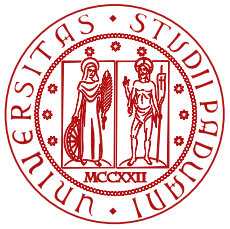


Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Master degree in Computer Engineering

Big Data Computing

University of Padova

Academic Year
2023/2024

Contents

1

MapReduce and Partitioning

In a wide spectrum of domains there was an increasing need to analyze large amounts of data.
Available tools and commonly used platforms could not practically handle very large datasets.
Was important to find powerful computing systems, which feature multiple processors, multiple storage devices, and high-speed communication networks.
However, these powerful systems had several problems like the maintenance cost, huge programming skills required for the task allocation for leveraging on parallel computing and, moreover, **fault-tolerance** becomes serious issue: a large number of components implies a low **Mean-Time Between Failures (MTBF)**.
When we talk about fault-tolerance we mean the ability of a system to continue functioning properly in the event of the failure of some of its components.
From the moment that there are a large number of components, it implies a very low MTBF that is a measure used in engineering to quantify the reliability of a system. It represents the average time elapsed between two consecutive failures of a component within the system. In other words, it indicates how long a component is expected to operate before it fails.
On the notebook is shown an example for this concept.

1.1 MapReduce in general

MapReduce was firstly introduced as a programming framework for big data processing on distributed platforms, where large datasets are processed in parallel.
Being a programming framework means that MapReduce provides both a model for writing applications and the infrastructure to execute, based on clusters them. The framework handles many of the complexities involved with running parallel algorithms, including fault tolerance, data distribution, and task scheduling which are hidden to the programmer.
Over time, as the field of big data evolved, the concept of MapReduce was integrated into larger, more versatile systems. Now MapReduce applications runs on clusters of commodity processors (as a laptop) and cloud infrastructures.
Several software frameworks have been proposed to support MapReduce programming. For example Apache Spark or Apache Hadoop.
How is the cluster's architecture: A cluster refers to a computing platform consisting of multiple interconnected compute nodes via a fast network.
Typically, these nodes are traditional computers (often referred to as "commodity hardware"), each with its own RAM, disk, and operating system.
This collection of nodes becomes a single computing platform - a multiprocessor - that can be used to run a given application, thanks to the nodes communicating with each other via the network.
When a large data set is involved, the processing job can be divided into smaller chunks, which are then processed simultaneously by different nodes in the cluster. This parallelism significantly speeds up processing time compared to doing the same job on a single machine.
A typical **cluster architecture** is composed by **racks** of 16-64 compute nodes (CPU or more CPUs) connected by switches to work together, as shown in Figure 1.1 Moreover, a fundamental part of the cluster is the **Distributed File System**: the role of the DFS in a cluster environment, especially in large-scale data processing tasks like those performed by MapReduce, is vital. It allows the system to efficiently distribute the workload across multiple nodes by ensuring that the necessary data is available locally where the tasks are executed, reducing network traffic and speeding up processing times.
The DFS subdivides the dataset into **chunk** (e.g. of 64 MB) and replicates each chunk in different nodes in order to ensure **fault-tolerance**. An example of DFS

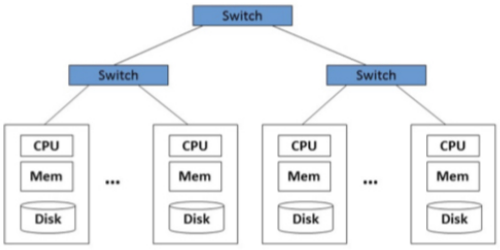


Figure 1.1: Cluster architecture

is **Hadoop DFS**.

What is a cloud infrastructure? As said before, MapReduce applications runs not only on clusters of commodity processors but also on **cloud infrastructure**. Cloud infrastructure refers to the virtualized and scalable resources that are provided over the internet by cloud service providers.

These resources include computing power, storage, networking, and even entire platforms and software that run on remote data centers.

The basic idea is to provide users the capability to access and manage resources without the need to own and maintain physical hardware.

We will use MapReduce as a **conceptional model** for high-level algorithm (more abstract) design and analysis and **Spark** as a programming framework for implementing MapReduce algorithms. The most important thing for MapReduce is the **data centric view**. Moreover it is inspired by functional programming (use of maps and reduce functions).

1.2 MapReduce as a conceptional model

A MapReduce computation can be viewed as a **sequence of rounds**.
A round transforms **a set of key-value pairs** into another set of key-value pairs (data centric view), through the following two phases:

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

Terminology: the application of the reduce function to a pair (k, L_k) is referred to as reducer.
In Figure 1.2 the diagram representation of one round of computation.

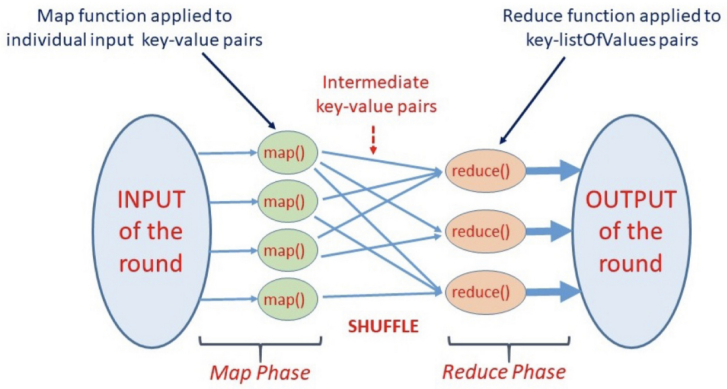


Figure 1.2: Single round computation

We can give also a representation of multiple round: the input of a round comprises input data and/or data from the ouputs of the previous round. It's convenient to define **shared variables**, available to all executors, which maintain global information.
In Figure 1.3 is shown this concept.
It's natural to question why we use key-value pairs instead of representing data as object. The answer is since MapReduce has a data-centric view the **keys** are needed as **address** to reach objects and as **labels** to define the groups in the

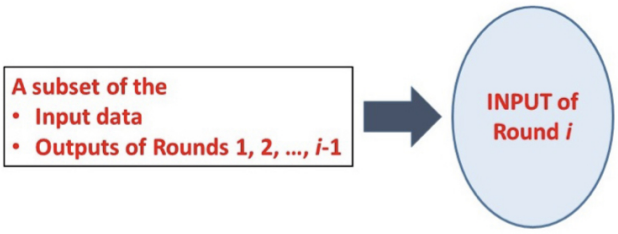


Figure 1.3: More round computation

reduce phases.
The domains for key and values can change from one round to another and, using frameworks as Spark, it's possible to manage data without using keys.
In a MapReduce algorithm must be specified so that:

- input and output clearly defined
- sequence of rounds executed has to be unambiguously implied
- for each round are clear:
  - input, intermediate and output sets of key-value pairs
  - function applied in the two phases
  - bounds for key performance indicators

In order to specify a MR algorithm with a fixed number of rounds, we will use pseudocode as in Figure 1.4:

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

Round 1:

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

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

Figure 1.4: Pseudocode

We can give give an example. Suppose to take the example shown in Figure 1.5, in which we have to count words in some documents.

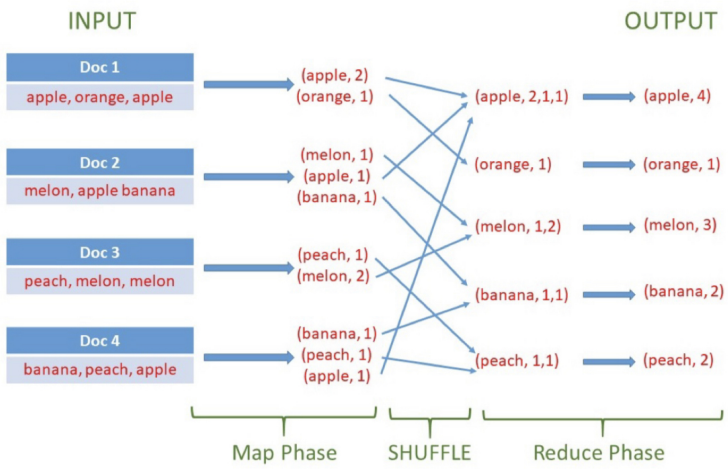


Figure 1.5: Cluster architecture

Input: k documents $D_1, D_2, ..., D_k$ where each document D_i is a key-value pair (name, list of word) and N is the total size of documents, the number of all words.
Output: $\{(w, c(w)) : w$ is a word, $w \in D_1 \cup D_2 \cup ... \cup D_k, c(w) =$ # occurences of w in all doc $\}$.
For the example above, $N = 12$ and $k = 4$.
Round 1:

- Map Phase: for each $1 \leq i \leq k$ separately do $D_i \hookrightarrow \{(w, c_i(w))$ with the arrow symbol represents the **map function**
- Reduce Phase: for each w separately let L_w the list of values of intermediate key-value pairs i.e $L_w =$ list of counts $c_i(w)$.
  $(w, L_w) \hookrightarrow (w, \sum_{c_i(w) \in L_w} c_i(w))$ where the arrow symbol represents the **reduce function**

When we run our algorithm on **distributed platform**:

- **Input and output files reside on a Distributed File System, while intermediate data are stored on the executors' local memories/disks.**
- **Map phase:** to each executor is assigned a subset of input pairs and applies the map function to them sequentially, one at a time.
- **Shuffle:** the system collects the intermediate pairs from the executors' local spaces and groups them by key.

- **Reduce phase:** each executor is assigned a subset of (k, L_k) pairs, and applies the reduce function to them sequentially, one at a time.

In order to analyze the MR algorithm is needed to estimate the **key performance indicators**:

- Number of rounds R; rough estimate of the running time under the assumption that **SHUFFLES** dominate the complexity and have comparable costs across different rounds.
- Local space M_L that is the maximum amount of main memory required by a single invocation of a map or reduce function. The maximum is taken over all rounds and all invocations at each round. In other words is the maximum amount of **main memory** needed at each executor, i.e., at each compute node.
- Aggregate space M_A that is the maximum amount of space which is occupied by stored data at the beginning/end of a map or reduce phase. The maximum is taken over all rounds. It's the overall amount of **disk space** needed in the system, i.e., the capacity of the DFS.

As example, we can analyze the example above for Word Counting: obviously the number of rounds is 1. For providing M_L we define N_i that is the number of words of a single document. We take the maximum across all documents that is N_{MAX}.
So in the Map Phase we have $O(N_{MAX})$ local space and then, in the Reduce Phase we have $O(k)$ local space.
We take the maximum among these two and is asintotic to $O(N_{MAX} + k)$.
In order to compute M_A we define N as the total number of words and, space occupied for input, intermediate and output phase is always $O(N)$ so $M_A = O(N)$.

1.2.1 Design goals for MapReduce algorithms

A simple observation: for every problem solvable by a sequential algorithm in space S there exists a 1-round MapReduce algorithm with both M_L and M_A proportional to S since assigning the same key key $\emptyset$ to all input elements and use the sequential algorithm as reduce function implicate that.
However this is not practical for very large input due to these reasons:

- very large memory platform is needed
- no parallelism

In Figure 1.5 the design goals for MapReduce algorithms
These goals are conflicting, so we search for a tradeoff between them. Supposing that they are enumerated with a growing order, we can say that:

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

Figure 1.6: Design goals

- 1 and 4 aim to time efficiency, with 1 aims to low communication costs and 4 aims to low computational costs
- 2 and 3 aim at space efficiency, with 2 aims to a low main memory requirement and 3 to low data replication

Why is MapReduce suitable for big data processing?

- Data-centric view: Algorithm design focuses on data transformations targeting the above design goals.
- Usability: program has not to allocate task to nodes, handling failure etc.
- Portability/Adaptability
- Cost

However, there are some **drawbacks**:

- Running time is only coarsely captured by R, we should use more sophisticated time-performance
- In some cases, one or a few reducers may be much slower than the others, thus delaying the end of the round. Algorithm designers should aim at balancing the load among reducers
- MapReduce is not suitable for applications that require very high performance

1.3 Partitioning - an approach for satisfying space goal

A typical approach to obtain the stated goal on local space is to:

- **subdivide** the relevant data in small **partitions** either deterministically or randomly
- **make** reduce functions work separately in individual partitions

Although partitioning may increase the number of rounds and rarely the aggregate space, but, a suitable trade-off can be find.

1.3.1 Deterministic Partitioning

Class Count Problem: given a set S of N objects with class labels, count how many objects belong to each class.
More precisely:
Input: Set S of N objects represented by pairs $(i, (\gamma_i, o_i))$ for $0 \leq i \leq N$, where γ_i is the class of the $i-th$ object o_i
Output: The set of pairs $\gamma, c(\gamma)$, where γ is a class labelling some object of S and $c(\gamma)$ is the number of objects of S labeled with γ.
On notebook and example of different approach for solving one exercise: **growing the number of round, we can reduce the local space, as proven by analysis**.
The choice of $\sqrt{N}$ partitions (which we will often make) provides an example of a simple tradeoff between local space and number of rounds.
If more stringent bounds on the local space are imposed, different partitionings may be needed, yielding different round-space tradeoffs.
In practice, the number of partitions is fixed as a multiple of the number of available workers, as long as local space is not an issue.

1.3.2 Random Partitioning

It's a translation of deterministic partitioning and the main difference is that the input pairs does not have the integer key $[0, N)$.
More precisely:
Input: Set S of N objects represented by pairs (γ_i, o_i) for $0 \leq i \leq N$, where γ_i is

the class of the $i-th$ object o_i
Output: The set of pairs $\gamma, c(\gamma)$, where γ is a class labelling some object of S and $c(\gamma)$ is the number of objects of S labeled with γ.
We can employ the second round as before, but the partitioning of round 1 will be achieved by **assigning a random key** from $[0, l)$ to each of the intermediate pair.

1.3.3 Analysis of methods

On the notebook it is shown random partitioning on class count problem and it's emerged that can be only worse than deterministic one!
Resuming, for the example seen, M_A has the same bound for both methods, but, M_L is quite different:

- for deterministic partitioning $M_L = O(\sqrt{N})$
- for random partitioning $M_L = O(max\{m, l\})$

We note that $m \geq \frac{N}{l}$ since when partitioning N objects into l groups there must exist a group with at least $\frac{N}{l}$ objects. Based on this consideration our hope is that $m \approx \frac{N}{l}$.
It follows an important theorem in Figure 1.6

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

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

Figure 1.7: Theorem

Based on this theorem, if we will set $l = \sqrt{N}$ we get

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

with a probability very close to 1 for a larger N, so we get the same result of deterministic method.
In the notebook it is proven this theorem.
The conclusion is that if we assign to N objects a random partition among l possibilities, and $\frac{N}{l}$ is sufficiently large, with high probability there are not

partitions with an excessive number of data.
We can do some observations on different types of partitioning:

- Deterministic partitioning: applicable whenever input pairs contain pieces of data which can be mapped into partition indices that ensure a sufficiently even partitioning of the pairs.
- Random partitioning: always applicable, but to ensure a sufficiently even partitioning, the expected size of each partition should

2

Coreset and Composable Coresets

Suppose that we want to solve a problem Λ on instances P which are too large to be processed by known algorithms.
The **coreset technique** consists on:

- Extract a small subset T from P, that's called coreset, making sure that it's a good representation of P
- Run the best known, possibly slow, sequential algorithm for Λ on the small coreset T, rather than on the entire istance

The technique is effective if T can be extracted efficiently by processing P, possibly in a distributed or streaming fashion and if the solution computed on T is a good solution for Λ w.r.t. the entire input P.
In Figure 2.1 the representation of the construction behind the idea.

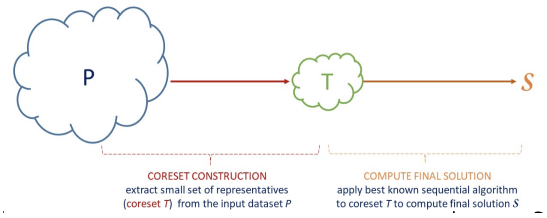


Figure 2.1: Representation of the idea

In general, T can be the result of some processing of the elements of P, and, for instance, may contains elements of P plus additional information.
The coreset technique can be enhanced with **composable coreset** techniques:

- Partition P into l subsets $P_1, P_2, .., P_l$ and extract a small coreset T_i to each P_i, ensuring that P_i is well represented

- Run best known sequential algorithm for Λ on $T = \cup T_i$, rather than on P

The technique is effective if each T_i can be extracted efficiently from P_i in parallel for all i and if the final coreset T is still small and the solution computed on T is a good solution for Λ.
In Figure 2.2 the representation of the construction behind the idea.

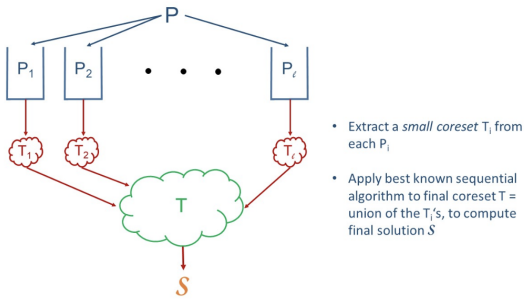


Figure 2.2: Representation of the idea

2.1 Case study: k-center clustering

Clustering is a fundamental primitive for data analysis but many applications of clustering are NP hard and benefit from the use of composable coreset technique.
The general definition of clustering is: 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" and points in different clusters are "distant".
Rather than a distance function, it's possible also to use a **similarity function**: close points are similar, distant points are dissimilar.
A clustering problem is usually defined by requiring that clusters optimize a given objective function, and/or satisfy certain properties.
In Figure 2.3, an example of clustering problem.
Typically, the input of a clustering problem consists of a set of points from a **metric space**: a metric space is an ordered pair (M, d) where M is a set and d is a distance function defined as $d : MxM \hookrightarrow \mathbb{R}$ such that the following property are valid.

- is symmetric $d(\vec{x}, \vec{x'}) = d(\vec{x'}, \vec{x})$ for all $\vec{x}, \vec{x'} \in M$

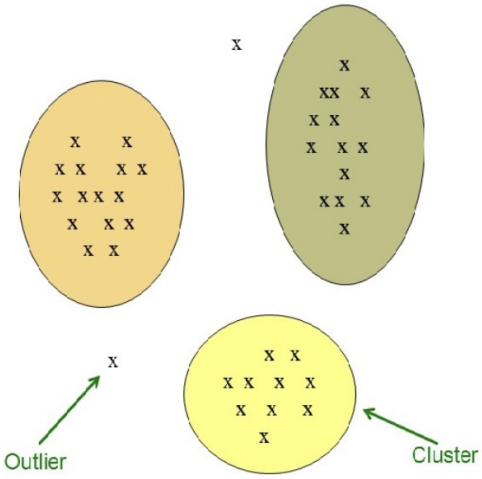


Figure 2.3: Clustering problem

- $d(\vec{x}, \vec{x}) = 0$ for all $\vec{x} \in M$
- d satisfies the triangle inequality $d(\vec{x}, \vec{x'}) \leq d(\vec{x}, \vec{z}) + d(\vec{z}, \vec{x'}) =$

The choice of the function may have a significant impact on the effectiveness of the cluster analysis.
The triangle inequality is very important in the big data context in order to prune or make smoother the computation.
There are different distance function available and we list most used:

- **Minkowski distance**: The Minkowski distance, also known as the L_r norm, is a generalization of different distance metrics, including Manhattan distance ($r = 1$) and Chebyshev distance ($r \to \infty$). It measures the distance between two points X and Y in n-dimensional space using the formula:

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

where $X = (x_1, x_2, ..., x_n)$ and $Y = (y_1, y_2, ..., y_n)$ are two points in n-dimensional space.
For $r = 1$, the Minkowski distance reduces to the Manhattan distance, which is commonly used in grid-like environments. The formula for Manhattan distance is:

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

For $r = \infty$, the Minkowski distance becomes the Chebyshev distance, which measures the maximum absolute differences of coordinates between two points. The formula for Chebyshev distance is:

$$d_{L_r}(X, Y) = \max_{i=1}^{n} |x_i - y_i|$$

In Figure 2.4 an example of Minkowski distance and a comparison between Minkowski distances, for points at distance ≤ 1 from center O

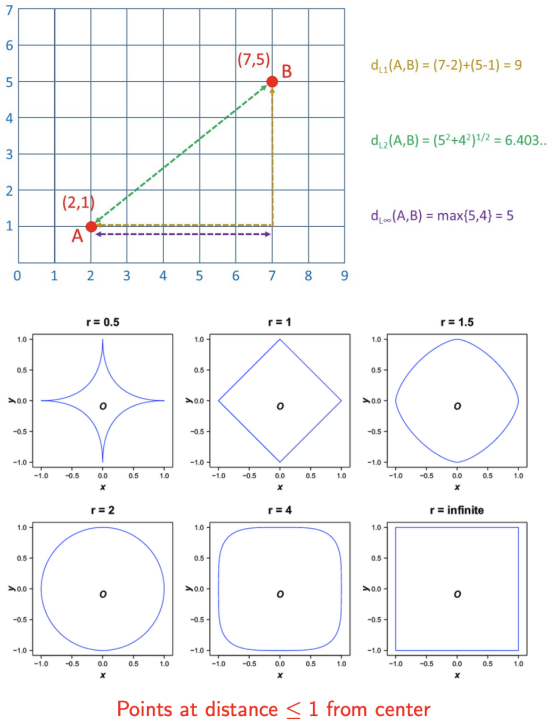


Figure 2.4: Minkowski distances

- The angular distance between two vectors X and Y in n-dimensional space is calculated as the angle between them. Mathematically, it is given by:

$$d_{\text{angular}}(X, Y) = \arccos\left(\frac{X \cdot Y}{\|X\| \cdot \|Y\|}\right) \in [0, \pi]$$

where $X = (x_1, x_2, ..., x_n)$ and $Y = (y_1, y_2, ..., y_n)$ are two vectors in n-dimensional space, and $\cdot$ represents the dot product. The dot product of X and Y is given by:

$$X \cdot Y = \sum_{i=1}^{n} x_i y_i$$

And the norm (magnitude) of a vector X is given by:

$$\|X\| = \sqrt{\sum_{i=1}^{n} x_i^2}$$

For non-negative coordinates, the angular distance $d_{\text{angular}}(X, Y)$ lies in the range $[0, \frac{\pi}{2}]$, while for arbitrary coordinates, the range is $[0, \pi]$, as shown in Figure 2.5
To normalize values in the range $[0, 1]$, $d_{\text{angular}}(X, Y)$ is often divided by π

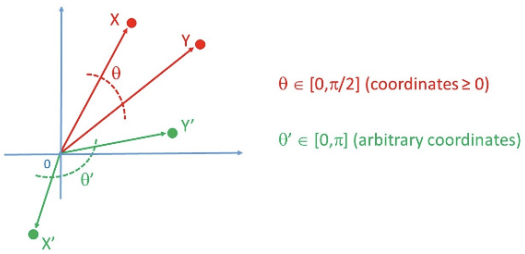


Figure 2.5: Angular distance

or $\frac{\pi}{2}$ depending on whether vectors have arbitrary or non-negative coordinates, respectively.

Additionally, to satisfy the second property of metric spaces, scalar multiples of a vector are regarded as the same vector.

It's important to note that this distance metric is commonly used in information retrieval for documents. A document X over a vocabulary of n words can be represented as an n-vector, where x_i is the number of occurrences of the i-th word of the vocabulary in X.

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

$$d_{\text{Hamming}}(X,Y) = |\{i \mid x_i \neq y_i\}|$$

For example, for $X = (0,1,1,0,1)$ and $Y = (1,1,1,0,0)$, the Hamming distance $d_{\text{Hamming}}(X,Y)$ is:

$$d_{\text{Hamming}}(X,Y) = |\{1,5\}| = 2$$

Observation: The Hamming distance $d_{\text{Hamming}}(X,Y)$ is equivalent to the Manhattan distance $d_{L_1}(X,Y)$.

- **Jaccard distance**: Used when points are sets (e.g., documents seen as bags 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:

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

Note that the distance ranges in the interval [0,1], and it is 0 if $S = T$ and 1 if S and T are disjoint. The value $\frac{|S \cap T|}{|S \cup T|}$ is referred to as the Jaccard similarity of the two sets. An example is provided in Figure 2.6

Minkowski and Angular distances are used when an object is characterized by the numerical values of its features (e.g., frequency).

Hamming and Jaccard distances are used when an object is characterized by having or not having some features (no multiplicity).

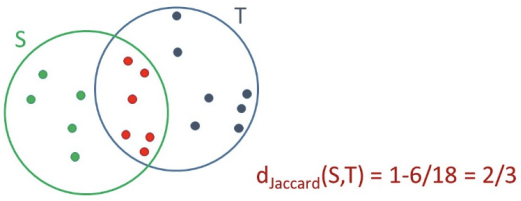


Figure 2.6: Jaccard distances

Moreover, there are not only different distance functions: it's important to evaluate different types of clusterings.
Given a set of points in a metric space, a clustering problem often specifies an objective function to optimize.

A common approach in clustering is:

- define a cost function over possible partitions of the objects
- find the partition (=clustering) of minimal cost

Clustering problems can be categorized based on whether or not

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

It's important to recall the definition of an **optimization problem** Λ that is defined as:

- a set of instances I as possible inputs
- $\forall i \in I$, a set of S_i feasible solutions
- an objective function $\Phi(\cdot)$

The problem is, given an instance $i \in I$, to find a solution in the set of solution $s \in S_i$ such that, $\Phi(s)$ is minimum among all possible $s' \in S_i$ (**minimization problem**). Notice that for an instance i there might me more than a single optimal solution.
It's crucial also to define the definition of **c-approx** algorithm A for a problem Λ: given a $c \geq 1$,(the **approximation ratio**) A is an algorithm that given $i \in I$, returns a solution $A(i) \in S_i$ s.t

$$\Phi(A(i)) \leq c \cdot min\{\Phi(s')|s' \in S\}$$

for minimization problem.

Observations

Approximation algorithms play a crucial role in big data computing for the following reasons:

- For several important optimization problems, finding optimal solutions for large inputs is very time-consuming. For instance:
  - NP-hard problems for which it is likely that exponential time is required.
  - Problems for which polynomial-time algorithms exist but require a superlinear number of operations (e.g., quadratic, cubic).
- Allowing for approximations widens the spectrum of possible solutions and makes it easier to find one.
- In real-world instances, often affected by noise or uncertainty, the true optimum may not be well-defined.

2.1.1 Center-based clustering

Let P be a set of N points in a 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 suitably selected centers for the clusters, where $c_i \in C_i$ for every $1 \leq i \leq k$.

Remark: In the above definition, the centers are points of P. In some cases (e.g., when points are from Euclidean spaces), centers outside P may be allowed.

For a given input point set P, a center-based clustering problem aims at finding a k-clustering of P which optimizes a certain objective function. The three most popular objectives are:

- **k-center clustering**: Aims at minimizing the maximum distance of any point from the center of its cluster.
- **k-means clustering**: Aims at minimizing the sum of the squared distances of the points from the centers of their respective clusters (also known as Sum of Squared Errors (SSE)).

- **k-median clustering**: Aims at minimizing the sum of the distances of the points from the centers of their respective clusters.

Observation: For k-means and k-median, minimizing the sum of (squared) distances is equivalent to minimizing the average (squared) distance. In Figure 12.7 examples for both objective functions

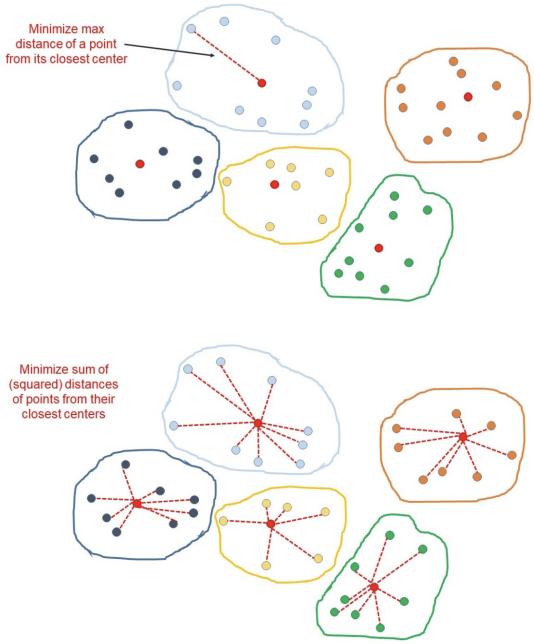


Figure 2.7: Different clustering examples

We want to provide the formal definition of the problem: for any point $x \in P$ and any subset $S \subseteq P$ we define:

$$d(x, S) = min_{y \in S} d(x, y)$$

so it's the distance between the point x and the closest point of S.
Let (M, d) be a metric space. The k-clustering problems are optimization problem that, given a finite pointset $P \subseteq M$ and an integer k, require tor return the subset S of k centers which minimizes the following objective functions:

- For k-center clustering:

$$\Phi_{\text{kcenter}}(P, S) = \max_{x \in P} d(x, S)$$

- For k-means clustering:

$$\Phi_{\text{kmeans}}(P, S) = \sum_{x \in P} (d(x, S))^2$$

- For k-median clustering:

$$\Phi_{\text{kmedian}}(P, S) = \sum_{x \in P} d(x, S)$$

For the k-center/means/median clustering problems, we define:

INPUT: Pointset P and integer k.

FEASIBLE SOLUTIONS: All subsets $S \subseteq P$ of size k.

OPTIMAL SOLUTION: S^* (i.e., the feasible solution which minimizes the appropriate objective function).

We also define:

- $\Phi^{opt}_{\text{kcenter}}(P, k) = \Phi_{\text{kcenter}}(P, S^*)$.
- $\Phi^{opt}_{\text{kmeans}}(P, k) = \Phi_{\text{kmeans}}(P, S^*)$.
- $\Phi^{opt}_{\text{kmedian}}(P, k) = \Phi_{\text{kmedian}}(P, S^*)$.

For the above 3 clustering problems, given a set of centers S, the best clustering of P around centers of S is the one that assigns each $x \in P$ to the cluster of the closest center, which can be computed efficiently.
Taking as input P, S, as output we get the assignment of each $x \in P$ to a center $x.center$

Observations

- The k-center/means/median problems are NP hard
- There are several efficient approximation algorithms that in practice return good quality solutions but dealing with large inputs it's a challenge
- k-center useful when we need that every point is close to a center, but it's sensitive to noise. k-means/median provide gurantees on the average distances.

2.2 K-CENTER

2.2.1 FARTHEST-FIRST TRAVERSAL: ALGORITHM

- Popular 2-approximation sequential algorithm
- Simple and, somewhat fast, implementation when the input fits in main memory.
- Powerful primitive for extracting samples for further analyses.

Input Set P of N points from a metric space (M, d), integer $k > 1$.
Output A set S of k centers which is a good solution to the k-center problem on P (i.e., $\Phi_{\text{kcenter}}(P, S)$ "close" to $\Phi^{opt}_{\text{kcenter}}(P, k)$)
In Figure 2.8 there is the algorithm

```
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 S
```

Figure 2.8: FFT algorithm

The algorithm explanation is the following: we start by initializing randomly the first center, then we find the last k-1 centers by taking, for each iteration, the farthest point from the set S.
Observation: The best clustering around the centers of S can be computed by invoking $Assign(P, S)$. In fact, the assignment of each point to the closest center can be easily maintained in every iteration of the for-loop.
In Figure 2.9 an example of this concept applied
An important theorem is provided below:
Theorem: Let S be the set of centers returned by running Farthest-First Traversal on P. Then:

$$\Phi_{\text{kcenter}}(P, S) \leq 2 * \Phi^{opt}_{\text{kcenter}}(P, k)$$

and so the FFT is a 2-approximation algorithm.
The proof is on the notebook.
Observations:

- The k-center objective focuses on worst-case distance of points from their closest centers.

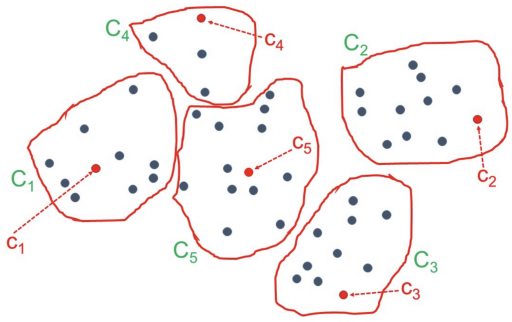


Figure 2.9: Example

- Farthest-First Traversal's approximation guarantees are almost the best one can obtain in practice. It was proved that computing a c-approximate solution to k-center is NP-hard for any fixed c < 2.
- The k-center objective is very sensitive to noise. For noisy datasets (e.g., with outliers), the clustering which optimizes the k-center objective may obfuscate some "natural" clustering inherent in the data. An example in Figure 2.10

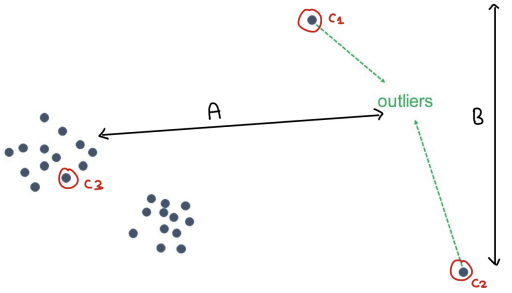


Figure 2.10: Example

The natural clustering would be, with $k = 3$, the two clusters on the left and the two utliers in a single cluster.
However, with A, B very large, the optimal solution is the one provided in the image

Moreover, Farthest-First Traversal requires $k - 1$ scans of the pointset P: impractical for massive P and k not small. How can we compute a "good" solution to k-center for a pointset P which is too large for a single machine? We resort to the **composable coreset technique**.

2.2.2 MapReduce-FFT

Let P be a set of N points (N large!) from a metric space (M, d), and let $k > 1$ be an integer.
In Figure 2.11 the **Algorithm MR FFT**.

Algorithm MR-Farthest-First Traversal

- **Round 1:**
  - Map Phase: Partition P arbitrarily into ℓ subsets of equal size $P_1, P_2, \ldots, P_\ell$.
  - Reduce Phase: for every $i \in [1, \ell]$ separately, run Farthest-First Traversal on P_i to determine a set $T_i \subseteq P_i$ of k centers.
- **Round 2:**
  - Map Phase: empty.
  - Reduce Phase: gather the coreset $T = \cup_{i=1}^{\ell} T_i$ (of size $\ell \cdot k$) and run, using a single reducer, Farthest-First Traversal on T to determine a set $S = \{c_1, c_2, \ldots, c_k\}$ of k centers, and return S as output.

Figure 2.11: MP-FFT algorithm

Theorem: The 2-round MR-FFT algorithm can be implemented using local space $M_L = O(\sqrt{Nk})$ and aggregate space $M_A = O(N)$
In the notebook there is the complete analysis.
Let T be the union of the coresets T_i computed by MR-Farthest-First Traversal on input P. The following lemma establishes the quality of T. Remember that each T_i is the set of centers for partition P_i, find by FFT-MR.
Lemma: For every $x \in P$ we have that:

$$d(x, T) \leq 2 \cdot \Phi^{opt}_{\text{kcenter}}(P, k)$$

where $d(x, T)$ is the distance between a generic point and the point at minimum distance between x of T (set of centers of different partitions).
Observation: This implies that T is a good representative of P w.r.t the k-center in the sense that each $x \in P$ has a "close-by" representative in T.
In the notebook the proof of this lemma.
Theorem: Let S be the set of k centers returned by running MR-Farthest-First Traversal on P. Then:

$$\Phi_{\text{kcenter}}(P, S) \leq 4 \cdot \Phi^{opt}_{\text{kcenter}}(P, k)$$

That is, MR-Farthest-First Traversal is a 4-approximation algorithm.
In the notebook, the proof of this theorem.

Observations on MR-FFT:

- Farthest-First Traversal provides good coresets T_i's, hence a good final coreset T since it ensures that any point not belonging to T is 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

When P has low dimensionality (distance function easier), the quality of the solution returned by MR-FFT can be made arbitrarily close to 2 by selecting a **slightly larger coreset**, while still ensuring sublinear local space and linear aggregate space.
If in each partition P_r we use FFT to extract $k' > k$ centers (selecting more centers than the minimum required (i.e., k' centers when you only need k)) can lead to a finer subdivision of the data in each partition P_r. This means each sub-cluster or group within the partition is smaller and more tightly grouped around its center.

By having more centers than necessary, every point in the partition is likely to be closer to a center because there are more centers available, covering the space more comprehensively. This reduces the maximum distance any point in P has to be from the nearest center in the coreset T. Essentially, more centers allow for a tighter cover of the dataset.
With this modify, the max distance between any point $x \in P$ from the coreset T, now of dimension $k!l$, decreases **sharply** for low-dimensional datasets and the approximation ratio becomes close to 2 with a moderate increase in space and time complexity.
Another important question is: does a random sample provide a good coreset? Let $T \subseteq P$ be a coreset of $|T| = \sqrt{Nk}$ points, selected at random from P independently, with replacement and with uniform probability. Consider a set S of k centers computed by running Farthest-First Traversal on T. Is S a good solution to k-center on P?
The answer is no and on the notebook is shown an extreme case which gives an exhaustive answer to this question.

2.3 k-center as a Coreset Primitive, applications

In MR-FFT, k-center is employed to extract the coreset for solving k-center itself. Now, we will see that it is also useful to extract coresets for other problems as well.

2.3.1 Diameter Problem

Problem: Given a set P of N points from a metric space (M, d) determine its diameter

$$d_{\max} = \max_{x,y \in P} d(x, y),$$

i.e., the maximum distance between two points, as shown in Figure 2.12

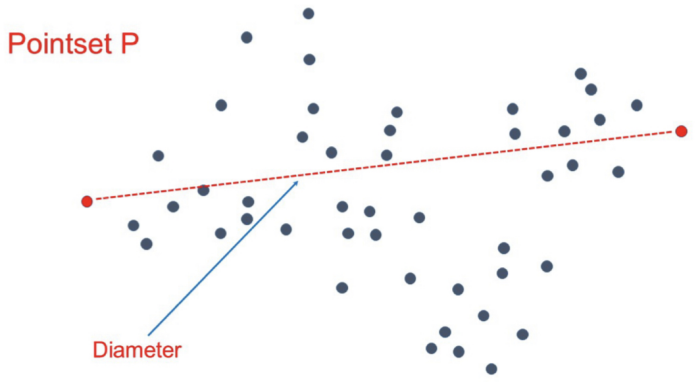


Figure 2.12: Diameter problem

Sadly, the computation of the exact diameter requires almost quadratic operations (except for special cases such as the low-dimensional Euclidean spaces), hence it is impractical for very large pointsets.
2-approximation to the Diameter: For an arbitrary $x_i \in P$ define

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

Lemma: For any $0 \leq i < N$ we have $d_{\max} \in [d_{\max}(i), 2d_{\max}(i)]$. The proof of this lemma is given in the notebook.

Better, coreset-based diameter approximation

Coreset-based approximation:

- Fix a suitable $k \geq 2$.
- Extract a coreset $T \subseteq P$ of size k by running a k-center clustering algorithm on P and taking the k cluster centers as set T.
- Return $d_T = \max_{x,y \in T} d(x,y)$ as an approximation of $d_{\max}$.

Can the approximation be computed efficiently?

If $k = O(1)$, d_T can be computed

- Sequentially: in $O(N)$ time (using Farthest-First Traversal)
- In MapReduce: in 2 rounds, with local space $M_L = O(\sqrt{N})$ and aggregate space $M_A = O(N)$ (through MR-Farthest-First Traversal)

Is d_T a good approximation of $d_{\max}$? Remembering that for sure $d_T \leq d_{max}$, being an approximation, in the notebook that is a good approximation.

2.3.2 Maximization of Minimum Distance

Objective: Determine the *most diverse* subset of k (for a given small k).

Formal definition of the problem

Diversity maximization problem: remote clique variant Given a set P of points from a metric space $(\mathcal{M}, d)$ and a positive integer $k < |P|$, return a subset $S \subseteq P$ of k points, which maximizes the diversity function

$$\text{div}(S) = \sum_{x,y \in S} d(x,y).$$

Observations:

- NP-hard problem
- A c-approximation algorithm is known with $c = 2 - \frac{2}{k}$ which, however, requires quadratic time (impractical for large inputs!)
- Other variants: different functions div$(\cdot)$ to maximize (all NP-hard)

Coreset-based approach to diversity maximization

For a given input P, define the value of the optimal solution as:

$$\text{div}^{\text{opt}}(P,k) = \max_{\substack{S \subseteq P, \\ |S|=k}} \text{div}(S).$$

Can we extract a good solution from a small coreset $T \subseteq P$? Yes, as long as T is a good coreset. The quality of a coreset is quantified as follows.

$(1+\varepsilon)$-coreset: Let $\varepsilon \in (0,1)$ be an accuracy parameter. A subset $T \subseteq P$ is an $(1+\varepsilon)$-coreset for the diversity maximization problem on P if

$$\text{div}^{\text{opt}}(T,k) \geq \frac{1}{1+\varepsilon}\text{div}^{\text{opt}}(P,k).$$

Obs.: since we are solving a maximization problem, the smaller the ε the better the optimal solution in T approximates the optimal solution in P.

The following fact, establishes an interesting relation between the k-center and the diversity maximization problems.

Fact: Given a set P, for any k, the optimal value of the k-center objective, denoted $\phi^{\text{opt}}_{\text{kcenter}}(P,k)$, is at most the optimal value of the diversity objective, denoted $\text{div}^{\text{opt}}(P,k)$, divided by $\binom{k}{2}$:

$$\phi^{\text{opt}}_{\text{kcenter}}(P,k) \leq \frac{\text{div}^{\text{opt}}(P,k)}{\binom{k}{2}}$$

This property, is crucially exploited by the coreset-based approach described (at a high level) in the following slide.

Coreset-based algorithm for maximization problem: Given P and k:

- Run Farthest-First Traversal to extract a set T' of $h > k$ centers from P, for a suitable value $h > k$.
- Consider the h-clustering induced by T' on P and select k arbitrary points from each cluster (or all points in the cluster if they are less than k).
- Gather the at most $h \cdot k$ points selected from the h clusters into a coreset T, and extract the final solution S by running the best sequential algorithm for diversity maximization on T

What is a good choice for h? In the notebook is shown that.

2.4 K-MEANS AND K-MEDIAN

Until now, we have talked only about k-center. However, we've already seen that there are other two objective functions very used: **k-means** and **k-median**. Let us review the two problems: let (M, d) be a metric space. The k-clustering problems are optimization problem that, given a finite pointset $P \subseteq M$ and an integer k, require tor return the subset S of k centers which minimizes the following objective functions:

- For k-means clustering:
$$\Phi_{\text{kmeans}}(P, S) = \sum_{x \in P} (d(x, S))^2$$
- For k-median clustering:
$$\Phi_{\text{kmedian}}(P, S) = \sum_{x \in P} d(x, S)$$

Observation: depending on the application and/or the algorithm used, the requirement that S be a subset of P may be lifted, and the centers can be allowed to be arbitrarily points of M.
Now we list current popular algorithm for k-means/k-median:
Lloyd's algorithm:

- **APPLICABILITY.** Used only for k-means when $M = \mathbb{R}^D$, $d(\cdot, \cdot)$ is the standard Euclidean distance (L_2-distance), and centers can be selected outside P.
- **ACCURACY.** If initial centers are selected well (e.g., through k-means++), it usually provides good solutions, but, if not, it may be trapped into local optima.
- **EFFICIENCY.** Efficient implementations are provided by most common software packages, but a limit on the number of iterations is needed in case of very slow convergence.
- **SUITABILITY FOR MASSIVE INPUTS.** Yes if data can be processed by a distributed platform and only few iterations are executed. However, it is not suitable to process data streams.

In Figure 2.13 the pseudocode of this algorithm
It's important to remember that in k-means centers are not part of the input; in fact:

Input: data points $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m\}$; $k \in \mathbb{N}^+$
Output: clustering $C = (C_1, C_2, \ldots, C_k)$ of $\mathcal{X}$; centers $\mu_1, \mu_2, \ldots, \mu_k$ with μ_i center for C_i, $1 \leq i \leq k$;

randomly choose $\mu_1^{(0)}, \ldots, \mu_k^{(0)}$;
for $t \leftarrow 0, 1, 2, \ldots$ **do** `/* until convergence */`
  for $i = 1, \ldots, k$: $C_i \leftarrow \{\mathbf{x} \in \mathcal{X} : i = \arg\min_j d(\mathbf{x}, \mu_j^{(t)})\}$;
  for $i = 1, \ldots, k$: $\mu_i^{(t+1)} \leftarrow \frac{1}{|C_i|} \sum_{\mathbf{x} \in C_i} \mathbf{x}$;
  if *convergence reached* **then**
    return $C = (C_1, \ldots, C_k)$ and $\mu_1^{(t+1)}, \mu_2^{(t+1)}, \ldots, \mu_k^{(t+1)}$

Figure 2.13: Lloyd's algorithm

Proposition: Given a cluster C_i, the center μ_i that minimizes the function $\sum_{\vec{x} \in C_i} d(\vec{x}, \mu_i)^2$ is

$$\mu_i = \frac{1}{C_i} \sum_{\vec{x} \in C_i} \vec{x}$$

K-means ++ algorithm:

- **APPLICABILITY.** It can be used for both k-means and k-median, and enforces $S \subset P$.
- **ACCURACY.** It provides decent solutions, but it is often useful to refine them to get better accuracy.
- **EFFICIENCY.** Very easy and efficient implementation.
- **SUITABILITY FOR MASSIVE INPUTS.** Yes if data can be processed by a distributed platform and k is small. However, it is not suitable to process data streams.

In Figure 2.14 the algorithm for k-means ++

$c_1 \leftarrow$ random point chosen from P with uniform probability;
$S \leftarrow \{c_1\}$;
for $2 \leq i \leq k$ **do**
  foreach $x \in P - S$ **do** $\pi(x) \leftarrow (d(x, S))^2 / \sum_{y \in P - S} (d(y, S))^2$;
  $c_i \leftarrow$ random point in $P - S$ according to distribution $\pi(\cdot)$;
  $S \leftarrow S \cup \{c_i\}$
return S

Figure 2.14: k-means ++

Partitioning Around Medoids (PAM) algorithm (a.k.a. k-medoids) It is based

on a local search strategy which starts from an arbitrary solution S and progressively improves it by performing the best swap between a point in S and a point in $P - S$, until no improving swap exists.

- **APPLICABILITY.** Mainly used for k-median, and enforces $S \subset P$.
- **ACCURACY.** It provides good solutions.
- **EFFICIENCY.** Very slow since each iteration requires checking about $N \cdot k$ possible swaps and the convergence can be very slow
- **SUITABILITY FOR MASSIVE INPUTS**: Not at all!

2.4.1 Coreset-based approach for k-means/median

The **composable coreset technique** can be employed to devise k-means/median algorithms which are able: (i) to handle massive data (in a distributed setting); and (ii) to provide good accuracy.

- Each point $x \in T_i$ is given a weight $w(x)$ = the number of points in P for which x is the closest representative in T_i.
- The local coresets T_i's are computed using a sequential algorithm for k-means/median.
- The final solution S is also computed using a sequential algorithm for k-means/median, adapted to handle weights.

Why weights are needed?: In the coreset technique, the final solution S is computed on a small coreset.
In order for the technique to work successfully, we need to ensure that S is a good solution for P and, to this purpose, we need that:

$$\Phi(T, S) \approx \Phi(P, S)$$

where Φ is the objective function.
Suppose that each $x \in P$ has a "very close" representative in T (there is a point in T that is very close for each point in P). So we can say that:
k-center: $\Phi_{kcenter}(T, S) \approx \Phi_{kcenter}(P, S)$ and we have already shown that before.
Can we say the same thing for k-median and k-means? So that

$$\Phi_{k-means/median}(T, S) \approx \Phi_{k-means/median}(P, S)$$

In general, the two values may be very different if we have no information about the mass of points of $P - T$ (no coreset points) which every $y \in T$ represents. But if we use weights the scenario changes, as shown in Figure 2.15

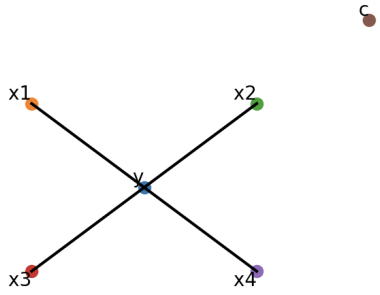


Figure 2.15: Example with weights

where each x_i is a point in $P - T$, $y \in T$ and c is the selected center closest to y. In this case $w(y) = 5$ since y represents itself and the x_i's. So, if c is a center in the final solution, we can say that:

$$d(y,c) + \sum_{i=1}^{4} d(x_i,c) \approx w(y) \cdot d(y,c)$$

where the second term is the contribution of $(y, w(y))$ to the objective function $\Phi_{k-means/k-median}(T,S)$ and it's a good approximation of the first term, that is the contribution of $(y, x_1, ..., x_4)$ to $\Phi_{k-means/k-median}(P,S)$.
From now we will focus only to k-means, with similar result for k-median with some adaptations.

2.4.2 Weighted variant of k-means clustering

Input:

- Set P of N points from $\mathbb{R}^D$.
- Integer weight $w(x) > 0$ for every $x \in P$.
- Target number k of clusters.

Output: Set S of k centers in $\mathbb{R}^D$ **minimizing**

$$\Phi^{w}_{kmeans}(P,S) = \sum_{x \in P} w(x) \cdot (d(x,S))^2.$$

Observations:

- This formulation allows centers outside P.
- If $w(x) = 1$ for every $x \in P$ we have the standard k-means clustering problem.

The adaptation of Lloyd's algorithm is quite simple:

- $\Phi_{k-means}(P, S) \hookrightarrow \Phi^{w}_{k-means}(P, S)$
- Centroid for cluster $C = \{x_1, x_2, x_3, ..., x_n\}$:

$$\frac{1}{t}\sum_{i=1}^{t} x_i \hookrightarrow \frac{1}{\sum_{i=1}^{t} w(x_i)}\sum_{i=1}^{t} w(x_i) \cdot x_i$$

Instead, for k-means++, it's sufficient to change the probability distribution $\pi(\cdot)$ as follows:

$$\pi(x) = \frac{d(x, S)^2}{\sum_{y \in P-S} d(y, S)^2} \hookrightarrow \frac{w(x) \cdot d(x, S)^2}{\sum_{y \in P-S} w(y) \cdot d(y, S)^2}$$

For PAM algorithm (k-median) modifications to account for weights are alse very simple

2.4.3 Coreset-based MapReduce algorithm for k-means

MR-kmeans(A): MapReduce algorithm for k-means which uses a sequential algorithm A for k-means, as an argument (functional approach!), such that:

- $\mathcal{A}$ solves the more general weighted variant.
- $\mathcal{A}$ requires space proportional to the input size.

Input: Set P of N points in $\mathbb{R}^D$, integer $k > 1$, sequential k-means algorithm $\mathcal{A}$.

Output: Set S of k centers in $\mathbb{R}^D$ which is a good solution to the k-means problem on P.

In Figure 2.16 the **MR-kmeans algorithm**

An example is given in Figure 2.16: in the first round is computed the set $T_i = \{a, b, c, d\}$ with given $w(a) = 10, w(b) = 6, w(c) = 4, w(d) = 5$ and in case a, b, c, d do not belong to P, they should not contribute to the weights. **Analysis of the method**: Assume $k \leq \sqrt{N}$. By setting $\ell = \sqrt{\frac{N}{k}}$, it is easy to see that MR-kmeans($\mathcal{A}$) requires

Round 1:

- Map Phase: Partition P arbitrarily in ℓ subsets of equal size $P_1, P_2, \ldots, P_\ell$.
- Reduce Phase: for every $i \in [1, \ell]$ separately, run $\mathcal{A}$ on P_i (with unit weights) to determine a set $T_i \subseteq P_i$ of k centers, and define
  - For each $x \in P_i$, the proxy $\tau(x)$ as x's closest center in T_i.
  - For each $y \in T_i$, the weight $w(y)$ as the number of points of P_i whose proxy is y.

Round 2:

- Map Phase: empty.
- Reduce Phase: gather the coreset $T = \cup_{i=1}^{\ell} T_i$ of $\ell \cdot k$ points, together with their weights, and run, using a single reducer, $\mathcal{A}$ on T (with the given weights) to determine a set $S = \{c_1, c_2, \ldots, c_k\}$ of k centers, which is then returned as output.

Figure 2.16: MR-kmeans(A)

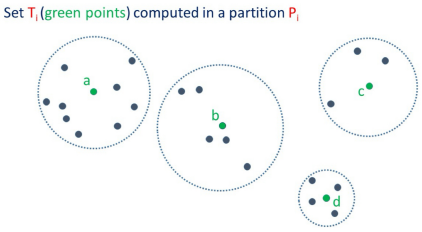


Figure 2.17: Example

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

That is the same analysis that we did for MR-FFT assuming that each P_i has size $O\frac{N}{l}$ and that A requires linear space in its input size.

Does MR-kmeans(A) provide accurate solutions?

Assume that A is an α-approximation algorithm for weighted k-means, for some $\alpha > 1$.

Theorem: Let S be the set of k centers returned by MR-kmeans(A) on input P. Then:

$$\Phi_{\text{kmeans}}(P, S) = O(\alpha^2) \cdot \Phi_{\text{opt kmeans}}(P, k)$$

.

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

Why is this result useful?:

- It's useful because MR-kmeans enables the processing of very large datasets that cannot fit into main memory

- If $\mathcal{A}$ is very accurate but slow, then MR-kmeans allows us to retain (in part) its accuracy, confining its execution to small subsets, hence attaining higher efficiency

The above theorem is proved by combining the following two lemmas. **Lemma 1: coreset quality**: Let $T = \bigcup_{i=1}^{l} T_i \subseteq P$ be the coreset computed by MR-kmeans(A) and let $\tau : P \rightarrow T$ be the **associated proxy function**.
Then, T is an α-coreset for P, k and the k-means objective, that is:

$$\sum_{p \in P} (d(p, \tau(p)))^2 \leq \alpha \cdot \Phi^{opt}_{\text{kmeans}}(P, k)$$

where α is the approximation ratio for A. This leamma is quantifying the quality of T w.r.t the objective function of k-means, in terms of α. **Lemma 2: final solution quality** Let $T \subseteq P$, with associated proxy function $\tau : P \rightarrow T$, be an α-coreset for P, k and the k-means objective. Suppose that the points of T are weighted according to the proxy function τ.
Then, the solution S computed by A on the weighted coreset T, is an $O(\alpha^2)$-approximate solution to k-means for P, that is

$$\Phi_{\text{kmeans}}(P, S) = O(\alpha^2) \cdot \Phi^{opt}_{\text{kmeans}}(P, k)$$

Observations on MR-kmeans(A):

- For k-means (as well as for k-center and k-median) good coresets are obtained by combining solutions to the same problem on smaller partitions. However, this is not always the case for other problems (e.g., diameter, diversity maximization).
- 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 accurate.
- If $k' > k$ centers are selected from each P_i in R1 (e.g., through k-means++), the quality of T, hence the quality of the final clustering, improves.

3

Streaming

The **typical scenario** is that data to be processed arrive as a *continuous stream* because they are generated by some **evolving process** or because their *massive volume* discourages random access.
Therefore, data analysis needs to be performed in real-time using limited memory resources, without storing all the data for later offline processing.
For better visualizing the task, one possible application is the one related to **IoT** where there is a stream of data generated by thousands of sensors and the task is to learn from this data.
In Figure 3.1 there is the **streaming model**: a sequential machine with limited amount of working memory and an input provided as a **continuous one way stream**.

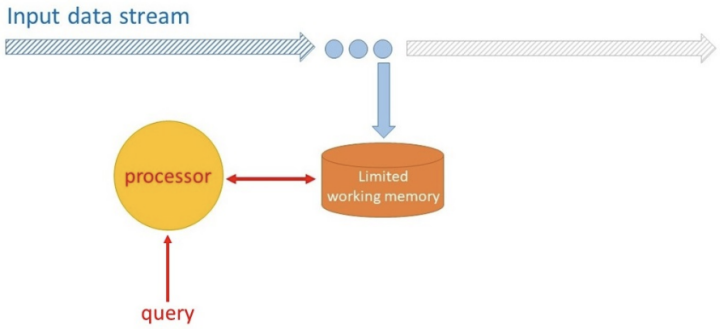


Figure 3.1: Streaming model

Data may arrive very fast, hence they must be processed very fast.
Let $\Sigma = x_1, x_2, ..., x_n$ denote the **input stream** received sequentially where x_i is

the i-th element received. Upon receiving x_i, the streaming model performs two tasks:

1. (**Update Task**): Whenever a new data item x_i is received, the data structures in memory are updated to incorporate this new information. The choice of data structure depends on the specific requirements and constraints of the application.
   The choice of data structure is critical because it can significantly impact the performance of an application, especially in how quickly it can process and retrieve data.
2. (**Query Task**): After updating the data structures with x_i, if a query or computation is required, it is performed based on all the data items received up to that point (x_1 through x_i). The purpose of this task is to derive some result or output based on the accumulated data.

We can consider three **metrics** for giving a numeric performance to the stream:

- **Metric 1**: Size s of the working memory , with the aim of $s <<< n$
- **Metric 2**: Number p of sequential passes (scan of the data) over Σ. This metric indicates how many times the input stream is processed or passed through. Ideally, this number should be 1, meaning the stream is processed in a single pass without needing to revisit or reprocess the data.
- **Metric 3**: Processing time per item T with the aim is $T = O(1)$

As **algorithm design technique** we want

- **Approximate solutions**: when dealing with large data streams, it is often impractical to seek exact solutions due to the linear space they may require. Instead, approximate solutions that utilize significantly less space are preferred.
- **Synopsis Data Structures**: To maintain a summary of the data stream Σ efficiently, synopsis data structures are employed. These structures provide a lossy summary (a summary of information with some losses), meaning that they capture essential information with some permissible loss of detail to reduce memory usage. Synopsis data structures could be *Random Sample, Hash based skecthes*.

Typical data analysis tasks are identification of frequent items or patterns, extracting useful statistics or optimizing as clustering. The **goal** is a suitable tradeoffs between accuracy, working memory, and processing time per item (i.e., throughput).

Examples:

- In $\Sigma = A, B, A, C, A, D, A, A$, the majority element is A.
- In $\Sigma = A, B, C, D, D, E, E$, a majority element does not exist.

Figure 3.2: Example

3.1 Warm-up: finding the majority element in a stream

The **majority problem** is, given a stream, to identify the element x that occurs $> \frac{n}{2}$ times in Σ, if any. In Figure 3.2 an example

Standard off-line setting (So when all data are collected): Easily solved using *linear space* in $0(n \log n)$ time using sorting, or $O(n)$ time through hashing.

How do we solve it if we don't have the full data available?

Streaming setting: We need 1 or 2 passes approach, depending on goals:

1. First pass (**Boyer-Moore algorithm**): find an element x which is the possible majority element, if one exists
2. Second pass: check whether x is the majority element

In Figure 3.3 the **Boyer-Moore algorithm**

The Boyer-Moore algorithm maintains 2 integers *cand* and *count*:

- *cand* contains a candidate majority element (*the true majority element, if one exists*);
- *count* is a counter;

Initialization: $cand \leftarrow$ null and $count \leftarrow 0$.

For each x_t in Σ **do**
  if $count = 0$ **then** $\{cand \leftarrow x_t;\ count \leftarrow 1;\}$
  else {
    if $cand = x_t$ **then** $count \leftarrow count + 1$
    else $count \leftarrow count - 1$
  }

At the end: return *cand*

Figure 3.3: Boyer-Moore Algorithm

Theorem: Given a stream Σ which contains a majority element m, the Boyer-Moore algorithm returns m using:

- working memory of size $s = O(1)$

- 1 pass
- $O(1)$ time per element

Observation: If we don't know whether a majority exists, we need second pass to ascertain whether the returned element occurs $\geq \frac{|\Sigma|}{2} + 1$ times.
Here the straightforward proof for the metrics and, in the notebook, the correctness of this theorem.

- **Working Memory** $O(1)$:
  - The algorithm uses a constant amount of additional memory, specifically a few variables to keep track of the current candidate and its count. This is independent of the input size.

- **1 Pass**:
  - The algorithm processes the input array in a single pass. It iterates through the array exactly once to determine the majority candidate.

- $O(1)$ **Time per Element**:
  - Each element is processed in constant time. The algorithm performs simple comparison and increment/decrement operations for each element.

In the notebook there is also an example of application of Boyer-Moore algorithm.

3.2 Sampling

Definition: Given a set X of n elements and an integer $1 \leq m < n$ an $m - sample$ of X is a random subset $S \subset X$ of size m, such that for each $x \in X$, we have $P[x \in S] = \frac{m}{n}$ (**uniform sampling**).
In the **offline setting**, where the entire set X is known in advance, an $m - sample$ S of X can be easily computed by selecting m elements from X with uniform probability (since we already have all data)
However, in **streaming setting**, things get harder, especially in application scenarios where streams are potentially unbounded and at every time step a random sample of the elements seen so far is required. Elements arrive one at a time, and the total number of elements n is not known in advance—or it could be

potentially unbounded.
In this setting, X represent the prefix of Σ of dimension n.
Sampling problem: Given a (possibly unbounded) stream $\Sigma = x_1, x_2, ...$ and an integer $m < |\Sigma|$, maintain, for every $t \geq m$, an $m - sample$ S of the prefix $\Sigma_t = x_1, x_2, ..., x_t$.
We write $t \geq m$ due to the fact we need to have enough elements.
For a fixed t, the solution is easy: before the stream starts, compute an $m-sample$ I of the set of integers $\{1, ..., t\}$ and, for each entry $x_i \in \Sigma_t$ with $i \in I$, we add x_i to S. But how we can maintain S for every t?
In Figure 3.4 the **Reservoir Sampling algorithm**

Initialization: $S \leftarrow \emptyset$.

For each x_t in Σ **do**
  if $t \leq m$ **then** add x_t to S;
  else with probability m/t do the following: {
    evict a random element x from S
    add x_t to S
  }

Figure 3.4: Reservoir Sampling Algorithm

Reservoir Sampling is an effective technique for maintaining a random subset of a streaming dataset. It adjusts the sample dynamically as new data arrives, ensuring that each element has an equal probability of being included in the sample at any point.
Algorithm Description

- **Initialize:** Start with the first m elements of the stream as your initial sample S.
- **For each new element x_t at position $t > m$:**
  - Generate an index $\in [1, t]$, so with probability $\frac{m}{t}$ we take an element of the m-sample
  - If we get the index of the m-sample, we replace it with x_t, otherwise nothing.

The algorithm ensures that every element in the stream has an equal chance of being included in the reservoir.

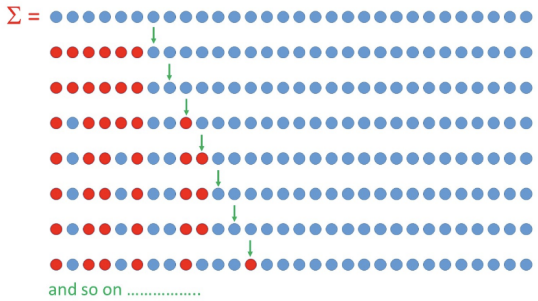


Figure 3.5: Example

In Figure 3.5 an example of the process
Theorem: Let $\Sigma = x_1, x_2, ...$ For any time $t \geq m$, the set S maintained by the reservoir sampling algorithm is an $m - sample$ of $Sigma_t = x_1, x_2, ..., x_t$.
In the notebook there is the proof of this theorem.

3.2.1 Frequent Items Problem

An important problem with stream is the **frequent items problem**: given a stream $\Sigma = x_1, x_2, ..., x_n$ of n items and a frequency threshold $\varphi \in (0, 1)$, determine all distinct items that occur at least $\varphi \cdot n$ times in Σ (we call it **frequent items**).
Suppose a stream Σ with $n = 100$ items and $\varphi = 0.3$: supposing 30 occurrences of A, 30 occurrences of B, 20 occurrences of C, 10 occurrences of D and 10 occurrences of E, **frequent items** are $\{A, B\}$.
Observations

- There are at most $\frac{1}{\varphi}$ frequent items. Let k be the number of frequent items. Each frequent item must appear in at least $\varphi \times N$ transactions.

  To find the maximum number of frequent items, we need to ensure that the sum of the supports of all frequent items does not exceed the total number of transactions N.

  $$k \times (\varphi \times N) \leq N$$

  Dividing both sides by N:

  $$k \times \varphi \leq 1$$

  Solving for k:

  $$k \leq \frac{1}{\varphi}$$

Therefore, the number of frequent items k must be at most $\frac{1}{\varphi}$.

- Any **EXACT STRATEGY** would be potentially too expensive in terms of space. Why? Since it would need to store a counter for each distinct item and the number of distinct items could be proportional to n in the worst case.

Frequent Items Through Reservoir Sampling: Frequent items in a data stream can be approximated by:

1. Extracting an m-sample S from Σ with reservoir sampling. It is important to note that the same element may occur multiple times in the sample.
2. Returning the subset $S' \subseteq S$ consisting of distinct items in the sample.

How well does S' approximate the set of frequent items?

- If $m \geq \frac{1}{\varphi}$, for any given frequent item a, we expect to find at least one copy of a in S (hence in S').
  A frequent item a is an item that appear more than $\varphi \cdot m$ times in the m-sample S computed using Reservoir Sampling. So if $m \geq \frac{1}{\varphi}$ we have that in expectation the number of times of a in S is $m \cdot \varphi \geq \frac{1}{\varphi} \cdot \frac{1}{\varphi} = 1$
- However, some frequent items may be missed, and S' might contain items with very low frequency.

ϵ-Approximate Frequent Items (ϵ-AFI) Problem Given a stream $\Sigma = x_1, x_2, \ldots, x_n$ of n items, a frequency threshold $\varphi \in (0, 1)$, and an accuracy parameter $\epsilon \in (0, \varphi)$, we aim to return a set of distinct items that:

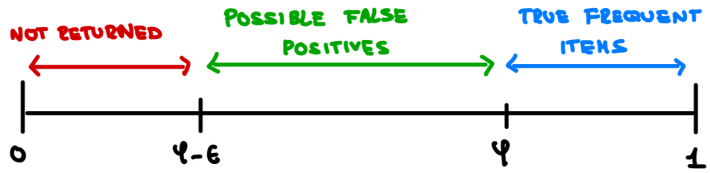
- Includes all items occurring at least $\varphi \cdot n$ times in Σ.
- Contains no item occurring less than $(\varphi - \epsilon) \cdot n$ times in Σ.

Suppose a stream Σ with $n = 100$ items and $\varphi = 0.3$: supposing 30 occurrences of A, 30 occurrences of B, 20 occurrences of C, 10 occurrences of D and 10 occurrences of E, **ϵ-frequent items** legal outputs are $\{A, B\}$,$\{A, B, C\}$. D, E cannot be included in the output because they are to infrequent.

Remark: With this formulation, we avoid false negatives but tolerate high-frequency false positives.

Sticky Sampling for the ϵ-AFI Problem Sticky Sampling provides a probabilistic solution to the ϵ-AFI problem with the following main ingredients:

- A confidence parameter $\delta \in (0, 1)$.
- A hash table S, with entries being pairs $(x, f_e(x))$ where:



  - x (the key) is an item.
  - $f_e(x)$ (the value) is a lower bound on the number of occurrences of x seen so far.

- A **sampling rate** chosen to ensure that, with probability at least $1 - \delta$, the frequent items are in the sample.

In Figure 3.6 the **Sticky Sampling algorithm**:

Consider stream $\Sigma = x_1, x_2, \ldots, x_n$, and assume n known!

Initialization:

- $S \leftarrow$ empty Hash Table);
- $r = \ln(1/(\delta\varphi))/\epsilon$; // *sampling rate* $= r/n$.

For each x_t in Σ **do**

if $(x_t, f_e(x_t)) \in S$ **then** $f_e(x_t) \leftarrow f_e(x_t) + 1$;
else add $(x_t, 1)$ to S with probability r/n; /* (*start tracking* x_t) */

At the end: return all items x in S with $f_e(x) \geq (\varphi - \epsilon)n$

Figure 3.6: Stickly Sampling Algorithm

The sampling rate r in the Sticky Sampling algorithm is given by:

$$r = \frac{\ln(1/(\delta\varphi))}{\epsilon}$$

The actual sampling rate is:

$$\text{sampling rate} = \frac{r}{n} = \frac{\ln(1/(\delta\varphi))}{\epsilon n}$$

If we talk about **accuracy**, increasing the sampling rate is a good thing. However, as we will see in the theorem below, increasing the sampling rate means an increase of **working memory size in expectation**.
Theorem: Stickly sampling solves the ϵ-AFI problem correctly with probability at least of $1 - \delta$ and requires:

- working memory of size $O(\frac{\ln(1/(\delta\varphi))}{\epsilon})$ in expectation
- 1 pass
- $O(1)$ expected processing time per element

Observations: ϵ and δ regulate the accuracy and confidence and the smaller they are, the better accuracy and confidence are, at the expense of the space. Query time (to compute the actual items to return) is $O(|S|)$ as opposed to the processing time per element, which is $O(1)$ in expectation.
Stickly Sampling algorithm with unknown n: Suppose that n is not known by the algorithm or Σ is unbounded.
The **goal** is, after processing $x_1, ..., x_n$, to be able to derive a solution to the ϵ-AFI problem for $x_1, ..., x_n$ for every $n \geq 1$ The idea is shown in Figure 3.7

IDEA. Maintain sampling rate always $\geq r/n$, with

$$r = \ln(1/(\delta\varphi))/\epsilon,$$

as follows.

- Subdivide the stream Σ into batches $B_0, B_1, B_2, \ldots$ of geometrically growing size: namely: $|B_0| = 2r$ and $|B_i| = 2^i r$, for every $i \geq 1$.
- Sample each batch B_i with geometrically decreasing rate $1/2^i$.
- Recalibrate S at the beginning of each batch.

Figure 3.7: Stickly Sampling Idea for unknown n

Instead, in Figure 3.8, the **Stickly Sampling algorithm** with unknown n The second for each means the **recalibration**. Recalibration needed to decrease the sampling probability of the items in S to $\frac{1}{2^i}$.
Observations on Sticky Sampling with unknown n: It can be shown that:

1. The expected size of the working memory is $\leq 2r$ where $r = \ln(1/(\delta\varphi))/\epsilon$
2. After processing the first n items in the stream $x_1, \ldots, x_n$, the items in S with count $> (\varphi - \epsilon)/n$ is a set of ϵ-AFI problem for $x_1, \ldots, x_n$ with probability $\geq 1 - \delta$
3. The current sampling rate is in $[r/n, 2r/n]$

Initialization:

- $S \leftarrow$ empty Hash Table);
- $r = \ln(1/(\delta\varphi))/\epsilon$;

For each $x_t \in \Sigma$ **do**

Let B_i the batch of x_t;
if x_t is the first element of B_i **then** {
  For each $(x, f_e(x)) \in S$ **do**
    Let τ = number of tails before head of an unbiased coin.
    if $f_e(x) - \tau > 0$ **then** $f_e(x) = f_e(x) - \tau$
    else delete $(x, f_e(x))$ from S
}
if $(x_t, f_e(x_t)) \in S$ **then** $f_e(x_t) \leftarrow f_e(x_t) + 1$;
else add $(x_t, 1)$ to S with probability $1/2^i$;

Figure 3.8: Stickly Sampling Algorithm for unknown n

3.3 SKETCHING

Before starting, it's important to define what is a **sketch**: space-efficient **data structure** that can be used to provide **(typically probabilistic)** estimates of (statistical) characteristics of a data stream.
Consider a stream $\Sigma = x_1, x_2, \ldots, x_n$, whose elements belong to a universe **U**. (In fact, Σ could represent an arbitrary prefix of a longer, possibly unbounded, stream.)

For each $\mathbf{u} \in \mathbf{U}$ occurring in Σ define its **(absolute) frequency**

$$f_u = \left|\{j : x_j = \mathbf{u}, 1 \leq j \leq n\}\right|,$$

i.e., the number of occurrences of **u** in Σ.

Definition: Frequency Moments

For every $\mathbf{k} \geq 0$, the **k**-th frequency moment F_k of Σ is

$$F_k = \sum_{\mathbf{u} \in \mathbf{U}} f_u^{\mathbf{k}}.$$

(We assume $0^0 = 0$)

So we could extract **relevant information from frequency moments**:

- F_0 is the **number of distinct items in** Σ.

- $F_1 = |\Sigma|$, trivial to maintain with a counter
- $1 - \frac{F_2}{|\Sigma|^2}$ is the **Gini index of** Σ. It provides information on **data skew** (skewness refers to the asymmetry in the distribution of data values): the closer to 0, the higher is the skew.

Observations on the Gini index:

- If $x_1 = x_2 = \ldots = x_n$

$$\Rightarrow F_2 = n^2 = |\Sigma|^2$$
$$\Rightarrow \text{GINI INDEX} = 1 - \frac{F_2}{|\Sigma|^2} = 0 \quad \text{(MAXIMUM SKEW)}$$

- If Σ contains m distinct elements, each occurring n/m times

$$\Rightarrow F_2 = m \cdot \left(\frac{n}{m}\right)^2 = \frac{n^2}{m}$$
$$\Rightarrow \text{GINI INDEX} = 1 - \frac{\left(\frac{n^2}{m}\right)}{n^2} = 1 - \frac{1}{m} \quad \text{(MAXIMUM BALANCE)}$$

3.3.1 Estimating F_0 for Σ (i.e. distinct elements)

The exact computation use $|U|$ counters or a dictionary with $|F_0|$ elements and it's not suitable for streaming settings.
So, as we already seen for other problems, we could leverage on **approximation algorithm**, that, in our case, is the **probabilistic counting algorithm** which uses $O(\log |U|)$ working memory and F_0 estimated within a factor c, with probability $\geq 1 - 2/c$ **(accuracy-confidence tradeoff)**.
Main idea:

- Map each $u \in U$ to a random integer $h(u) \in [0, |U| - 1]$. $\Rightarrow h(u)$ is a $O(\log |U|)$-bit binary string. So, for example, if the universe is the alphabet, we need 5 bit to represent all possible instances in binary strings.
- The more distinct elements in Σ, the more likely to have a $u \in \Sigma$ mapped to a string with many trailing 0's (zero in the tail of the binary string). This is why, if we have distinct elements, we have more different binary string, so more probability.

We need the following ingredients:

- Array C of $\lceil \log_2 |U| \rceil + 1$ bits, all initialized to zero.
- Hash Function $h : U \rightarrow [0..|U| - 1]$. We assume:

- For every $u \in U$, $h(u)$ has uniform distribution in $[0..|U|-1]$
- All $h(u)$'s are pairwise independent.

Notation: for $i \in [0..|U|-1]$, define

$$tr(i) = \text{number of trailing zeroes in binary representation of } i$$

e.g., $12 = (1100)_2 \Rightarrow tr(12) = 2$
In Figure 3.9 the **probabilistic counting algorithm**:

Algorithm: **For each** $x_j \in \Sigma$ **do** $C[tr(h(x_j))] \leftarrow 1$

After processing x_n, estimate F_0 as

$$\tilde{F}_0 = 2^R,$$

where R is the largest index of C with $C[R] = 1$.

Figure 3.9: Probabilistic counting algorithm

So, in other words, we put the cell, corresponding to the number of trailing zeroes of an item, of the array C equal to 1 and we use the largest index (so the higher number of trailing zeroes) to estimate $\tilde{F}_0$.
In the notebook there is an example and the intuition for this algorithm. **Theorem**: For a stream Σ of n elements, the **probabilistic counting algorithm** returns a value $\tilde{F}_0$ such that, for any $c > 2$:

$$\Pr(\tilde{F}_0 < F_0/c) \leq 1/c \quad \text{and} \quad \Pr(\tilde{F}_0 > cF_0) \leq 1/c,$$

hence

$$\Pr(F_0/c \leq \tilde{F}_0 \leq cF_0) \geq 1 - 2/c.$$

The algorithm requires a working memory of $O(\log |U|)$ bits, 1 pass, and $O(\log |U|)$ time per element.
The Theorem provides a confidence interval for $\tilde{F}_0$:

$$\tilde{F}_0 \in \left[\frac{F_0}{c}, cF_0\right]$$

or, equivalently, for F_0:

$$F_0 \in \left[\frac{\tilde{F_0}}{c}, c\tilde{F_0}\right]$$

When c is large, the confidence $1 - 2/c$ is close to 1 but the interval is wider.

For what concerns the other statements, we have that:

- Working memory: $O(\log(|U|))$ bits required to store C (although only the value R could be maintained) and to represent $h(x)$ when computed.
- $O(\log(|U|))$ time per element: required to compute the binary representation of $h(x)$, hence the number of trailing 0's.

3.3.2 Estimating individual frequencies and F_2 for Σ

Now, our **objective** is in one pass, over Σ, to compute a **small sketch** that enables to derive **unbiased estimates** of:

- f_u for any given $u \in U$ (**individual frequencies**)
- $F_2 = \sum_{u \in U}(f_u)$ (**second moment**)

with provable space-accuracy tradeoffs.

It's important to do an observation: the **exact computation** of these things might require space proportional to $|\Sigma|$.

Count-min sketch: The first approach we consider is based on the count-min sketch. Main ingredients are:

- $d \times w$ matrix C of counters ($O(\log n)$ bits each)
- d hash functions: $h_0, h_1, \ldots, h_{d-1}$, with
$$h_j : U \to \{0, 1, \ldots, w-1\},$$
for every j.

Note that d and w are *design parameters* that regulate the space/time-accuracy tradeoff.

Initialization: $C[j,k] = 0$, for every $0 \leq j < d$ and $0 \leq k < w$.

For each x_t **in** Σ **do**

For $0 \leq j \leq d-1$ **do** $C[j, h_j(x_t)] \leftarrow C[j, h_j(x_t)] + 1$;

At the end of the stream: for any $u \in U$, its frequency f_u can be estimated as:

$$\tilde{f_u} = \min_{0 \leq j \leq d-1} C[j, h_j(u)].$$

Initialization: $C[j,k] = 0$, for every $0 \leq j < d$ and $0 \leq k < w$.

For each x_t in Σ **do**

  For $0 \leq j \leq d-1$ **do** $C[j, h_j(x_t)] \leftarrow C[j, h_j(x_t)] + 1$;

At the end of the stream: for any $u \in U$, its frequency f_u can be estimated as:

$$\tilde{f}_u = \min_{0 \leq j \leq d-1} C[j, h_j(u)].$$

Figure 3.10: Count-min sketch algorithm

In Figure 3.10 the **count-min sketch algorithm**:
So, for each item, we add a 1 in the cell corresponding to the row j, with $0 \leq j \leq d-1$, in the column indicated by $h_j(\cdot)$. It's useful, if we see that in the stream there are for example five times an item, to skip the computation of all items and inserting directly $5 \cdot item$.
At the end, we estimate for each element u its $\tilde{f}_u$ that is the minimum value of the cell where the element u has contributed.
In the notebook there is an example.
Count-min sketch analysis: We assume that

1. For each $j \in [0, d-1]$ and each $u, v \in U$ with $u \neq v$, $h_j(u)$ and $h_j(v)$ are independent random variables uniformly distributed in $[0, w-1]$. Uniform distribution means that every possible hash value in the range $[0, w1]$ is equally likely to occur. There is no bias towards any particular value within this range.

2. The d hash functions $h_1, h_2, \ldots, h_d$ are mutually independent.

Theorem: Consider a $d \times w$ count-min sketch for a stream Σ of length n, where $d = \log_2(1/\delta)$ and $w = 2/\epsilon$, for some $\delta, \epsilon \in (0,1)$. The sketch ensures that for any given $u \in U$ occurring in Σ

$$\tilde{f}_u - f_u \leq \epsilon \cdot n,$$

with probability $\geq 1 - \delta$.

Observation: The bias in the estimated frequencies discourages their use to estimate the second moment F_2. However, in which sense this algorithm is **biased**?
A count-min sketch uses multiple hash functions to map elements to a set of counters.

Due to the limited number of counters (cells), different elements might map to the same counter, causing **hash collisions**. When multiple elements map to the same counter, the counter value accumulates the frequencies of all those elements, not just the target element.
Because the counters may include contributions from other elements due to collisions, the estimated frequency is typically higher than the true frequency. This overestimation is a systematic error, or bias.
In the notebook there is the proof for this theorem.
Count sketch: Count sketch **can be seen as an unbiased variant of the count-min sketch.**
The idea is, for each item $u \in U$, to multiply its contribution to each row by a value in $\{-1, +1\}$ randomly collected, so to canel out collisions.
Main ingredients are:

- $d \times w$ array C of counters ($O(\log n)$ bits each)
- d hash functions: $h_0, h_1, \ldots, h_{d-1}$, with
$$h_j : U \to \{0, 1, \ldots, w-1\}$$
for every j.
- **NEW**: d hash functions: $g_0, g_1, \ldots, g_{d-1}$, with
$$g_j : U \to \{-1, +1\}$$
for every j.

In Figure 3.11 the **count-sketch algorithm**:

Initialization: $C[j,k] = 0$, for every $0 \le j < d$ and $0 \le k < w$.

For each x_t in Σ **do**
  For $0 \le j \le d-1$ **do** $C[j, h_j(x_t)] \leftarrow C[j, h_j(x_t)] + g_j(x_t)$;

At the end of the stream: for any $u \in U$ and $0 \le j < d$, let
$$\tilde{f}_{u,j} = g_j(u) \cdot C[j, h_j(u)].$$
The frequency of u can be estimated as:
$$\tilde{f}_u = \text{median of the } \tilde{f}_{u,j}\text{'s}$$

Figure 3.11: Count sketch algorithm

It's equal as before, with the difference that now it's not added directly a 1 but

could be a 1 or a -1.
At the end of the stream, the estimated frequency is the result of a median operation between cells where u compared multiplied by the value of $h(u)$ for that cell.
Count-min sketch analysis: We assume that for both sets of hash functions (the h_j's and the g_j's) independence and uniform distribution, which we made for the h_j's in the analysis of the count-min sketch.
Theorem: Consider a $d \times w$ count sketch for a stream Σ of length n, where $d = \log_2(1/\delta)$ and $w = O\left(\frac{1}{\epsilon^2}\right)$, for some $\delta, \epsilon \in (0, 1)$. The sketch ensures that for any given $u \in U$ occurring in Σ:

(A) $\mathbb{E}[\tilde{f_{u,j}}] = f_u$, for any $j \in [0, d-1]$, i.e., $\tilde{f_{u,j}}$ is an unbiased estimator of f_u;

(B) With probability $\geq 1 - \delta$,

$$|\tilde{f_u} - f_u| \leq \epsilon \cdot \sqrt{F_2},$$

where $F_2 = \sum_{u \in U} (f_u)^2$ (true second moment).

Intuition. Due to the random signs, on average the "noise" created by several items colliding on the same column as u, cancel out.
In the notebook there is the proof for (A) statement.

- **Bias**: count-min sketch provides biased estimates while count sketch provides unbiased ones.
- **Accuracy**: with probability $\geq 1 - \delta$
  - count-min sketch: $|\tilde{f_u} - f_u| \leq \epsilon n \quad (n = |\Sigma|)$
  - count sketch: $|\tilde{f_u} - f_u| \leq \epsilon \sqrt{F_2}$
  - and
$$F_2 = \sum_{u \in U} (f_u)^2 = n \Rightarrow \sqrt{F_2} \leq n$$

  If the distribution of frequencies is not too skewed, $\sqrt{F_2} \ll n$

Estimation of F_2: Given a $d \times w$ count sketch for Σ, define

$$\tilde{F_{2,j}} = \sum_{k=0}^{w-1} (C[j,k])^2 \quad \text{for } 0 \leq j < d$$

We can derive the following estimate for the true second moment F_2:

$$\tilde{F_2} = \text{median of the } \tilde{F_{2,j}}'s$$

An example is shown in the notebook.

Analysis of $\tilde{F}_2$: The following theorem can be proved under the same assumptions made for the analysis of the count sketch.

Theorem:Consider a $d \times w$ count sketch for a stream Σ of length n, where $d = \log_2(1/\delta)$ and $w = O\left(\frac{1}{\epsilon^2}\right)$, for some $\delta, \epsilon \in (0, 1)$. The sketch ensures that:

- $\mathbb{E}[\tilde{F_{2,j}}] = F_2$ for any $0 \leq j < d$. That is, any $\tilde{F_{2,j}}$ is an unbiased estimator of F_2.
- With probability $\geq 1 - \delta$,

$$|\tilde{F_2} - F_2| \leq \epsilon \cdot \sqrt{F_2}.$$

Analysis of performance metrics

Both count-min and count sketches can be computed in **1 pass**.

To assess space and time performance, we assume:

- Each hash function can be applied in constant time.
- The space occupied by the sketch dominates over the one needed to store the hash functions.

For both sketches we have:

- **Working memory:** $O(d \cdot w)$, which becomes $O(\log(1/\delta)/\epsilon)$, for the count-min sketch, and $O(\log(1/\delta)/\epsilon^2)$, for the count sketch, in order to attain the probabilistic accuracy stated before.
- **Processing time per element:** $O(d) = O(\log(1/\delta))$,

Moreover, given the sketch, **the estimates $\tilde{f}_u$'s (individual frequencies) and $\tilde{F}_2$ (second moment) can be computed in $O(d)$ and $O(d \cdot w)$ time, respectively.**

3.4 Filtering

For many applications, processing a data stream $\Sigma = x_1, x_2, \ldots$ essentially involves identifying if the x_i's satisfy a certain criterion.

Some criteria can be checked very easily with a minimum cost in terms of space and time. However, this is not always the case.

Example. Suppose that the x_i's are email addresses and that when x_i arrives we need to check whether it belongs to a set S of verified addresses. If S is very large (e.g., 1 billion addresses of approximately 20 bytes each), we face two issues:

- If S does not fit into main memory, it must be stored on disk.
- Standard exact techniques to check $x_i \in S$, especially if S is on disk, may be time consuming and not compatible with a high arrival rate.

Can we check membership efficiently with reasonable accuracy?

3.4.1 Bloom filter

Approximate membership problem: Given a stream $\Sigma = x_1, x_2, \ldots$ of elements from some universe U, and let S be a set of m elements from U. Store S into a compact data structure that, for any given x_i, allows to check whether $x_i \in S$ with

- no error, when $x_i \in S$ **(No false negatives)**
- small probability error, when $x_i \notin S$ **(Small false positive rate)**

A solution to the problem comes from the **Bloom filter**. Its **main ingredients** are:

- Array A of n bits, all initially 0.
- k hash functions: $h_0, h_1, \ldots, h_{k-1}$, with

$$h_j : U \rightarrow \{0, 1, \ldots, n-1\} \text{ for every } 0 \leq j < k$$

Note that n and k are **design parameters** that regulate the tradeoff between space/time and accuracy. The **Bloom filter algorithm** is shown in Figure 3.12

Initialization:

For each $e \in S$ **do**

For $0 \leq j < k$ **do** $A[h_j(e)] \leftarrow 1$;

Membership test: for any x_i in Σ if

$$x_i \in S \quad \Leftrightarrow \quad A[h_0(x_i)] = A[h_1(x_i)] = \cdots = A[h_{k-1}(x_i)] = 1$$

Figure 3.12: Bloom filter algorithm

Straightforward properties:

- The approach ensures that there are no false negatives.
- Assuming that $k \ll n$, and that the hash functions can be stored compactly, the required working memory is dominated by the storage of $A \Rightarrow n$ bits.

- Assuming that each hash function can be applied in $O(1)$ time, the membership test requires $O(k)$ time.

Bloom filter analysis of false positive rate

Assumptions: for the set of hash functions (the h_j's) we make the same assumptions of independence and uniform distribution, which we made in the analysis of the count-min sketch.

Theorem: Suppose that n is sufficiently large. For any given x_i which does not belong to S, the probability that x_i is erroneously claimed to be in S is

$$\Pr(A[h_j(x_i)] = 1 \text{ for each } 0 \leq j < k) \simeq (1 - e^{-km/n})^k$$

This probability is referred to as **false positive rate**.

Email example: In the case of email addresses mentioned before, $m = 10^9$ and storing the entire set S would require 20GB (assuming that each email takes 20 bytes). Using a Bloom filter with $n = 8m$ (hence $|A| = 1GB$), and $k = 6$, the false positive rate is about 2.15

In the notebook the proof for this theorem.

4

Similarity Search

We'll go through the **introduction to similarity search**.
An useful example is when data are for example **points of interest** in a map and we have a distance function and a threshold. The goal is to find points of interest near my current position (at distance $\leq r$).
This problem is an example of **similarity search**: given a set of objects, find items that are **similar or near**.
Obviously a distance function is needed to evaluate similarity of two objects: *two objects are similar if their distance is below a given threshold*.
r-Near neighbor search
For a given metric space (M, d), point $q \in M$, and distance threshold $r > 0$ define the **ball of radius** r around q as

$$B_r(q) = \{p \in M : d(p, q) \leq r\}$$

Definition: r-Near neighbor search (r-NNS) problem Given a set P of n points from the metric space (M, d), construct a data structure that, given a query point $q \in M$ and a distance threshold $r > 0$, returns:

- a point $p \in B_r(q) \cap P$, if any such point exists;
- `null` if $B_r(q) \cap P = \emptyset$.

So, substantially, we want a of P inside the ball of radius r centered in q.
Observations:

- The query is unknown at the construction of the data structure.

- If there are more points in $B_r(q) \cap P$, an arbitrary one is returned.
- The distance function and the threshold r depend on the application.

An example of r-NNS is given in Figure 4.1

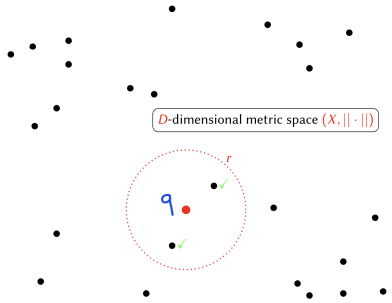


Figure 4.1: Example of r-NNS

There are several other variants of similarity search problems e.g.,:

- **r-near neighbor reporting**: given the set P, a query point q, and a threshold r, return **all points** in $B_r(q) \cap P$.
- **Nearest neighbor search**: given the set P and query point q, find the closest point to q in P. That is, return $\arg\min_{p \in P} d(p, q)$.
  Searching for the nearest point does not require a threshold r; however, it is usually performed by reducing it instances of r-NNS, using a suitable sequence of values of r.
- **k-nearest neighbor search**: given the set P, a query point q, and an integer $k \geq 1$, find the top-k closest points to q in P.
- **Similarity join**: given two sets P and Q and a threshold r, find all similar pairs between the two sets. That is, find all $p \in P$, $q \in Q$ with $d(p, q) \leq r$.

We will compute it in **sequential setting** and we'll use the following **performance indicators**:

- **Construction time**: the time to construct the initial data structure;
- **Space**: the space (in memory words) to store the data structure;
- **Query time**: the time to execute one query.

For points in $\mathbb{R}^D$ we assume that $d(x, y)$ can be computed in $O(D)$ time and that a point is stored in $O(D)$ memory space.

Remark: Similarity search has been addressed in the MapReduce and streaming frameworks.
A **brute force solution** could be the following:

- **Construction**: Store points of P in a list.
- **Query** q: Compute $d(q,p)$ for all $p \in P$ returning the first point $p' \in P$ with $d(q,p') \leq r$, if one exists, and returning `null`, otherwise.

If $P \subset \mathbb{R}^D$ and it contains n points, the brute force approach requires:

- $O(n)$ construction time;
- $O(Dn)$ space;
- $O(Dn)$ query time.

IMPORTANT: *The query time is too high!*

4.1 Similarity search in low dimensions

To solve more efficiently the r-NNS problem in $\mathbb{R}^D$ under Euclidean distance, we use the **kd-tree**.
A kd-tree is a **binary tree** that recursively partition the space into rectangular regions.
These trees are used to solve a slightly different problem named **Range Reporting** but it's useful also for our initial problem.
Definition: Range Reporting (RR) problem $\hookrightarrow$ Given a set $P \subset \mathbb{R}^D$ of n points, construct a data structure that, given a rectangular region R, returns **all points of** P **contained in** R.

Note: a rectangular region $R = [x_{1,1} : x_{1,2}] \times \ldots \times [x_{D,1} : x_{D,2}] \subset \mathbb{R}^D$ is the region including all points $p = (p_1, \ldots, p_D)$ with $p_i \in [x_{i,1} : x_{i,2}]$ for all $i \in \{1, \ldots, D\}$.
An example of Range Reporting in $\mathbf{R}^2$ is given in Figure 4.2

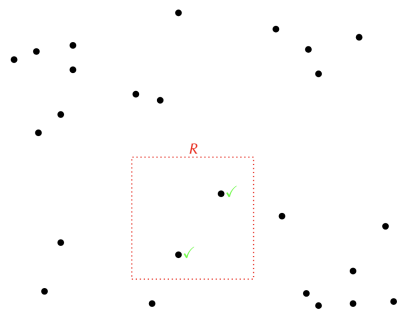


Figure 4.2: Range reporting in R^2

Observations: Range Reporting vs r-NNS: In the case of range reporting the query

region is rectangular and the task to be solved is to return all points in R. Instead, for r-NNS problem, the query region is a circle and the task to be solved is to return a single point contained in a circle of radius r centered around the query point q.

kd-trees data structure: For simplicity we define the kd-tree for the case $D = 2$. The definition can be extended to $D > 2$.

A kd-tree for a set P of n points in $\mathbb{R}^2$ is defined as follows (the definition can be extended to higher values of D):

- Each node v represents a subset $P_v \subseteq P$. In particular, the root represents P and the leaves represent distinct individual points (so there are n leaves).
- Each internal node v represents the set of points $P_v \subseteq P$ that are found in the leaf nodes of its subtree.
  Each internal node is associated with a line l (vertical or horizontal, depending on whether v is respectively at even or odd depth) which splits P_v into 2 subsets $P_{v,1}$ and $P_{v,2}$ of size $\lceil |P_v|/2 \rceil$ and $\lfloor |P_v|/2 \rfloor$, respectively. The left child of v represents $P_{v,1}$ and the right child represents $P_{v,2}$.
  If l is a vertical line at abscissa x_v, then
  - $P_{v,1}$ contains all points of P_v with abscissa $\leq x_v$ (so if a point is on the line, it's considered as "owned" by left child);
  - $P_{v,2}$ contains all points of P_v with abscissa $> x_v$.

  The case when is horizontal is analogous. An example can be seen in Figure 4.3.

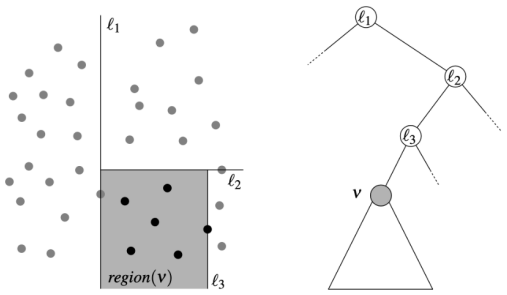


Figure 4.3: kd-tree

As we can see l_1 is a vertical line, while l_2 a horizontal line (justified by their depth).
Each node v represents a rectangular region defined by the vertical/horizontal lines associated to path from the root to v.

- The construction process is recursive and continues until each subset contains exactly one point. When a subset contains only one point, this point is stored in a leaf node.

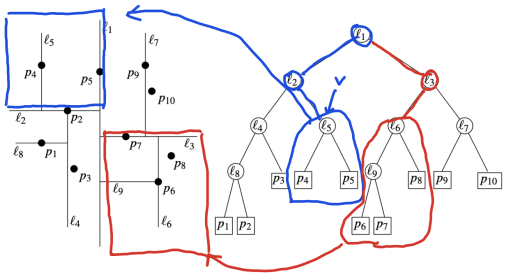


Figure 4.4: Example of kd-tree

A clearer example is the one in Figure 4.4

The main idea for the **kd-tree query algorithm** is the following:

- Given a query region $R = [x_1, x_2] \times [y_1, y_2]$, the query algorithm visits all nodes whose regions intersect R.
- Given an internal node v associated with a vertical line at abscissa x_v (similarly for a horizontal line):
  - If $x_1 \leq x_v$, we recursively search v's left subtree;
  - If $x_2 > x_v$, we recursively search v's right subtree;

  Note that *recursion might proceed on both subtrees if the line is inside the range*.
- When the recursive algorithm reaches a region entirely contained in R, it returns all points in it.

An example is shown in Figure 4.5

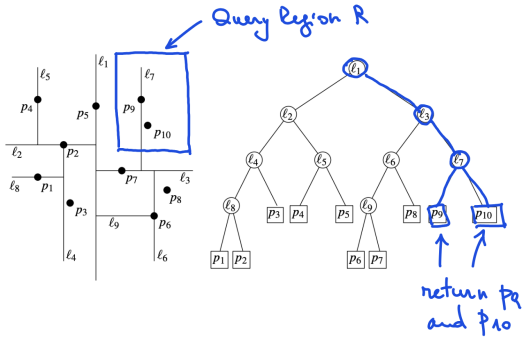


Figure 4.5: Example of the query of algorithm

Focusing more on details of **query algorithm**, consider a kd-tree T for a set of n points P.

Let `ReportAll`(v) be a primitive that given a node $v \in T$ returns all points associated with the leaves of the subtree of v.

It is not difficult to show that `ReportAll`(v) can be implemented in time proportional to the number of returned points (which is also proportional to the number of nodes of the subtree of v).
In Figure 4.6 the **kd-tree query algorithm**

```
SearchKdTree(v, R)      //  first invocation: v = root
Input: A node v in the kd-tree, and a range R
Output: The set of points p ∈ R from the subtree of v

out ← ∅;
if (R_v intersects R) then out ← ReportAll(v);
else
    if (v is internal) then
        Let u, w be the left and right child of v, respectively;
        Let R_u, R_w be regions represented by u and w, respectively;
        if (R_u intersects R) then
            if (R_u ⊆ R) then out ← out ∪ ReportAll(u);
            else out ← out ∪ SearchkdTree(u,R);
        if (R_w intersects R) then
            if (R_w ⊆ R) then out ← out ∪ ReportAll(w);
            else out ← out ∪ SearchkdTree(w,R);
return out
```

Figure 4.6: Query algorithm kd-tree

Theorem-Performance in $\mathbb{R}^2$: Let P be a set of n points in $\mathbb{R}^2$. Then, the Range Reporting problem can be solved using the kd-tree data structure with the following performance:

- **Construction time**: $O(n \log n)$
- **Space**: $O(n)$
- **Query time**: $O(\sqrt{n} + k)$, where k is the number of reported points.

The intuition about query time is the following: consider the execution of `SearchKdTree` on T storing a set P of n points for a query region R.

Let $Q(R)$ be the set of nodes of T on which `SearchKdTree` or `ReportAll` is called.

We have that $Q(R) = Q_1(R) \cup Q_2(R)$, where

- $Q_1(R)$ = nodes whose regions intersect R but are not entirely contained in R;
- $Q_2(R)$ = nodes whose regions are entirely contained in R (hence `ReportAll` is run on each of them);

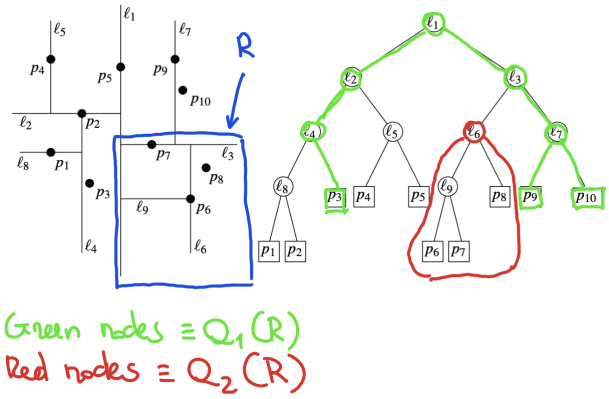


Figure 4.7: Idea for evaluating

In Figure 4.7 the idea.
It can be argued that:

1. $|Q_1(R)| = O(\sqrt{N})$ (green points)
2. The call on each node of $Q_1(R)$ contributes $O(1)$ time to the overall query
3. The calls on all nodes of $Q_2(R)$, or rather red points, contributes aggregately $O(k)$ time to the overall query time.

The above considerations immediately imply that the query time is $O(\sqrt{n}+k)$ However, we start with the require of solving kd-tree for r-NNS: to solve an r-NNS query with center $q \in \mathbb{R}^2$ on the input set P we have to

- Compute the smallest square R_q enclosing the ball of radius r and center q
- Let S be the output of an RR query with range R_q on a kd-tree tree of P
- Return a point $p \in S \cap B_r(q)$, if any, or null if $S \cap B_r(q) = \emptyset$

The solution works efficiently from a practical point of view.

Let k_R be the number of points returned by the R_q query, and let k_q be the actual number of points in the ball with center q and radius r.

The above solution requires $O(\sqrt{n}+k_R)$ not $O(\sqrt{n}+k_q)$; in some pathological cases, $k_R \gg k_q$.

Not clear how to close the gap from a theoretical point of view.

- The kd-tree is defined in a similar fashion, but each level takes care of a distinct coordinate, in cycling order (i.e., 1, 2, ..., D, 1, 2...).
- `SearchKdTree` remains unchanged.

The following theorem generalizes the one for $\mathbb{R}^2$.

Theorem: Let P be a set of n points in $\mathbb{R}^D$. Then, the Range Reporting problem can be solved using the kd-tree data structure with the following performance:

- Construction time: $O(Dn \log n)$
- Space: $O(Dn)$
- Query time: $O(Dn^{1-1/D} + k)$, where k is the number of reported points. (For $D = 2$: $O(\sqrt{n})$)

Curse of dimensionality

- The query cost of kd-tree converges to linear scanning time when D increases.
- There is a strong conjecture that the exact r-NNS problem cannot be solved in (truly) sublinear query time when D is large, i.e., query time $O(n^{\epsilon})$ for a constant $\epsilon < 1$.

4.2 r-NNS in high dimensions

The curse of dimensionality can be tackled resorting to approximate approaches.

How can approximation affect the solution to the r-NNS problem for a query point q and radius r?

- The returned point p might be at distance $> r$ from q (false positive). This can be easily detected and it should be avoided that $d(p, q) >> r$.
- No r-near neighbors are returned while some exists (false negatives). This should be avoided.

Definition: (c, r)-Approximate Near Neighbor Search ((c, r)-ANNS): Given a set P of n points from the metric space (M, d), construct a data structure that, given a query point $q \in M$ and a distance threshold $r > 0$, provides the following answer for a certain constant $c \geq 1$:

- If there are points in $B_r(q) \cap P$, it returns a point $p \in P$ with $d(p, q) \leq cr$;
- If there are no points in $B_r(q) \cap P$ it may either return a point $p \in P$ with $d(p, q) \leq cr$, if one exists, or null.

Remark: In this case, a null answer is always acceptable, even if a point $p \in P$ with $d(p, q) \leq cr$ exists.

Observation: in all cases, the data structure never returns a point far away from the query point q (i.e., at distance $> cr$).
In Figure 4.8 three examples, one with near points, one without near points and one with only far points.

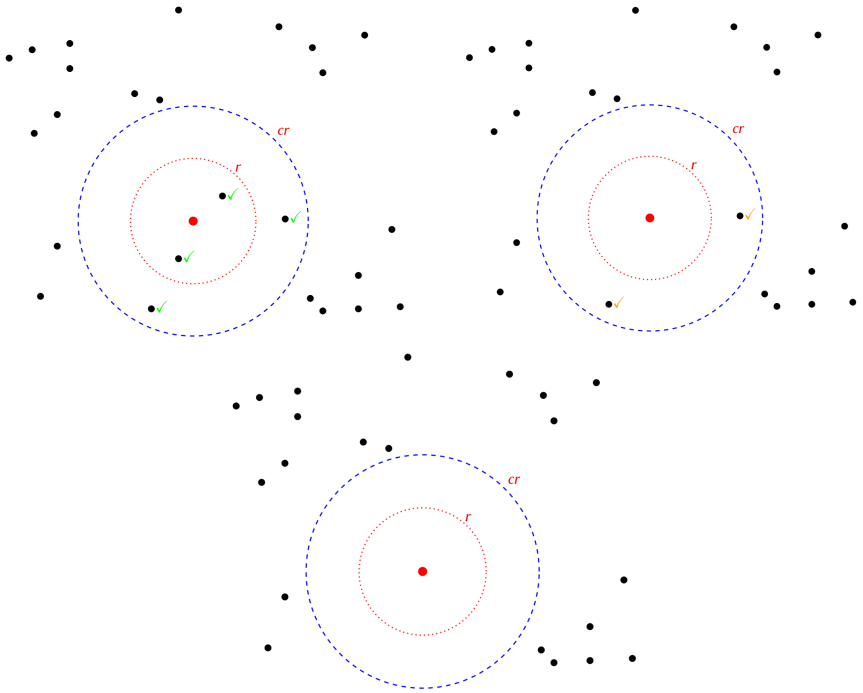


Figure 4.8: Examples

We present a solution for (c, r)-ANN which is based on **Locality Sensitive Hashing (LSH)**, a popular tool for similarity searching in high dimensions.

Main ideas:

- The entire space is partitioned into *regions* through a *hash function* h randomly extracted from a suitable family $\mathcal{H}$.
- The partitioning ensures that: (1) two near points are likely to be mapped by h to the same region; (2) two far points are likely to be mapped by h to different regions;
- The neighbors of the query point q are searched for in the region identified by $h(q)$.

Observation: In traditional hashing, the primary goal is to distribute data as evenly as possible across a set of buckets to minimize the chances that any

two distinct elements (keys) collide, i.e., map to the same bucket. The hash function in this case is designed to minimize collisions overall, regardless of any relationship or similarity between the elements.

Locality Sensitive Hashing (LSH), on the other hand, has a different objective. The key idea behind LSH is to increase the probability that similar items (those that are close to each other in the high-dimensional space) will be hashed into the same bucket. In contrast, dissimilar items (those that are far apart) will be hashed into different buckets with high probability.

Definition of LSH: Consider a metric space $(\mathcal{M}, d)$ and a family of hash functions

$$\mathcal{H} = \{h : \mathcal{M} \to S\},$$

where S is a given domain (subset) (e.g., integers in some range $[0, t]$). So, a single hash function h maps each point to a number and that's why, the family H, has as codomain a subset S of possible points.

For any two points $p, q \in \mathcal{M}$ denote by

$$\Pr_{h \in \mathcal{H}}[h(p) = h(q)]$$

the probability that a hash function h extracted from $\mathcal{H}$ uniformly at random maps p and q to the same value.

Definition: (p_1, p_2, c, r)**-Locality Sensitive Hashing**: Given parameters $p_1, p_2 \in [0, 1]$ with $p_1 > p_2$, $c > 1$ and $r > 0$, we say that $\mathcal{H}$ is (p_1, p_2, c, r)-locality sensitive if for any $p, q \in \mathcal{M}$:

- If $d(p, q) \leq r$, then $\Pr_{h \in \mathcal{H}}[h(p) = h(q)] \geq p_1$.
- If $d(p, q) > cr, then \Pr_{h \in \mathcal{H}}[h(p) = h(q)] \leq p_2$.

So, if two points are close, we want that the hash function gives, with probability greater than p_1, the same region for both points. Instead, if these point's are distant, the probability has to be small.

In Figure 4.9, the LSH for $(c, r) - ANNS$ and an example of the same problem. The answer is not correct only when null is returned but there exists a point $p \in B_r(q) \cap P$. In the example, the hash function that allows results (probabilities above) illustrated above is the following:

$h(p)$ = integer represented by the 2nd and 3rd bits starting from the left

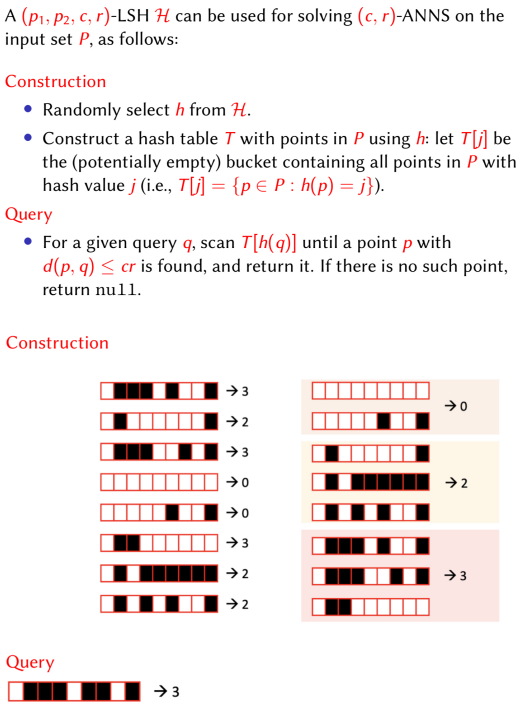


Figure 4.9: Algorithm and example

Consider a query point q. Which points do we expect to see in $T[h(q)]$?

- A near point p (i.e., $d(p, q) \leq r$) will be in $T[h(q)]$ with probability at least p_1, but it might also end up in a different bucket.
- A far point p' (i.e., $d(p', q) > cr$) might end up in $T[h(q)]$, but only with probability at most p_2.

Worst case scenario: there exists only one near point $p \in P$ ($d(q, p) \leq r$) and $n - 1$ far points p' (i.e., $d(p', q) > cr$). In expectation, at most np_2 far points collide with q.

ANNS with LSH: performance: We make the following reasonable assumptions:

- $\mathcal{M} = \mathbb{R}^D$ for some (large) D
- Each point of $\mathcal{M}$ requires $O(D)$ words to be stored.
- A hash value $h(p)$ for some $p \in \mathcal{M}$ can be computed in $O(D)$ time.
- The set S of possible hash values is $\ll n$, where n is the number of points of P.

Theorem

Let P be a set of n points in a metric space $(\mathcal{M}, d)$, and let $\mathcal{H}$ be a (p_1, p_2, c, r)-locality sensitive family of hash functions. Using $\mathcal{H}$ and the above approach to (c, r)-ANNS, a query is answered successfully with probability $\geq p_1$. Moreover the following performance is obtained:

- **Construction time:** $O(Dn)$
- **Space:** $O(Dn)$
- **Query time:** $O(Dnp_2)$ in expectation.

In the notebook is shown the proof for this theorem.
Let now see two family of hash function, the **LSH for Hamming and for Euclidean**. **LSH for Hamming - bit sampling**: Let p be a D-dimensional Boolean vector and let p_i its i-th bit.
Consider the following family of hash functions:

$$\mathcal{H}_{\mathcal{H}} = \{h_i(p) = p_i\}$$

Given two points p, q, the probability of collision is

$$\Pr_{h \in \mathcal{H}_{\mathcal{H}}}[h(p) = h(q)] = 1 - \frac{d_H(p, q)}{D},$$

where $d_H()$ denotes the Hamming distance.
For any two points p, q we have that:

- If $d_H(p, q) \leq r$, then

$$\Pr_{h \in \mathcal{H}_{\mathcal{H}}}[h(p) = h(q)] = 1 - \frac{d_H(p, q)}{D} \geq 1 - \frac{r}{D} \stackrel{\text{def}}{=} p_1$$

- If $d_H(p, q) \geq cr$, then

$$\Pr_{h \in \mathcal{H}_{\mathcal{H}}}[h(p) = h(q)] = 1 - \frac{d_H(p, q)}{D} \leq 1 - \frac{cr}{D} \stackrel{\text{def}}{=} p_2$$

Therefore $\mathcal{H}_{\mathcal{H}}$ is $(1 - r/D, 1 - cr/D, c, r)$-locality sensitive.
Observation: A (p_1, p_2, c, r)-locality sensitive family $\mathcal{H}$ of hash functions is effective when $p_1 \gg p_2$.

A parameter which is typically used to measure the quality of such a family is the ρ factor defined as:

$$\rho = \frac{\log_2 p_1}{\log_2 p_2}$$

The ρ factor for bit sampling is:

$$\rho = \frac{\log_2 p_1}{\log_2 p_2} = \frac{\log_2(1 - r/D)}{\log_2(1 - cr/D)} \sim \frac{r/D}{cr/D} = \frac{1}{c}.$$

LSH for Euclidean distance - random projection Consider points p in $\mathbb{R}^D$. For a fixed value $w > 0$ define the family $\mathcal{H}_\mathcal{E}$ of hash functions

$$h_{a,b}(p) = \left\lfloor \frac{\langle a, p \rangle + b}{w} \right\rfloor, \quad (\langle , \rangle \text{ denotes the inner product})$$

where a and b are sampled as: $a \sim \mathcal{N}^D(0, 1)$ (normal distribution), $b \sim U[0, w]$ (uniform distribution).

Observation: there is a distinct hash function for every pair (a, b).

It can be shown that $\mathcal{H}_\mathcal{E}$ is (p_1, p_2, c, r)-locality sensitive with $\rho = O(1/c)$. There are better LSH families of hash functions for Euclidean distance with $\rho = O(1/c^2)$.

4.2.1 Improving the data structure

Ideally, LSH should have p_1 close to 1 and p_2 close to 0. However, a family H might not provide this type of guarantees.
We now outline two improvements to respectively increase the collision probability of near points and decrease the collision probability of far points, which, combined together, yield better LSH.
Improvement 1: increase collision probability of near points: *Idea* $\hookrightarrow$ repeat search $\ell > 1$ times using ℓ distinct hash tables based on independent hash functions chosen uniformly at random from a (p_1, p_2, c, r)-locality sensitive family $\mathcal{H}$.

This technique is called **repetition or OR construction**.

- The probability that a given near point p (i.e., with $d(p, q) \leq r$) collides with the query point q in at least one hash table increases with ℓ (see next slides).
- However, checking ℓ buckets, one for each hash table, becomes computationally expensive.

Improvement 2: decrease collision probability of far points: *Idea* $\hookrightarrow$ use a family $\mathcal{G}$ of hash functions obtained by concatenating $k \geq 1$ independent

hash functions chosen uniformly at random from a (p_1, p_2, c, r)-locality sensitive family $\mathcal{H}$. Namely,

$$\mathcal{G} = \{g \in \mathcal{H}^k\} = \{g(p) = (h_1(p), \ldots, h_k(p)) \mid h_i \in \mathcal{H}\}$$

It is immediate to establish that for a random $g \in \mathcal{G}$ and any two points p, q with $p \neq q$,

- If $d(p, q) \leq r$, then $\Pr[g(p) = g(q)] \geq p_1^k$;
- If $d(p, q) > cr$, then $\Pr[g(p) = g(q)] \leq p_2^k$.

This technique is called **concatenation or AND construction**.

LSH and (c,r)-ANNS: performance: Let P be a set of n points in a metric space (M, d), and let $\mathcal{H}$ be a (p_1, p_2, c, r)-locality sensitive family of hash functions. Fix

$$k = \log_{1/p_2} n \quad \text{and} \quad \ell = 2p_1^{-k} = 2n^{\rho},$$

where $\rho = \log_2 p_1 / \log_2 p_2$. Using the above approach to (c, r)-ANNS, a query is answered successfully with probability $\geq 1/2$. Moreover the following performance is obtained:

- Construction time: $O\left(Dn^{1+\rho} \log_{1/p_2} n\right)$
- Space: $O\left(Dn + n^{1+\rho} \log_{1/p_2} n\right)$
- Query time: $O\left(Dn^{\rho} \log_{1/p_2} n\right)$ in expectation.

5

Spark for exam

Spark can run on a single machine, in **local mode**, or on a cluster managed by a cluster **manager**, which controls the physical machines.
The heart of Spark is the **driver**, which creates the Spark context, a channel to access to Spark functionalities. Moreover, it distributes tasks to executors and monitors the status of execution.
There are the **executors**, which execute the task assigned and report the result to the driver.
A dfundamental abstraction in Spark is the **RDD**, a collection of elements of the same type, partitioned and distirbuted across several machines.
An RDD can be created from data or from other RDD through transformation. They are read only and are materialized only when needed (**lazy evaluation**).
Each RDD is broken into **chunks** called **partitions** which are distributed among the available machines. The number can be specified or not and the subdivision is created by default. **Partitioning** is exploited to enhance performance.
There are different types of operation on an RDD A:

- **TRANSFORMATIONS**: A transformation generates a new RDD B starting from the data in A.
  There are two types of transformation: **narrow**, where each partition of A contributes to create B or **wide transformation**, where each partition of A may contribute to many partitions of B, so shuffling across machines may be required. An example of narrow is Map and an example of wide is ReduceByKey.
- **ACTIONS**: An action is a computation on the elemnts of A which return a value, e.g. count.
  The **lazy evaluation** means that an RDD A is materialized only when an

action is performed. In order to evict this thing we can do a lazy action and then save the RDD in data memory with cache or persist.