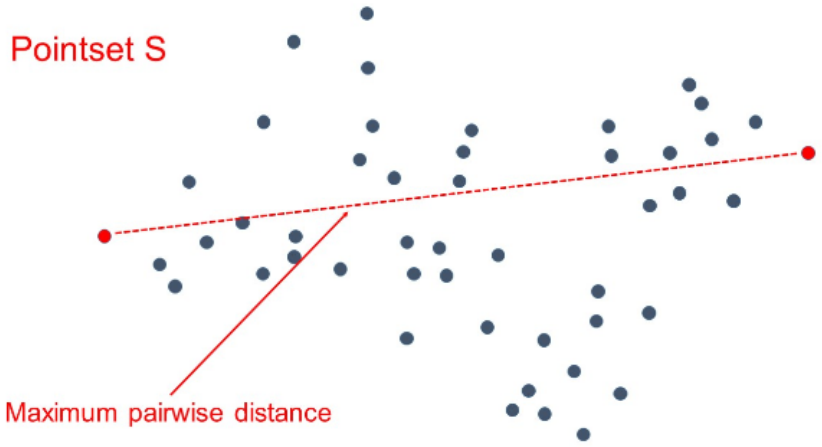


Figura 4: Maximum pairwise distance between two points

**Lemma 1**

For any $0 \leq i < N$ we have $max_{0 \leq j < N} \in [d_{max}(i), 2d_{max}(i)]$.

In other words if you compute the maximum distance starting from a given point, then the final result will be not too far from the maximum pairwise distance. Let's prove this lemma.

*Dimostrazione.* It is immediate to see that $d_{max} \geq d_{max}(i)$. Suppose that $d_{max} = d(x_r, x_t)$ for some $r, t$. By the triangle inequality we have

$$d(x_r, x_t) \leq d(x_r, x_i) + d(x_i, x_t) \leq 2d_{max}(i)$$

□

We can then exploit this lemma for deriving an efficient Map Reduce algorithm by computing the maximum pairwise distance from a given point $x_0$. After computing the distance from $x_0$ to every other point, we have an approximation to $d_{max}$.

**MR algorithm for Maximum Pairwise distance** The idea is to split the $n$ points in $\sqrt{n}$ groups of points. Then we can compute the maximum distance starting from a point $x_0$ within each group. If the groups don't include $x_0$, we'll need to replicate it inside every group. We'll then have the points with the maximum distance from $x_0$ for every group. Then we just compute the maximum between the "winner" for each group and we have the maximum pairwise distance. Let's see how this get implemented in Map Reduce:

- **Round 1:**
  - *Map phase:* map each pair $(i, x_i)$ into the intermediate pair $(imod\sqrt{N}, (i, x_i))$. Also, create the $\sqrt{N} - 1$ pairs $j, (0, x_0))$, for $1 \leq j \leq \sqrt{N}$.

- *Reduce phase:* For each key $j \in [0, \sqrt{N})$ gather the set $S^{(j)}$ of all intermediate pairs with key $j$ (which include $(j, (0, x_0))$) and produce the pair $(0, d_{max}(0, j))$ where $d_{max}(0, j)$ is the maximum distance between $x_0$ and the points associated with pairs in $S^(j)$.

- **Round 2:**
  - *Map phase:* empty.
  - *Reduce phase:* gather all pairs $(0, d_{max}(0, j))$ with $j \in [0, \sqrt{N})$ and return the output pair $(0, max_{0 \leq j < \sqrt{N}} d_{max}(0, j))$.

- **Analysis:** $R = 2$, $M_L = O(\sqrt{N}$ and $M_A = O(N)$.

# 3 Exploiting samples

When facing a big-data processing task we should ask ourselves the following question: can we profitably process a small sample of the data? In many cases, the answer is yes. Specifically, a small sample, suitably extracted, could be exploited for the following purposes:

- To subdivide the dataset in smaller subsets to be analyzed separately;
- To provide a succint yet accurate representation of the whole dataset, which contains a good solution to the problem and filters out noise and outliers, thus allowing the execution of the task on the sample.

## 3.1 Sorting

Sorting in Map Reduce is defined this way:

- **Input:** Set $S = \{s_i : 0 \leq i < N\}$ of $N$ distinct sortable objects (each $s_i$ represented as a pair $(i, s_i)$);
- **Output:** Sorted set $\{(i, s_{\pi(i)} : 0 \leq i \leq N\}$, where $\pi$ is a permutation such that $s_{\pi(1)} \leq s_{\pi(2)} \leq \ldots \leq s_{\pi(N)}$.

The Map Reduce algorithm is based on the following idea:

- Fix a suitable design parameter $K$;
- Randomly select some objects ($K$ on average) as splitters;
- Partition the objects into subsets on the ordered sequence of splitter;
- Sort each subset separately and compute the final ranks.

Therefore, the algorithm is as follows:

- **Round 1:**
  - *Map phase:* for each pair $(i, s_i)$, transform the pair into $(i \cdot mod K, (0, s_i))$ (call it regular pair). Then with probability $p = \frac{K}{N}$, independently from other objects, select $s_i$ as a splitter and, if selected, create $K$ splitter pairs $(j, (1, s_i))$, with $0 \leq j < K$. (note that a binary flag is used to distinguish between splitter and regular pairs). In particular, let $t$ be the number of splitters selected in the Map phase. At the end of the phase, the splitter pairs represent $K$ copies of the splitters, one for each key $j$.

- *Reduce phase:* For $0 \leq j < K$ do the following: gather all regular and splitter pairs with key $j$; sort the $t$ splitters and let $x_1 \leq x_2 \leq \ldots \leq x_t$ be the splitters in sorted order. Transform each regular pair $(j, (0, s))$ into the pair $(\ell, s)$, where $\ell \in [0, t]$ is such that $x_\ell < s \leq x_{\ell+1}$ (assume that $x_0 = -\infty$ and $x_{t+1} = +\infty$).

- **Round 2:**
  - *Map phase:* empty;
  - *Reduce phase:* For every $0 \leq \ell \leq t$ gather, from the output of the previous round, the set of pairs $S^{(\ell)} = \{(\ell, s)\}$, compute $N_\ell = |S^{(\ell)}|$, and create $t + 1$ replicas of $N_\ell$ (use suitable pairs for these replicas).

- **Round 3:**
  - *Map phase:* empty;
  - *Reduce phase:* For every $0 \leq \ell \leq t$ do the following: gather $S^{(\ell)}$ and the values $N_0, N_1, \ldots, N_t$; sort $S^{(\ell)}$, then compute the final output pairs for the objects in $S^{(\ell)}$ whose ranks start from $1 + \sum_{h=0}^{\ell-1} N_h$.

An example of the above algorithm can be found on the course slides.

**Analysis of MR Sample Sort** The number of rounds is clearly $R = 3$. To bound the maximum amount of local space $M_L$ required by the algorithm we can examine each round:

- In the first round, we need $K$ copies of each splitter to be created, and each reducer must store all splitter pairs and a subset of $\frac{N}{K}$ intermediate pairs. Therefore, the space is on the order of $O(K + t + \frac{N}{K})$.

- In the second round, each reduced must gather one $S^{(\ell)}$. Therefore, the space is on the order of $O(max\{N_\ell; 0 \leq \ell \leq t\})$.

- In the third round, each reducer needs to store all $N_\ell$'s and one $S^{(\ell)}$. Therefore, the space is on the order of $O(t + max\{N_\ell; 0 \leq \ell \leq t\})$.

Therefore the overall $M_L = O(t + \frac{N}{K} + max\{N_\ell; 0 \leq \ell \leq t\})$. The problem here is that we don't know the exact value of $t$, but only an expectation of it. The aggregate space $M_A$ is $O(N + t \cdot K + t^2))$, since in round 1 each splitter is replicated $K$ times, and in round 3 each $N_\ell$ is replicated $t + 1$ times. The objects are never replicated. The following lemma shows we can provide an upper bound to $t$ and $N_\ell$.

**Lemma 2**

With reference to the Map Reduce algorithm, for any $K \in (2 \log N, N)$ the following two inequalities hold with high probability (at least $1 - \frac{1}{N}$):

- $t \leq 6K$;
- $\{N_\ell; 0 \leq \ell \leq t\} \leq 4(\frac{N}{K}) \log N$.

*Dimostrazione.* We show that each inequality holds with probability at least $1 - \frac{1}{2N}$:

- $t$ is a Binomial $(N, \frac{K}{N})$ random variable with $\mathbb{E}[t] = K > 2 \log N$. By the Chernoff bound (first inequality with $\delta_1 = 5$) we have that $t > 6K$ with probability at most $\frac{1}{2N}$.

- View the sorted sequence of objects as divided into $\frac{K}{2\log N}$ contiguous segments of length $N^{'} = 2(\frac{N}{K}\log N$ each, and consider one arbitrary such segment. The probability that no splitter is chosen among the objects of the segment is

$$\left(1-\frac{K}{N}\right)^{N^{'}} = \left(1-\frac{1}{(N/K)}\right)^{(N/K)2\log N} \leq \left(\frac{1}{e^{\log N}}\right)^{2} = \frac{1}{N^2}$$

  So, there are $\frac{K}{2\log N}$ segments and we know that, for each segment, the event "no splitter falls in the segment" occurs with probability $\leq \frac{1}{N^2}$. Hence, the probability that any of these $\frac{K}{2\log N}$ events occurs is $\leq \frac{K}{N^2 2\log N} \leq \frac{1}{2N}$ (following the union bound). Therefore, with probability at least $(1-\frac{1}{2N}$, at least 1 splitter falls in each segment, which implies that each $N_\ell$ cannot be larger than $4\frac{N}{K}\log N$. Hence, we have that the second inequality stated in the lemma holds with probability at least $(1-\frac{1}{2N}$.

In conclusion, we have that the probability that at least one of the two inequalities does not hold, is at most $2\frac{1}{2N} = \frac{1}{N}$, and the lemma follows. □

From this lemma and the previous analysis, we can express the following theorem, which provides an upper bound on the memories of SampleSort:

**Theorem 3.** *By setting $K = \sqrt{N}$, MR SampleSort runs in 3 rounds, and, with high probability, it requires local space $M_L = O(\sqrt{N}\log N)$ and aggregate space $M_A = O(N)$.*

# 4 Spark

Spark is an open-source cluster-computing framework. Originally developed at the UC Berkeley in 2009, it was later donated to the Apache Software Foundation. It provides a unified set of computing engine and has a set of APIs for data analysis. Spark runs on the Java Virtual Machine. Spark does not come with a storage system (unlike Hadoop) but can run on the Hadoop Distributed File System (HDFS) as well as on other systems.The main features of Spark are:

- Fault tolerance;
- In-memory caching, which enables efficient execution of multiround algorithms: Spark tries to keep the distributed file system in the main memory to avoid the slowdown of the disk;

Spark can run in a single machine (local mode) or on a cluster manager. In particular, a singular machine does not have enough power and resources to process big data efficiently. A cluster is a group of machines whose power and resources are pooled to provided one powerful execution platform. Spark can be viewed as a tool for managing and coordinating the execution of (big data) jobs on a cluster. Without careful management and coordination, the potential added power coming from the combination of individual machines is likely to be wasted.

**Spark application** A typical spark application is composed by a driver process, which is responsible for ruining the user code. It also dispatch the code into several executors. Both he driver process and the executors interact with the cluster manager which provide the interface to the physical machine. In particular:

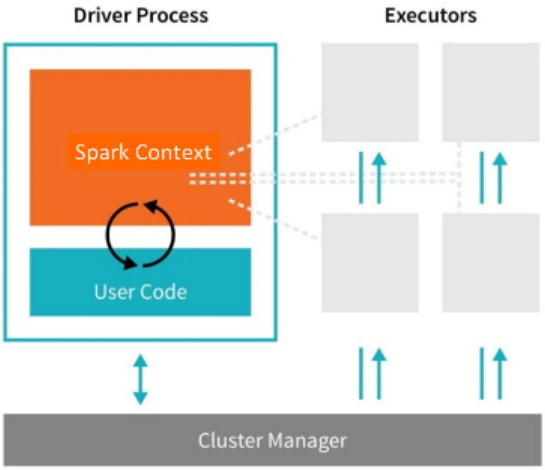


Figura 5: Spark application

- The **driver process** (master process in MapReduce) runs the *main()* function and sits on a node in the cluster. It is the heart of the application and it is responsible for maintaining information about the application, responding to a user's program or input and analysing, distributing and scheduling the work across executors. The driven process is represented by an object called Spark Context which can be regarded as a channel to access all Spark functionalities.
- The **executor processes** (worker processes in MapReduce) are responsible for actually executing the work that the driven assign them. Each executor is responsible for executing the code assigned to it by the driver and reporting the state of its computation back to the driver.
- The **cluster manager** controls physical machines and allocates resources to applications.

The driver and executors are simply processes which can run on the same machine or on different machine. While executors run Scala code, the driven can be driven from different languages from the Spark APIs.

## 4.1 Resilient Distributed Dataset

This is a fundamental abstraction in Spark. We can view a RDD as a collection of elements of the same type, which are partitioned and (possibly) distributed across several machines. An RDD provides an interface based on coarse-grained transformations. Moreover, RDDs ensure fault-tolerance. When we talk about RDD, there is one key ingredient which behind which is the data partitioning: each RDD is broken into chunks called partitions which are distributed among the available machines. A program can specify the number of partition for each RDD and decide whether using a default HashPartitioner, based on hash codes, or a custom partitioner. The number of partition is usually around 2 or 3 times the number of cores, which helps balancing the work. Partitioning is crucial for obtaining:

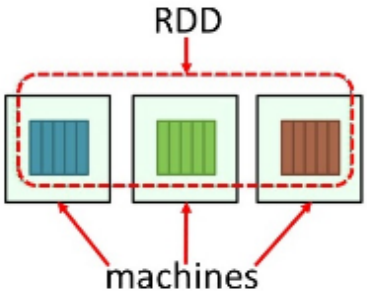


Figura 6: Resilient Distributed Dataset

- *Data reuse*: In iterative (multi-round) applications data are kept in the executor's main memories as much as possible, with the intent to avoid expensive accesses to secondary storage;
- *Parallelism*: Some data transformations are applied independently in each partition thus exploiting the parallelism offered by the underlying platform without incurring a slowdown for data movements.

RDDs are also immutable (read-only) and can be created either by loading data from a stable storage or from other RDDs trough transformations. RDDs need not be materialized at all times. Each RDDs maintains enough information regarding the sequence of transformations that generated it (its lineage), which enable the recomputation of its partitions from data in stable storage. In other words, an RDD can always be reconstructed after a failure (unless the failure affects the stable storage). Programmers can control the actual saving of RDDs in memory through the methods persist or cache.

**Operations on RDDs** The following types of operations can be performed on a RDD $A$:

- **Transformations**: a transformation generates a new RDD $B$ from the data in $A$. We distinguish between:
  - *Narrow transformations.* Each partition of $A$ contributes to at most one partition of $B$. Hence, no shuffling of data across machines is needed (maximum parallelism). E.g., the *map* method which transforms each element of $A$;
  - *Wide transformations.* Each partition of $A$ may contribute to many partitions of $B$. Hence, shuffling of data across machines may be required. E.g., the *groupByKey* method, in case $A$ consists of key-value pairs, which, for every key $k$ occurring in $A$, groups all elements of $A$ with key $k$ into a key-value pair $(k, L_k)$ of $B$, where $L_k$ is the set of values associated with $k$ in $A$.
- **Actions**: An action launches a computation on the data in $A$ which returns a value to the application (e.g., the count method which returns the number of elements in the RDD). It is at this point that the RDD $A$ is actually materialized (lazy evaluation).

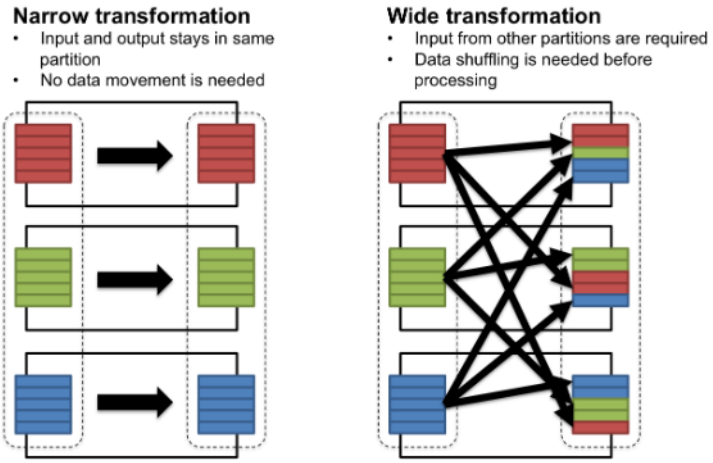


Figura 7: Transformation on RDDs

### 4.2 Implementing MapReduce algorithms in Spark

Spark enables the implementation of MapReduce algorithms but also offers a much richer spectrum of processing methods. Spark uses a core idea of functional programming, so it allows you to use functions as arguments to other functions. To implement a MapReduce round which transform a set $S$ of key-value pairs into a new set $S'$ of key-value pairs in Spark we operate as follows:

- Firstly we store $S$ in an RDD;
- **Map phase:** from the RDD, invoke one of the several map methods (narrow transformations) offered by the spark API. These methods require the map function as argument.
- **Reduce phase:** on the RDD resulting from the Map Phase, first group the key-value pairs by key (e.g., using the *groupByKey* method, a wide transformation), then on each group seen as one key-value pair (one key, many values) apply a map method. Some methods (e.g., reduceByKey) allow you to do both steps at once, in some cases.

Global variables and data structures (e.g., the dataset size $N$),stored in the driver's memory, can be used by the transformations. These global data can be defined and updated through actions.

## 5 Clustering

Given a set of points belonging to some space, with a notion of distance between points, clustering aims at grouping the points into a number of subsets (clusters) such that

- Points in the same cluster are "close" to one another;
- Points in different clusters are "distant" from one another.

The distance capture a notion of similarity: close points are similar, distant points are dissimilar. In particular, a clustering problem is usually defined by requiring that clusters optimize a given function, and/or satisfy certain properties. Numerous clustering problems have been defined,

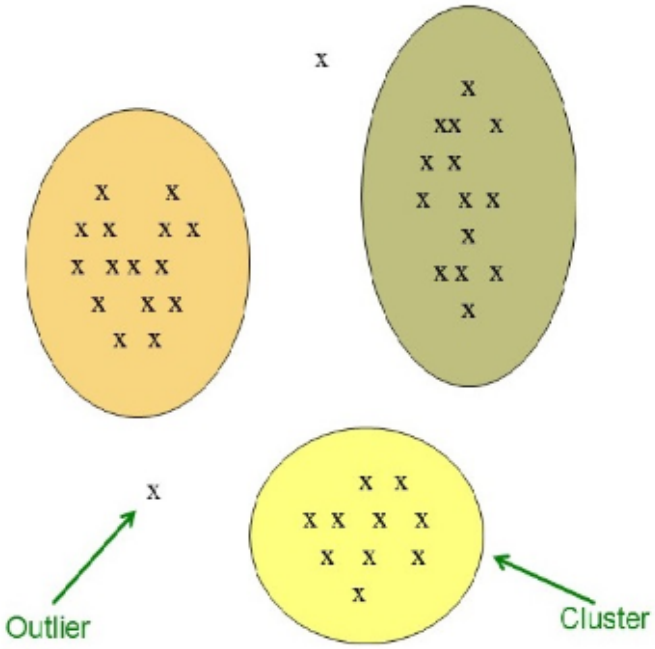


Figura 8: Example of clustering

studied and employed in applications over the years. Clustering is a hard problem: points can come in large size and multiple dimensions, making the problem complex.

**Applications of clustering** Clustering is a fundamental task in data analysis, which finds applications in many different fields, like marketing, information retrieval, image processing or biology.

## 5.1 Metric space

Typically, the input of a clustering problem is a set of points from a metric space. In particular, a metric space is an ordered pair $(M, d)$ where $M$ is the set of points we want to capture, and $d(\cdot)$ is a metric, a function which captures the distances of the points in $M$:

$$d : M \times M \longrightarrow \Re$$

such that for every $x, y, z \in M$ the following holds:

- $d(x, y) \geq 0$;
- $d(x, y) = 0$ if and only if $x = y$;
- $d(x, y) = d(y, x)$;
- $d(x, z) \leq d(x, y) + d(y, z)$ (triangle inequality).

One such class of distance functions is the

**Minkowski distances:** Let $X, Y \in \Re^n$, with $X = (x_1, x_2, \ldots, x_n)$ and $Y = (y_1, y_2, \ldots, y_n)$. For $r > 0$ the $L_r$-distance between $X$ and $Y$ is

$$d_{L_r}(X, Y) = \left( \sum_{i=1}^{n} |x_i - y_i|^r \right)^{\frac{1}{r}}$$

Let's focus now on three important cases of the minkowski distances:

- When $r = 2$ we have the euclidean distance. This is the standard distance in $\Re^n$;
- When $r = 1$ we have the Manhattan distance. This is the sum of the absolute differences of coordinates in each dimension. This is used in grid-like environments;
- When $r = \infty$ we have the Chebyschev distance. This is the maximum absolute differences of coordinates, over all dimensions.

Another class of distance funtion is the

**Cosine distance:** It is used when points are vectors in $\Re^n$ (or, in general in an inner-product space). Given two vectors $X = (x_1, x_2, \ldots, x_n)$ and $Y = (y_1, y_2, \ldots, y_n)$ in $\Re^n$, their cosine distance is the angle between them, that is

$$d_{cosine} = arcos\left( \frac{X \cdot Y}{\|X\| \cdot \|Y\|} \right) = arcos\left( \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} (x_i)^2} \sqrt{\sum_{i=1}^{n} (y_i)^2}} \right)$$

The cosine distance is often used in information retrieval to assess similarity between documents. Consider an alphabet of $n$ words. A document $X$ over this alphabet can be represented as an n-vector, where $x_i$ is the number of occurrences in $X$ of the $i-$th word of the alphabet.

**Intuition on the cosine distance** Let $X$ and $Y$ be two vectors with unit norm ($\|X\| = \|Y\| = 1$). The cosine distance measures the angle between the vectors represented by $X$ and $Y$. The inner product $X \cdot Y$ represents the projection of $Y$ on $X$(or vice versa). By definition of cosine, we have that $X \cdot Y = cos\theta$, and hence $\theta = arcos(X, Y)$. If the norm is not 1, the angle does not change: first normalize the vectors, and then compute the angle.

**Observations on the cosine distance** For non-negative coordinates it takes values in $[0, \frac{\pi}{2}]$, while for arbitrary coordinates the range is $[0, \pi]$. Dividing by $\pi/2$ (or $\pi$) the range becomes $[0, 1]$. In order to satisfy the second property of distance functions for metric spaces, scalar multiples of a vector must be regarded as the same vector.

**Hamming distance:** Used when points are binary vectors over some $n$-dimensional space, that is $X, Y \in \{0, 1\}^n$. The Hamming distance between two vectors is the number of coordinates in which they differ:

$$d_{hamming}(X, Y) = |\{i | x_i \neq y_i\}| = \sum_{i=1}^{n} |x_i - y_i| = d_{L_1}(X, Y)$$

where the equality follows beacuse $X, Y \in \{0, 1\}^n$.

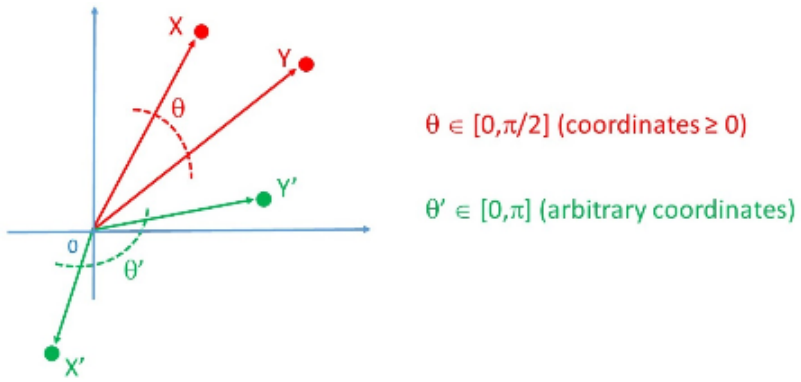


Figura 9: cosine distance transformation

**Jaccard distance:** Used when points are sets (for example documents seen as bag of words). Let $S$ and $T$ be two sets over the same ground set of elements. The Jaccard distance between $S$ and $T$ is defined as

$$d_{Jaccard}(S,T) = 1 - \frac{|S \cap T|}{|S \cup T|}$$

Note that the distance ranges in $[0,1]$, and it is 0 if and only if $S = T$ and 1 if and only if $S$ and $T$ are disjoint. The fraction in the distance definition is referred to as the Jaccard similarity of the two sets.

**Choosing the distance function** Hamming and Jaccard distances are used when an object is characterized by having or not having some features. In particular,

- *Hamming distance:* it measures the total number of differences between two objects. Examples: fixed-length binary words, genetic distance;
- *Jaccard distance:* it measures the ratio between the number of differences between two sets and the total number of points in the sets. Example: the topics describing a document.

On the other hand, Euclidean and Cosine distances are used when an object is characterized by the numerical values of its features:

- *Euclidean distance:* it aggregates the gap in each dimension between two objects. Examples: GPS coordinates, measurements of physical/chemical processes;
- *Cosine distance:* it measures the ratios among feature's values, rather than their absolute values. Example: relevance of a document since documents can have different lengths.

**Edit distance:** Used for strings. Given two strings $X$ and $Y$ , their edit distance is the minimum number of deletions and insertions that must be applied to transform $X$ into $Y$. An equivalent definition of edit distance is: Given strings $X$ and $Y$, let their Longest Common Subsequence $LCS(X,Y)$ be the longest string $Z$ whose characters occur in both $X$ and $Y$ in the same order but not necessarily contiguously. For example,

$$LCS(ABCDE, ACFDEG) = ACDE$$

It is easy to see that

$$d_{edit}(X, Y) = |X| + |Y| - 2|LCS(X, Y)|$$

$LCS(X, Y)$ can be computed in $O(|X| \cdot |Y|)$ time trough dynamic programming.

## 5.2 Curse of dimensionality

The curse of dimensionality refers to issues that arise when processing data in high-dimensional spaces. Those issues can affects both quality (distance functions may loose effectiveness in assessing similarity) and performance (the running times may have a linear or even exponential dependency on $n$). In particular,

**Quality** Random points in a high-dimensional metric space tend to be

- Sparse;
- Almost equally distant from one another;
- Almost orthogonal (as vectors) to one another.

As a consequence, in high dimensions distance functions may loose effectiveness in assessing similarity/dissimilarity. However, this holds in particular when points are random, and it might be less of a problem in some real-world datasets.

**Performance** Performance issue may affect

- *Clustering:* In many cases, the running time of a clustering algorithm can be expressed as $O(C_{dist} \cdot N_{dist})$, where $N_{dist}$ is the number of distance computations and $C_{dist}$ is the cost of a distance computation ($C_{dist} = \theta(n)$ for an $n$-dimensional Minkowski/Cosine/Hamming/Jaccard distances).
- *Nearest Neighbour search problem:* Given a set $S$ of $m$ points in $\Re^n$, construct a data structure that, for a given query point $q \in \Re^n$, efficiently returns the closest point $x \in S$ to $q$. The problem requires time

$$\theta(min\{n \cdot m, 2^n\})$$

## 5.3 Types of clustering

Given a set of points in a metric space, hence a distance between points, a clustering problem often specifies an objective function to optimize. The objective function also allows to compare different solutions.Objective functions can be categorized based on whether or not

- A target number $k$ of clusters is given in input;
- Disjoint clusters are sought;
- For each cluster a center must be identified.

When centres are required, the value of the objective function depends on the selected centres. Cluster centres must belong to the underlying metric space but, sometimes, they are constrained to belong to the input set.

## 5.4 Center based clustering

Let $P$ be a set of $N$ points in metric space $(M, d)$ and let $k$ be the target number of clusters, $1 \leq k \leq N$. We define a $k-$clustering of $P$ as a tuple $C = (C_1, C_2, \ldots C_k; c_1, c_2, \ldots c_k)$ where

- $(C_1, C_2, \ldots C_k)$ defines a partition of $P$, i.e $P = C_1 \cup C_2 \cup \ldots \cup C_k$;
- $c_1, c_2, \ldots c_k$ are suitable selected centers for the clusters, where $c_i \in C_k$, for every $1 \leq i \leq k$.

In particular, the definition above requires the centers to belong to the clusters, hence the pointset. We'll cover the 3 most popular center-based clustering problems:

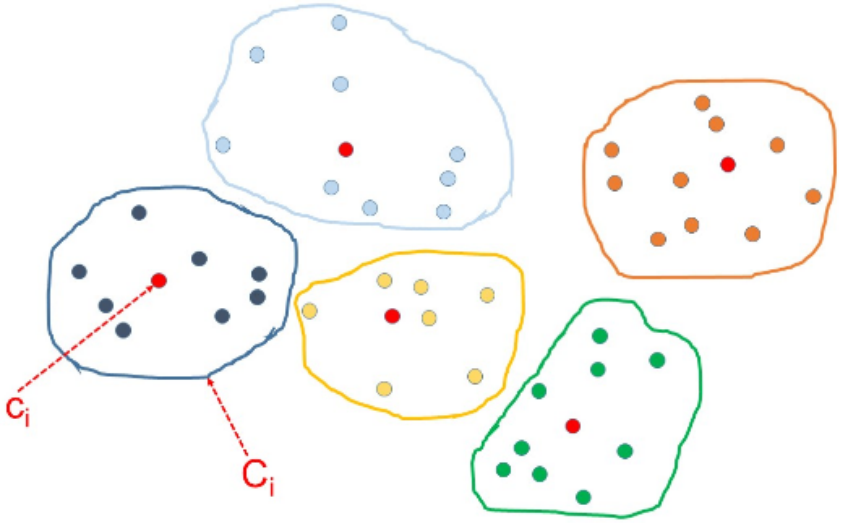


Figura 10: Example of Center based clustering

- **k-center clustering:** requires to minimize the maximum distance of any point from the closest center;
- **k-means clustering:** requires to minimize the sum of the squared distances of the points from their closest centers;
- **k-median clustering:** requires to minimize the sum of the distances of the points from their closest centers.

In particular, for k-means and k-median, minimizing the sum of(squared) distances is equivalent to minimizing the average (squared) distance.

**Center based clustering formal definition** Consider a metric space $(M, d)$ and an integer $K > 0$. A $k-$clustering problem for $(M, d)$ is an optimization problem whose instances are the finite pointsets $P \subseteq M$. For each instance $P$, the problem requires to compute a $k-$clustering $C$ of $P$ (feasible solution) which minimizes a suitable objective function $\phi(C)$. The following are the three popular objective functions:

- For *k-center clustering*, the objective function is

$$\phi_{kcenter}(C) = max_{i=1}^{k} max_{a \in C_i} d(a, c_i)$$

- For *k-means clustering*, the objective function is

$$\phi_{kmeans}(C) = \sum_{i=1}^{k} \sum_{a \in C_i} (d(a, c_i))^2$$

- For *k-median clustering*, the objective function is

$$\phi_{kmeans}(C) = \sum_{i=1}^{k} \sum_{a \in C_i} d(a, c_i)$$

Therefore our three clustering problems will require to minimize those functions. Unfortunately, all of those problems are NP-hard. However, there are several efficient approximation algorithms have been devised. Dealing efficiently with large inputs can still a challenge anyway. Regarding the choice of the problem, k-center is useful when we need to guarantee that every point is close to a center. This is also sensitive to noise. K-means/median on the other hand provide guarantees on the average (square) distances. K-means is more sensitive to noise but there exist faster algorithms to solve it.
In particular, k-center and k-median belong to the family of facility-location problems, which, given a set $F$ of candidate locations of facilities and a set $C$ of customers, require to find a subset of $k$ locations where to open facilities, and an assignment of customers to them, so to minimize the max/avg distance between a customers and its assigned facility.

**Notation** Let's introduce some useful notation. Let $P$ be a pointset from a matrix space $(M, d)$. For every integer $k \in [1, |P|]$ we define

- $\phi^{opt}_{kcenter}(P, k)$ is the minimum value of $\phi_{kcenter}(C)$ over all possible $k-$clustering $C$ of $P$;
- $\phi^{opt}_{kmeans}(P, k)$ is the minimum value of $\phi_{kmeans}(C)$ over all possible $k-$clustering $C$ of $P$;
- $\phi^{opt}_{kmedian}(P, k)$ is the minimum value of $\phi_{kmedian}(C)$ over all possible $k-$clustering $C$ of $P$;

Also, for any point $p \in P$ and any subset $S \subseteq P$ we define

$$d(p, S) = min\{q \in S : d(p, q)\}$$

which generalizes the distance function to distances between points and sets.

**Partitioning primitive** Let $P$ be a point set and $S \subseteq P$ a set of $k$ selected centers. For all previously defined clustering problems, the best k-clustering around these centers is the one where each point is assigned to the cluster of the closest center (ties broken arbitrarily). We will use primitive Partition(P,S) to denote this task:

Partition $(P, S)$

Let $S = c_1, c_2, \ldots, c_k \subseteq P$
for $i \longleftarrow 1$ to $k$ do $C_i \longleftarrow \{c_i\}$

```
for each p ∈ P − S do
    ℓ ⟵ argmin_{i=1,k}{d(p, c_i)} //ties broken arbitrarily
    C_ℓ ⟵ C_ℓ ∪ {p}
C ⟵ (C_1, C_2, ..., C_k; c_1, c_2, ..., c_k)
return C
```

## 5.5 Farthest First traversal algorithm

FFT is a popular 2-approximation sequential algorithm, which is simple and has a fast implementation when the input fits in the main memory. Besides being a good algorithm for k-center, it is also a powerful primitive for extracting samples for further analysis. In particular

- **Input:** Set $P$ of $N$ points from a metric space $(M, d)$, integer $k > 1$.
- **Output:** $k-$clustering $C = (C_1, C_2, \dots C_k; c_1, c_2, \dots c_k)$ of $P$ which is a good approximation to the k-center problem for $P$.

The pseudo code is as follows:

```
S ⟵ {c_1} // c_1 ∈ P arbitrary point
for i ⟵ 2 to k do
        Find the point c_i ∈ P − S that maximizes d(c_i, S)
        S ⟵ S ∪ {c_i}
return Partition (P, S)
```

The invocation of $Partition(P, S)$ at the end can be avoided by maintaining the best assignment of the points to the current set of centers in every iteration of the for-loop.

**Theorem 4.** *Let $C_{alg}$ be the $k$-clustering of $P$ returned by the Farthest First traversal algorithm. Then*

$$\phi_{kcenter}(C_{alg}) \leq 2 \cdot \phi^{opt}_{kcenter}(P, k)$$

*That is, Farthest-First Traversal is a 2-approximation algorithm.*

*Dimostrazione.* Let $S = \{c_1, \dots, c_k\}$ be the of centers returned by Farthest-First Traversal with input $P$, and let $S_i = \{c_1, c_2, \dots, c_i\}$ be the partial set of centers determined up to iteration $i$ of the for-loop. It is easy to see that:

$$d(c_i, S_{i-1}) \leq d(c_j, S_{j-1}) \;\; \forall 2 \leq k < i \leq k$$

Let $q$ be the point with maximum $d(q, S)$. Therefore $\phi_{kcenter}(C_{alg}) = d(q, S) = d(q, S_k)$. Let $c_i$ be the closest center to $q$, that is $d(q, S) = d(q, c_i)$. We now show that the $k+1$ points $c_1, \dots, c_k, q$ are such that any two of them are at distance at least $\phi_{kcenter}(C_{alg})$ from one another. Indeed, since we assumed $c_i$ to be the closest center to $q$, we have

$$\phi_{kcenter}(C_{alg}) = d(q, c_i) \leq d(q, c_j) \; \forall j \neq i$$

Note that $q$ would be the point selected by the algorithm as $(k+1)$-st center. Hence, by the first inequality from this proof, we have that for any pair of indexes $j1, j2$ with $\leq j1 < j2 \leq k$,

$$\phi_{kcenter}(C_{alg}) = d(q, S_k) \leq d(c_k, S_{k-1}) \leq d(c_{j2}, S_{j2-1}) \leq d(c_{j2}, c_{j1})$$

Now we know that all pairwise distances between $c_1, c_2, \ldots, c_k, q$ are at least $\phi_{kcenter}(C_{alg})$ Clearly, by the pigeon hole principle, two of these $k+1$ points must fall in the same cluster of the optimal clustering. Suppose that $x, y$ are two points in $\{c_1, c_2, \ldots, c_k, q\}$ which belong to the same cluster of the optimal clustering with center $c$. By the triangle inequality, we have that

$$\phi_{kcenter}(C_{alg}) \leq d(x,y) \leq d(x,c) + d(c,y) \leq 2\phi^{opt}_{kcenter}(P,k)$$

and the theorem follows. □

Let's go over some *observation* on k-center clustering. k-center clustering provides a strong guarantee on how close each point is to the center of its cluster. For noisy pointsets the clustering which optimizes the k-center objective may obfuscate some "natural" clustering inherent in the data. Moreover, for any fixed $\varepsilon > 0$ it is NP-hard to compute a $k-$clustering $C_{alg}$ of a pointset $P$ with

$$\phi_{kcenter}(C_{alg}) \leq (2-\varepsilon)\phi^{opt}_{kcenter}(P,k)$$

hence the Farthest-First Traversal is likely to provide almost the best approximation guarantee obtainable in polynomial time.

## 5.6 K-center clustering for big data

Farthest-First Traversal requires $k-1$ scans of pointset $P$ which is impractical for massive pointsets and not so small $k$. In these cases, we resort to the following general technique, which we call **coreset technique**. The idea is using sampling to extract a "small" subset of the input (called coreset), making sure it contains a good solution to the problem. Then we can run the best known (sequential) algorithm to solve the problem on the small coreset, rather than on the entire input, still obtaining a fairly good solution. The advantage is that the best known algorithm might be expensive, so when we use a reduced coreset it will not be a problem. In order for this technique to be effective we need to extract efficiently the coreset and, in particular, when it can be composed by combining smaller coreset independently extracted from some (arbitrary) partition of the input. How do we instantiate this idea for the k-center problem? Consider a

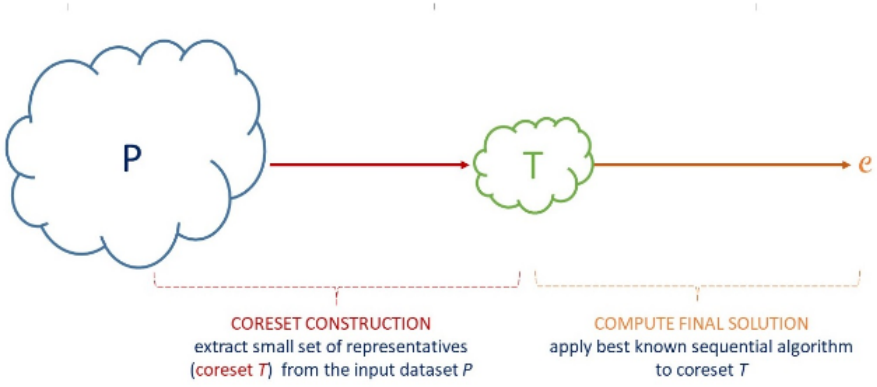


Figura 11: Graphical representation of the coreset technique

large pointset $P$ and a target granularity (number of clusters) $k$. We select a coreset $T \subset P$ making sure that for each $x \in P-T$ is close to some point of $T$ (close in the sense of approximately

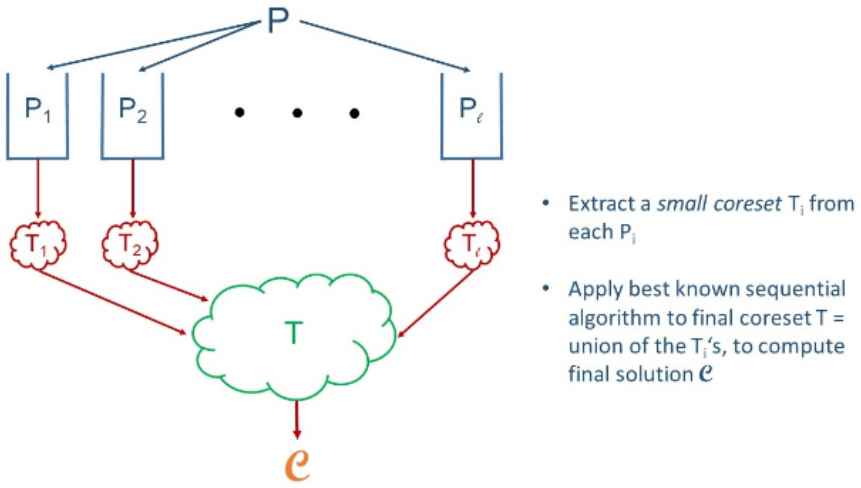


Figura 12: Graphical representation of the composable coreset technique

within $\phi^{opt}_{kcenter}(P,k)$). Then we search the set $S$ of $k$ final centers in $T$, so that each point of $T$ is at distance approximately within $\phi^{opt}_{kcenter}(P,k)$ from the closest center. By combining those two steps, we gather that each point of $P$ is at distance $O(\phi^{opt}_{kcenter}(P,k))$ from the closest center. The Map Reduce algorithm described next implements this idea using a composable coreset construction.

**Map Reduce Farthest-First Traversal** Let $P$ be a set of $N$ points from a metric space $(M,d)$, and let $k>1$ be an integer. The following MapReduce algorithm (MR-Farthest-First Traversal) computes a good k-center clustering:

- **Round 1:** Partition $P$ arbitrarily in $\ell$ subsets of equal size $P_1, P_2, \ldots, P_\ell$ and execute the Farthest-First Traversal algorithm on each $P_i$ separately to determine a set $T_i$ of $k$ centers, for $1 \leq i \leq \ell$.
- **Round 2:** Gather the coreset $T = \cup_{i=1}^{\ell} T_i$ (of size $\ell \cdot k$) and run, using a single reducer, the Farthest-First Traversal algorithm on $T$ to determine a set $S = \{c_1, c_2, \ldots, c_k\}$ of $k$ centers.
- **Round 3:** Execute $Partition(P,S)$.

Note that in Rounds 1 and 2 the Farthest-First traversal algorithm is used to determine only centers and not complete clusterings.

**Algorithm analysis** Assume $k \leq \sqrt{N}$. By setting $\ell = \sqrt{\frac{N}{k}}$, the 3-round MR-Farthest-First traversal algorithm uses

- Local space $M_L = O(\sqrt{N \cdot k}) = o(N)$. In detail, in round 1 we have $O(\frac{N}{\ell}) = O(\sqrt{N \cdot k})$, in round 2 we have $O(\ell \cdot k) = O(\sqrt{N \cdot k})$ , and in round 3 we have $O(\sqrt{N})$.

- Aggregate space $M_A = O(N)$.

We have a constant number of round, a linear aggregate space and a sublinear local space, but we need to verify if the solution is good. This is addressed by the following theorem:

**Theorem 5.** *Let $C_{alg}$ be the $k-$clustering of $P$ returned by the MR-Farthest-First Traversal algorithm. Then:*

$$\phi_{kcenter}(C_{alg}) \leq 4\phi^{opt}_{kcenter}(P,k)$$

*That is, MR-Farthest-First Traversal is a 4-approximation algorithm.*

*Dimostrazione.* For $1 \leq i \leq \ell$, let $d_i$ be the maximum distance of a point of $P_i$ from the centers $T_i$ determined in Round 1. By reasoning as in the analysis of Farthest-First Traversal, we can show that there exist $k+1$ points in $P_i$ whose pairwise distances are all $\leq d_i$. At least two such points, say $x, y$, must belong to the sae cluster of the optimal clustering for $P$ (not $P_i$), with center $c$. Therefore,

$$d_i \leq d(x,y) \leq d(x,c) + d(c,y) \leq 2\phi^{opt}_{kcenter}(P,k)$$

Consider now the set of centers $S$ determined in round 2 from the coreset $T$, and let $d_T$ be the maximum distance of a point of $T$ from $S$. The same argument as above shows that

$$d_T \leq 2\phi^{opt}_{kcenter}(P,k)$$

Finally, consider an arbitrary point $p \in P$ and suppose that $p \in P_i$, for some $1 \leq i \leq \ell$. By virtue of the two inequalities derived previously, we know that there must exist a point $t \in T_i$ (hence $t \in T$) and a point $c \in S$, such that

$$d(p,t) \leq \phi^{opt}_{kcenter}(P,k)$$

$$d(t,c) \leq \phi^{opt}_{kcenter}(P,k)$$

Therefore by triangle inequality, we have that

$$d(p,c) \leq d(p,t) + d(t,c) \leq 4\phi^{opt}_{kcenter}(P,k)$$

and this immediately implies that

$$\phi_{kcenter}(C_{alg}) \leq 4\phi^{opt}_{kcenter}(P,k)$$

□

**Observations on MR-Farthest-First Traversal** The sequential Farthest-First Traversal algorithm is used both to extract the coreset and to compute the final set of centers. It provides a good coreset since it ensures that any point not in the coreset be well represented by some coreset point. MR-Farthest-First Traversal is able to handle very large pointsets and the the final approximation is not too far from the best achievable one. By selecting $k' > k$ centers from each subset $P_i$ in Round 1, the quality of the final clustering improves. In fact, it can be shown that when $P$ satisfy certain properties and $k'$ is sufficiently large, MR-Farthest-First Traversal can almost match the same approximation quality as the sequential Farthest-First Traversal, while, still using sublinear local space and linear aggregate space.

## 5.7 K-means clustering

Given a pointset $P$ from a metric space $(M, d)$ and an integer $K > 0$, the $k-$means clustering problem requires to determine a $k-$clustering $C = (C_1, C_2, \dots C_k; c_1, c_2 \dots, c_k)$ which minimizes

$$\phi_{kmeans}(C) = \sum_{i=1}^{k} \sum_{a \in C_i} (d(a, c_i))^2$$

We'll focus on the euclidean spaces (we consider $\Re^d$ for the euclidean distance). Moreover, we'll focus on the unrestricted version of the problem where centers need not belong to the input $P$. K-means clustering aims at minimizing cluster variance, and works well for discovering ball-shaped clusters. K-means is also quite sensitive to outliers due to the quadratic dependence on distances (although less than k-center).
Let's briefly focus on some properties of Euclidean spaces:

**Properties of Euclidean spaces** Let $X = (X_1, X_2, \dots X_D)$ and $Y = (Y_1, Y_2, \dots, Y_D)$ be two points in $\Re^d$. Recall that their Euclidean distance is

$$d(X, Y) = \sqrt{\sum_{i=1}^{D} (X_i - Y_i)^2} = \|X - Y\|$$

Then we can define the **centroid** of a set $P$ of $N$ points in $\Re^d$ as the average of the all points in $P$:

$$c(P) = \frac{1}{N} \sum_{x \in P} x$$

where the sum is component-wise. In particular, the centroid doesn't necessary belongs to $P$. An important property we'll exploit in the LLoid's algorithm is the following:

**Lemma 3**

The centroid $c(P)$ of a set $P \subset \Re^d$ is the point of $\Re^d$ which minimizes the sum of the square distances to all points of $P$.

*Dimostrazione.* Consider an arbitrary point $Y \in \Re^d$. We have that

$$\sum_{X \in P} (d(X, Y))^2 = \sum_{X \in P} \|X - Y\|^2 = \sum_{x \in P} \|X - c_P + c_P - Y\|^2 = \sum_{X \in P} (X - c_P + c_P - Y) \cdot (X - c_P + c_P - Y)$$

By using the linearity of the inner product, we obtain

$$\sum_{X \in P} (d(X, Y))^2 = \sum_{X \in P} [(X - c_P) \cdot (X - c_P) + (X - c_P) \cdot (c_P - Y) + (c_P - Y) \cdot (X - c_P) +$$
$$+ (c_P - Y) \cdot (c_P - Y)] = \sum_{X \in P} \|X - c_P\|^2 + N\|c_P - Y\|^2 + 2(c_P - Y) \cdot \sum_{X \in P} (X - c_P)$$

By definition of $c_P$ we have that $\sum_{X \in P}(X - c_P) = (\sum_{X \in P} X) - N_{c_P}$ is the all 0's vector, hence

$$\sum_{X \in P} (d(X, Y))^2 = \sum_{X \in P} \|X - c_P\|^2 + N\|c_P - Y\|^2 \geq \sum_{X \in P} \|X - c_P\|^2 = \sum_{X \in P} (d(X, c_P))^2$$

□

The lemma implies that when seeking a $k-$clustering for points in $\Re^D$ which minimizes the kmeans objective, the best center to select for each cluster is its centroid (assuming that centers need not necessary belong to the input pointset).

**LLoyd's algorithm** Also known as k-means algorithm, it's one of the most popular algorithm for data mining. The structure of the algorithm is simple and leverage overt two properties: the partitioning, and the centroids. The LLoyd's algorithm uses those two properties in an iterative way: we start by selecting random centers in our domain $\Re^D$ and we partition our input points using those centers. Then, we compute the centroids and we replace the centers. We assign points according to the new centers and repeat until we have an improvement of the objective function. This is related to the Expectation-Maximization scheme.

**Pseudo code**

- **input:** Set $P$ of $N$ points from $\Re^d$, integer $k > 1$.
- **output:** $k-$clustering $C = (C_1, C_2, \ldots C_k; c_1, c_2 \ldots, c_k)$ of $P$ with small $\phi_{kmeans}(C)$. The centres does not needs to be in $P$.

```
S ⟵ arbitrary set of k points in ℜ^D
φ ⟵ ∞; stopping−condition ⟵ false
while (!stopping−condition) do
        (C_1, C_2, ..., C_K; S) ⟵ Partition (P, S)
        for i ⟵ 1 to k do
                c_i' ⟵ centroid of C_i
        C ⟵ C_1, C_2, ..., C_K; c_1', c_2', ..., c_k')
        S ⟵ {c_1', c_2' ... c_k'}
        if φ_kmeans(C) < φ then φ ⟵ φ_kmeans(C)
        else stopping−condition ⟵ true
return C
```

**Lloyd'salgorithm analysis** The following theorem holds:

**Theorem 6.** *The Lloyd's algorithm always terminates.*

*Dimostrazione.* Let $C_t$ be the $k-$clustering computed by the algorithm in iteration $t \geq 1$ of the while loop. We have that:

- By construction, the objective function values $\phi_{kmeans}(C_t)$'s, for every $t$ except for the last one, form a strictly decreasing sequence. Also, in the last iteration (say Iteration $t^*$) $\phi_{kmeans}(C_t^*) = \phi_{kmeans}(C_{t^*-1})$, since the invocation of Partition and the subsequent re-computation of the centroids in this iteration cannot increase the value of the objective function.
- Each $C_t$ is uniquely determined by the partition of $P$ into $k$ clusters computed at the beginning of iteration $t$.
- By the above two points we conclude that all $C_t$'s, except possibly the last one, are determined by distinct partitions of $P$.

The theorem follows immediately since there are $k^N$ possible partitions of $P$ into $\leq k$ clusters. □

**Observations** In particular, Lloyd's algorithm may be trapped into a local optimum whose value of the objective function can be much larger than the optimal value. Hence, no guarantee can be proved on its accuracy. While the algorithm surely terminates, the number of iterations can be exponential in the input size. Besides the trivial $k^N$ upper bound on the number of iterations, more sophisticated studies proved an $O(N^{kD})$ upper bound which improves upon the trivial one, in scenarios where $k$ and $D$ are small. Some recent studies proved also a $2\Omega(\sqrt{N})$ lower bound on the number of iterations in the worst case. Empirical studies show that, in practice, the algorithm requires much less than $N$ iterations, and, if properly seeded (see next slides) it features guaranteed accuracy. To improve performance, with a limited quality penalty, one could also stop the algorithm earlier. The quality of the solution and the speed of convergence of Lloyd's algorithm depend considerably from the choice of the initial set of centers. Typically, initial centers are chosen at random, but there are more effective ways of selecting them. One such way is the algorithm k-means++, a careful center initialization strategy that yields clusterings not too far from the optimal ones.

**K-Means++** Let $P$ be a set of $N$ points from $\Re^D$, and let $k > 1$ be an integer. Algorithm k-means++ computes a (initial) set $S$ of $k$ centers for $P$ using the following procedure (note that $S \subseteq P$):

$c_1 \longleftarrow$ random point chosen from $P$ with uniform probability
$S \longleftarrow \{c_1\}$
for $i \longleftarrow 2$ to $k$ do
    $c_i \longleftarrow$ random point chosen from $P - S$, where a point $p$ is chosen with probability $(d(p,S))^2 / \sum_{q \in P-S} (d(q,S))^2$
    $S \longleftarrow S \cup \{c_i\}$
return $S$

Altough k-means++ already provides a good set of centers for the k-means problem (see next theorem) it can be used to provide the initialization for Lloyd's algorithm, whose iterations can only improve the initial solution.

**Details on the selection of $c_i$ in k-means++** Consider the iteration $i$ of the for loop of k-means++ ($i \geq 2$). Let $S$ be the current set of $i-1$ already discovered centers. Let $P - S = \{p_j : 1 \leq j \leq |P - S|\}$ (arbitrary order), and define $\pi_j = (d(p_j, S))^2 / \sum_{q \in P-S} (d(q,S))^2$. Note that the $\pi_j$'s define a probability distribution, that is, $\sum_{j>1} \pi_j = 1$. Draw a random number $x \in [0,1]$ and set $c_i = p_r$, where $r$ is the unique index between 1 and $|P - S|$ such that

$$\sum_{j=1}^{r-1} \pi_j < x \leq \sum_{j=1}^{r} \pi_j$$

Let's consider now the following theorem:

**Theorem 7.** *Let $\phi^{opt}_{kmeans}(k)$ be the minimum value of $\phi_{kmeans}(C)$ over all possible k-clusterings $C$ of $P$, and let $C_{alg} = Partition(P, S)$ be the $k-$clustering of $P$ induced by the set $S$ of centers returned by the k-means++. Note that $C_{alg}$ is a random variable which depends on the random choices made by the algorithm. Then:*

$$E[\phi_{kmeans}(C_{alg})] \leq 8(\log k + 2) \cdot \phi^{opt}_{kmeans}(k)$$

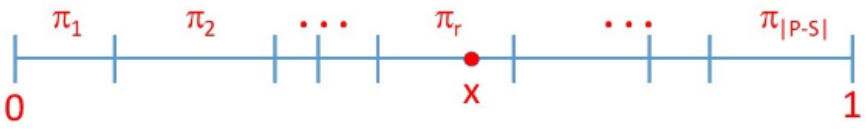


Figura 13: Selection of the random value $x$

*where the expectation is over all possible sets $S$ returned by k-means++, which depend on the random choices made by the algorithm.*

## 5.8 K-means clustering for big data

To compute a "good" k-means clustering of a pointset $P$ that is too large for a single machine we can either use:

- Distributed (MapReduce) implementation of Lloyd's algorithm and k-means++;
- Coreset-based approach, similar in spirit to what was done for k-center. This approach lends itself to efficient distributed as well as streaming implementations.

**Lloyd's and k-means++ in MapReduce** The algorithms are left as exercises (see slides).

**Coreset-based approach for k-means** We use a coreset-based approach similar to the one used fork-center. The difference with respect to the k-center case is that the local coresets $T_i$'s are computed using a sequential algorithm for k-means, but most importantly each point of $T$ is given a weight equal to the number of points in $P$ it represents, so to account for their cumulative contribution to the objective function.

**Weighted k-means clustering** We can define a weighted, more general, variant of the k-means clustering problem. Let $P$ be a set of $N$ points from a metric space $(M, d)$, and let $k < N$ be the target granularity. Suppose that for each point $p \in P$, an integer weight $w(p) > 0$ is also given. We want to compute the $k-$clustering $C = (C_1, C_2, \ldots C_k; c_1, c_2, \ldots, c_k)$ of $P$ which minimizes the following objective function

$$\phi^{w}_{kmeans}(C) = \sum_{i=1}^{k} \sum_{p \in C_i} w(p) \cdot (d(p, c_i))^2$$

That is, the weighted sum of the squared distances of the points from their closest centers. Note that the unweighted k-means clustering problem is a special case of the weighted variant where all weights are equal to 1. Most of the algorithms known for the standard k-means clustering problem (unit weights) can be adapted to solve the case with general weights.It is sufficient to regard each point $p$ as representing $w(p)$ identical copies of itself. The same approximation guarantees are maintained. For example:

- Lloyd's algorithm remains virtually identical except that the objective function $\phi_{wkmeans}(C)$ is used, and the centroid of a cluster $C_i$ becomes

$$\frac{1}{\sum_{p \in C_i} w(p)} \sum_{p \in C_i} w(p) \cdot p$$

- K-means++ remains virtually identical except that in the selection of the $i-$th center $c_i$, the probability for a point $p \in P - S$ to be selected becomes

$$\frac{w(p) \cdot (d(p,S))^2}{\sum_{q \in P-S}(w(q) \cdot (d(q,S))^2)}$$

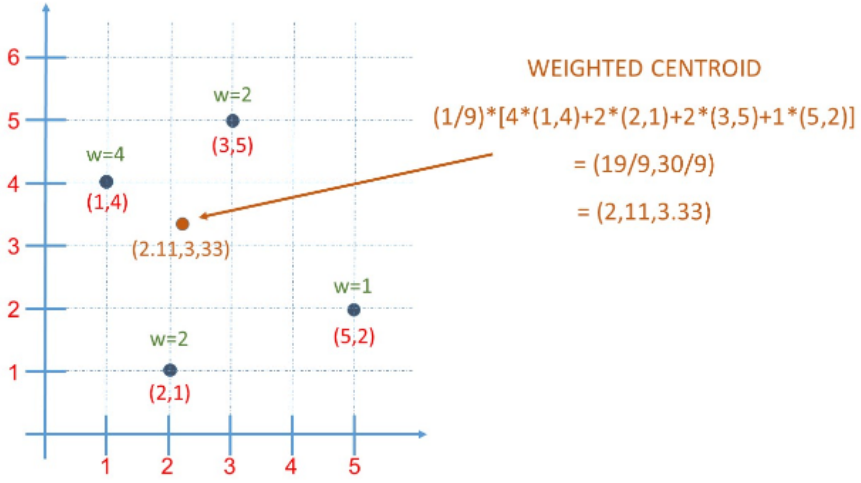


Figura 14: Example of weighted centroid

**Coreset-Based MapReduce algorithm for k-means** The Mapreduce algorithm for k-means uses as a parameter a sequential algorithm $A$ for k-means. We call this algorithm **MR-kmeans(A)**. In principle, one can instantiate $A$ with any sequential algorithm, as long as the following characteristics are met:

- $A$ solves the more general weighted variant maintaining the same approximation ratio;
- $A$ requires space proportional to the input size.

**MR-kmeans(A)**

- **Input**: Set $P$ of $N$ points from a metric space $(M, d)$, integer $K > 1$.
- **Output**: k-clustering $C = (C_1, C_2, \ldots C_k; c_1, c_2, \ldots, c_k)$ of $P$ which is a good approximation to the k-means problem for $P$.

The algorithm has 3 rounds:

- **Round 1:** partition $P$ arbitrarily in $\ell$ subsets of equal size $P_1, P_2, \ldots P_\ell$ and execute $A$ independently on each $P_i$ (with unit weights). For $1 \leq i \leq \ell$, let $T_i$ be the set of $k$ centers computed by $A$ on $P_i$. For each $p \in P_i$ define its proxy $\pi(p)$ as its closest center in $T_i$, and for each $q \in T_i$, define its weight $w(q)$ as the number of points of $P_i$ whose proxy is $q$.

- **Round 2:** Gather the coreset $T = \bigcup_{i=1}^{\ell} T_i$ of $\ell \cdot k$ points, together with their weights. using a single reducer, run $A$ on $T$, using the given weights, to identify a set $S = \{c_1, c_2, \ldots, c_k\}$ of $k$ centers.
- **Round 3:** run $Partition(P, S)$ to return the final clustering.

**Analysis of MR-kmeans(A)** Assume $k \leq \sqrt{N}$. By setting $\ell = \sqrt{N/k}$, it is easy to see that the 3-round MR-kmeans(A) algorithm can be implemented using:

- Local space $(M_L) = O(max\{N/\ell, \ell \cdot k, \sqrt{N}\}) = O(\sqrt{N \cdot k}) = o(N)$;
- Aggregate space $M_A = O(N)$.

The quality of the solution returned by the algorithm is stated in the following theorem, which is proved in the next few slides.

**Theorem 8.** *Suppose that $A$ is an $\alpha$-approximation algorithm for weighted k-means clustering, for some $\alpha > 1$, and let $C_{alg}$ be the k-clustering returned by MR-kmeans(A) with input $P$. Then:*

$$\phi_{kmeans}(C_{alg}) = O(\alpha^2 \cdot \phi^{opt}_{kmeans}(P, k))$$

*That is, MR-kmeans(A)is an $O(\alpha^2)$-approximation algorithm.*

In order to show this more clearly we need to introduce the following:

**Definition 2**

Given a pointset $P$, a coreset $T \subseteq P$ and a proxy function $\pi : P \longrightarrow T$, we say that $T$ is a $\gamma-$coreset for $P, k$ and the k-means objective if

$$\sum_{p \in P} (d(p, \pi(p))^2 \leq \gamma \cdot \phi^{opt}_{kmeans}(P, k)$$

Intuitively, the smaller $\gamma$, the better $T$ (together with the proxy function) represents $P$ with respect to the k-means clustering problem,and it is likely that a good solution for the problem can be extracted from $T$, by weighing each point of $T$ with the number of points for which it acts as proxy. The following lemma relates the quality of the solution returned by MR-kmeans(A) to the quality of the coreset $T$ and to the approximation guarantee of $A$.

**Lemma 4**

Suppose that the coreset $T$ computed in Round 1 of MR-kmeans(A) is a $\gamma$-coreset and that $A$ is an $\alpha$-approximation algorithm for weighted k-means clustering, for some $\alpha > 1$. Let $C_{alg}$ be the $k-$clustering of $P$ returned by MR-kmeans(A). Then:

$$\phi_{kmeans}(C_{alg}) = O(\alpha \cdot (1 + \gamma) \cdot \phi^{opt}_{kmeans}(P, k))$$

The lemma implies that by chosing a $\gamma$-coreset with small $\gamma$,the penalty in accuracy incurred by the MapReduce algorithm with respect to the sequential algorithm is small. This shows that accurate solutions even in the big data scenario can be computed.

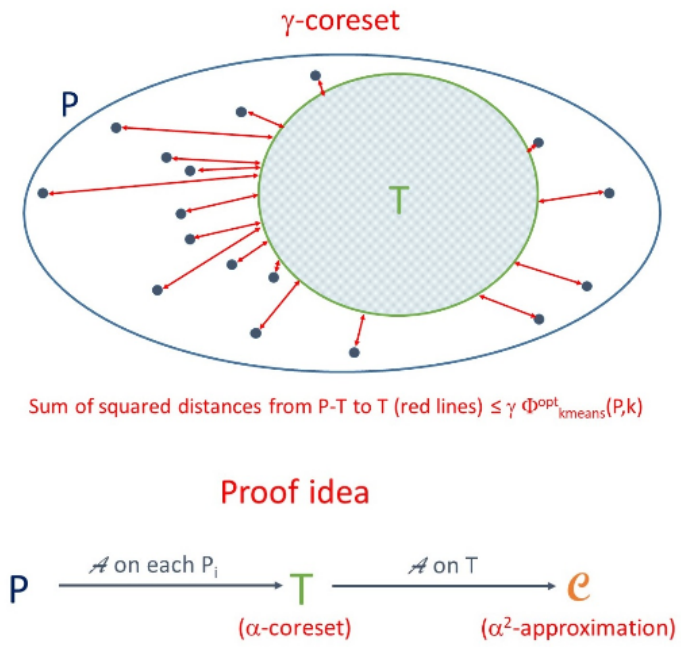


Figura 15: $\gamma$-coreset

**Lemma 5**

If $A$ is an $\alpha$-approximation algorithm for k-means clustering, then the coreset $T$ computed in Round 1 of MR-kmeans(A) is a $\gamma$-coreset with

$$\gamma = \alpha$$

*Dimostrazione.* Let $S^{opt}$ be the set of optimal centers for $P$. Let also $S_i^{opt}$ be the set of optimal centers for $P_i$. Recall that for every $1 \leq i \leq \ell$, for each $p \in P_i$ its proxy $\pi(p)$ is the point of $T_i$ closest to $p$, thus $d(p, \pi(p)) = d(p, T_i)$. Hence, we have

$$\sum_{p \in P} d^2(p, \pi(p)) = \sum_{i=1}^{\ell} \sum_{p \in P_i} (d(p, T_i))^2 \leq \sum_{i=1}^{\ell} \alpha \sum_{p \in P_i} (d(p, S_i^{opt}))^2 \leq \alpha \sum_{i=1}^{\ell} \sum_{p \in P_i} (d(p, S^{opt}))^2$$

$$= \alpha \sum_{p \in P} (d(p, S^{opt}))^2 = \alpha \phi_{kmeans}^{opt}(P, k)$$

□

The theorem proof follows from the lemmas.

## 5.9 Cluster Evaluation

When we talk about cluster evaluation we might refer to

- *Clustering tendency:* assessment whether the data contain meaningful clusters, namely clusters that are unlikely to occur in random data;
- *Unsupervised evaluation:* assessment of the quality of a clustering (or the relative quality of two clusterings) without reference to external information;
- *Supervised evaluation:* assessment of the quality of a clustering (or the relative quality of two clusterings) with reference to external information.

**Clustering tendency: Hopkins statistic** The goal of this statistic is to understand if our input set behaves as some random data. In particular, we know that randomly generated data are likely not having a clustering structure.
Let $P$ be a dataset of $N$ points in some metric space $(M, d)$. The *Hopkins statistic* measures to what extent $P$ can be regarded as a random subset from $M$. For some fixed $t \ll N$ (typically $t < 0.1 \cdot N$) let:

$$X = \{x_1, x_2, \ldots, x_t\} \text{ random sample from P}$$
$$Y = \{y_1, y_2, \ldots, y_t\} \text{ random set of points from M}$$
$$w_i = d(x_i, P - \{x_i\}) \text{ for } 1 \leq i \leq t$$
$$u_i = d(y_i, P - \{y_i\}) \text{ for } 1 \leq i \leq t$$

The Hopkins statistic is then defined as

$$H(P) = \frac{\sum_{i=1}^{t} u_i}{\sum_{i=1}^{t} u_i + \sum_{i=1}^{t} w_i}$$

If $P$ is truly random, the terms at the denominator will be similar, and the statistic will be around $\frac{1}{2}$. If we have separated cluster, the sum of $w_i$ will be small compared to the sum of $u_i$, and we'll have a Hopkins statistic of almost 1. If instead we have points which are not random but very spread, the sum of $w_i$ dominate the denominator so we get something close to zero.
$H(P)$ is, by definition, based on sampling, hence it is suited for use in large datasets.

**Unsupervised evaluation** We want to see if the cluster we obtained are meaningful, even when our input data have not a cluster structure. In the case of k-center, k-means, and k-median, the value of the objective function is the most natural metric to assess the quality a clustering or the relative quality of two clusterings. However, the objective functions of k-center/means/median capture intra-cluster similarity but do not assess whether distinct clusters are truly separated. Therefore we don't know what is the correct number of cluster. In what follows we present two popular approaches for measuring intra-cluster similarity and inter-cluster dissimilarity: *Cohesion/Separation metrics* and *Silhouette coefficient*.

**Sum of distances** The approaches to unsupervised evaluation which we will describe require numerous applications of a primitive that computes the sum of the distances between a given point and a subset of points.
Given a metric space $(M, d)$, a point $p \in M$, and a subset $C \subseteq M$ define

$$d_{sum}(p, C) = \sum_{q \in C} d(p, q)$$

Clearly, from $d_{sum}(p, C)$ one immediately obtains the average distance between $p$ and $C$ as $s_{sum}(p, C)/|C|$.
Computing one $d_{sum}(p, C)$ can be done efficiently, even for large $C$, but computing many of them at once may become too costly. Hence we will discuss how to get approximate estimates efficiently and how accurate these estimates are.

**Unsupervised evaluation: cohesion/separation** Let $C = (C_1, C_2, \ldots, C_k)$ be a k-clustering of a pointset $P$ from a metric space $(M, d)$ (ignoring centers, if any exist), and let $N_i = |C_i|$, $\forall i$. The *cohesion* of $C$ is the average distance between points in the same clusters. Therefore we have

$$cohesion(C) = \frac{\sum_{i=1}^{k} \frac{1}{2} \sum_{p \in C_i} d_{sum}(p, C_i)}{\sum_{i=1}^{k} \binom{N_i}{2}}$$

where the factor $\frac{1}{2}$ avoids counting the same distance twice. The *separation* of $C$ is the average distance between points in the different clusters. Therefore we have

$$separation(C) = \frac{\sum_{1 \le i < j \le k} \sum_{p \in C_i} d_{sum}(p, C_j)}{\sum_{1 \le i < j \le k} N_i \cdot N_j}$$

The larger the gap between cohesion and separation,the better the quality of the clustering.

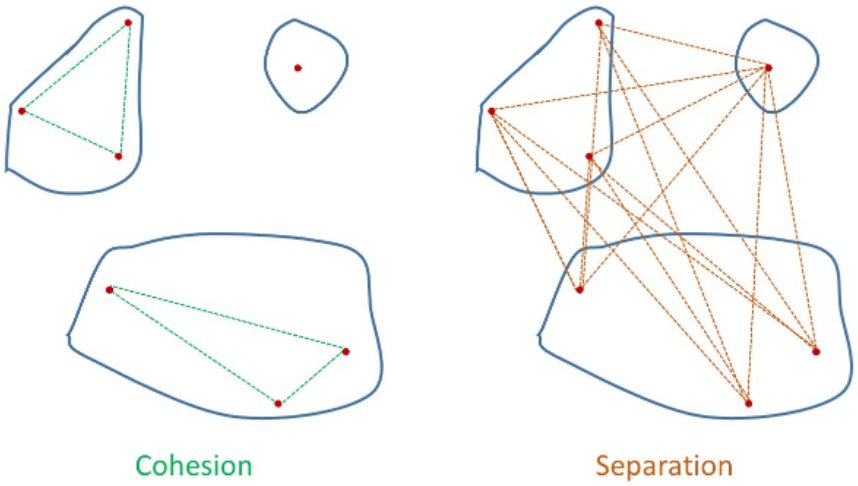


Figura 16: Example of cohesion and separation

**Unsupervised evaluation: silhouette** Let us consider the same clustering $C = (C_1, C_2, \ldots, C_k)$ as before. For a point $p \in P$ belonging to some cluster $C_i$ let $a_p$ be the average distance between $p$ and the other points in its cluster, and $b_p$ the minimum, over all clusters $C_j \neq C_i$ of the average distance between $p$ and the other points in $C_j$, that is:

$$a_p = d_{sum}(p, C_i)/N_i$$

$$b_p = min_{j \neq i} d_{sum}(p, C_j)/N_j$$

The silhouette coefficient for $p$ is

$$s_p = \frac{b_p - a_p}{max\{a_p, b_p\}}$$

which is a value between $-1$ (i.e., $b_p = 0$) and 1 (i.e., $a_p = 0$). $s_p$ is close to 1 when $p$ is much closer to the points of its cluster than to the points of the closest clusters (i.e.,$a_p \ll b_p$), while it is close to -1 when the opposite holds. To measure the quality of $C$ we define the *average*

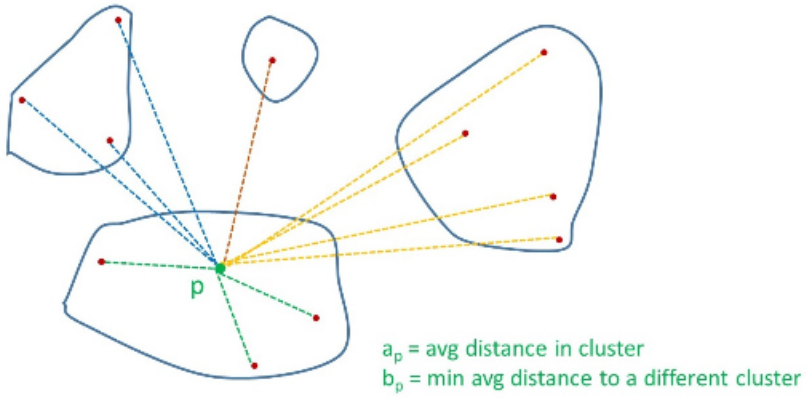


Figura 17: Example of silhouette

*silhouette coefficient* as

$$s_C = \frac{1}{|P|} \sum_{p \in P} s_p$$

A "good" clustering has $s_C \simeq 1$, i.e., for most points $p$, $b_p \ll a_p$.

**Approximating the sum of distances** Computing exactly $cohesion(C)$ and $separation(C)$, or the average sihouette coefficient $s_C$ for a clustering $C$ of $P$ essentially entails computing all pairwise distances between points and is impractical for a very large $P$. Since the metrics have been expressed in terms of sums of distances between individual points and subsets of points (clusters), let us see how to effectively and efficiently approximate these sums. Consider a point $p \in P$ and a subset $C \subseteq P$. Let $N = |C|$.

- Draw $t$ random points $x_1, x_2, \ldots, x_t$ from $C$, with replacement and uniform probability.
- Compute the approximation

$$\tilde{d}_{sum}(p, C, t) = \frac{n}{t} \sum_{i=1}^{t} d(p, x_i)$$

$\tilde{d}_{sum}(p, C, t)$ is a random variable, whose value depends on the choices of the random points.

**Proposition 1**

For any $t$, $\tilde{d}_{sum}(p, C, t)$ is an unbiased estimator of $d_{sum}(p, C)$.

*Dimostrazione.* We must prove that $E[\tilde{d}_{sum}(p, C, t)] = d_{sum}(p, C)$. We have:

$$E[\tilde{d}_{sum}(p, C, t)] = E[\frac{n}{t}\sum_{i=1}^{t} d(p, x_i)] = \frac{n}{t}\sum_{i=1}^{t} E[d(p, x_i)] =$$

$$\frac{n}{t}\sum_{i=1}^{t}\sum_{y\in C} d(p, y)Pr(x_i = y) = \frac{n}{t}\sum_{i=1}^{t}\sum_{y\in C} d(p, y)\frac{1}{n} = d_{sum}(p, C)$$

□

It is shown that given any $\varepsilon \in (0, 1)$, if we set $t \in \alpha \cdot logn/\varepsilon^2$, for a suitable constant $\alpha$, then with high probability

$$d_{sum}(p, C) - \delta \leq \tilde{d}_{sum}(p, C, t) \leq d_{sum}(p, C) + \delta$$

where $\delta = \varepsilon \cdot n \cdot max_{y\in C} d(p, y)$.

**Supervised evaluation: entropy** The goal of this method is to measure the impurity of our clustering when we're given some kind of ground-truth. Consider a clustering $C$ of a pointset $P$. Suppose that each point $p \in P$ is associated with a class label out of a domain of $L$ class labels. For each cluster $C \in C$ and class $i$, let

$$m_C = \text{ number of points in cluster } C$$

$$m_i = \text{ number of points of class } i$$

$$m_{C,i} = \text{ number of points of class i in cluster } C$$

Then we can define the *entropy of a cluster* $C$ as:

$$-\sum_{i=1}^{L} \frac{m_{C,i}}{m_C} \log_2 \frac{m_{C,i}}{m_C}$$

The entropy measures the impurity of $C$, ranging from 0 (i.e., minimum impurity when all points of $C$ belong to the same class), to $log_2 L$(i.e., maximum impurity when all classes are equally represented in $C$). The *entropy of a class* $i$ is defined as:

$$-\sum_{C\in C} \frac{m_{C,i}}{m_i} \log_2 \frac{m_{C,i}}{m_i}$$

This measures how evenly the points of class $i$ are spread among clusters, ranging from 0 (i.e., all points of class $i$ concentrated in the same cluster), to $log_2 K$(i.e., the points of class $i$ are evenly spread among all clusters), where $K$ is the number of clusters. It is defined also when points belong to multiple classes (e.g.,categories of wikipedia pages).
When the number of clusters/classes is small, their entropies can be computed efficiently in MapReduce using a simple adaptation of the class count problem.

# 6 Association Analysis

Association analysis aims at discover patterns in the data, which reveals interesting relations among individuals or groups. We'll focus on the traditional formulation, *Market-basket analysis*. The technique we'll study can be easily generalized to other domains.

## 6.1 Market Basket analysis

In this domain of problems, the data consist in a large set of *items* (for instance products sold in a supermarket) and a large set of *basket* (for instance a basket may represents what a customer bought). The goal is to analyze this data to extract *frequent itemsets*, which are subsets of items that occur together in a high number of baskets. We also look for *association rules*, which are correlations between subsets of items. A rigorous formulations can be expressed as follow. Let's consider a Dataset $T = \{t_1, t_2, \ldots, t_N\}$ of $N$ *transactions* (analogous term for baskets) over a set $I$ of $d$ items,with $t_i \subseteq I$, for $1 \leq i \leq N$.

**Definition 3**

**Items and its support** Let $X \subseteq I$ and let $T_X \subseteq T$ be the subset of transactions that contains $X$. Then, $X$ is an itemset and its support with respect to $T$, denoted by $Supp_T(X)$, is $\frac{|T_X|}{N}$, that is, the fraction of transactions of $T$ that contain $X$. (note that $Supp_T(\emptyset) = 1$)

You can think of support in this way: if you take a random transaction from $T$ with uniform probability, then the probability that this transaction contains $X$ is exactly $Supp_T(X)$.

**Definition 4**

**Association rule and its support and confidence** An association rule is a rule $r : X \longrightarrow Y$, with $X, Y \subset I$, $X, Y \neq \emptyset$, and $X \cap Y = \emptyset$. Its support and confidence with respect to $T$, denoted by $Supp_T(r)$ and $Conf_R(r)$, respectively, are defined as

$$Supp_T(r) = Supp_T(X \cup Y)$$

$$Conf_T(r) = \frac{Supp_T(X \cup Y)}{Supp_T(X)}$$

$X$ and $Y$ are called, respectively, the rule's *antecedent* and *consequent*.

Given the dataset $T$ of $N$ transaction over $I$, and given a *support threshold* $minsup \in (0, 1]$, and a *confidence threshold* $minconf \in (0, 1]$, the following two objectives can be pursued:

- Compute all frequent items, that is, the set of (non empty) itemsets $X$ such that $Supp_T(X) \geq minsup$. We denote this set by $F_{T,minsup}$.
- Compute all (interesting) association rules $r$ such that $Supp_T(r) \geq minsup$ and $Conf_T(r) \geq minconf$.

**Observations** Support and confidence measure the interestingness of a pattern (either itemset or rule). In particular, the threshold $minsup$ and $minconf$ define which patterns must be regarded as interesting. Ideally, we would like that the support and confidence (for rules) of the returned patterns be unlikely to be seen in a random dataset. (Hypothesis testing setting)
The choice of $minsup$ and $minconf$ directly influences

- The output size: low thresholds may yeld too many patterns (possibly exponential in the input size) which become hard to exploit efficiently;
- False positive/negatives: Low thresholds may yield a lot of uninteresting patterns (false positives), while high thresholds may miss some interesting patterns (false negatives).

## 6.2 Potential output explosion

**Theorem 9.** *For a set $I$ of $d$ items:*

- ***Number of distinct non-empty itemsets**: $2^d - 1$;*
- ***Number of distinct association rules**: $3^d - 2^d + 1$.*

Enumerating all itemsets/rules (to find the interesting ones) is out of the question even for small sizes datasets ($d > 40$). Moreover, the complexity analysis with respect to the input size might not be meaningful (if the output is large for example). Therefore, we consider efficient strategies those that require time/space polynomial in both the input and the output sizes.

*Dimostrazione.* The count of itemsets is trivial. For the count of the association rules:

- We define $n_{rules}(k) = |\{rules\ X \longrightarrow Y : |X| = k\}|$
- Total number of rules $= \sum_{k=1}^{d-1} n_{rules}(k)$.

How many rules $X \longrightarrow Y$ exists with $|X| = k$?

- $X$ can be fixed in $\binom{d}{k}$ ways.
- Once $X$ is fixed, $Y \neq \emptyset$ can be fixed in $2^{d-k} - 1$ ways.

Thus, $n_{rules}(k) = \binom{d}{k}(2^{d-k} - 1)$

$$\sum_{k=1}^{d-1} n_{rules}(k) = \sum_{k=1}^{d-1} \binom{d}{k}(2^{d-k} - 1) = \sum_{k=0}^{d} \binom{d}{k}(2^{d-k} - 1) - (2^d - 1)$$

$$= \left(\sum_{k=0}^{d} \binom{d}{k} 2^{d-k}\right) - \left(\sum_{k=0}^{d} \binom{d}{k}\right) - (2^d - 1) = \left(\sum_{k=0}^{d} \binom{d}{k} 1^k 2^{d-k}\right) - 2^d - (2^d - 1)$$

$$= 3^d - 2^d - (2^d - 1) = 3^d - 2^{d+1} + 1$$

□

## 6.3 Lattice of Itemsets

A family of itemsets under $\subseteq$ forms a **lattice**. A lattice is a mathematical abstract structure which is a partially ordered set. For each two elements $X, Y$ there is a *unique least upper bound* $(X \cup Y)$ and a *unique greatest lower bound* $(X \cap Y)$. The lattice can be graphically represented through the *Hasse diagram*. A property is the **Anti-monotonicity of support**: for every $X, Y \subseteq I$

$$X \subseteq Y \Rightarrow Supp(X) \geq Supp(Y)$$

There are important consequences to the previous property. For a given support threshold,

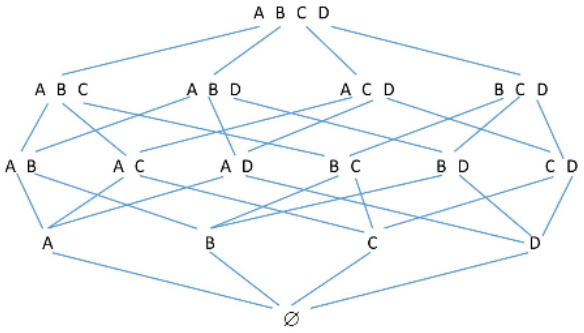


Figura 18: Example of Hasse diagram

- **downward closure**: If $X$ is frequent, every $W \subseteq X$ is frequent;
- If $X$ is not frequent, then every $W \supseteq X$ is not frequent.

Therefore, frequent itemsets form a sublattice closed downwards.

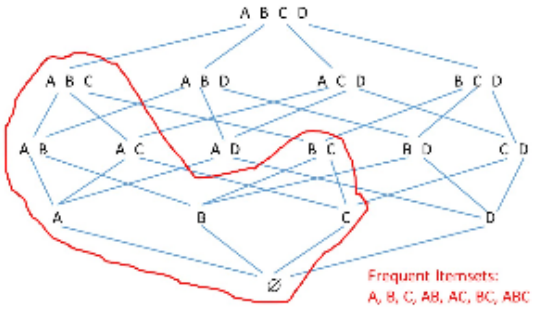


Figura 19: Example of subset in a Hasse diagram

## 6.4 Mining of frequent itemsets and association rules

The input of this problem is the dataset $T$ of $N$ transactions over $I$, and the $minsup$ and $minconf$ (for association rules). The output for Frequent Itemsets is

$$F_{T,minsup} = \{(X, Supp_T(X)) : X \neq \emptyset \wedge Supp_T(X) \geq minsup\}$$

The output for the association rules is

$$\{(r : X \leftarrow Y, Supp_T(r), Conf_T(r)) : Supp_T(r) \geq minsup \wedge Conf_T(r) \geq minconf\}$$

The goal is to obtain algorithm strategies which have time/space complexities polynomial in input/output sizes.
In particular, the most known algorithms to compute the association rules work in two phases:

1 Compute the frequent itemsets and their supports;

2 Extract the rules from the frequent itemsets and their supports.

**Efficient mining of Frequent Itemsets** Here the key objective is a careful exploration of the lattice of itemsets exploiting anti-monotonicity of support. There are two possible main approaches:

- **Breadth First**: compute the frequent items in a line order. This is usually the faster when there are fewer items;
- **Depth First**: Extends the frequent itemsets going "up" in the lattice. This is more convenient when there are more frequent itemsets. Due to the depth first exploration, at some point it may restrict the set of transactions used for support.

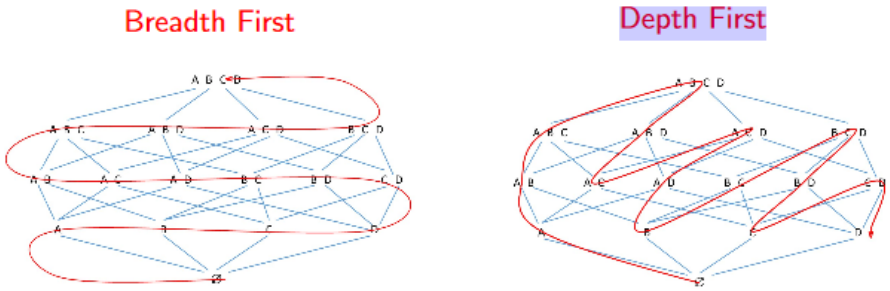


Figura 20: lattice exploration approaches

## 6.5 A-priori algorithm

**A-Priori** is a popular, paradigmatic data mining algorithm introduced in 1994, which uses the breadth first approach.To describe the algorithm is convenitent to introduce the following definition: for every itemset $X \subseteq I$, define its absolute support as the number of transactions that contain X

$$\sigma(X) = Supp(X) \cdot N$$

For convenience, we assume the existence of a total ordering of the items,and assume that transactions/itemsets are represented as sorted vectors. This assumption is not crucial but will simplify the statement.

**High-level strategy** The input of this problem is the dataset $T$ of $N$ transactions over $I$, and the *minsup*. The output is

$$F_{T,minsup} = \{(X, Supp_T(X)) : X \neq \emptyset \wedge Supp_T(X) \geq minsup\}$$

The algorithm will proceed as follows:

- Compute frequent itemsets by increasing length $k = 1, 2, \ldots$
- Frequent itemsets of length $k > 1$ (says $F_k$) are computed as follows: First we compute a set of candidates $C_k \supseteq F_k$ from $F_{k-1}$. The anti-monotonicity property is used to make $C_k$ small. Then, we extract $F_k$ from $C_k$ by computing the support of each $X \in C_k$.

**Pseudo-code** The A-priori algorithm pseudo-code is the following:

```
k ⟵ 1
Compute F_1 = {i ∈ I; Supp({i}) ≥ minsup}
Compute O_1 = {(X, Supp(X)); X ∈ F_1}
repeat
        k ⟵ k + 1
        C_k ⟵APRIORI–GEN(F_{k−1}) /* Candidates */
        for each c ∈ C_k do σ(c) ⟵ 0
        for each t ∈ T do
                for each c ∈ C_k do
                        if c ⊆ t then σ(c) ⟵ σ(c) + 1
        F_k ⟵ {c ∈ C_k : σ(c) ≥ N · minsup}
        O_k ⟵ {(X, σ(X)/N); X ∈ F_k}
until F_k = ∅
return ⋃_{k≥1} O_k
```

## 6.6 APRIORI-GEN(F)

The main goal of APRIORI-GEN is to determine a very small set of candidates which includes all the frequent itemsets. Let's start by the high level strategy. We can divide it in tow phases:

- **Candidate generation**: In this first phase, we generate a first set of candidates by merging pairs from $F$ with all but 1 item in common. (for example, itemsets "ABCD" and "ABCE" in $F$ generate "ABCDE")
- **Candidate pruning**: Here we exclude all candidates containing an infrequent subset.(for example, "ABCDE" survives only if all length-4 subsets are in $F$)

The pruning is done a priori without computation of support. This explains the name of the algorithm. Moreover, both phases exploit anti-monotonicity.

**Pseudo-code** Let's look at the pseudo code for APRIORI-GEN.

```
Let ℓ − 1 be the size of each itemset in F
Φ ⟵ ∅
/* Candidate generation */
for each X, Y ∈ F such that X ≠ Y ∧ X[1...ℓ − 2] = Y[1...ℓ − 2] do
        add X ∪ Y to Φ
/* Candidate Pruning */
for each Z ∈ Φ do
        for each Y ⊂ Z such that |Y| = ℓ − 1 do
                if (Y ∉ F) then {remove Z from Φ; exit inner loop}
return Φ
```

**Correctness of A-Priori**

**Theorem 10.** *The A-Priori algorithm for mining frequent itemsets is correct.*

*Dimostrazione.* It suffices to show that for each $k \geq 1$ the set $F_k$ computed by the algorithm consists of all frequent itemsets of size $k$.
We proceed by induction on $k$:

- **Basis** $k = 1$ is trivial;
- **Induction step**. Fix $k > 1$
  - Inductive hypothesis: we assume the property holds up to index $k - 1$;
  - Inductive step: it is sufficient to prove that the set $C_k$ returned by APRIORI-GEN($F_{k-1}$) contains $F_k$.

To prove that $F_k \subseteq C_k$ consider an arbitrary $X \in F_k$. Let $X = X[1]X[2]\dots X[k]$ and define

$$X(1) = X[1]X[2]\dots X[k-2]X[k-1]$$

$$X(2) = X[1]X[2]\dots X[k-2]X[k]$$

Clearly:

- $X = X(1) \cup X(2)$.
- Both $X(1)$ and $X(2)$ are frequent (by anti-monotonicity), thus $X(1), X(2) \in F_{k-1}$

Therefore:

- $X = X(1) \cup X(2)$ is added to $C_k$ by the candidate generation phase.
- $X$ cannot be eliminated by the candidate pruning phase since all of its subsets are frequent.

□

**Efficiency of A-Priori** A-Priori owes its popularity to a number of features that yield efficient performance especially for sparse (few frequent itemsets) datasets:

- It Only does a few passes over the dataset: namely $n_{max} + 1$ passes, where $k_{max}$ is the length of the longest frequent itemset. Note that if the number of frequent itemsets is small, $k_{max}$ must also be small;
- Support is computed only for a few non-frequent itemsets: (see the enxt lemma) this is due to candidate generation and pruning which exploiting anti-monotonicity of support;
- Several optimizations are known for the computation of supports of candidates: this is typically the most time-consuming task.

**Lemma 6**

Consider the execution of A-Priori on a dataset $T$ of transactions over a set$I$ of $d$ items with support threshold *minsup*, and suppose that $M$ frequent itemsets are returned at the end. Then, the algorithm computes the support for $\leq d + min\{M^2, dM\}$ itemsets.

The following theorem is an easy consequence of the lemma.

**Theorem 11.** *he A-Priori algorithm for mining frequent itemsets can be implemented in time polynomial in both the input size (sum of all transaction lengths)and the output size (sum of all frequent itemsets lengths).*

The proof of the lemma follows:

*Dimostrazione.* The algorithm computes the support of

- All $d$ items (to determine $F_1$);
- All candidates in $C_k$, for $k = 2, 3, \ldots$

Let $m_k$ be the number of frequent itemsets of length $k$. Therefore,

$$M = \sum_{k \geq 1} m_k$$

We now show that $|C_k| \leq min\{(m_{k-1})^2, dm_{k-1}\}$, for $k > 1$. For each candidate $Z \in C_k$ there exists $X, Y \in F_{k-1}$:

- $Z = X \cup Y$.
- $Z = Xa$, with $a$ last item of $Y$ (since $X$ and $Y$ share the first $k-1$ items).

Therefore, the following 2 upper bounds holds:

- $|C_k| \leq \binom{m_{k-1}}{2} \leq (m_{k-1})^2$;
- $|C_k| \leq dm_{k-1}$

n conclusion, an upper bound to the number of itemsets for which the support is computed is

$$d + \sum_{k>1} |C_k| \leq d + \sum_{k>1} min\{(m_{k-1})^2, dm_{k-1}\} \leq d + min\{\sum_{k>1} (m_{k-1})^2, \sum_{k>1} dm_{k-1}\}$$

$$\leq d + min\{(\sum_{k>1} m_{k-1})^2, \sum_{k>1} dm_{k-1}\} \leq d + min\{M^2, dM\}$$

□

**Optimizations of A-Priori** The most time-consuming step in A-Priori is the counting of supports of candidates (sets $C_2, C_3, \ldots$). In practice, has however performance issues. A-priori requires a pass over the entire dataset checking each candidate against each transaction. The name of candidates can be still very large (even considering the upper-bound), stressing storage and computing capacity. In particular the issue may become critical for the set $C_2$, which contains all pairs of frequent items. As $k$ grows larger, the cardinality of $F_{k-q}$, hence of $C_k$, drops.
In order to tackle this issues, some solutions have been developed. Primarily, we can use of a trie-like data structure (called HASH TREE) to store $C_k$ and to speed-up support counting. A trie is a tree structure which is very convenient to use for store strings. Moreover, in order to avoid storing all pairs of items in $C_2$, we can use the following strategy:

- While computing the supports of individual items, builds a small hash table to upper bound the supports of pairs;
- In $C_2$ includes only pairs with upper bound of support $\geq$ *minsup*.

**Other approaches to frequent itemsets mining** Other approaches to frequent itemsets mining Several algorithms based on depth-first mining strategies have been devised and tested. Those algorithm avoid several passes over the entire dataset of transactions. This is done by confining the support counting of longer itemsets to suitable small projections of the dataset, typically much smaller than the original one. Some prominent example are *FP-Growth* and *Patricimine*.

## 6.7 Mining association rule

Let $T$ be a set of $N$ transactions over the set $I$ of $d$ items. Let $minsup, minconf \in (0,1)$ be the support and confidence thresholds. WE must compute

$$\{r : X \longrightarrow Y \text{ with } Supp_T(r) \geq minusp \wedge Conf_T(r) \geq minconf\}$$

In particular, recall that:

$$Supp_T(r) = Supp_T(X \cup Y)$$

$$Conf_T(r) = Supp_T(X \cup Y)/Supp_T(X)$$

If $F$ is the set of frequent itemsets with respect to the $minsup$, then for every rule $r : X \longleftarrow Y$ that we target, we have that

$$X, X \cup Y \in F$$

The mining strategy consist in the following:

- Compute the frequent itemsets with respect to $minsup$ and their supports;
- For each frequent itemset $Z$ compute

$$\{r : Z - Y \longrightarrow Y \text{ such that } Y \subset Z \wedge Conf(r) \geq minconf\}$$

Note that each rule derived from $Z$ automatically satisfies the support constraint since $Z$ is frequent. Conversely, rules derived from itemsets which are not frequent need not to be checked, since they would not satisfy the support constraint.
Regarding Step 2, we can observe that it is usually not expensive since there are few frequent itemsets. Nevertheless, it can be implemented efficiently exploiting a special anti-monotonicity property for rules.
Consider a frequent itemset $Z$ and two non-empty subsets $Y^{'} \subset Y \subset Z$. The two rules:

$$r_1 = Z - Y^{'} \longrightarrow Y^{'}$$

$$r_2 = Z - Y \longrightarrow Y$$

are such that $Conf_T(r_2) \leq Conf_T(r_1)$ (Ant-monotonicity property for rules). This is true because

$$Conf_T(r_2) = \frac{Supp_T(Z)}{Supp_T(Z - Y)} \leq \frac{Supp_T(Z)}{Supp_T(Z - Y')} = Conf_T(r_1)$$

since $Z - Y' \supset Z - Y$ and by anti-monotonicity of support. As a consequence, it holds that

$$Conf_T(r_1) < minconf \Rightarrow Conf_T(r_2) < minconf$$

**Efficient mining of rules** The high-level strategy for mining rules is the following: For each frequent itemset $Z$,

- Check confidence of rules $Z - Y \longrightarrow Y$ by increasing length of $Y$;
- Once all consequents $Y$ of length $k$ have been checked, generate "candidate" consequents of length $k+1$ to be checked, using APRIORI-GEN.

This strategy requires time polynomial in the input/output.

## 6.8 Frequent itemset mining for big data

When the dataset $T$ is very large one can follow two approaches:

- **Partition-based approach:** We partition $T$ into $K$ subset, where $K$ is a design parameter. Then, we mine frequent itemsets independently in each subset and extract the final output from the union of the frequent itemsets computed before.
- **Sampling approach:** We compute the frequent itemsets from a small sample of $T$.

We remarks that the partitioning approach computes the exact output but may be inefficient, while the sampling approach computes an approximation but is efficient.

**Partition-Based approach** The MapReduce algorithm is based on SON's algorithm. The settings are the following:

- $T$ is a set of $N$ transaction over the set $I$ of $d$ items. Assume

  $$T = \{(i, t_i) : 0 \leq i < N\}$$

  where $t_i$ is the $i$-th transaction;
- $minusp \in (0, 1)$ is the support threshold;
- $K$ is a suitable design parameter with $1 < K < N$.

Then, the algorithm consist in 4 rounds, and the high-level description is the following:

- **Round 1:**
  - Partition $T$ into $K$ subsets $T_0, T_1, \ldots T_{k-1}$;
  - In each $T_i$ compute the set $\Phi_i$ of frequent itemsets with respect to $minsup$. Note that the same itemset can be extracted from multiple subsets, and that an itemset frequent in $T_i$ is not necessarily frequent in $T$.
- **Round 2:** Gather $\Phi = \bigcup_{i=0}^{K-1} \Phi_i$ and eliminate the duplicates.
- **Round 3:** For every $0 \leq i < K$independently do the following:
  - Gather $T_i$ and $\Phi$;
  - For each $X \in \Phi$ count the number of transactions of $T_i$ that contains $X$ (call this number $\sigma(X, i)$).
- **Round 4:** For each $X \in \Phi$, gather all $\sigma(X, i)$'s, compute the final support $Supp_T(X) = \frac{1}{N} \sum_{j=0}^{K-1} \sigma(X, j)$ and output $X$ if $Supp_T(X) \geq minsup$.

**Partition based approach analysis** The correctness of the algorithms is immediate from the following observations:

- For each itemset $X \in \Phi$ the exact support is computed in Round 3 + round 4;
- A "true" frequent itemset $X$ must belong to some $\Phi_i$, and hence in $\Phi$: Since $|T_X| \geq N \cdot minsup$, there must be a subset $T_i$ where $X$ appears in $\frac{|T_X|}{K} \geq \frac{N}{K} \cdot minusp$ transactions, and thus $X$ is in $\Phi_i$.

Remark that $\Phi$ may contain infrequent itemset (false positives).
The number of rounds is 4, and the space requirements depend on the size of $\Phi$, which cannot be easily predicted. The larger $K$, the more false positive are likely to occurs.

**Sampling approach analysis** This is based on the idea that we don't really need to process the entire dataset, but it's enough to consider an approximate set of frequent itemset (but haing under control the quality of the approximation).

**Definition 5**

**Approximate frequent itemset** Let $T$ be a dataset of transactions over the set of items $I$ and $minsup \in (0, 1]$ a support threshold. Let also $\varepsilon > 0$ be a suitable parameter. A set $C$ of pairs $(X, s_X)$, with $X \subseteq I$ and $s_X \in (0, 1]$, is an $\varepsilon$-approximation of the set of frequent itemsets and their supports if the following conditions are satisfied:

1 For each $X \in F_{T,minsup}$ there exists a pair $(X, s_X) \in C$;

2 For each $(X, s_X) \in C$,

- $Supp(X) \geq minsup - \varepsilon$;
- $|Supp(C) - s_X| \leq \varepsilon$;

where $F_{T,minsup}$ is the true set of frequent itemsets with respect to $T$ and and $minsup$.

The first condition ensures that the approximate set $C$ comprises all true frequent itemsets and we don't have false negatives. The second conditions ensures that $C$ does not contain itemsets of very low support, and that for each itemset $X$ such that $(X, s_X) \in C$, then $s_X$ is a good estimate of its support.

**Simple algorithm** Let $T$ be a dataset of $N$ transactions over $I$, and $minsup \in (0, 1]$ a support threshold. Let also $f(minsup) < minsup$ be a suitably lower support threshold. Let $S \subseteq T$ be a sample drawn at random with uniform probability and with replacement. We return the set of pairs:

$$C = \{(X, s_X = Supp_S(X)) : X \in F_{S,f(minsup)}\}$$

where $F_{S,f(minsup)}$ is the set of frequent itemsets with respect to $S$ and $f(minsup)$.

**Theorem 12.** ***Riondato-Upfal*** *Let $h$ be the maximum transaction lenght and let $\varepsilon, \delta$ be suitable design parameters in $(0, 1)$. There is a constant $c > 0$ such that if*

$$f(minsup) = minsup - \frac{\varepsilon}{2} \quad AND \quad |S| = \frac{4c}{\varepsilon^2}\left(h + log\frac{1}{\delta}\right)$$

*then with probability at least* $1-\delta$ *the set* $C$ *returned by the algorithm is an* $\varepsilon$*-approximation of the set of frequent itemset and their supports.*

The result steted in the theorem relies on the following lemma:

**Lemma 7**

Using the sample size $|S|$ specified in the above theorem, the algorithms ensures that with probability $\geq 1-\delta$ for each itemset $X$ it holds that $|Supp_T(X) - Supp_S(X) \leq \frac{\varepsilon}{2}$.

Let's see the proof of the theorem now.

*Dimostrazione.* By using the result stated in the lemma we have that

- For each frequent itemset $X \in F_{T,minsup}$,

$$Supp_S(X) \geq Supp_T(X) - \frac{\varepsilon}{2} \geq minsup - \frac{\varepsilon}{2} = f(minsup)$$

  hence, the pair $(X, s_X = Supp_S(X))$ is returned by the algorithm.

- For each pair $(X, s_X = Supp_S(X))$ returned by the algorithm

$$Supp_T(X) \geq Supp_S(C) - \frac{\varepsilon}{2} = s_X - \frac{\varepsilon}{2} = s_X - \frac{\varepsilon}{2} \geq f(minsup) - \frac{\varepsilon}{2} \geq minsup - \varepsilon$$

  and

$$|Supp_T(X) - s_X| = |Supp_T(X) - Supp_S(X)| \leq \frac{\varepsilon}{2} < \varepsilon$$

Hence, the output of the algorithm is an $\varepsilon$-approximation of the true frequent itemsets and their supports. □

**Observations** The size of the sample is independent of the support threshold *minsup* and of the number $N$ of transactions. It only depends on the approximation guarantee embodied in the parameters $\varepsilon$, $\delta$, and on the max transaction length $h$, which is often quite low.
Tighter bounds on the sample size are known, but we use this since it's sample.
The sample-based algorithm yields a 2-round MapReduce algorithm:in first round the sample of suitable size is extracted; in the second round the approximate set of frequent itemsets is extracted from the sample (e.g., through A-Priori) within one reducer.

## 6.9 Limitations of the Support/Confidence framework

1 **Redundancy:** many returned patterns may characterize the same subpopulation of data;

2 **Difficult control of output size:** it is hard to predict how many patterns will be returned for given support/confidence thresholds.

3 **Significance:** are the returned patterns significant, interesting?

We'll address the first and the second issues and we'll docus on frequent itemsets, even if some ideas can be extended to association rules.

**Closed itemset** Closed itemset aims at avoid redundancy and (attempt to) control the output size. Let's consider the dataset $T$ of $N$ transactions over $I$, and support threshold *minsup*.

**Definition 6**

An itemset $X \subseteq I$ is closed with respect to $T$ if for each superset $Y \supset X$ we have $Supp_T(Y) < Supp_T(X)$.

Therefore, an itemset $X$ is closed if its support decreases as soon as an item is added. We'll use the following notation:

- *Set of closed itemset:* $CLO_T = \{X \subseteq I : X \text{ is closed with respect to} T\}$;
- *Set of frequent closed itemset:* $CLO - F_{T,minsup} = \{X \in CLO_T : Supp_T(X) \geq minsup\}$.

**Maximal itemsets** Maximal itemsets purse the same goals as closed itemset but using a more drastic approach.

**Definition 7**

An itemset $X \subseteq I$ is maximal with respect to $T$ and $minsup$ if $Supp_T(X) \geq minsup$ and for each superset $Y \supset X$ we have $Supp_T(Y) < minsup$.

Therefore, $X$ is maximal if it is frequent and becomes infrequent as soon as an item is added. As a notation, we call

$$MAX_{T,minsup} = \{X \subseteq I : X \text{ is maximal with respect to } T \text{ and } minsup\}$$

**Properties of closed/maximal itemsets** **Property 1:** $MAX \subseteq CLO - F \subseteq F$.
This property follows immediately from the definitions of frequent closed and maximal itemsets.

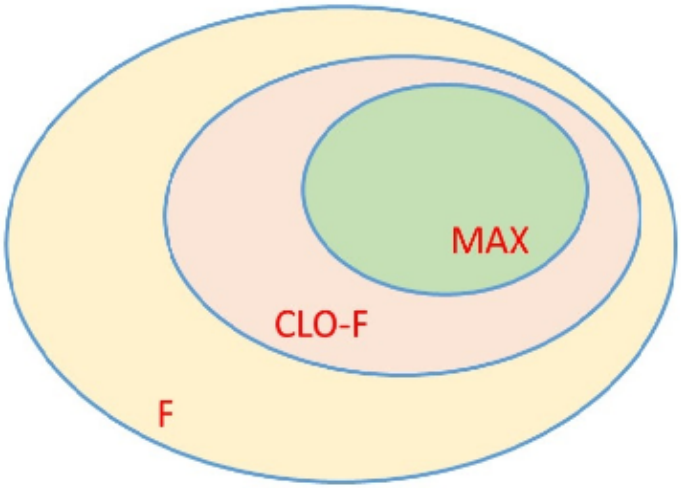


Figura 21: First property

item **Property 2:** for each itemset $X$ there is a closed itemset $X'$ such that

- $X \subseteq X'$;
- $Supp(X') = Supp(X)$.

**Property 3:** for each frequent itemset $X$ there is a maximal itemset $X'$ such that

$$X \subseteq X'$$

Let's see the demonstration of the second property (the proof for the third propery is almost identical).

*Dimostrazione.* Consider an itemset $X$.

- If $X$ is closed, then $X' = X$;
- If $X$ is not closed we construct $X'$ as follows:
  1 Find $X' \supset X$ with $Supp(X') = Supp(X)$;
  2 If $X'$ is closed we are done;
  3 If not we iterate the procedure.

□

**Consequences of the properties** For a given dataset and a given support threshold, $MAX$ and $CLO - F$ provide succinct and lossless representations of $F$: We can determine $F$ by taking all subsets of $MAX$ or $CLO - F$.
Moreover, $CLO - F$, together with their supports, provides a succinct and lossless representation of $F$ together with their supports. We can do this by deriving $F$ from $CLO - F$ as above. Then, for each $X \in F$ compute

$$Supp(X) = max\{Supp(Y) : Y \in CLO - F \text{ and } X \subseteq Y\}$$

In particular, we need to remember that from $MAX$ and their supports we can derive $F$ but not their supports.

**Representativity of closed itemset** For $X \subseteq I$, let $T_X$ denote the set of transactions where $X$ occurs. Then we define

**Definition 8**

**Closure**

$$Closure(X) = \bigcap_{t \in T_X} t$$

**Theorem 13.** *Let $X \subseteq I$. If $Supp(X) > 0$ we have*

*1 $X \subseteq Closure(X)$;*

*2 $Supp(Closure(X)) = Supp(X)$;*

*3 $Closure(X)$ is closed.*

*Dimostrazione.* 1 Immediate, since $X \subseteq t$, for each $t \in T_X$.

2 We have

- $Supp(X) \geq Supp(Closure(X))$ is immediate from the first property;
- $Supp(X) \leq Supp(Closure(X))$ derives from the fact that, by definition, $Closure(X)$ is contained in each transaction $t$ that contains $X$.

3 By definition, $Closure(X) = \bigcap_{t \in T_X} t$, which implies immediately that

- $Closure(X)$ is contained in every transaction of $T_X$;
- For each item $a$ not in $Closure(X)$, at least one transaction of $T_X$ will not contain $Closure(X) \cup \{a\}$.

$$\Rightarrow \text{ every } Y \supset Closure(X) \text{ has smaller support than } Closure(X)$$

$$\Rightarrow Closure(X) \text{ is closed}$$

□

**Observations** Each closed itemset $Y$ represents compactly all (possibly many) itemsets $X$ such that $Closure(X) = Y$.
There exist efficient algorithms for mining maximal or frequent closed itemsets.
Notions of closure similar to the ones used for itemsets are employed in other mining contexts.

**Output explosion** On the matter of output explosion regarding maximal/frequent closed itemset, in practice they yeld a quite effective reduction of the pattern size. however, there are non-trivial pathological cases where their number is exponential in theinput size (an example can be found on the slide).
To solve this, we can impose explicitly a limit on the output size.

**Top-K frequent itemsets** Let $T$ be a dataset of $N$ transactions over a set $I$ of $d$ items

**Definition 9**

**Top-K frequent itemsets** Let $X_1, X_2, \ldots$ an enumeration of the non-empty itemsets in non-increasing order of support (ties broken arbitrarily). For a given integer $K \geq 1$. Let

$$s(K) = Supp(X_K)$$

The Top-K frequent itemsets with respect to $T$ are all itemsets of support $\geq s(K)$.

According to this definition, we take all the itemset with support equal to $K$. We therefore get more than $K$ itemsets. The number of Top-K frequent itemsets is $\geq K$.
s(K) can be defined equivalently as the maximum support threshold that provides at least $K$ frequent itemsets.

**Top-K frequent closed itemsets** We can give the same definition for closed itemsets.

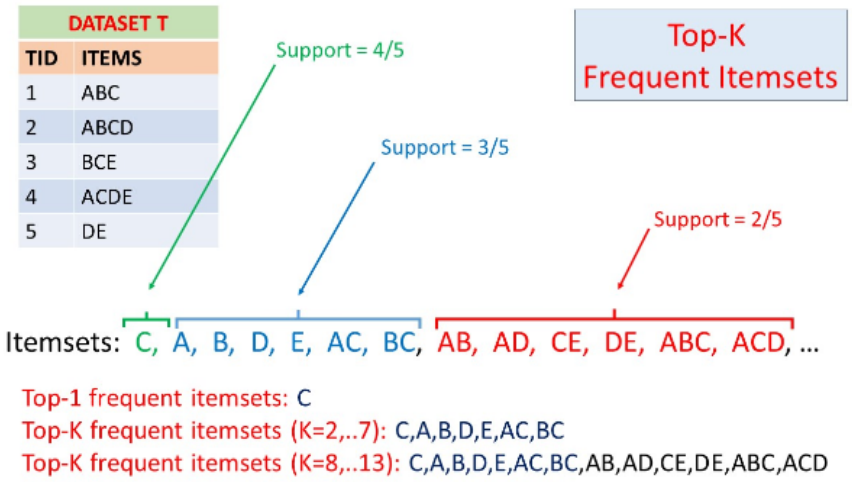


Figura 22: Top-K frequent itemset example

**Definition 10**

**Top-K frequent closed itemsets** Let $X_1, X_2, \ldots$ an enumeration of the non-empty itemsets in non-increasing order of support (ties broken arbitrarily). For a given integer $K \geq 1$. Let

$$sc(K) = Supp(X_K)$$

The Top-K frequent closed itemsets with respect to $T$ are all itemsets of support $\geq sc(K)$.

**Remarks** There are efficient algorithms for mining the Top-K frequent(closed) itemsets. A popular strategy is the following: we generate a (closed) itemsets in non-increasing order of support (using a priority queue); then, we stop at the first itemset with smaller support than the $K$-th one. The notion of Top-K patterns (w.r.t. some ranking) is used a lot in many different contexts.
The Top-K formulation solve the problem of the output explosion in practice. However, for Top-K frequent closed itemsets there is a tight polynomial bound on the output size, while for Top-K frequent itemsets there are pathological cases with exponential output size.

**Control of the output size** The next theorem shows that for Top-K frequent closed itemsets,$K$ provides tight control on the output size.

**Theorem 14.** *For $K > 0$, the Top-K frequent closed itemsets are $O(d \cdot K)$, where $d$ is the number of items.*

Hence, for small $K$(at most polynomial in the input size) the number of Top-K frequent closed itemsets will be polynomial in the input size.

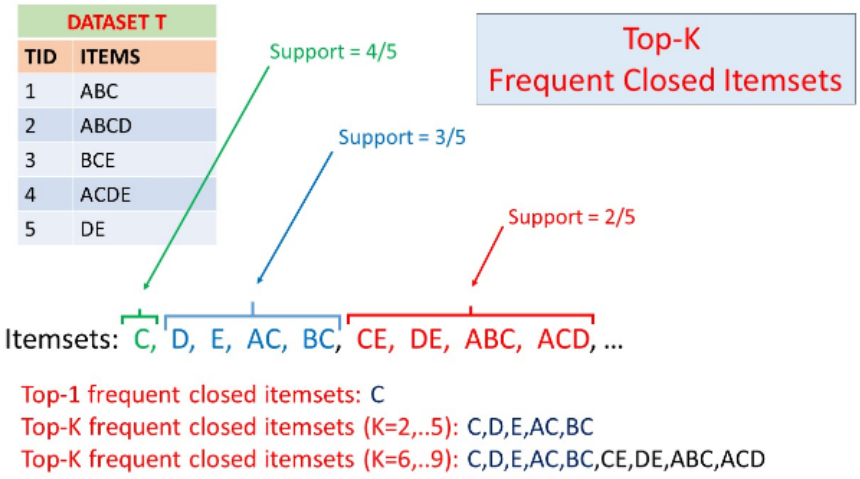


Figura 23: Top-K frequent closed itemset example

**Recap** Let's recap the upside and downside of our procedures:

1 **Maximal itemsets:**

- Lossless representation of the frequent itemsets (lossy, if supports are required);
- In practice, much fewer than the frequent itemsets, but, in pathological cases, still exponential in the input size.

Thus, we obtain a reduction of redundancy.

2 **Frequent closed itemsets:**

- Lossless representation of the frequent itemsets with supports;
- In practice, much fewer than the frequent itemsets, but, in pathological cases, still exponential in the input size.

Thus, we obtain a reduction of redundancy.

3 **Top-K frequent (closed) itemsets:**

- The output size is on the order of $O(d \cdot K)$, if restricted to closed itemsets, otherwise small in practice but exponential in $d$ for pathological cases.

Thus, we obtain a reduction of redundancy, and control on the output size (with closed itemsets).