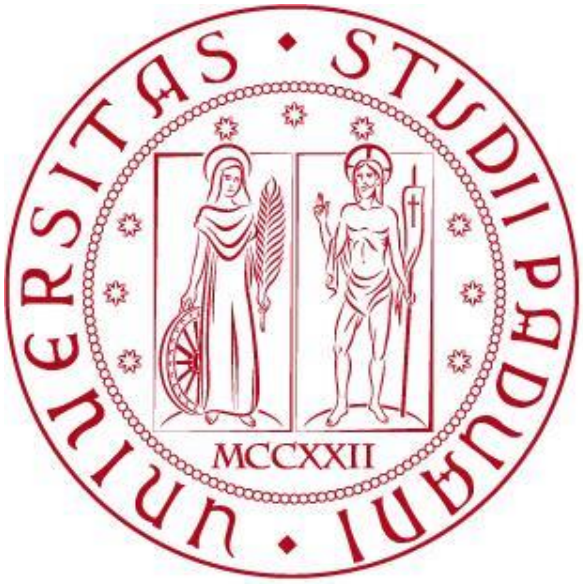


UNIVERSITÀ
DEGLI STUDI
DI PADOVA

NOTES OF

BIG DATA COMPUTING

Map Reduce, Clustering, Association Analysis

(Version 26/06/2020)

Edited by:
Stefano Ivancich

CONTENTS

1. **Map Reduce** 1
   - 1.1. **Intro** 1
   - 1.2. **MapReduce computation** 2
   - 1.3. **Basic Techniques and Primitives** 6
     - 1.3.1. **Partitioning Technique** 6
     - 1.3.2. **Efficiency-accuracy tradeoffs** 7
     - 1.3.3. **Exploiting samples** 7
2. **Clustering** 9
   - 2.1. **Distance** 9
   - 2.2. **Center-based clustering** 11
   - 2.3. **k-center clustering** 12
   - 2.4. **k-means clustering** 14
   - 2.5. **k-median clustering** 17
   - 2.6. **Cluster Evaluation** 19
     - 2.6.1. **Clustering tendency** 19
     - 2.6.2. **Unsupervised evaluation** 19
     - 2.6.3. **Supervised evaluation** 20
3. **Association Analysis** 21
   - 3.1. **Basics** 21
   - 3.2. **Mining of Frequent Itemsets and Association Rules** 23
     - 3.2.1. **Mining Frequent Itemsets** 23
     - 3.2.2. **Mining association rules** 26
   - 3.3. **Frequent itemset mining for Big Data** 27
   - 3.4. **Limitations** 29
     - 3.4.1. **Closed and Maximal itemsets** 29
     - 3.4.2. **Top-K frequent itemsets** 31

This document was written by students with no intention of replacing university materials. It is a useful tool for the study of the subject but does not guarantee an equally exhaustive and complete preparation as the material recommended by the University.
The purpose of this document is to summarize the fundamental concepts of the notes taken during the lesson, rewritten, corrected and completed by referring to the slides to be used in the design of Big Data Systems as a "practical and quick" manual to consult. There are no examples and detailed explanations, for these please refer to the cited texts and slides.

If you find errors, please report them here:
www.stefanoivancich.com/
ivancich.stefano.1@gmail.com
The document will be updated as soon as possible

1.Map Reduce

1.1. Intro

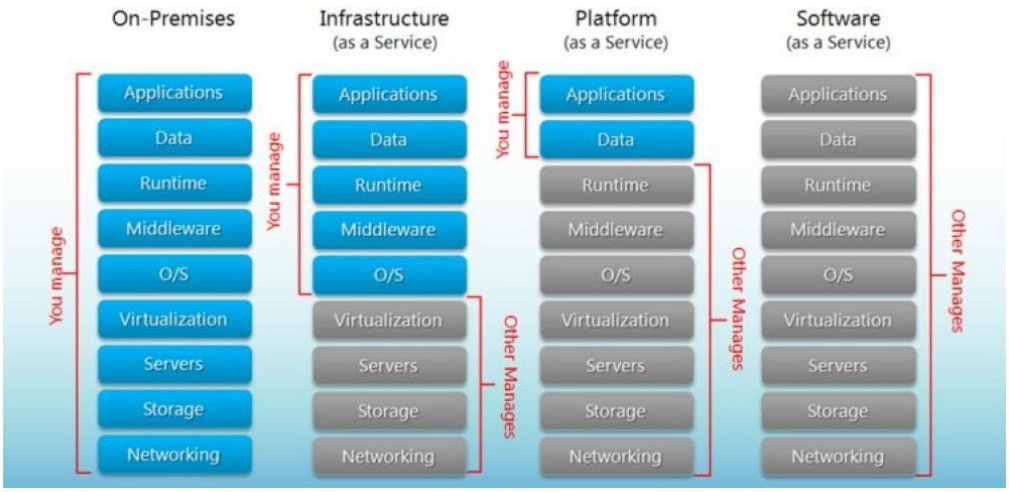
Programming framework for big data processing on distributed platforms.

Main features:

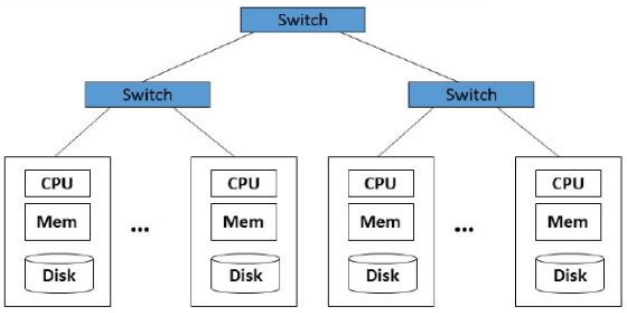
- Data centric view
- Inspired by functional programming (map, reduce functions)
- Ease of algorithm/program development. Messy details (e.g., task allocation; data distribution; fault-tolerance; load-balancing) are hidden to the programmer

Runs on:

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



Typical cluster architecture:



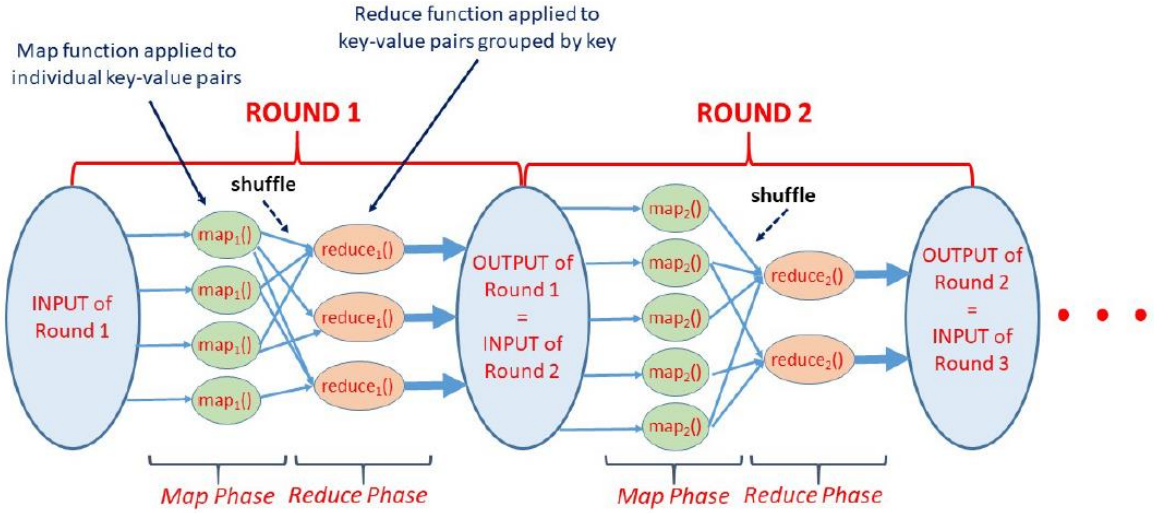
- **Racks of 16-64 compute nodes** (commodity hardware), connected (within each rack and among racks) by fast switches (e.g., 10 Gbps Ethernet)
- **Distributed File System:**
    - Files can be enormous, possibly a terabyte.
    - Files are rarely updated
    - Files are divided into chunks (e.g., 64MB per chunk)
    - Each chunk is replicated (2x or 3x) with replicas in different nodes and, possibly, in different racks.
    - To find the chunks of a file, there is a **master node file** which is also replicated. A directory (also replicated) records where all master nodes are.
    - Examples: Google File System (GFS); **Hadoop Distributed File System (HDFS)**

Software Implementations: Apache Hadoop (old, works BAD), **Apache Spark** (uses the HDFS).

1.2. MapReduce computation

Map-Reduce computation:

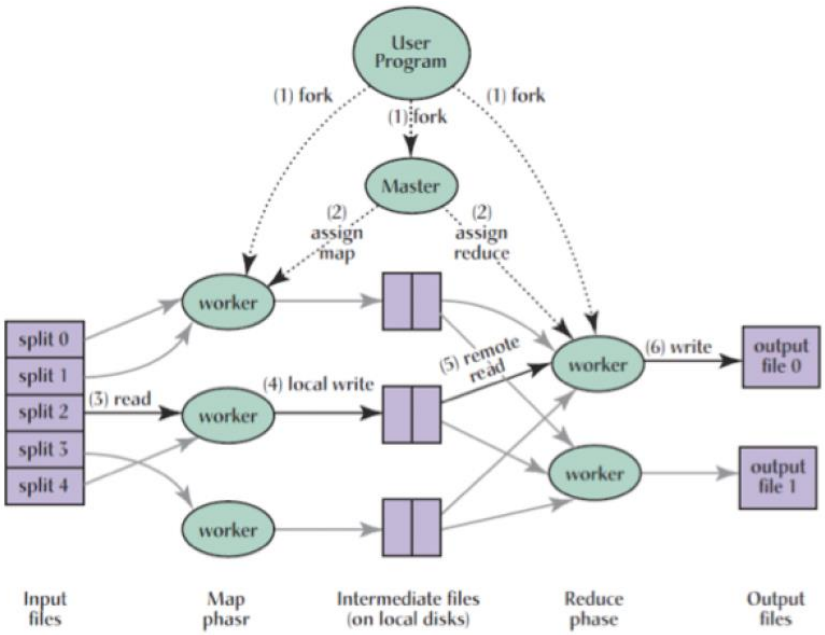
- Is a sequence of rounds.
- Each round transforms a set of key-value pairs into another set of key-value pairs (data centric view), through the following two phases:
  - **Map phase:** a user-specified **map function** is applied separately to each input key-value pair and produces $\geq \mathbf{0}$ 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 ≥ 0 other key-value pairs, which is the output of the round.
- The output of a round is the input of the next round.



Implementation of a round:

- **Input** files are elements of any type (tuple, documents, …). A chunk is a collection of elements, and no element is stored across two chunks. Keys of input elements are not relevant and we tend to ignore them.
- Input file is split into X chunks and each chunk is 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 produces in its local disk.
  A Map task can produce several key-value pairs with the same key, even from the same element.
- The **intermediate key-values pairs**, while residing in the local disks, are **grouped by key**, and the values associated with each key are formed into a list of values.
  This is done by partitioning into Y buckets through a hash function h.
  (k, v) is putted in the Bucket $i = h(k) \bmod Y$
  Hashing helps balancing the load of the workers.
- Each bucket is the input of a different **reduce task** which is assigned to a worker.
  The input of the reduce function is a pair of the form $(k, [v_1, \ldots, v_k])$
  The output of the Reduce function is a sequence of ≥ 0 key-value pairs.
  Reducer: application of the reduce function to a single key and its associated list of values.
  Reduce task executes one or more reducers.

- The outputs from all the Reduce tasks are merged into a single file written on DataFileSystem.
- The user program is forked into a **master process** and several **worker processes**. The master is in charge of assigning map and reduce tasks to the various workers, and to monitor their status (idle, in-progress, completed).
- Input and output files reside on a Distributed File System, while intermediate data are stored on the workers' local disks.
- The round involves a data shuffle 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).
  Shuffle is often the most expensive operation of the round
- The values X and Y are design parameters.



Dealing with faults: Master pings workers periodically to detect failures.

- **Worker failure:**
  - Map tasks completed or in-progress at failed worker are reset to idle and will be rescheduled.
  - Reduce tasks in-progress at failed worker are reset to idle and will be rescheduled.
- **Master failure:** the whole MapReduce task is aborted

Specifications of a MapReduce algorithm:

- Input and Output
- The sequence of rounds
- For each round
  - input, intermediate and output sets of key-value pairs
  - functions applied in the map and reduce phases.
- Meaningful values (or asymptotic) bounds for the key performance indicators (defined later).

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

For example, an MR algorithm with a fixed number of rounds R, we can use the following style:

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

Round 1:

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

Round 2: as before ..

...

Round R: as before ...

Analysis of a MapReduce algorithm: estimate these Key Performance Indicators:

- **Number of rounds $\boldsymbol{R}$**.
- **Local space $\boldsymbol{M_L}$**: maximum amount of main memory required, in any round, by a single invocation of the map and reduce function used in that round, for storing the input and any data structure needed by the invocation.
  Bounds the amount of **main memory** required at each worker.
- **Aggregate space $\boldsymbol{M_A}$**: maximum amount of (disk) space which is occupied by the stored data at the beginning/end of the map and reduce phase of any round.
  Bounds the overall amount of (disk) space that the **executing platform must provide**.

These are usually estimated through asymptotic analysis (either worst-case or probabilistic) as functions of the instance size.

Example: 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.

Observation: 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:

- $R = 1$ (trivial)
- $M_L = O(\max\{N_{\max}, k\}) = O(N_{\max} + k)$. The 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.

Observation: in principle, we may have $N_{\max} + k = \Theta(N)$, which implies linear local space. When this happens, most of the advantages of MapReduce vanish (see more in the next two slides). In practice, however it is likely that $k, N_{\max} = o(N)$, so the above algorithm is efficient. We'll see later how to improve the algorithm ensuring sublinear M_L even for large k.

Design goals for MapReduce algorithms

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

But this is impractical for very large inputs because a platform with very large main memory is needed and No parallelism is exploited.

So, we break 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.

Goals:

- $\boldsymbol{R = O(1)}$
- **Sublinear local space**: $M_L = O(|input|^{\epsilon})$ with $\epsilon \in (0,1)$
- **Linear aggregate space**: $M_A = O(|input|)$, or slightly superlinear
- **Polynomial complexity** of each map or reduce **functions**

Often, small M_L enables high parallelism but may incur large R.

Pros of Map Reduce:

- **Data-centric view:** Algorithm design can **focus on data transformations**, targeting the design goals. Programming frameworks (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

Cons of Map Reduce:

- **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** may be 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.

1.3. Basic Techniques and Primitives

Chernoff bound: Given a Binomial random variable $Bin(n,p)$, $\mu = E[X] = np$
For every $\delta_1 \geq 5, \delta_2 \in (0,1)$:

- $\Pr(X \geq (1+\delta_1)\mu) \leq 2^{-(1+\delta_1)\mu}$
- $\Pr(X \leq (1-\delta_2)\mu) \leq 2^{-\mu\delta_2^2/2}$

Union Bound: $\Pr(\bigcup_i A_i) \leq \sum_i \Pr(A_i)$

1.3.1. 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.

- **Map** (i, elem) --> ($i \bmod \sqrt{N}$, elem)
- **Map** (i, elem) --> (j, elem) where j =random $\in [0, \sqrt{N})$

Eg.1: Improved Word Count (Random key)

Consider the word count problem, k documents and N word occurrences overall, in a realistic scenario where $k = \Theta(N)$ and N is huge (e.g., *huge number of small documents*). 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 counts 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 occurring 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, 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

- $R = 2$
- $M_L = O\left(\max_{i=1,k} N_i + m + \sqrt{N}\right)$.
- $M_A = O(N)$

How large can m be? We need a *very useful* technical tool.

Let $N' \leq N$ be the number of intermediate pairs produced by the Map phase of Round 1, and consider an arbitrary key $x \in [0, \sqrt{N})$.

Crucial observation: m_x is a Binomial$(N', 1/\sqrt{N})$ random variable with expecation $\mu = N'/\sqrt{N} \leq \sqrt{N}$.

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 union bound we have that

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

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

Eg.2: Class Count ($i \bmod \sqrt{N}$ key)

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

Observation: the straightforward 1-round algorithm may require $O(N)$ local space, in case a large fraction of objects belong to the same class. In the next slide we will see a more efficient algorithm.

Round 1:

- *Map phase*: map each pair $(i, (o_i, \gamma_i))$ into the intermediate pair $(i \bmod \sqrt{N}, (o_i, \gamma_i))$ (mod = reminder of 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 γ labeling some object in $S^{(j)}$, produce the pair $(\gamma, c_j(\gamma))$, where $c_j(\gamma)$ is the number of objects of $S^{(j)}$ labeled with γ.

Round 2:

- *Map phase*: empty
- *Reduce phase*: for each class γ, gather the at most $\sqrt{N}$ pairs $(\gamma, c_j(\gamma))$ resulting at the end of the previous round, and return the output pair $(\gamma, \sum_j c_j(\gamma))$.

1.3.2. Efficiency-accuracy tradeoffs

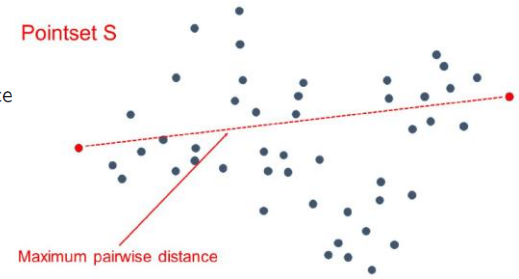
For some problems an exact MapReduce algorithm may be too costly (require large R, M_L, or M_A)
If is acceptable for the application, we **relax the condition of exact solution** (we search for an approximate solution).

E.g.: Maximum pairwise distance

Suppose that we are given a set S of N points from some metric space (e.g., $\Re^3$) and we want to determine the maximum distance between two points, for a given distance function $d(\cdot,\cdot)$.

More precisely:

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: $(0, \max_{0\leq i,j<N} d(x_i, x_j))$.



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

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

Lemma

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

Proof.

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

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

Round 1:

- *Map phase*: map each pair (i, x_i) into the intermediate pair $(i \bmod \sqrt{N}, (i, x_i))$. Also, create the $\sqrt{N} - 1$ pairs $(j, (0, x_0))$, for $1 \leq j < \sqrt{N}$.
- *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: $R = 2$, $M_L = O\left(\sqrt{N}\right)$, $M_A = O(N)$.

1.3.3. Exploiting samples

If we can profitably process a small sample of the data, with that sample we can

- subdivide the dataset in smaller subsets and analyze them separately.
- or, give an 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.

E.g.: Sorting

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 $\{(i, s_{\pi(i)}) : 0 \leq i < N\}$, where π is a permutation such that $s_{\pi(1)} \leq s_{\pi(2)} \leq \cdots \leq s_{\pi(N)}$.

The MR SampleSort algorithm is based on the following idea:

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

Round 1:

- *Map phase*: for each pair (i, s_i) do the following:
  - transform the pair into $(i \bmod K, (0, s_i))$ (call it *regular pair*);
  - with probability $p = K/N$, independently of 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 from regular pairs).

  Obs. 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 $x_1 \leq x_2 \leq \cdots x_t$ be the splitters in sorted order. Transform each regular pair $(j, (0, s))$ into the pair (ℓ, s), where $\ell \in [0, t]$ is such that $x_\ell < s \leq x_{\ell+1}$ (assume $x_0 = -\infty$ 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_ℓ (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 $S^{(\ell)}$; and compute the final output pairs for the objects in $S^{(\ell)}$ whose ranks start from $1 + \sum_{h=0}^{\ell-1} N_h$.

- Number of rounds: $R = 3$
- Local Space M_L:
  - Round 1: $O(K + t + N/K)$, since K copies of each splitter are created, and each reducer must store all splitter pairs and a subset of N/K intermediate pairs.
  - Round 2: $O(\max\{N_\ell \,;\, 0 \leq \ell \leq t\})$ since each reducer must gather one $S^{(\ell)}$.
  - Round 3: $O(t + \max\{N_\ell \,;\, 0 \leq \ell \leq t\})$, since each reducer must store all N_ℓ's and one $S^{(\ell)}$.

  $\Rightarrow$ overall $M_L = O(t + N/K + \max\{N_\ell \,;\, 0 \leq \ell \leq t\})$
- Aggregate Space M_A: $O(N + t \cdot K + t^2))$, since in Round 1 each splitter is replicated K times, and in Round 3 each N_ℓ is replicated $t+1$ times. The objects are never replicated.

Lemma

With reference to the MR SampleSort algorithm, for any $K \in (2 \ln N, N)$ the following two inequalities hold with high probability (i.e., probability at least $1 - 1/N$):

1. $t \leq 6K$, *and*
2. $\max\{N_\ell \,;\, 0 \leq \ell \leq t\} \leq 4(N/K) \ln N$.

Theorem

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

2. 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 captures a notion of similarity: close points are similar, distant point are dissimilar.

2.1. Distance

Metric Space: ordered pair (M, d) where:

- M is a set
- $d(\cdot)$ is a metric on M. $d: M \times M \to \mathbb{R}$ s.t. $\forall x, y, z \in M$
  - $d(x, y) \geq 0$
  - $d(x, y) = 0 \iff x = y$
  - $d(x, y) = d(y, x)$ symmetry
  - $d(x, z) \leq d(x, y) + d(y, z)$ triangle inequality

Different distance functions can be used to analyze dataset through clustering. The choice of the function may have a significant impact on the effectiveness of the analysis.

Distance functions:

- **Minkowski distances**: $d_{Lr}(X, Y) = (\sum_{i=1}^{n} |x_i - y_i|^r)^{1/r}$ where the vectors $X, Y \in \mathbb{R}^n, r > 0$
  - $r = 2$: Euclidean distance. (Linea d'aria) $\|\cdot\|$
    Aggregates the gap in each dimension between two objects.
  - $r = 1$: Manhattan distance. Used in grid-like environments.
  - $r = \infty$: (Chebyshev distance)
- **Cosine (or angular) distance**: $d_{cosine}(X, Y) = \arccos\left(\frac{X * Y}{\|X\| * \|Y\|}\right) = \arccos\left(\frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}\right)$
  is the angle between the vectors $X, Y \in \mathbb{R}^n$
  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.
  Measures the ratios among features' values, rather than their absolute values.
- **Hamming distance**: $d_{Hamming}(X, Y) = |\{i: x_i \neq y_i\}| = d_{L1}(X, Y) = \sum_{i=1}^{n} |x_i - y_i|$
  Used when points are binary vectors.
  Measures the total number of differences between two objects.
- **Jaccard distance**: $d_{Jaccard}(S, T) = 1 - \frac{|S \cap T|}{|S \cup T|} \in [0,1]$
  Used when points are sets (e.g., documents seen as bags of words).
  Measures the ratio between the number of differences between two sets and the total number of points in the sets.
- **Edit distance:** minimum number of deletions and insertions that must be applied to transform X into Y.
  Used in Strings.
  Longest Common Subsequence $LCS(X, Y)$ longest string Z whose characters occur in both X and Y in the same order but not necessarily contiguously.

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

Hamming and Jaccard distances are used when an object is characterized by having or not having some features.
Euclidean and Cosine distances are used when an object is characterized by the numerical values of its features.

Minimum distance of a point $p \in P$ from a set $S \subseteq P$: $d(p,S) = \min\{q \in S : d(p,q)\}$

Curse of dimensionality: issues that arise when processing data in high-dimensional spaces.

- **Quality**: Distance functions may lose effectiveness in assessing similarity. Points in a high-dimensional metric space tend to be Sparse, Almost equally distant from one another, Almost orthogonal (as vectors) to one another.
- **Performance**: The running times may have a linear or even exponential dependency on n.
    - Running time of a clustering $O(C_{dist} * N_{dist})$ =cost distance computation*#of distance computations
    - Nearest Neighbor Search problem: given a set $S \subseteq \mathbb{R}^n$ of m points, construct a data structure that, for a given query point $q \in \mathbb{R}^n$, efficiently returns the closest point $x \in S$ to q. The problem requires time: $\Theta(\min\{n * m, 2^n\})$

Objective functions can be categorized based on whether or not:

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

Combinatorial optimization problem: $\Pi(I, S, \phi, m)$ NP-HARD

- instances I
- solutions S
- objective function ϕ that assigns a real value to each solution $s \in S$
- $m \in \{min, max\}$
- $\forall i \in I \; \exists S_i \subseteq S$ of feasible solutions
- given $i \in I$ find a feasible solution $s \in S_i$ which minimize $\phi(s)$ or maximize.

c-approximation algorithm A: $\forall i \in I$ returns a feasible solution $A(i) \in S_i$ s.t. $\phi\big(A(i)\big) \leq c \min_{s \in S_i} \phi(s)$

2.2. Center-based clustering

Family of clustering problems used in data analysis.

***k*-clustering** of a set of points P is a tuple $C = (C_1, \dots, C_k; c_1, \dots, c_k)$ where:

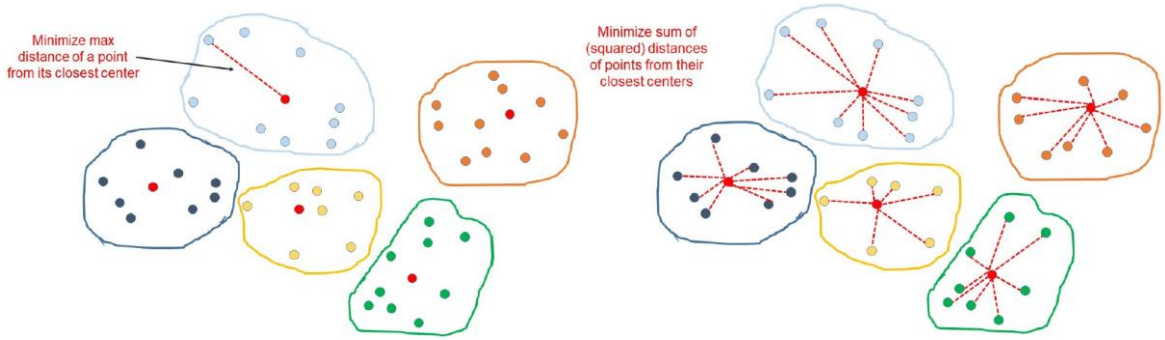
- $|P| = N$
- $(C_1, \dots, C_k)$ is a partition of $P = C_1 \cup \dots \cup C_k$
- $c_1, \dots, c_k$ are centers of the clusters, $c_i \in C_i$

Minimize one of those Objective functions: NP-HARD

- **k-center clustering:** $\phi_{kcenter}(C) = \max_{i=1,\dots,k} \max_{a \in C_i} d(a, c_i)$ minimize the maximum distance of any point from the closest center.
  Useful when we need to guarantee that every point is close to a center.
  Sensitive to noise.
- **k-means:** $\phi_{kmeans}(C) = \sum_{i=1}^{k} \sum_{a \in C_i} d(a, c_i)^2$ minimize the sum of the squared distances of the points from their closest centers.
  More sensitive to noise but there exist faster algorithms to solve it.
- **k-median:** $\phi_{kmedian}(C) = \sum_{i=1}^{k} \sum_{a \in C_i} d(a, c_i)$ minimize the sum of distances of the points from their closest centers.

Each point is at distance at most ϕ from its closest center.

k-center and k-median belong to the family of **facility-location problems**: 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.



Partitioning primitive: assign each point $x \in P$ to the Cluster C_i which its center c_i is closest to x

Partition(P, S)

Let $S = \{c_1, c_2, \dots, c_k\} \subseteq P$
for $i \leftarrow 1$ to k do $C_i \leftarrow \{c_i\}$
for each $p \in P - S$ do
  $\ell \leftarrow \text{argmin}_{i=1,k}\{d(p, c_i)\}$ // *ties broken arbitrarily*
  $C_\ell \leftarrow C_\ell \cup \{p\}$
$\mathcal{C} \leftarrow (C_1, C_2, \dots, C_k; c_1, c_2, \dots, c_k)$
return $\mathcal{C}$

2.3. k-center clustering

Provides a strong guarantee on how close each point is to the center of its cluster.
For noisy pointsets (e.g., pointsets with outliers) the clustering which optimizes the k-center objective may obfuscate some “natural" clustering inherent in the data.

Farthest-First Traversal Algorithm

- **Input**: set P of N points from a metric space (M, d), integer $k > 1$
- **Output**: k-clustering $C = (C_1, \dots, C_k; c_1, \dots, c_k)$ of P

$S \leftarrow \{c_1\}$ // $c_1 \in P$ *arbitrary point*
for $i \leftarrow 2$ **to** k **do**
  Find the point $c_i \in P - S$ that maximizes $d(c_i, S)$
  $S \leftarrow S \cup \{c_i\}$
return Partition(P, S)

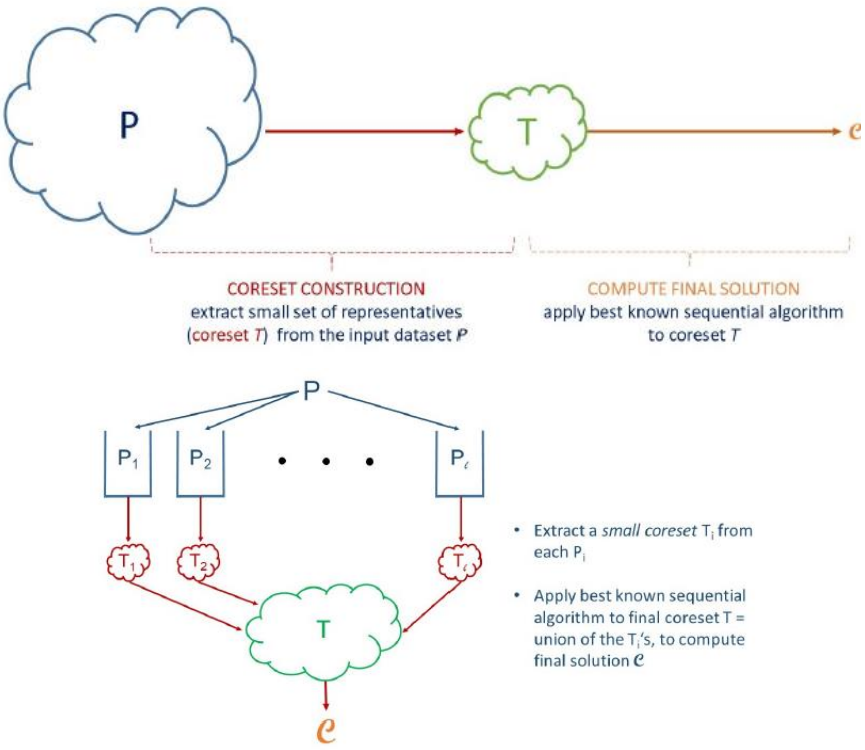
Is a 2-approximation algorithm: $\phi_{kcenter}(C_{alg}) \leq \mathbf{2} * \boldsymbol{\phi}^{\boldsymbol{opt}}_{\boldsymbol{kcenter}}(\boldsymbol{P}, \boldsymbol{k})$
It means that each point is at distance at most 2ϕ from its closest center.

Dealing with Big Data: pointset P is too large for a single machine.
Farthest-First Traversal requires $k - 1$ scans of pointset P which is impractical for massive pointsets and not so small k.

Coreset technique: (is a variant of sampling)

- Extract a "small" subset of the input (dubbed coreset), making sure that it contains a good solution to the problem.
- Run best known (sequential) algorithm, possibly expensive, to solve the problem on the small coreset, rather than on the entire input, still obtaining a fairly good solution.

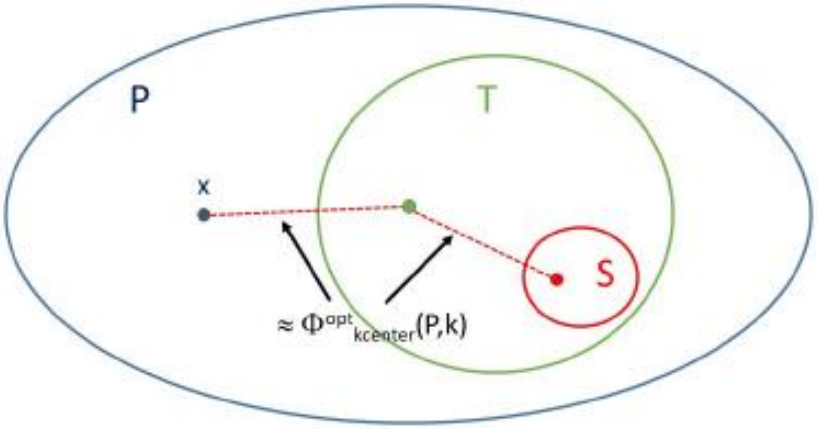
The technique is effective when the coreset can be extracted efficiently and, in particular, when it can be composed by combining smaller coresets independently extracted from some (arbitrary) partition of the input.



Coreset for K-Center:

- Select coreset $T \subset P$ making sure that for each $x \in P - T$ is close to some point of T (approx. within $\phi^{opt}_{kcenter}(P,k)$)
- Search the set S of k final centers in T, so that each point of T is at distance approx. within $\phi^{opt}_{kcenter}(P,k)$ from the closest center.

By combining the above two points, we gather that each point of P is at distance $O\left(\phi^{opt}_{kcenter}(P,k)\right)$ from the closest center.



MapReduce-Farthest-First Traversal

- Round 1: Partition P arbitrarily in ℓ 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) (see previous exercise on how to implement Partition(P, S) in 1 round).

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

Analysis of MR-Farthest-First Traversal

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

- Local space $M_L = O\left(\sqrt{N \cdot k}\right) = o(N)$. Indeed,
  R1: $O(N/\ell) = O\left(\sqrt{N \cdot k}\right)$
  R2: $O(\ell \cdot k) = O\left(\sqrt{N \cdot k}\right)$
  R3: $O\left(\sqrt{N}\right)$ (from exercise).
- Aggregate space $M_A = O(N)$

Is a **4-approximation** algorithm: $\phi_{kcenter}(C_{alg}) \leq 4 * \phi^{opt}_{kcenter}(P,k)$
It means that each point is at distance at most 4ϕ from its closest center.

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.
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.

2.4. k-means clustering

Aim at optimizing average distances.

Finds a k-clustering $C = (C_1, \dots, C_k)$ which minimizes $\phi_{kmeans}(C) = \sum_{i=1}^{k} \sum_{a \in C_i} d(a, c_i)^2$

- Assumes that **centers need not** necessarily **belong to the input** pointset.
- Aims at minimizing cluster variance and works well for discovering ball-shaped clusters.
- Because of the quadratic dependence on distances, k-means clustering is rather sensitive to outliers (though less than k-center).

Centroid of a set P: $c(P) = \frac{1}{N} \sum_{X \in P} X$ (component wise sum)

Is the point that minimizes the sum of the square distances to all points of P. $\sum_{X \in P} d(X, c_P)^2 \leq \sum_{X \in P} d(X, Y)^2$

Lloyd's algorithm

Input Set P of N points from $\Re^D$, integer $k > 1$

Output k-clustering $\mathcal{C} = (C_1, C_2, \dots, C_k; c_1, c_2, \dots, c_k)$ of P with small $\Phi_{\text{kmeans}}(\mathcal{C})$. Centers need not 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
```

- Always terminates but the **number** of **iterations** can be **exponential** in the input size.
- May be trapped into a **local optimum**.
- The **quality** of the solution **and** the **speed** of convergence of Lloyd's algorithm **depends** considerably from the choice of the **initial set of centers**

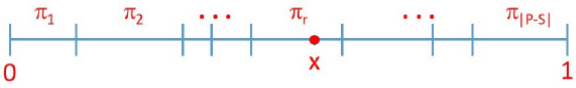
k-means++: center initialization strategy that yields clustering not too far from the optimal ones.

Computes a (initial) set S of k centers for P

```
c_1 ← random point chosen from P with uniform probability
S ← {c_1}
for i ← 2 to k do
  c_i ← random point chosen from P − S, where a point p
        is chosen with probability (d(p,S))^2 / Σ_{q∈P−S} (d(q,S))^2
  S ← S ∪ {c_i}
return S
```

Probability of a point j is selected: $\pi_j = \frac{\left(d(p_j,S)\right)^2}{\sum_{q \in P-S} (d(q,S))^2}$ where $\sum_{j \geq 1} \pi_j = 1$

Pick $x \in [0,1]$ and set $c_i = p_r$ where $r \in [1, |P - S|]$ such that $\sum_{j=1}^{r-1} \pi_j < x \leq \sum_{j=1}^{r} \pi_j$



Let C_{alg} = Partition(P; S) be the k-clustering of P induced by the set S of centers returned by the k-means++. $\boldsymbol{\phi_{kmeans}(C_{alg})}$ **is a random variable**. $\boldsymbol{E[\phi_{kmeans}(C_{alg})] \leq 8(\ln k + 2)\phi^{opt}_{kmeans}(k)}$

Dealing with Big Data: pointset P is too large for a single machine.

- Distributed (e.g., MapReduce) implementation of Lloyd's algorithm and k-means++.
- Coreset-based approach

Parallel k-means: variant in which a set of $> k$ candidate centers is selected in $O(\log N)$ parallel rounds, and then k-centers are extracted from this set using a weighted version of k-means++.

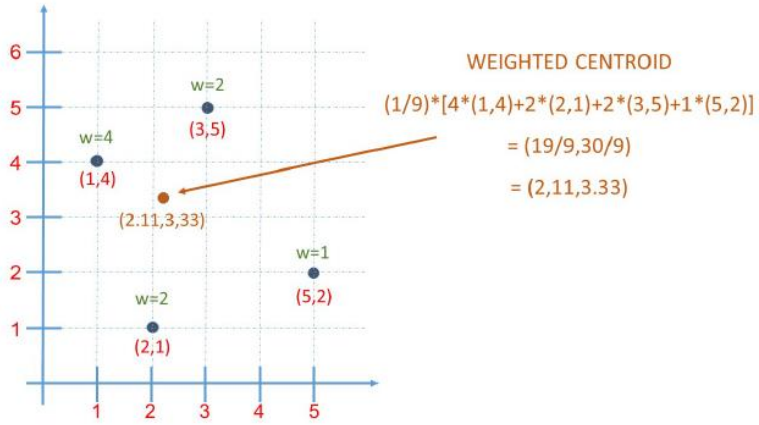
Weighted k-means: for each point $p \in P$ is given an integer weight $w(p) > 0$. It minimizes:

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

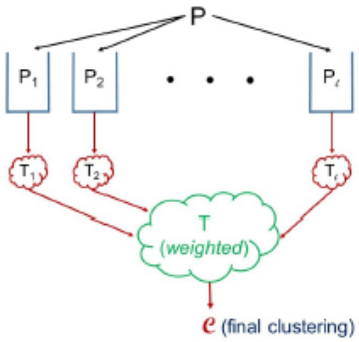
Most of the algorithms known for the standard k-means clustering problem (unit weights) can be adapted to solve the case with general weights.
It is sufficient to regard each point p as representing $w(p)$ identical copies of itself.
Lloyd's algorithm remains virtually identical except that:

- C_i becomes $\frac{1}{\sum_{p \in C_i} w(p)} \sum_{p \in C_i} w(p) * p$
- The objective function $\phi^{w}_{kmeans}(C)$ is used

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



Coreset-based MapReduce algorithm for k-means:



- The local coresets T_i's are computed using a sequential algorithm for k-means.
- 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

Uses, as a parameter, a sequential algorithm A for k-means: MR-kmeans(A).

MR-kmeans($\mathcal{A}$)

Input Set P of N points from a metric space (M, d), integer $k > 1$

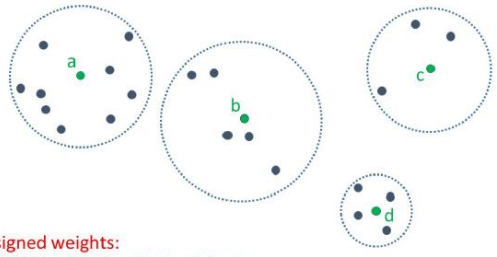
Output k-clustering $\mathcal{C} = (C_1, C_2, \ldots, C_k; c_1, c_2, \ldots, c_k)$ of P which is a good approximation to the k-means problem for P.

Algorithm:

- Round 1: Partition P arbitrarily in ℓ subsets of equal size $P_1, P_2, \ldots, P_\ell$ and execute $\mathcal{A}$ 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 $\mathcal{A}$ 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 $T = \cup_{i=1}^{\ell} T_i$ of $\ell \cdot k$ points, together with their weights. Using a single reducer, run $\mathcal{A}$ 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.

Example for Round 1

Set T_i (green points) computed in a partition P_i



Assigned weights:
w(a)=10, w(b)=6, w(c)=4, w(d)=5

Analysis of MR-kmeans($\mathcal{A}$)

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

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

Is a $O(\alpha^2)$-**approximation** algorithm: $\phi_{kmeans}(C_{alg}) = O\left(\alpha^2 * \phi^{opt}_{kmeans}(P, k)\right)$

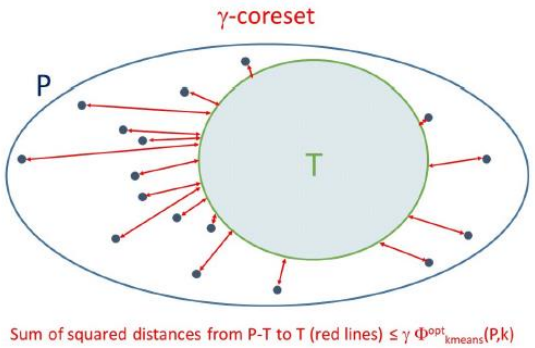
Where A is an α-approximation algorithm for Weighted k-means clustering with $\alpha > 1$.
It means that each point is at distance at most α^2 from its closest center.

T is a γ-**coreset** for P, k and ϕ_{kmeans} **if** $\sum_{p \in P}\left(d(p, \pi(p))\right)^2 \leq \gamma * \phi^{opt}_{kmeans}(P, k)$ where:

- $T \subseteq P$
- $\pi: P \to T$ proxy function

The smaller γ, the better T represents P wrt the k-means problem, and it's likely that a good solution can be extracted from T, by weighing each point of T with the # of points for which it acts as proxy.

The coreset computed in Round 1 is a γ-coreset, with $\gamma = \alpha$.



Proof idea

Observations:

- The constants hidden in the order of magnitude for the approximation factor are not large.
- Two different k-means sequential algorithms can be used in R1 and R2. For example, one could use k-means++ in R1, and k-means++ plus LLoyd's in R2.
- In practice, the algorithm is fast and quite accurate.
- If $k' > k$ centers are selected from each P_i in Round 1 (e.g., through k-means++), the quality of T, hence the quality of the final clustering, improves. In fact, for low dimensional spaces, a γ-coreset T with $\gamma \ll \alpha$ can be built in this fashion using a not too large k'.

2.5. k-median clustering

Finds a k-clustering $C = (C_1, \dots, C_k)$ which minimizes $\phi_{kmedian}(C) = \sum_{i=1}^{k} \sum_{a \in C_i} d(a, c_i)$

- require that cluster centers belong to the input pointset.
- Uses an arbitrary metric spaces (like Manhattan distance)

Partitioning Around Medoids (PAM) algorithm:

Input Set P of N points from (M, d), integer $k > 1$

Output k-clustering $\mathcal{C} = (C_1, C_2, \dots, C_k; c_1, c_2, \dots, c_k)$ of P with small $\Phi_{\text{kmedian}}(\mathcal{C})$. Centers must belong to P.

```
S ← {c1, c2, ..., ck} (arbitrary set of k points of P)
C ← Partition(P, S)
stopping-condition ← false
while (!stopping-condition) do {
  stopping-condition ← true
  for each (c ∈ S and p ∈ P − S) do {
    S' ← (S − {c}) ∪ {p}
    C' ← Partition(P, S')
    if Φkmedian(C') < Φkmedian(C) then {
      stopping-condition ← false
      C ← C'; S ← S'
      exit the for-each loop
    }
  }
}
return C
```

- Is a **5-approximation** algorithm: $\phi_{kmendian}(C_{alg}) \leq 5 * \phi^{opt}_{kmedian}(P, k)$
- is **very slow** in practice especially for large inputs since:
  - the local search may require a large number of iterations to converge $\binom{N}{k}$. This can be solved by stopping the iteration when the objective function does not decrease significantly.
  - in each iteration up to $k(N-k)$ swaps may need to be checked, and for each swap a new clustering must be computed.
- Faster alternative: adaptation of the LLoyd's algorithm where the center of a cluster C is defined to be the point of C that minimizes the sum of the distances to all other points of C.

Dealing with Big Data: uses coreset like k-median

- $\phi^{w}_{kmedian}(C) = \sum_{i=1}^{k} \sum_{p \in C_i} w(p) * d(p, c_i)$
- Let B be a sequential algorithm which solves the weighted variant of k-median using space linear in the input size.
- MR-kmedian(B): 3-round MapReduce algorithm for k-median, which uses B as a subroutine.

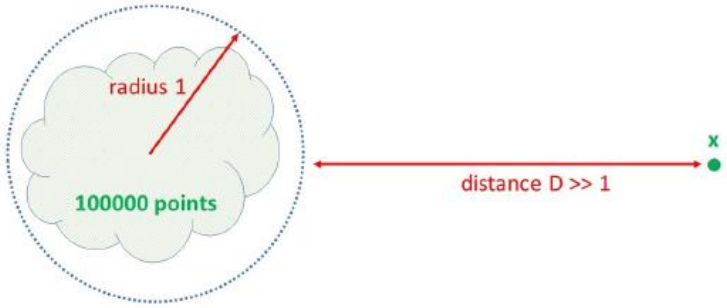
Analysis of MR-kmedian($\mathcal{B}$) algorithm

Assume $k \leq \sqrt{N}$. Then, MR-kmedian($\mathcal{B}$) requires 3 rounds, local space $M_L = O\left(\sqrt{N \cdot k}\right)$, and aggregate space $M_A = O(N)$, where N is the input size.

Is a $O(\alpha^2)$-**approximation** algorithm: $\phi_{kmedian}(C_{alg}) = O\left(\alpha^2 * \phi^{opt}_{kmedian}(P, k)\right)$

k-center vs k-means vs k-median

- All 3 problems tend to return spherically shaped clusters.
- They differ in the center selection strategy since, given the centers, the best clustering around them is the same for the 3 problems.
- **k-center** is useful when we want to optimize the distance of every point from its centers, hence minimizes the radii of the cluster spheres. It is also useful for coreset selection.
- **k-median** is useful for facility-location applications when targeting minimum average distance from the centers. However, it is the most challenging problem from a computational standpoint.
- **k-means** is useful when centers need not belong to the pointset and minimum average distance is targeted. Optimized implementations are available in many software packages (unlike k-median)



For $k \geq 2$, the relevance of the outlier x in the 3 problems is different

- **k-center:** x must be always selected as center!
- **k-means:** x must be selected as center if $D \geq 1000$!
- **k-median:** x must be selected as center if $D \geq 1000000$!

2.6. Cluster Evaluation

2.6.1. Clustering tendency

Assess whether the data contains meaningful clusters, namely clusters that are unlikely to occur in random data.

Hopkins statistic: measures to what extent P can be regarded as a random subset from M

$H(P) = \frac{\sum_{i=1}^{t} u_i}{\sum_{i=1}^{t} u_i + \sum_{i=1}^{t} w_i} \in [0,1]$ for some fixed $t \ll N$ (typically $t < 0.1 * N$), where

- P set of N points in some metric space (M, d)
- $X = \{x_1, \dots, x_t\}$ random points from P
- $Y = \{y_1, \dots, y_t\}$ random set of points from M (metric space)
- $w_i = d(x_i, P - \{x_i\})$ for $1 \leq i \leq t$
- $u_i = d(y_i, P - \{y_i\})$ for $1 \leq i \leq t$

$H(P) \cong 1$: P is likely to have a clustering structure.
$H(P) \cong 0.5$: P is likely to be a random set.
$H(P) \ll 0.5$: points of P are likely to be well (i.e., regularly) spaced.

$H(P)$ is suited for big data.

2.6.2. Unsupervised evaluation

Assess the quality of a clustering (or the relative quality of two clusterings) without reference to external information.

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.
Approaches for measuring intra-cluster similarity and inter-cluster dissimilarity:

- **Cohesion/Separation** metrics
- **Silhouette** coefficient

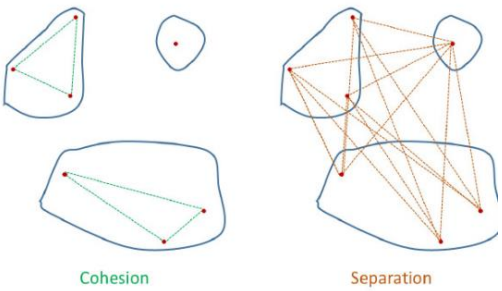
Sum of the distances between a given point and a subset of points $d_{sum}(p, C) = \sum_{q \in C} d(p, q)$
Average distance between p and C: $d_{sum}(p, C)/|C|$
Computing one $d_{sum}(p, C)$ can be done efficiently, even for large C, but computing many of them at once may become too costly.

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

- average distance between points in the same cluster.
  $C = (C_1, \dots, C_k)$ k-clustering of a pointset P, $N_i = |C_i|$

Separation: $\text{separation}(C) = \frac{\sum_{1 \leq i < j \leq k} \sum_{p \in C_i} d_{sum}(p, C_j)}{\sum_{1 \leq i < j \leq k} N_i * N_j}$

- average distance between points in different clusters.



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

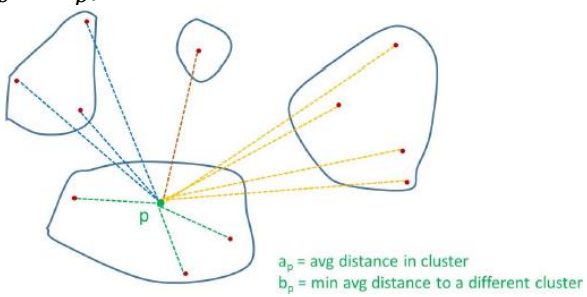
Silhouette coefficient of a point $p \in C_i$: $s_p = \frac{b_p - a_p}{\max\{a_p, b_p\}} \in [-1,1]$

- $a_p = d_{sum}(p, C_i)/N_i$ average distance between p and the other points in its cluster
- $b_p = \min_{j \neq i} d_{sum}(p, C_j)/N_j$ minimum, over all clusters $C_j \neq C_i$ of the average distance between p and the other points in C_j

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.

Average silhouette coefficient: $s_C = \frac{1}{|P|}\sum_{p \in P} s_p$

- measure the quality of the clustering C
- Good: $s_C \cong 1$ (i.e.,$a_p \ll b_p$)



Computing exactly cohesion, separation or the average silhouette coefficient require to compute all pairwise distances between points and its **impractical for very large P.**
Approximation of these sums: $\tilde{d}_{sum}(p, C, t) = \frac{n}{t}\sum_{i=1}^{t} d(p, x_i)$

- $n = |C|$
- t random points $x_1, \ldots, x_t$ from C, with replacement and uniform probability.
- is a random variable
- is an unbiased estimator of $d_{sum}(p, C)$

For any $\epsilon \in (0,1)$, if we set $t \in \alpha * \log n/\epsilon^2$, then with high probability:

$$d_{sum}(p, C) - \delta \leq \tilde{d}_{sum}(p, C, t) \leq d_{sum}(p, C) + \delta$$
$$\text{where } \delta = \epsilon * n * \max_{y \in C} d(p, y)$$

2.6.3. Supervised evaluation

Assess the quality of a clustering (or the relative quality of two clusterings) with reference to external information (e.g., class labels).

Entropy of a cluster C: $-\sum_{i=1}^{L} \frac{m_{C,i}}{m_C} \log_2 \frac{m_{C,i}}{m_C} \in [0, \log_2 L]$

- L class labels
- m_C =#points in cluster C
- m_i =#points of class i
- $m_{C,i}$ =#points of class i in cluster C

Measures the impurity of C, ranging from 0 (i.e., min impurity when all points of C belong to the same class), to $\log_2 L$ (i.e., max impurity when all classes are equally represented in C).

Entropy of a class i: $-\sum_{C \in C} \frac{m_{C,i}}{m_C} \log_2 \frac{m_{C,i}}{m_i} \in [0, \log_2 K]$

measures how evenly the points of class i are spread among clusters: 0=all in 1 cluster.

3.Association Analysis

3.1. Basics

Set I of d items
Transaction $t \subseteq I$
Dataset $T = \{t_1, \dots, t_N\}$ of N transactions over I, with $t_i \subseteq I$ for $1 \leq i \leq N$

Itemset $X \subseteq I$

- $T_X \subseteq T$ is the subset of transactions that contains X
- **Support** of X w.r.t. T: $\text{Supp}_T(X) = \frac{|T_X|}{N}$ is the fraction of transactions of T that contain X.
  - $\text{Supp}_T(\emptyset) = 1$

Association rule: $r: X \rightarrow Y$ with $X, Y \subseteq I$, $X, Y \neq \emptyset$ and $X \cap Y = \emptyset$

- **Support** of r w.r.t. T: $\text{Supp}_T(r) = \text{Supp}_T(X \cup Y)$
- **Confidence** of r w.r.t. T: $\text{Conf}_T(r) = \text{Supp}_T(X \cup Y)/\text{Supp}_T(X)$

- **Dataset T**

TID	Items
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

- **Itemset:** $X = \{\text{Milk,Diaper,Beer}\}$

$$\text{Supp}_T(X) = 2/5$$

- **Association rule:** $r = \{\text{Milk,Diaper}\} \rightarrow \{\text{Beer}\}$

$$X = \{\text{Milk,Diaper}\}$$
$$Y = \{\text{Beer}\}$$
$$\text{Conf}_T(r) = (2/5)/(3/5) = 2/3$$

Problem formulation: Given

- the dataset T of N transactions over I
- Support threshold $\text{minsup} \in (0,1]$
- Confidence threshold $\text{minconf} \in (0,1]$

Compute:

- The set $F_{T,\text{minsup}}$ composed by the **frequent itemsets** w.r.t. T and minsup.
  That are all (non empty) itemsets X such that $\text{Supp}_T(X) \geq \text{minsup}$
- All (interesting) **association rules** r such that $\text{Supp}_T(r) \geq \text{minsup}$ and $\text{Conf}_T(r) \geq \text{minconf}$

Example ($\text{minsup} = \text{minconf} = 0.5$)

Dataset T	
TID	Items
1	ABC
2	AC
3	AD
4	BEF

Frequent Itemsets	
Itemset	Support
A	3/4
B	1/2
C	1/2
AC	1/2

Association Rules		
Rule	Support	Confidence
A → C	1/2	2/3
C → A	1/2	1

Support and confidence measure the interestingness of a pattern (itemset or rule).

- thresholds minsup and $\mathrm{minconf}$ 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.
The choice of minsup and $\mathrm{minconf}$ directly influences:

- Output size: low thresholds may yield too many patterns, which become hard to exploit effectively.
- False positive/negatives: low thresholds may yield a lot of uninteresting patterns (false positives), while high thresholds may miss some interesting patterns (false negatives).

Potential output explosion
For set I of d items:

- Number of distinct non-empty itemsets = $2^d - 1$
- Number of distinct association rules = $3^d - 2^{d+1} + 1$

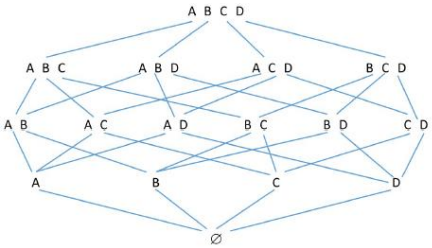
Enumeration of all itemsets/rules (to find the interesting ones) is not possible even for small sizes (say $d > 40$).
Complexity analysis w.r.t. input size might be not meaningful.
We consider efficient strategies those that require time/space polynomial in both the input and the output sizes.

Lattice of Itemsets
The family of itemsets under $\subseteq$ forms a **lattice**:

- 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** ($X \cap Y$)

Represented through the **Hasse diagram:**

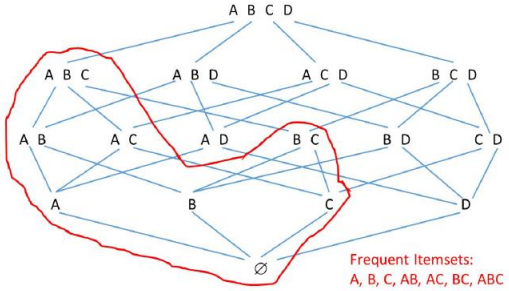
- nodes contain itemsets
- 2 nodes are connected if one is contained in the other.



Anti-monotonicity of support: for every itemsets $X, Y \subseteq I$: $\boldsymbol{X \subseteq Y \Rightarrow \mathrm{Supp}_T(X) \geq \mathrm{Supp}_T(Y)}$
Because, if X is contained in Y, then every transaction that contain Y also contains X.
For a given support threshold:

- X is frequent ⇒ every $W \subseteq X$ is frequent
- X is not frequent ⇒ every $W \supseteq X$ is not frequent

Frequent itemsets form a sublattice closed downwards



3.2. Mining of Frequent Itemsets and Association Rules

Input: Dataset T of N transactions over I, minsup and minconf
Output (Frequent Itemsets): $F_{T,\text{minsup}} = \{(X, \text{Supp}_T(X)) : X \neq \emptyset \; and \; \text{Supp}_T(X) \geq \text{minsup}\}$
Output (Association Rules):

$$\{(r: X \rightarrow Y, \text{Supp}_T(r), \text{Conf}_T(r)) : \text{Supp}_T(r) \geq \text{minsup} \; and \; \text{Conf}_T(r) \geq \text{minconf}\}$$

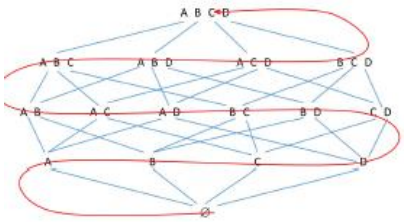
Most algorithms to compute the association rules, work in two phases:

- Compute the frequent itemsets (w.r.t minsup) and their supports (the hardest phase!).
- Extract the rules from the frequent itemsets and their supports.

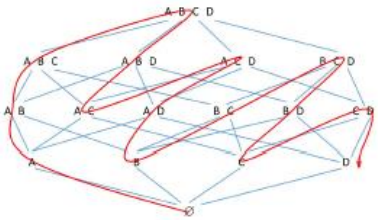
3.2.1. Mining Frequent Itemsets

Objective: Careful exploration of the lattice of itemsets exploiting anti-monotonicity of support

Breadth First



Depth First



Absolute support of $X \subseteq I$: $\boldsymbol{\sigma(X) = \text{Supp}_T(X) * N}$. = # of transactions that contain X
Assume the existence of a total ordering of the items and assume that transactions/itemsets are represented as sorted vectors.

A-Priori algorithm (Breadth First)

- Compute frequent itemsets by increasing length $k = 1,2, \ldots$
- Frequent itemsets of $k > 1$ (F_k) are computed as follows:
  - 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.
  - Extract F_k from C_k computing the support of each $X \in C_k$

$k \leftarrow 1$
Compute $F_1 = \{i \in I \; ; \; \text{Supp}(\{i\}) \geq \text{minsup}\}$
Compute $O_1 = \{(X, \text{Supp}(X)) \; ; \; X \in F_1\}$
repeat
  $k \leftarrow k + 1$
  $C_k \leftarrow$ APRIORI-GEN(F_{k-1}) /* *Candidates* */
  for each $c \in C_k$ **do** $\sigma(c) \leftarrow 0$
  for each $t \in T$ **do**
    for each $c \in C_k$ **do**
      if $c \subseteq t$ **then** $\sigma(c) \leftarrow \sigma(c) + 1$
  $F_k \leftarrow \{c \in C_k \; : \; \sigma(c) \geq N \cdot \text{minsup}\}$;
  $O_k \leftarrow \{(X, \sigma(X)/N) \; ; \; X \in F_k\}$
until $F_k = \emptyset$
return $\bigcup_{k \geq 1} O_k$

apriori-gen(F):

- **Candidate set generation:** by merging pairs from F with all but 1 item in common.
  I.e., itemsets "ABCD", "ABCE" in F generate "ABCDE"
- **Candidate pruning:** Exclude candidates containing an infrequent subset.
  I.e., "ABCDE" survives only if all lenght-4 subsets are in F

  The pruning is done a priori without computation of support.

```
Let ℓ − 1 be the size of each itemset in F
Φ ← ∅
/* Candidate Generation */
for each X, Y ∈ F s.t. 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 s.t. |Y| = ℓ − 1 do
    if (Y ∉ F) then {remove Z from Φ; exit inner loop}
return Φ
```

Efficiency of A-Priori:

- just $k_{max} + 1$ passes over the dataset, where k_{max} is the length of the longest frequent itemset.
- Support is computed only for a few non-frequent itemsets: this is due to candidate generation and pruning which exploiting antimonotonicity of support.
- Several optimizations are known for the computation of supports of candidates this is typically the most time-consuming task.
- The algorithm computes the support for $\leq d + \min\{M^2, d * M\}$ itemsets. Where M is the # of frequent itemsets returned at the end. d is the number of items in I.
- 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).

Main performance issues:

- most time-consuming step is the counting of supports of candidates (sets $C_1, C_2, \ldots$)
- A-Priori requires a pass over the entire dataset checking each candidate against each transaction (a lot for large input/output sizes)
- The number of candidates may still be very large stressing storage and computing capacity.

Optimizations of A-Priori:

- Use of a trie-like data structure (called HASH TREE) to store C_k and to speed-up support counting.
- Use of a strategy to avoid storing all pairs of items in C_2. Namely,
  - 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

DATASET T	
TID	ITEMS
1	ACD
2	BEF
3	ABCEF
4	ABF

- $N = 4$ **transactions**, $d = 6$ **items**.
- Let's fix minsup $= 0.5$.
- **Observation:** a brute force strategy would compute the support of $2^d - 1 = 63$ itemsets.

Example: computation of F_1

DATASET T	
TID	ITEMS
1	ACD
2	BEF
3	ABCEF
4	ABF

ITEMSET	SUPPORT
A	3/4
B	3/4
C	1/2
~~D~~	~~1/4~~
E	1/2
F	3/4

Set F_1	
ITEMSET	SUPPORT
A	3/4
B	3/4
C	1/2
E	1/2
F	3/4

Example: computation of F_2

Set C_2		
ITEMSET		SUPPORT
After Generation	After Pruning	
AB	AB	
AC	AC	
AE	AE	
AF	AF	
BC	BC	
BE	BE	
BF	BF	
CE	CE	
CF	CF	
EF	EF	

Observation: C_2 consists always of all pairs of frequent items.

Set C_2		
ITEMSET		SUPPORT
After Generation	After Pruning	
AB	AB	1/2
AC	AC	1/2
~~AE~~	~~AE~~	~~1/4~~
AF	AF	1/2
~~BC~~	~~BC~~	~~1/4~~
BE	BE	1/2
BF	BF	3/4
~~CE~~	~~CE~~	~~1/4~~
~~CF~~	~~CF~~	~~1/4~~
EF	EF	1/2

Example: computation of F_2

Set F_2	
ITEMSET	SUPPORT
AB	1/2
AC	1/2
AF	1/2
BE	1/2
BF	3/4
EF	1/2

Example: computation of F_3

Set C_3		
ITEMSET		SUPPORT
After Generation	After Pruning	
ABC		
ABF	ABF	1/2
ACF		
BEF	BEF	1/2

DATASET T	
TID	ITEMS
1	ACD
2	BEF
3	ABCEF
4	ABF

Example: computation of F_4

Set F_3	
ITEMSET	SUPPORT
ABF	1/2
BEF	1/2

C_4 is empty $\Rightarrow$ F_4 is empty and the algorithms STOPS.

Observations:

- Note that although ABF and BEF share 2 out of 3 items, they are not mergeable because they don't share the first 2 items.

  Their union ABEF is surely not frequent since if it were frequent ABE and ABF would be frequent, and ABEF would be generated by these itemsets
- Overall, the support of "only" 18 (< 63 !) itemsets has been computed

Other approaches to frequent itemsets mining: depth-first mining strategies

- avoiding several passes over the entire dataset of transactions.
- confining the support counting of longer itemsets to suitable small projections of the dataset, typically much smaller than the original one.

3.2.2. Mining association rules

Let

- T set of N transactions over the set I of d items
- $\text{minsup}, \text{minconf} \in (0,1)$ support/confidence thresholds.
- $\text{Supp}_T(r) = \text{Supp}_T(X \cup Y)$
- $\text{Conf}_T(r) = \text{Supp}_T(X \cup Y)/\text{Supp}_T(X)$

Compute: $\{(r: X \to Y, \text{Supp}_T(r), \text{Conf}_T(r)): \text{Supp}_T(r) \geq \text{minsup } and \text{ Conf}_T(r) \geq \text{minconf}\}$
If F is the set of frequent itemsets w.r.t. minsup, then for every rule $r: X \to Y$ that we target, we have that $X, X \cup Y \in F$

Mining strategy:

- Compute the frequent itemsets w.r.t. minsup and their supports
- For each frequent itemset Z compute $\{r: Z - Y \to Y \; st. Y \subset Z \; and \; \text{Conf}_T(r) \geq \text{minconf}\}$

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.
If

- $Y' \subset Y \subset Z$
- $r_1 = Z - Y' \to Y'$
- $r_2 = Z - Y \to Y$

Then: $\text{Conf}_T(r_1) < \text{minconf} \Rightarrow \text{Conf}_T(r_2) < \text{minconf}$

Frequent itemsets	
ITEMSET	SUPPORT
A	1/2
B	3/4
C	3/4
E	3/4
AC	1/2
BC	1/2
BE	3/4
CE	1/2
BCE	1/2

$Z = BCE$ and $\text{minconf} = 3/4$.

$$r_1 = BC \to E$$
$$r_2 = B \to CE$$

The confidences are:

$$\text{Conf}(r_1) = \frac{1}{2}/\frac{1}{2} = 1$$
$$\text{Conf}(r_2) = \frac{1}{2}/\frac{3}{4} = \frac{2}{3}$$

Also, the fact that

$$\text{Conf}(BE \to C) = 2/3 < 3/4$$

implies immediately $\text{Conf}(r_2) < 3/4$

High-level strategy for mining rules
For each frequent itemset Z: (polynomial time)

- Check confidence of rules $Z - Y \to Y$ by increasing length of Y ($|Y| = 1,2, \ldots$)
- Once all consequents Y of length k have been checked generate "candidate" consequents of length $k + 1$ to be checked, using apriori-gen.

3.3. Frequent itemset mining for Big Data

When the dataset T is very large.

Partition-based approach: (Exact solution but inefficient)

- Partition T into K subsets (K is a design parameter)
- Mine frequent itemsets independently in each subset
- Extract the final output from the union of the frequent itemsets computed before.

Sampling approach: compute the frequent itemsets from a small sample of T.

- approximation but is efficient

Partition-based approach

Setting

- T = set of N transactions over the set I of d items.
  Assume :
  $$T = \{(i, t_i) : 0 \leq i < N\},$$
  where t_i is the ith transaction.
- $\text{minsup} \in (0, 1)$: support threshold.
- K = suitable *design parameter* $(1 < K < N)$.

Round 1:

- Partition T arbitrarily into K subsets $T_0, T_1, \ldots T_{K-1}$.
- In each T_i compute the set Φ_i of frequent itemsets w.r.t. 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 $\Phi = \cup_{i=0}^{K-1} \Phi_i$ and eliminate the duplicates.

Round 3: For every $0 \leq i < K$ independently do the following:

- Gather T_i and Φ
- For each $X \in \Phi$ count the number of transactions of T_i that contain X (call this number $\sigma(X, i)$).

Round 4: For each $X \in \Phi$, gather all $\sigma(X, i)$'s, compute the final support $\text{Supp}_T(X) = (1/N) \sum_{i=0}^{K-1} \sigma(X, j)$ and output X if $\text{Supp}_T(X) \geq \text{minsup}$.

Correctness: immediate from the following observations:

- For each itemset $X \in \Phi$ the exact support is computed in R3+R4
- A "true" frequent itemset X must belong to some Φ_i, and hence in Φ.: since $|T_X| \geq N \cdot \text{minsup}$, there must be a subset T_i where X appears in $|T_X|/K \geq (N/K) \cdot \text{minsup}$ transactions, and thus X is in Φ_i.

Remark: Φ may contain many infrequent itemsets (*false positives*)!

Number of rounds: 4.

Space requirements:

- Depend on the size of Φ, which cannot be easily predicted
- The larger K, the more false positive are likely to occur.

Remark: *while the algorithm may work well in practice, it does not feature strong theoretical guarantees.*

Sampling-based approach

Definition (Approximate frequent itemsets)

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

1. For each $X \in F_{T,\text{minsup}}$ there exists a pair $(X, s_X) \in C$
2. For each $(X, s_X) \in C$,
   - $\text{Supp}(X) \geq \text{minsup} - \epsilon$
   - $|\text{Supp}(X) - s_X| \leq \epsilon$,

where $F_{T,\text{minsup}}$ is the true set of frequent itemsets w.r.t. T and minsup.

- Condition (1) ensures that the approximate set C comprises all true frequent itemsets
- Condition (2) ensures that: (a) C does not contain itemsets of very low support; and (b) for each itemset X such that $(X, s_X) \in C$, 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 $f(\text{minsup}) < \text{minsup}$ be a suitably lower support threshold

- Let $S \subseteq T$ be a sample drawn at random with uniform probability and with replacement
- Return the set of pairs

$$C = \left\{ (X, s_x = \text{Supp}_S(X)) \ : \ X \in F_{S,f(\text{minsup})} \right\},$$

where $F_{S,f(\text{minsup})}$ is the set of frequent itemsets w.r.t. S and $f(\text{minsup})$.

Theorem (Riondato-Upfal)

Let h be the maximum transaction length and let ϵ, δ be suitable design parameters in $(0,1)$. There is a constant $c > 0$ such that if

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

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

The result stated in the theorem relies on the following lemma which can be proved through an elegant exploitation of the notion of VC-dimension (see [RU14] for details).

Lemma

Unsing the sample size $|S|$ specified in the above theorem, the algorithm ensures that with probability $\geq 1-\delta$ for each itemset X it holds that $|Supp_T(X) - Supp_S(X)| \leq \epsilon/2$.

- 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 ϵ, δ, and on the max transaction length h, which is often quite low.
- Tighter bounds on the sample size are known.
- 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 reduce.

3.4. Limitations

Limitations of the Support/Confidence framework:

- **Redundancy**: many returned patterns may characterize the same subpopulation of data (e.g., transactions/customers).
- **Difficult control of output size**: it is hard to predict how many patterns will be returned for given support/confidence thresholds.
- **Significance**: are the returned patterns significant, interesting?

3.4.1. Closed and Maximal itemsets

Avoid redundancy and (attempt to) control the output size.

Itemset $X \subseteq I$ is **Closed** wrt T: if **for each** superset $Y \supset X$ we have $\text{Supp}_T(Y) < \text{Supp}_T(X)$
It means that X is **closed** if its support decreases as soon as an item is added.

- $\text{CLO}_T = \{X \subseteq I : X \text{ is closed wrt } T\}$ set of **closed** itemsets
- $\text{CLO-F}_{T,\text{minsup}} = \{X \in \text{CLO}_T : \text{Supp}_T(X) \geq \text{minsup}\}$ set of **frequent closed** itemsets

Itemset $X \subseteq I$ is **Maximal** wrt T and minsup: if $\text{Supp}_T(X) \geq \text{minsup}$ and **for each** superset $Y \supset X$ we have $\text{Supp}_T(Y) < \text{minsup}$
It means that X is **maximal** if it is frequent and becomes infrequent as soon as an item is added.

- $\text{MAX}_{T,\text{minsup}} = \{X \subseteq I : X \text{ is maximal wrt } T \textit{ and } \text{minsup}\}$

Dataset T	
TID	ITEMs
1	ABC
2	ABCD
3	BCE
4	ACDE
5	DE

For minsup $= 2/5$, identify:

(a) A maximal itemset
(b) A frequent closed itemset which is not maximal
(c) A closed itemset which is not frequent

Answer

(a) ACD (Support $= 2/5$)
(b) AC (Support $= 3/5$)
(c) ACDE (Support $= 1/5$)

Dataset T	
TID	ITEMs
1	ABC
2	ABCD
3	BCE
4	ACDE
5	DE

For minsup $= 2/5$, identify:

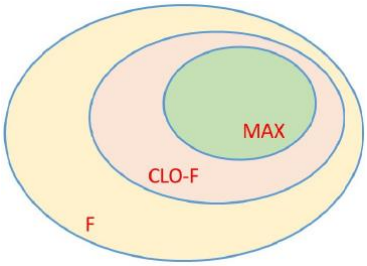
(d) Set F (frequent itemsets)
(e) Set CLO-F
(f) Set MAX

Answer

(d) F = A,B,C,D,E,AB,AC,AD,BC,CD,CE,DE,ABC,ACD
(e) CLO-F = C,D,E,AC,BC,CE,DE,ABC,ACD
(f) MAX = CE, DE, ABC,ACD

Properties of Closed/Maximal itemsets:

- **Property 1**: $\text{MAX} \subseteq \text{CLO-F} \subseteq \text{Frequent Itemset}$
- **Property 2:** for each itemset X there is a **closed** itemset X' such that
  - $X \subseteq X'$
  - $\text{Supp}_T(X') = \text{Supp}_T(X)$
- **Property 3:** for each **frequent** itemset X there is a **maximal** itemset X' such that
  - $X \subseteq X'$



Dataset T	
TID	ITEMs
1	ABC
2	ABCD
3	BCE
4	ACDE
5	DE

- X=AE, X'=ACDE (support $= 1/5$)
- X=AD, X'=ACD (support $= 2/5$)
- X=C, X'=C (support $= 4/5$)

minsup $= 2/5$

- X=AC (support $= 3/5$), X'=ACD (support $= 2/5$)

Consequences of the properties:

MAX and CLO-F provide a compact and lossless representations of F:

- Determine F by taking all subsets of MAX or CLO-F.

CLO-F (+ supports) provides a compact and lossless representation of F (+ supports):

- Determine F from CLO-F as above
- For each $X \in F$ compute $\text{Supp}_T(X) = \max\{\text{Supp}_T(Y): Y \in \text{CLO-F and } X \subseteq Y\}$

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

If $\text{Supp}_T(X) > 0$:

- $X \subseteq \text{Closure}(X)$
- $\text{Supp}_T(\text{Closure}(X)) = \text{Supp}_T(X)$
- $\text{Closure}(X)$ is closed

Dataset T	
TID	ITEMs
1	ABC
2	ABCD
3	BCE
4	ACDE
5	DE

- $X = AB$
- $\text{Closure}(X) = ABC$

Each closed itemset Y represents compactly all (many) itemsets X such that $\text{Closure}(X) = Y$

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.

Maximal/frequent closed itemsets yield a quite effective reduction of the output size.

However, there are non-trivial pathological cases where their number is exponential in the input size.

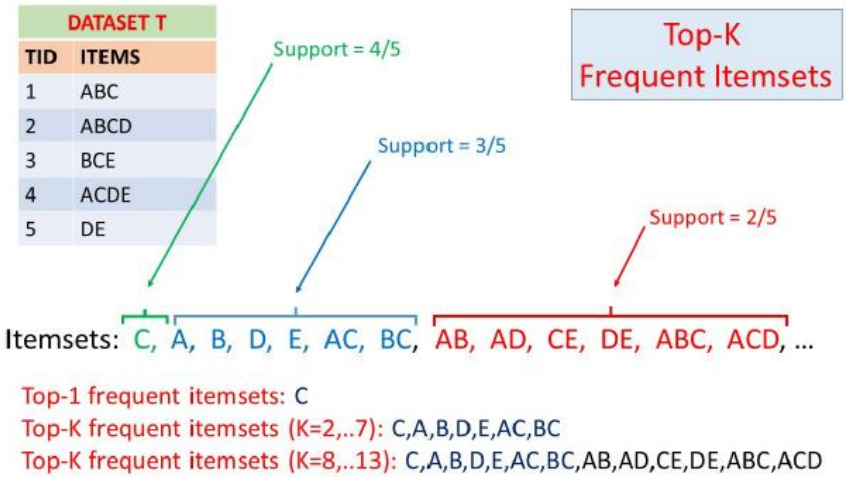
3.4.2. Top-K frequent itemsets

Controls output size.

Let

- $X_1, X_2, \ldots$ numeration of the non-empty itemsets in non-increasing order of support.
- For a given integer $K \geq 1$
- $s(K) = \mathrm{Supp}_T(X_K)$

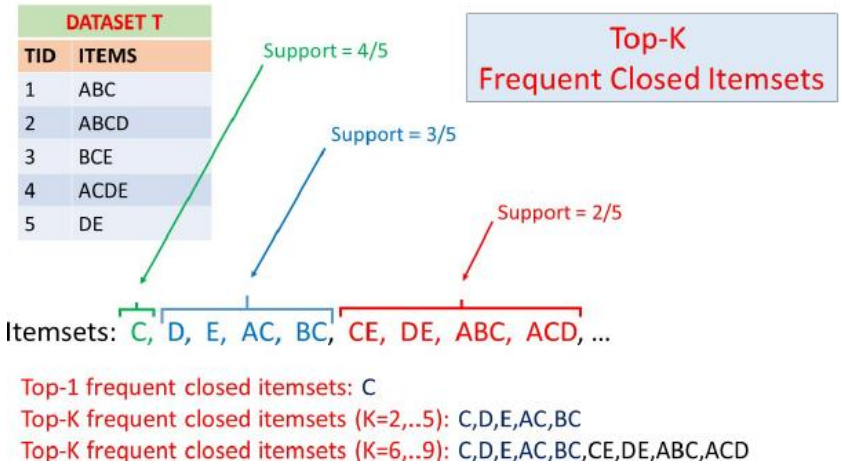
The **Top-K frequent itemsets** w.r.t. T are all itemsets of support $\geq s(K)$
The number of Top-K frequent itemsets is $\geq K$



Let

- $X_1, X_2, \ldots$ numeration of the non-empty **closed** itemsets in non-increasing order of support
- For a given integer $K \geq 1$
- $sc(K) = \mathrm{Supp}_T(X_K)$

The **Top-K frequent Closed itemsets** w.r.t. T are all itemsets of support $\geq sc(K)$



There are efficient algorithms for mining the Top-K frequent (closed) itemsets. A popular strategy is this:

- Generate (closed) itemsets in non-increasing order of support (using a priority queue).
- Stop at the first itemset smaller support than the K-th one.

Top-K frequent closed itemsets there is a tight polynomial bound on the output size.
For Top-K frequent itemsets there are pathological cases with exponential output size.

Control on the output size

For the Top-K frequent closed itemsets, K provides tight control on the output size.

For $K > 0$, the **# of Top-K frequent closed itemsets** = $\boldsymbol{O(d * K)}$ where d=#of items

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.

Maximal itemsets:

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

Frequent Close 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.
- Reduction of redundancy

Top-K frequent (closed) itemsets:

- Output size: $O(d * k)$ if restricted to closed itemsets, otherwise small in practice but exponential in d for pathological cases.
- Reduction of redundancy and control on the output size (with closed itemsets)