BDC

Davide Volpi

May 30, 2025

Contents

1 Introduction

MTBF = mean time between failure
n components $c_1, c_2, \ldots c_n$
p = probability that a given machine fails at given time
p_n = probability that a machine fails at a given time = $1 - (1-p)^n$.
x = time before failure
$MTBF = E(x) = \sum_{t=1}^{+\infty} t(1-p)^{n(t-1)} \rightarrow Geo(p_n) = \frac{1}{1-(1-p)^n}$ when the sum goes to $+\infty = 1$.

2 MapReduce

Is a **programming framework** for big data processing on distributed platforms. Its original formulation contained:

- A **programming model**.
- An **implementation** cluster-based computing environment.

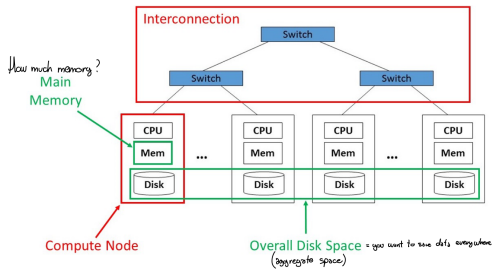
Is employed on **clusters of commodity processors** and **cloud infrastructures** for a **wide array of big data applications**.
The main features are:

- **Data centric view**.
- Inspired by **functional development**.
- **Ease of algorithm/program development**. Messy details are hidden.

2.1 Platforms

The MapReduce applications run on:

- **Clusters** of commodity processors.



- Platforms from cloud provisors.
  - **IaaS (Infrastructure as a Service):** provides computing infrastructure and physical/virtual machines (ex: CloudVeneto).
  - **Paas (Platform as a Service):** provides the user computing platforms.

2.2 Distributed File System (DFS)

- Files divided into **chunks**.
- Each chunk is **replicated** in **different nodes to ensure fault-tolerance**.

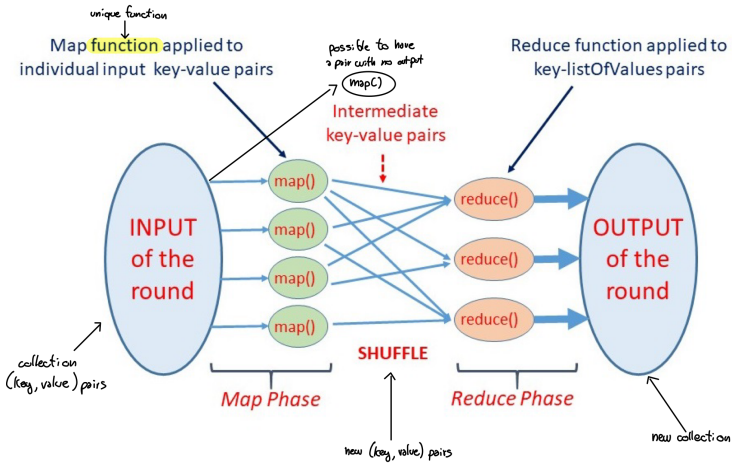
2.3 MapReduce-Hadoop-Spark

Several software frameworks have been proposed to support MapReduce programming:

- **Apache Hadoop:** aimed at improving its initially very poor performance.
- **Apache Spark:** it supports implementation of MapReduce computations, but provides a much richer programming environment.

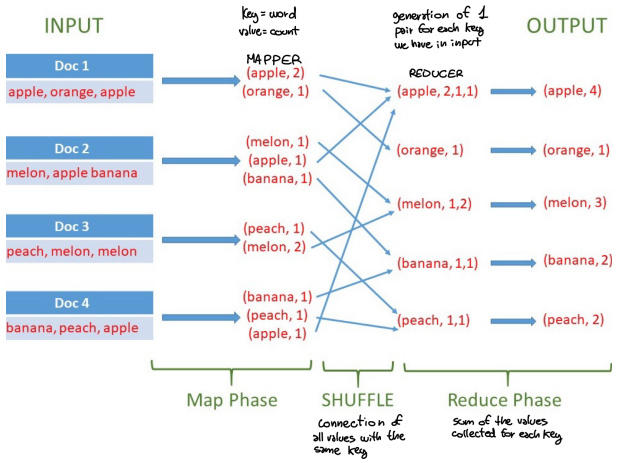
2.4 MapReduce Computation

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



A MapReduce computation may require multiple rounds. If so, the input of a round comprises input data and/or data from outputs of the previous rounds (round i depends on round i-1).
It is often convenient to define shared variables (read only possibly), available to all executors, which maintain global information across different rounds.

2.4.1 MapReduce Example: Word Count



INPUT: n documents $D_1, \dots D_k$ D_i = (name;list of words) where *name*=key and *list of words*=values.
OUTPUT: $\{(w, c(w))\}$ w word occurrence in at least one document ; $c(w)$ = # of occurrence in D_i.
ROUND 1:

- Map phase: produces intermediate pairs.
- Input → documents D_i.
- $\forall$ doc D_i, separately count $(w, c(w))$ where w is a word in D_i

REDUCE PHASE:

- (w, L_w) where $\forall$ word w let L_w = list of values from pairs with key w.
- L_w → list of partial counts
- Emit → $(w, \sum_{c \in L_w} c_i)$

2.4.2 Analysis of Word Count

With n_{max} =max num words per doc, n =tot number of words, k =number of map invocations.
In the case of 1 Round we have $M_L = max\{n_{max}, k\} = O(n_{max} + k)$ and $M_A = O(n)$.
If $k \sim n$ each doc with $O(1)$ words we will have $M_L = M_A = O(n)$.

2.5 MapReduce Algorithm

A MapReduce algorithm should be specified so that:

- The **input and output**.
- The **sequence of rounds** executed for any given input instance is unambiguously implied by the specification.
- For each round are clear **the input,intermediate and output** set of key-value pairs, the **functions applied in the map and reduce phase**.
- Meaningful values bounds for the key **performance indicators can be derived**.

2.6 Execution of a Round on a Distributed Platform

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

2.7 Performance Analysis

The analysis of an MR algorithm aims at estimating:

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

2.8 Design Goals

- Few rounds (constant/logarithmic).
- **Sublinear local space** ($M_L = O(|input|^{\epsilon}$, with $\epsilon < 1$).
- **Linear aggregate space** or only **slightly sublinear** ($M_A = O(|input|)$.
- **Low complexity** of each mac or reduce function (linear time on map/reduce input).

Observation: For every problem solvable by a sequential algorithm in space S there exists a 1-round MapReduce algorithm with both M_L and M_A proportional to S.
Remark: the trivial solution implemented by the observation is not practical for very large inputs because a platform with very large main memory is needed and no parallelism is exploited.

2.9 Pros and Cons

Is MapReduce suitable for Big Data?

- **Data-Centric view**
- **Usability**
- **Portability/Adaptability**
- **Cost**

Main drawbacks:

- **Running time is only coarsely captured by R**. More sophisticated time-performance measures should be used.
- **Curse of the last reducer**. In some cases, one or few reducers may be much slower than the others.
- **Not suitable for applications that require very high performance**.

2.10 Deterministic Partitioning

Problem: given a set S of N objects with class labels, count how many objects belong to each class.
Input: Set S of N objects represented by pairs $(i, (\gamma_i, o_i))$, for $0 \leqslant i < N$, where γ_i is the class of the i-th object (o_i).
Output: The set of pairs $(\gamma, c(\gamma))$ where γ is a class labelling some objects of S and $c(\gamma)$ is the number of objects of S labelled with γ.

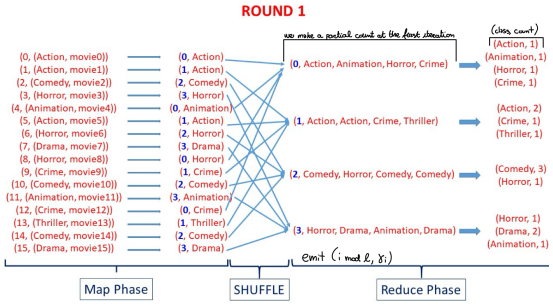
2.10.1 Class Count in 2 Rounds with Deterministic Partitioning

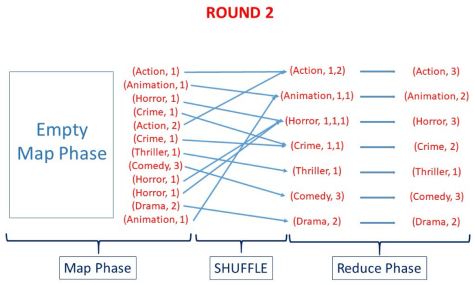
INPUT: $(i, (\gamma_i, o_i))$ with $i \in \{0, 1, \ldots, N-1\}$, l partitions.
OUTPUT:$(\gamma, c(\gamma))$
ROUND 1:

- **Map:** $(i, (\gamma_i, o_i)) \rightarrow (i \ mod \ l, \ \gamma_i)$
- **Reduce:** $\forall$ key j $\in \{0, 1, \ldots, l-1\}$
  - Let L_j = list of values (class) with the key j
  - $\forall$ unique class γ in L_γ emit $(\gamma, c_{L_j}(\gamma))$
  - $c_{L_j}(\gamma)$ = number of γ occurrences in L_j with $(|L_J| = N/l)$

ROUND 2:

- **Map:** $(\gamma, c_\gamma) \rightarrow (\gamma, c(\gamma))$
- **Reduce:** $\forall$ key γ let
  - L_γ = list of partial counts with key γ with $L_gamma \leqslant l$
  - Emit $(\gamma, \sum_{t \in L_\gamma} t)$





2.10.2 Analysis of Class Count Deterministic

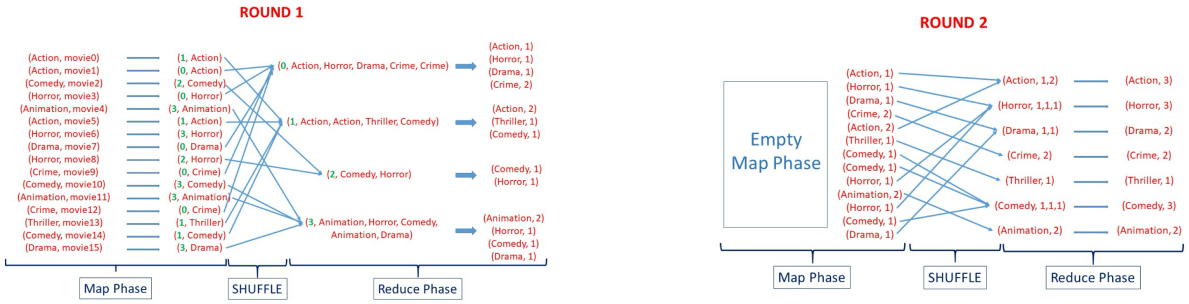
R=2
$M_L = max\{map\ R1:\ O(1), red\ R1:\ O(N/L), map\ R2:\ O(1), red\ R2:\ O(l)\} = O(N/l + l) \rightarrow l = \sqrt{N}$
$M_A = O(N)$
In the extreme case, with $\log_2(N)$ rounds you have constant memory (you create a tree).
In this case you need to have a priori the keys that are unique integers from 0 to n-1 or in that form.

2.11 Random Partitioning

Consider the class count problem, but assume that the input pairs are provided without the integer keys in $[0, N)$.
INPUT: Set S of N objects represented by pairs (γ_i, o_i), for $0 \leqslant i < N$, where γ_i is the class of the i-th object (o_i).
OUTPUT: The set of pairs $(\gamma, C(\gamma))$ where γ is a class labelling some object of S and $C(\gamma)$ is the number of objects of S labelled with γ.
We can employ the 2-round approach seen before, but the partitioning in Round 1 will be achieved now by assigning a **random key** in $[0, l)$ to each of the intermediate pair.
ROUND 1:

- **Map:** $(\gamma_i, o_i) \rightarrow (random(l), o_i)$ where random is $P[random(l) = i] = \frac{1}{l}$

2.11.1 Class Count in 2 Rounds with Random Partitioning



2.11.2 Analysis of Class Count Random

Let:

- m_x = number of intermediate pairs with random key x.
- $m = max\{m_x : 0 \leqslant x < l\}$
- $M_L = max\{O(1); O(m) = O(N/l); O(1); O(l)\}$

Theorem: Fix $l = \sqrt{N}$ 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 $1 - \frac{1}{N^5}$ we have

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

Based of the theorem and on the previous analysis, by setting $l = \sqrt{N}$ we get

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

with probability at least $1 - \frac{1}{N^5}$ (very close to 1, for large N).

2.12 Useful Probabilistic Tools

2.12.1 Union Bound

Given a countable set of events $E_1, \ldots, E_r$, we have $Pr(\cup_{i=1}^{r} E_i) \leqslant \sum_{i=1}^{r} Pr(E_i)$, where E_i ="something bad happens".

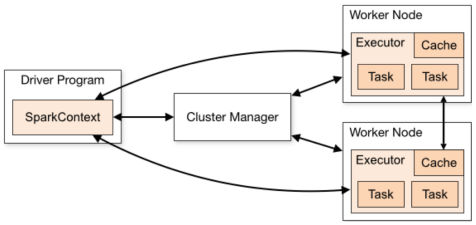
2.12.2 Chernoff Bound

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

- $Pr(X \geqslant \delta_1 \mu) \leqslant 2^{-\delta_1 \mu} \rightarrow$ probability of being in the upper tail of the distribution.
- $Pr(X \leqslant (1 - \delta_2)\mu) \leqslant 2^{-\delta_2^2 \mu/2} \rightarrow$ probability of being in the lower tail of the distribution.

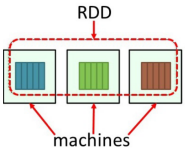
3 Apache Spark Fundamentals

- **Driver (master):** is the hearth of the application, it **creates the spark context**, an object which can be regarded as a channel to access all Spark functionalities; **distributes tasks to the executors**; **monitors the status of the execution**.
- **Executors (workers):** execute the tasks assigned by the driver, and report their status to the driver.
- **Cluster manager:** when the application is run on a distributed platform, the cluster manager controls the physical machines and allocates resources to applications.



3.1 Resilient Distributed Dataset (RDD)

- **Fundamental abstraction** in spark. Is a collection of elements of the same type, partitioned and distributed across several machines.



- An RDD provides an interface based on **coarse-grained transformations**.
- RDDs ensure **fault-tolerance**.

3.1.1 Main Characteristics

- RDDs are **created**
  - From data in stable storage.
  - from other RDDs.
- RDDs are **immutable**.
- RDDs are **materialized only when needed** (lazy evaluation).
- Spark **maintains the lineage of each RDD**, namely the sequence of transformations that generate it, which enable to materialize it or reconstruct it after a failure, **starting from data in stable storage**.

3.1.2 Partitioning

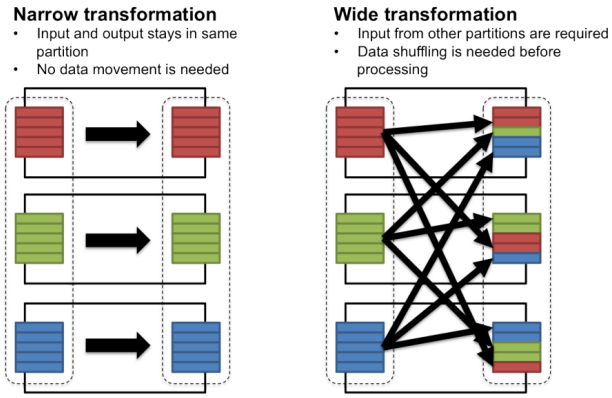
Each RDD is **broken into chunks** called **partitions** which are distributed among the available machines.
A program can specify the **number of partitions for each RDD**.
Partitions are **created by default or through a custom partitioner**.
The typical number of partitions is $2x/3x$ the number of cores.

- Spark creates map tasks so to make each executor apply the map function on data from a locally stored partition.
- To implement algorithms that require partitioning, the programmer can explicitly access RDD partitions.
- RDD partitions are exploited implicitly by some ready-make spark aggregation primitives.

3.1.3 Operations

The following types of operations can be performed on an RDD A:

- **TRANSFORMATIONS:** A transformation generates a new RDD B starting from the data in A. We distinguish between:
  - **Narrow Transformation:** Each partition of A contributes to one partition of B, which is stored in the same machine. No shuffling of data across machines is needed.
  - **Wide Transformation:** Each partition of A may contribute to many partitions of B. Hence, shuffling of data across machines may be required.



- **ACTIONS:** An action is a computation on the elements of A which returns a value to the application.
  - **Lazy Evaluation:** RDD A is materialized only when an action is performed.
- **PERSISTENCE:** Methods like cache()(or persist() will save the RDD data in memory after the subsequent action.
  - **cache():** data are stored asin RAM. Data that do not fit are recomputed when needed.
  - **persist():** data are stored as instructed. There are several options.

3.2 Implementing MapReduce Algorithms in Spark

- **Map Phase:** on the input RDD X we invoke one of map methods offered by Spark passing the desired map function as argument.
- **Reduce Phase:** on the RDD X' resulting from the map phase
  - Invoke one of grouping methods offered by Spark to group the key-value pairs into Key-ListOfValues pairs.
  - Invoke one of map methods offered by Spark to apply the desired reduce function to each Key-ListOfValues.

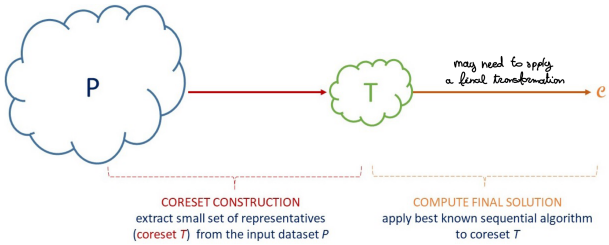
Alternatively, one can use one of the ready-made reduce primitives offered by Spark.

4 Coreset Technique

Suppose that we want too to solve a problem Π on instances P which are too large to be processed by known algorithms. With the coreset technique we:

1. Extract a small subset T from P (dubbed coreset), making sure that it represents P well.
2. Run best known (possibly slow) sequential algorithm for Π on the small coreset T.

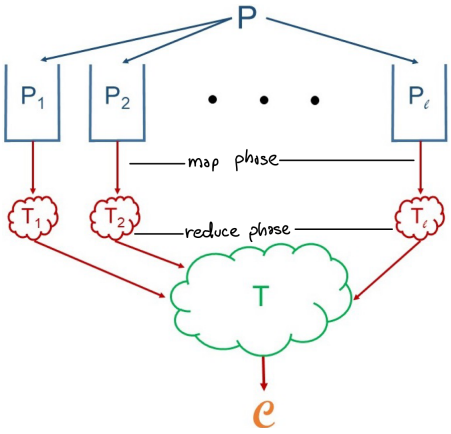
This technique is effective if T can be extracted efficiently by processing P and the solution computed on T is a good solution for Π.



4.1 Composable Coreset Technique

1. Partition P into l subsets $P_1, P_2, \ldots, P_l$, and extract a small coreset T_i from each P_i, making sure that it represents P_i well.
2. Run the best known (possibly slow) sequential algorithm for Π on $T = \cup_{i=1,l} T_i$.

This technique is effective if each T_i can be extracted efficiently from P_i in parallel for al i's and the final coreset T is still small and the solution computed on T is a good solution for Π.



- Extract a small coreset T_i from each P_i.
- Apply the best known sequential algorithm to final coreset T = union of the T_i's, to compute the final solution l.

4.2 Metric Space

A metric space is an order pair (M, d) where M is a set and $d(\cdot)$ is a metric on M, in example $d : M \times M \rightarrow R$. such that for every $x, y, x \in M$ holds:

- $d(x, y) \geqslant 0$
- $d(x, y) = 0$ if and only if $x = y$
- $d(x, y) = d(y, x)$ (symmetry)
- $d(x, z) \leqslant d(x, y) + d(y, z)$ (triangle inequality)

4.3 Distance Functions

4.3.1 Minkowski Distances

$$\textit{Minkowski Distances:}\quad d_{Lr}(X,Y) = (\sum_{i=1}^{n} |x_i - y_i|^r)^{\frac{1}{r}}$$

- **Euclidean Distance** $r = 2$: standard distance in R^2.
- **Manhattan Distance** $r = 1$: used in grid-like environments.
- **Chebyshev Distance** $r = \infty$: maximum absolute differences of coordinates ($max\{|x_i - y_i|\}\forall_i$).

It aggregates the gap in each dimension between two objects.

4.3.2 Angular Distance

$$\textit{Angular Distance:}\quad d_{angular}(X,Y) = arccos(\frac{X \cdot Y}{||X|| \cdot ||Y||}) = arccos(\frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n}(x_i)^2}\sqrt{\sum_{i=1}^{n}(y_i)^2}}) \in [0, \pi]$$

Represents the angle between two vectors.
It measures the rations among features' values, rather than their absolute value.

4.3.3 Hamming Distance

$$\textit{Hamming Distance:}\quad d_{hamming}(X,Y) = |\{i|x_i \neq y_i\}|$$

Used when points are binary vectors over some n-dimensional space. Represents the number of coordinates in which they differ.

4.3.4 Jaccard Distance

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

Used when points are sets.
It measures the ratio between the number of differences between two sets and the total number of points in the set.
The value $|S \cap T|/|S \cup T|$ is referred to a the **Jaccard Similarity** of the two sets.

5 Clustering

5.1 Types of Clustering

A clustering problem often specifies an objective function to optimize. This problems can be categorized on whether or not:

- A target number k of clusters is given in input.
- For each (disjoint) cluster center must be identified.

5.1.1 The Optimization Problem

Given a set I of instances (inputs) and $\forall_i \in I$, set s_i of solutions, we call ϕ : objective function and $\phi : s \to R$.
Problem: given an instance i, find a solution $s^* \in s_i$ such that:

- $\phi(s^*) = min\{\phi(s), s \in s_i\}$ if is a **minimization problem**.
- $\phi(s^*) = max\{\phi(s), s \in s_i\}$ if is a **maximization problem**.

There can be several optimal solutions $s_1^*, s_2^* \dots$ with the same cost.

5.1.2 Approximation Algorithm

Beyond worst case: design and analyze so that performance are input dependent.
Let π be an optimization problem. A c-approximation algorithm ($c \geqslant 1$) is an algorithm that $\forall_i \in I$, it returns a solution $A(i)$:

- $\phi(A(i)) \leqslant c \cdot min\{\phi(s) : s \in s_i\}$ in the case of a minimization problem.
- $\phi(A(i)) \geqslant \frac{1}{c} \cdot max\{\phi(s) : s \in s_i\}$ in the case of a maximization problem.

c is called the approximation ratio/factor and can be: constant (i.e. $c = 2$), parameter (i.e. $c = 1 + \epsilon$ and if $\epsilon \to 0$ we have a better approximation but high running time), size dependent (i.e. $c = 1 + \frac{1}{m}$).

5.2 Center Based Clustering

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

- $(C_1, C_2, \dots, C_k)$ defines a partition of P.
- $c_1, c_2, \dots, c_k$ are suitably selected centers for the clusters, where $c_i \in C_i, \forall_i$.

For a given input pointset P, a center-based clustering problem aims at finding a k-clustering of P which optimizes a certain objective function. The 3 most popular are:

- **K-center clustering:** aims at minimizing the maximum distance of any point from the center of its cluster.
  - $\phi_{kcenter}(P, S) = max_{x \in P} d(x, S)$
  - Useful when we need to guarantee that every point is close to a center.
- **K-means clustering:** aims at minimizing the sum of the squared distances of the points from the centers of their respective clusters.
  - $\phi_{kmeans}(P, S) = \sum_{x \in P} (d(x, S))^2$
  - Provide guarantees on the average squared distances.
- **K-median clustering:** aims at minimizing the sum of the distances of the points from the centers of their respective clusters.
  - $\phi_{kmedian}(P, S) = \sum_{x \in P} d(x, S)$
  - Provide guarantees on the average distances.

5.3 K-Center Clustering

Clustering is an important primitive and apply to Big Data. It can be speed up with coresets and can be a coreset for other problems.
Clustering aims to grouping the points into a number of subsets (clusters) such that:

- Points in the same cluster are close to one another (close points are similar).
- Points in different clusters are distant from one other (distant points are dissimilar).

5.3.1 Farthest-First Traversal (FFT) Algorithm

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 of the k-center problem on P.

1: $S \leftarrow \{c_1\}$ // $c_1 \in P$ arbitrary point
2: **for** $i \leftarrow 2$ to k do **do**
3: Find the point $c_i \in P - S$ that maximizes $d(c_i, S)$
4: $S \leftarrow S \cup \{c_i\}$
5: **end for**
6: return S

Theorem: Let S be the set of centers returned by running FFT on P. Then:

$$\phi_{kcenter}(P, S) \leqslant 2 \cdot \phi^{opt}_{kcenter}(P, k)$$

This is a 2-approximation algorithm, but is very sensitive to noise.
This algorithm focuses on worst-case distance of points from their closest centers.
The FFT approximation guarantees are the almost the best one can obtain in practice.

5.3.2 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.
Round 1:

- **Map phase:** Partition P arbitrarily into l subsets of equal size $P_1, P_2, \ldots, P_l$ (size of $P_i \sim N/l$).
- **Reduce Phase:** for every $i \in [1, l]$ separately, fun FFT on P_i to determine a set $T_i \subset P_i$ of k centers.

Round 2:

- **Map phase:** empty
- **Reduce phase:** gather the coreset $T = \cup_{i=1}^{l} T_i$ (of size $l \cdot k$) and run, using a single reducer, FFT on T to determine a set $S = \{c_1, c_2, \ldots, c_k\}$ of k centers, and return S as output.

Theorem: The 2-round MR-FFT algorithm can be implemented using local space $M_L = O(\sqrt{Nk})$ and aggregate space $M_A = O(N)$, and $l = \sqrt{\frac{N}{k}}$.
Let T be the union of the coresets T_i computed by MR-FFT on input P. The following lemma establishes the quality of T:

$$d(x, T) \leqslant 2\phi^{opt}_{kcenter}(P, k), \forall_x \in P$$

Theorem: let S be the set of k centers returned by MR-FFT on P. Then

$$d(P, S) \leqslant 4\phi^{opt}_{kcenter}(P, k)$$

The MR-FFT is a 4-approximation algorithm.
FFT 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-FFT is able to handle very large pointsets and the final approximation is not too far from the best achievable one.

5.3.3 Low-Dimensional Pointsets

When P has low dimensionality (2/3 dimensions or particular properties) the quality of the solution returned by MR-FFT can be made arbitrarily close to 2 by selecting a slightly larger coreset.

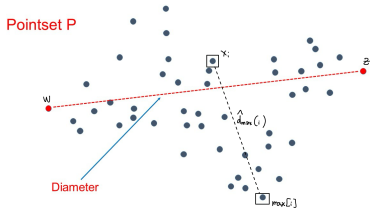
- $\forall_{partition} P_l$, construct coreset T'_l with $k' > k$ points, $T'_l = FFT(P_l, k')$
- $T' = \cup_{T'_l}, |T| = k' \cdot l > k \cdot l.$
- OUTPUT: $FFT(T', k)$
- As k' increases, approximate $MR - FFT \rightarrow 2$.

5.4 Diameter Computation

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

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

The computation of the exact diameter requires almost quadratic operations, hence it is impractical for every large pointset.
For an arbitrary $x_i \in P$ define $d_{max}(i) = max\{d(x_i, x_j) : 0 \leqslant j \leqslant N\}$
Lemma: for any $0 \leqslant i \leqslant N$ we have $d_{max} \in [d_{max}(i), 2d_{max}(i)]$

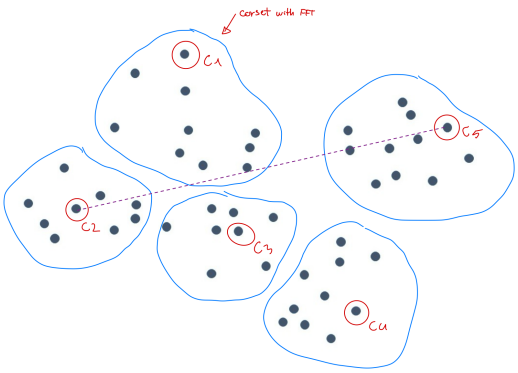


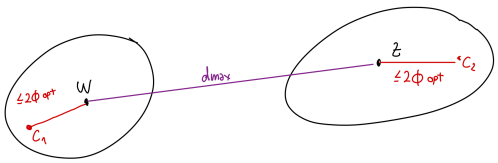
5.4.1 Coreset-Based Diameter Approximation

- Fix a suitable $k \geqslant 2$.
- Extract a coreset $T \subset P$ 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}.

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

- **Sequentially:** in $O(N)$ time using FFT.
- **MapReduce:** in 2 rounds, using $M_L = O(\sqrt{N})$ and $M_L = O(N)$ using MF-FFT.





We have $T = \{c_1, \ldots, c_n\}$, $d_{max} = d(w, z)$.
$d_{max} = d(w, z) \leqslant d(w, c_1) + d(c_1, c_2) + d(c_2, z) \leqslant d_{max}(T) + 4\phi^{opt}(P, k)$, as $k \rightarrow \infty$, approx to 1.

5.5 Diversity Maximization Problem

The objective is to determine the most diverse subset of size k.
Given a set P of points from a metric space (M, d) and a positive integer $k < |P|$, return a subset $S \subset P$ of k points, which maximizes the diversity function

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

A c-approximation algorithm is known with $c = 2 - 2/k$ which, however, requires quadratic time.

5.5.1 Coreset-based Approach to Diversity Maximization

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

$$div^{opt}(P, k) = \max_{S \subset P, |S|=k} div(S)$$

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

$$div^{opt}(T, k) \geqslant \frac{1}{1+\epsilon} div^{opt}(P, k) \quad or \quad \geqslant (1 - \epsilon) div^{opt}(P, k)$$

An interesting relation between the k-center and the diversity maximization problems is

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

where the right part is the average distance in S for each pair.
Given P and k:

- Run FFT to extract T' of $h > k$ centers from P.
- Consider the h-clustering included by T' on P and select k arbitrary points from each cluster.
- 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.

5.6 K-Means and K-Median

A current popular algorithm for the k-means/median is the **Lloyd's algorithm**, but has some problems with the initial random selection of the centroids.

5.6.1 K-Means and K-median ++

If the centroids are selected well it usually provides a good solution, the **k-means/median ++**:

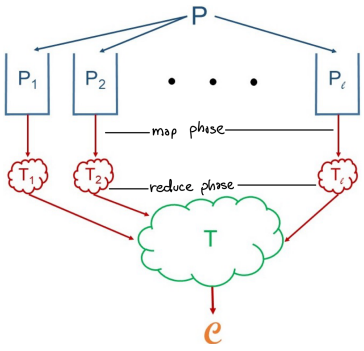
$c_1 \leftarrow$ random point chosen from P with uniform probability.
$S \leftarrow \{c_1\}$
for $2 \leqslant i \leqslant k$ **do**
  for $x \in P - S$ **do** $\pi(x) \leftarrow (d(x, S))^2 / \sum_{y \in P-S} (d(y, S))^2$
  end for
  $c_i \leftarrow$ random point in P-S according to distribution $\pi(\cdot)$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

K-means ++ is a randomized algorithm and it is known that in expectation, the returned solution is an α-approximation, with $\alpha = \Theta(\ln k)$.

5.6.2 Partitioning Around Medoids (PAM)

Is an algorithm 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.

5.6.3 Coreset-based Approach for K-Means and K-Median



- Each point $x \in T_i$ is given a weight $w(x)$ = the number of points in P_i 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, adopted to handle weights.

To each $y \in T$ we give an importance weight equal to the number of points in P associated to the point:

$$\forall y \in T : w(y) : \textit{number of points in } S \textit{ s.t. } y \textit{ is the closest point in } T$$

5.7 Weighted K-Means

Input:

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

Output: Set S of k centers in R^D minimizing

$$\phi^w_{kmeans}(P, S) = \sum_{x \in P} w(x) \cdot (d(x, S))^2$$

This formulation allows centers outside P and if $w(x) = 1$ for every $x \in P$ we have the standard k-means. The K-Means++ with weights become:

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

5.7.1 Coreset-based MapReduce Algorithm for K-Means

$MR - kmeans(A_1, A_2)$ which 2 sequential algorithms A_1 and A_2 such that:

- A_1 solves the standard variant without weights.
- A_2 solves the more weighted variant.

Both algorithms require proportional space to the input size.
Input: Set P of N points in R^D, integer $k > 1$, sequential k-means algorithms A_1, A_2.
Output: Set S of k centers in R^D which is a good solution to the k-means problem on P.
Round 1:

- **Map Phase:** Partition P arbitrarily in l subsets of equal size $P_1, \ldots, P_l$.
- **Reduce Phase:** for every $i \in [1, l]$ separately, run A_1 on P_i to determine a set $T_i \subset P_i$ 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.
  - Emit $(0, T_i)$.

Round 2:

- **Map Phase:** empty..
- **Reduce Phase:** using a single reducer, A_2 on $T = \cup_{i=1}^{l} T_i$ to determine a set $S = \{c_1, \ldots, c_k\}$ of k centers, which is then returned as output.
- Emit the solution (0,S)

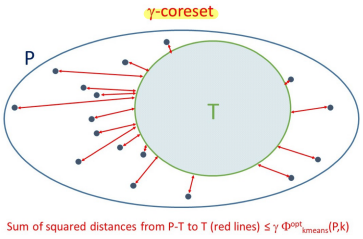
5.7.2 Analysis of the MR-KMeans

Assume $k \leqslant \sqrt{N}$. By setting $l = \sqrt{\frac{N}{k}}$, it is easy to see that $MR-kmeans(A_1, A_2)$ requires linear space:

- $M_L = O(\max\{Red1 : N/L, Red2 : l \cdot k\}) = O(\sqrt{N \cdot k}) = o(N)$
- $M_A = O(N)$

Given a pointset P, a coreset $T \subset P$ and a proxy function $\tau : P \to T$, T is a γ-coreset for P,k and the k-means objective if

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



Suppose that:

- A_1 is a γ-approximation algorithm for the unweighted k-means problem.
- A_2 is a α-approximation algorithm for the weighted k-means problem.

Then:

- The coreset T computed in Round 1 is a γ-coreset.
- The solution S computed in Round 2 is such that

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

5.7.3 Final Observations

Typically we use the k-Means++ as A_1 in Round 1 and a weighted version of k-Means++ plus LLoyd's as A_2 in Round 2.
If $k' > k$ centers are selected from each P_i in Round 1, the quality of T, hence the quality of the final clustering, improves.

6 Streaming

6.1 Streaming Model

- Sequential machine with limited amount of working memory.
- Input provided as a continuous (one-way) stream.

Let $\sum = x_1, \ldots, x_n, \ldots$ denote the input stream received sequentially. Upon receiving x_n:

- Suitable data structures stored in the working memory are updated → **Update task**.
- If required, a solution for the problem at hand is computed from the data stored in the working memory → **Query task**.

The **Key Performance Indicators** are:

- Size s of the working memory (aim $s << |\sum|$).
- Number p of sequential passes over $\sum$ (aim p=1).
- Update time T_u per item (aim $T_u = O(1)$.
- Query time T_q to return a solution after seeing $x_1, \ldots, x_n$ (aim T_1 independent of n).

The typical data analysis task are:

- Identification of frequent items.
- Statistics (frequency moments, quantiles, histograms)
- Optimization and graph problems (clustering, triangle counting)

Our goal are suitable tradeoff between accuracy, working memory, update/query time.

6.2 Boyer-Moore for Finding the Majority Element

Given a stream $\sum = x_1, \ldots, x_n$ return the element x (if any) that occurs $> n/2$ times in $\sum$:

- First pass with Boyer-Moore algorithm to find an element x which is the majority element, if one exists.
- Second pass to check whether x is indeed the majority element.

Algorithm 1 Boyer-Moore Algorithm

```
1: cand ← null and count ← 0
2: for x_t in ∑ do do
3:     If count = 0 then
4:         cand ← x_t
5:         count ← 1
6:     Else
7:         if cand = x_t then count ← count + 1
8:         else count ← count − 1
9: end for
10: return cand
```

Theorem: Given a stream $\sum$ which contains a majority element m, the Boyer-Moore algorithm returns m using working memory size of $O(1)$, 1 pass, $O(1)$ update and query times.

6.3 Sampling

Given a set X of n elements and an integer $1 \leqslant m < n$, an m-sample of X is a random subset $S \subset X$ of size m, such that for each $x \in X$, we have $Pr(x \in S) = m/n$ (uniform sampling).
In the streaming setting, where streams are potentially unbounded, at every time step a random sample of the elements seen so far is sought.

6.3.1 The Sampling Problem

Given a (possibly unbounded) stream $\sum = x_1, x_2, \ldots$ and an integer $m < |\sum|$, maintain, for every $t \geqslant m$, an m-sample S of the prefix $\sum_t = x_1, x_2, \ldots, x_t$.

6.3.2 Reservoir Sampling Algorithm

Performs a uniform sampling on a stream without knowing the size of the stream.

Algorithm 2 Reservoir Sampling Algorithm

```
1: Initialization: S ←
2: for x_t in ∑ do do
3:     If t ⩽ m then add x_t to S
4:     Else with probability m/t do
5:         evict an element x from S, chosen with uniform probability
6:         add x_t to S
7: end for
```

Theorem: Let $\sum = x_1, x_2, \ldots$. At any time $t \geqslant m$, the set S maintained by the reservoir sampling algorithm is an m-sample of $\sum_t = x_1, x_2, \ldots, x_t$. Moreover, the algorithm features $O(1)$ update and query times. $P[x \in S]$ at any moment t is always $\frac{m}{t}$.

6.4 The Frequent Items Problem

Given a stream $\sum = x_1, x_2, \ldots, x_n$ of n items from a universe U and a frequency threshold $\varphi \in (0, 1)$, determine all distinct items that occur at least $\varphi \cdot n$ times in $\sum$ (frequent items).

6.4.1 Frequent Items with Reservoir Sampling

Frequent items can be approximated through reservoir sampling:

1. Extract an m-sample S from $\sum$ with reservoir sampling.
2. Return the subset $S' \subseteq S$ of distinct items in the sample.

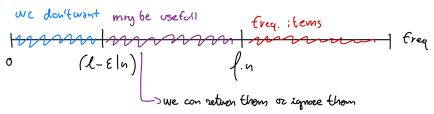
How well does S' approximate the set of frequent items?

- If $m \geqslant 1/\varphi$, for any given frequent item a, we expect to find at least one copy of a in S (hence in S').
- Some frequent item may be missed, and S' may contain items with very low frequency.

6.4.2 ϵ-Approximate Frequent Items (ϵ-AFI) Problem

Given a stream $\sum = x_1, x_2, \ldots, x_n$ of n items, a frequency threshold $\varphi \in (0, 1)$, and an accuracy parameter $\epsilon \in (0, \varphi)$, return a set of distinct items that

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



If $\epsilon = \varphi$ reservoir sampling don't work.

6.4.3 Sticky Sampling

Provides a probabilistic solution of the ϵ-AFI problem. The main ingredients are:

- **Confidence parameter** $\delta \in (0, 1)$.
- **Hash table S**, whose entries are pairs $(x, f_e(x))$ where
  - x (the key) is an item

- $f_e(x)$ (the value) is a lower bound to the number of the occurrences of x seen so far.

- **Sampling rate** chosen to ensure (with probability $\geqslant 1-\delta$) that the frequent items are in the sample.

Algorithm 3 Sticky Sampling Algorithm

1: Consider a stream $\sum = x_1, x_2, \ldots, x_n$ and assume n known
2: **Initialization:**
- $S \leftarrow$ empty hash table
- $r = \ln(1/(\delta\varphi))/\epsilon$ and sampling rate $= r/n$

3: **for** x_t in $\sum$ **do** do
4: **If** $(x_t, f_e(x_t)) \in S$ **then** $f_e \leftarrow f_e(x_t) + 1$
5: **Else** add $(x_t, 1)$ to S with probability r/n
6: **end for**
7: **return** all items x in S with $f_e(x) \geqslant (\varphi - \epsilon)n$

Theorem: Sticky sampling solves the ϵ-AFI problem correctly with probability at least $1-\delta$ and requires:

- Working memory of size $O(r)$ in expectation.
- 1 pass.
- $O(1)$ expected update time.
- $O(r)$ expected query times.

The space is independent of the stream length.

6.5 Sketching

A **sketch** is a space-efficient data structure that can be used to provide (typically probabilistic) estimates of (statistical) characteristics of a data stream.

6.5.1 Frequency moments

Consider a stream $\sum = x_1, x_2, \ldots, x_n$ whose elements belong to a universe U. For each $u \in U$ occurring in $\sum$ define its (absolute) frequency

$$f_u = |\{j : x_j = u, 1 \leqslant j \leqslant n\}|$$

i.e., the number of occurrences of u in $\sum$.
For every $k \geqslant 0$, the **k-th frequency moment** F_k of $\sum$ is

$$F_K = \sum_{u \in U} f_u^k$$

- F_0 = number of distinct items in $\sum$.
- $F_1 = |\sum|$ is trivial to maintain with a counter.
- $1 - F_2/|\sum|^2$ = Gini index of $\sum$. It provides info on data skew.

6.5.2 Estimating F_0 for $\sum$

Exact computation: use $|U|$ counters of dictionary with $|F_0|$ elements (not suited for streaming settings).
Approximation: probabilistic counting algorithm:

- Working memory $O(\log |U|)$.
- F_0 estimated within a factor c with probability $\geqslant 1 - 2/c$.

- The main idea is:
  - 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.
  - The more distinct elements in $\sum$, the more likely to have a $u \in \sum$ mapped to a string with many trailing 0's.

For this approximation we need:

- Array C of $\lceil \log_2 |U| \rceil + 1$ bits, all initialized to 0.
- Hash function $h : U \rightarrow [0 \ldots |U|-1]$. We assume:
  - For every $u \in U, h(u)$ has uniform distribution in $[0 \ldots |U|-1]$.
  - All $h(u)$'s are pairwise independent.
- For $i \in [0 \ldots |U|-1]$ define

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

Algorithm 4 Probabilistic Counting Algorithm

1: **For** $x_j \in \sum$ **do** $c[tr(h(x_j)] \leftarrow 1$
2: After processing x_n, estimate F_0 as

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

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

Theorem: for a stream $\sum$ of n elements, the probabilistic counting algorithm returns a value of $\tilde{F}_0$ such that, for any $c > 2$:

$$Pr(\tilde{F}_0 < F_0/c) \leqslant 1/c \quad \textit{and} \quad Pr(\tilde{F}_0 > cF_0) \leqslant 1/c$$

hence

$$Pr(F_0/c \leqslant \tilde{F}_0 \leqslant cF_0) \geqslant 1 - 2/c$$

6.5.3 Estimating Individual Frequencies and F_2 for $\sum$

In one pass over $\sum$ we want to compute a small sketch that enables to derive unbiased estimates of

- f_u for any given $u \in U$ (individual frequencies).
- $F_2 = \sum_{u \in U} (f_u)^2$

To compute these we need:

- $d \times w$ array C of counters ($O(\log n)$ bits each).
- d hash functions: $h_0, h_1, \ldots, h_{d-1}$, with

$$h_j : U \rightarrow \{0, 1, \ldots, w-1\}$$

Algorithm 5 Count-Min Sketch Algorithm

1: $C[j,k] = 0$ for every $0 \leqslant j < d$ and $0 \leqslant k < w$.
2: **For each** $x_t \in \sum$ **do**
3: **For** $0 \leqslant j \leqslant d-1$ **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:

$$\min_{0 \leqslant j \leqslant d-1} C[j, h_j(u)]$$

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

$$0 \leqslant \tilde{f}_u - f_u \leqslant \epsilon \cdot n \quad \textit{with probability} \geqslant 1 - \delta$$

The unbiased version of the count-min sketch algorithm is the count sketch algorithm. The main idea is for each item $u \in U$ multiply its contributions to each row by a value in $\{-1, +1\}$ randomly selected, so to cancel out collisions. The 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
$$h_j : U \to \{0, 1, \ldots, w-1\}$$
- d hash functions: $g_0, g_1, \ldots, g_{d-1}$, with
$$g_j : U \to \{-1, +1\}$$

Algorithm 6 Count Sketches Algorithm

1: $C[j,k] = 0$ for every $0 \leqslant j < d$ and $0 \leqslant k < w$.
2: **For each** $x_t \in \sum$ **do**
3: **For** $0 \leqslant j \leqslant d-1$ **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 $0 \leqslant j < d$, let

$$\tilde{f}_{u,j} = g_j(u) \cdot C[j, h_j(u)]$$

The frequency of u can be estimated as:

$$\tilde{f}_u = \textit{median of the } \tilde{f}_{u,j}\textit{'s}$$

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

- $E[\tilde{f}_{u,j} = f_u]$, for any $j \in [0, d-1]$, i.e., $\tilde{f}_{u,j}$ is an unbiased estimator of f_u.
- With probability $\geqslant 1 - \delta$,
$$|\tilde{f}_u - f_u| \leqslant \epsilon \cdot \sqrt{F_2}$$
where $F_2 = \sum_{u \in U} (f_u)^2$

6.5.4 Estimation of F_2

Given a $d \times w$ count sketch for $\sum$, define

$$\tilde{F}_{2,j} = \sum_{k=0}^{w-1} (C[j,k])^2 \quad \textit{for} \quad 0 \leqslant j < d \to \tilde{F}_2 = \textit{median of the } \tilde{F}_2\textit{'s}$$

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

- $E[\tilde{F}_{2,j}] = F_2$ for any $0 \leqslant j < d$. That is, any $\tilde{F}_{2,j}$ is an unbiased estimator of F_2.
- With probability $\geqslant 1 - \delta$,
$$|\tilde{F}_2 - F_2| \leqslant \epsilon \cdot \sqrt{F_2}$$

6.5.5 Analysis of the Performance Metrics

Both count-min and count sketches can be computed in 1 pass. For both sketches we have:

- **Working memory:** $O(d \cdot w)$, which becomes $O(\log(1/\delta)/\epsilon)$, for count-min sketch, and $O(\log(1/\delta)/\epsilon^2)$, for the count sketch.
- **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.

6.6 Filtering

6.6.1 Approximate membership problem

Given a stream $\sum = x_1, x_2, \ldots$ 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 false negative:** no error when $x_i \in S$.
- **Small false positive rate:** small probability of error when $x_i \notin S$.

The main ingredients are:

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

$$h_j : U \to \{0, 1, \ldots, n-1\} \quad \textit{for every } 0 \leqslant j < k$$

Algorithm 7 Bloom Filter Algorithm

1: **For each** $e \in S$ **do**
2: **For** $0 \leqslant j \leqslant k$ **do** $A[h_j(e)] \leftarrow 1$
3: **Membership test:** for any $x_i \in \sum$ if

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

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

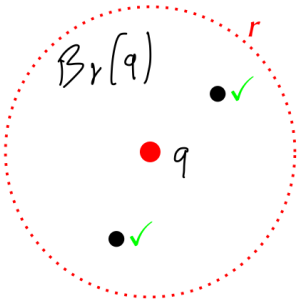
$$Pr[FP] = Pr(A[h_j(x_i)] = 1 \quad \textit{for each} \quad 0 \leqslant j < k) \simeq (1 - e^{-km/n})^k \to \quad \textit{if} \quad n = m \log(\frac{1}{\delta}) \quad \sim e^{-\frac{n}{m}}$$

7 Similarity Search in Low Dimensions

7.1 r-Near Neighbour 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 :

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

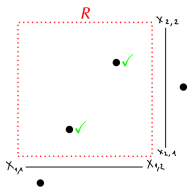


r-NNS Definition: Given a set P of n points from the metric space (M, d), construct a data structure that, given a query point $q \in M$ and a distance threshold $r > 0$, returns :

- A point $p \in B_r(q) \bigcap P$, if any such point exists.
- `null` if $B_r(q) \bigcap P = \varnothing$

7.2 Range Reporting (RR) Problem

Given a set $P \subset \mathbb{R}^D$ of n points, construct a data structure that, given a rectangular region $R = [X_{1,1} : X_{1,2}] \times \cdots \times [x_{D,1} : X_{D,2}]$, returns all points of P contained in R.



7.3 Kd-Tree

For a set P of n points in $\mathbb{R}^2$ is a binary tree such that :

- Each internal node v stores a line ℓ_v
  - If v is at even depth, ℓ_v is a vertical line at $x = x_v$.
  - If v is at odd depth, ℓ_v is horizontal line at $y = y_v$
- Each external node (leaf) v stores a point p_v.
- Each node v is associated with a rectangular region of $\mathbb{R}^2$, denoted as $region(v)$, and represents the subset of points $P_v = P \cap region(v)$.

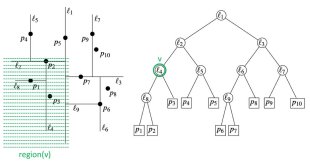
The rectangular regions define a hierarchical partitioning of P:

- For the root r, $region(r) = \mathbb{R}^2$.
- An internal node v storing a line ℓ_v and associated with $region(v)$ has two children u (left) and w (right) such that:
- If ℓ_v is vertical at abscissa x_v:

$$region(u) = region(v) \cap \mathbb{R}^2_{x \leqslant x_v}$$

$$region(w) = region(v) \cap \mathbb{R}^2_{x > x_v}$$

  where $\mathbb{R}^2_{x \leqslant x_v}$ and $\mathbb{R}^2_{x > x_v}$ are the two halfspaces defined by x_v.



- If ℓ_v is horizontal at ordinate y_v:

$$region(u) = region(v) \cap \mathbb{R}^2_{y \leqslant y_v}$$

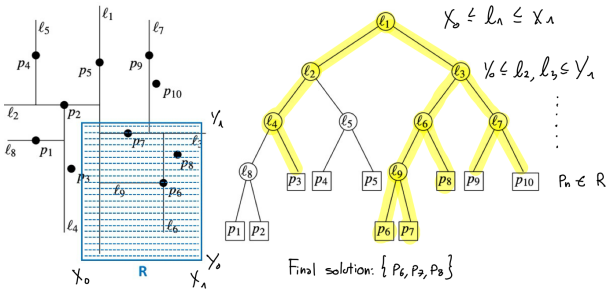
$$region(w) = region(v) \cap \mathbb{R}^2_{y > y_v}$$

  where $\mathbb{R}^2_{y \leqslant y_v}$ and $\mathbb{R}^2_{y > y_v}$ are the two halfspaces defined by y_v.

7.3.1 Query Algorithm

Given a rectangle $R = [x_1, x_2] \times [y_1, y_2]$, the query algorithm visits recursively all nodes whose regions intersect R, starting from the root:

- At leaf node v: return p_v if $p_v \in R$.
- At an internal node v: recurse on each child whose associated region intersect R. If $region(v) \cap R = \varnothing$, hence none of the children's regions intersect R, return $\varnothing$.



7.3.2 Performance in $\mathbb{R}^2$

The RR problem for a set P of n points in $\mathbb{R}^2$ can be solved using kd-tree data structure with:

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

7.3.3 Performance in $\mathbb{R}^2 D$

The RR problem for a set P of n points in $\mathbb{R}^D$ can be solved using kd-tree data structure with:

- **Construction time:** $O(Dn \log n)$ to sort all the points by x and y
- **Space:** $O(Dn)$
- **Query time:** $O(Dn^{1-1/D} + k)$, where k is the number of reported points. This converge to linear time when D increases.

7.3.4 Kd-Tree for r-NNS

1. Let R_q = smallest square enclosing ball $B_r(q)$.
2. Compute $S =$ `SearchKdTree(T.root(),`R_q`)`.
3. Return a point $p \in S \cap B_r(q)$, if any, or null if $S \cap B_r(q) = \varnothing$.

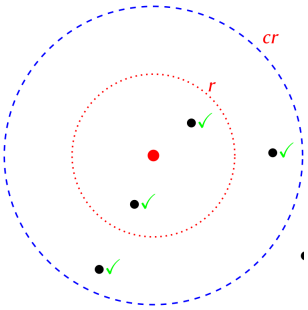
The solution works efficiently from a practical point of view.
Let $k_s = |S|$, and let $k_q = |S \cap B_r(q)|$. This solution requires $O(\sqrt{n} + k_s)$ and in some pathological cases, $k_s >> k_q$ and is unclear how to close the gap.

8 Similarity Search in High Dimensions

8.1 (c,r)-Approximate Near Neighbour Search (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 $c \geqslant 1$:

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



8.2 ANN with Locality Sensitive Hashing

Consider a metric space (M, d) and a family of hash functions

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

where S is a given domain. Fro any two points $p, d \in M$ denote by

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

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

8.3 (c, r, p_1, p_2)-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 (c, r, p_1, p_2)-locality sensitive if for any $p, q \in M$:

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

8.4 LSH for (c, r)-ANNS

A (c, r, p_1, p_2)-LSH $\mathcal{H}$ can be used for solving (c, r)-ANNS on the input set as follows:
Construction:

- Randomly select h from $\mathcal{H}$.
- Construct a hash table T with points in P using h: let $T[j]$ be the (potentially empty) bucket containing all points in P with hash value j.

Query:

- For a given query q, scan $T[h(q)]$ until a point p with $d(d, q) \leqslant cr$ is found, and return it. If there is no such point, return `null`.

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

- A near point p 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'* might end up in $T[h(q)]$, but only with probability at most p_2.

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

8.4.1 ANNS with LSH Performance

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

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

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

8.5 Improving the Data Structure

A family of $\mathcal{H}$ might not provide the guarantee that p_1 is close to 1 and p_2 close to 0.

8.5.1 Increase Collision Probability of Near Points

Idea: repeat search $\ell > 1$ times using ℓ distinct hast tables based on independent hash functions chosen uniformly at random from a $/c, r, p_1, p_2)$-locality sensitive family $\mathcal{H}$. This technique is called **repetition** or **OR construction**:

- The probability that a given near point p collide with the query point q in at least one hash table increases with ℓ.
- However, checking ℓ buckets, one for each hast table, becomes computationally expensive.

We want $\ell = \frac{1}{p_1}$ to be sure to have at least for sure one success.

8.5.2 Increase Collision Probability of Far Points

Idea: use a family $\mathcal{G}$ of hash functions obtained by concatenating $k \geqslant 1$ independent hash functions chosen uniformly at random from a (c, r, p_1, p_2)-locality sensitive family $\mathcal{H}$. Namely,

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

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

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

This technique is called **concatenation** or **AND construction**.
Suppose that we set

$$k = \log_{1/p_2} n = \frac{\log_2 n}{\log_2(1/p_2)}$$

Then, for a random $g \in \mathcal{G}$ and any two points p,q with $p \neq q$.

- If $d(p, q) \leqslant r$, then $Pr[g(p) = g(q)] \geqslant p_1^k = 1/n^\rho$.
- If $d(p, q) > cr$, the $Pr[g(p) = g(q)] \leqslant p_2^k = 1/n$.

Remark: with the above choice of k, the expected number of collisions of a query point q with far points in the same bucket is at most 1.

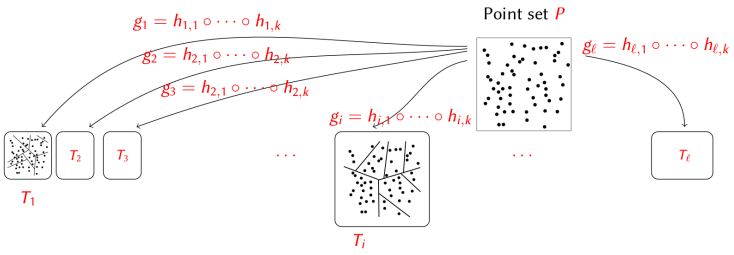
8.6 LSH and (c, r)-ANNS: A General Schema

We merge the two improvements to get the following schema.
Let $\mathcal{H}$ be a (c, r, P_1, p_2)-locality sensitive family of hash functions, and let k and ℓ be two values that will be set later.
Construction:

- Construct ℓ hash functions $g_1, \ldots, g_\ell$: each g_i consists of the concatenation of k hash functions randomly and independently selected from $\mathcal{H}$.
- For each g_i, construct a hash table T_i of points in P using g_i. We let $T_i[j]$ be the bucket containing all points in P with hash value j when using g_i.

Query q:

- Scan $T_1[g_1(q)], \ldots, T_\ell[g_\ell(q)]$ until a cr-near point p is found and return it, if no such point is found, return `null`.



8.6.1 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

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

Using the above approach to (c, r)-ANNS, a query is answered successfully with probability $\geqslant 1/2$. Moreover, the following performance is obtained:

- **Construction time:** $O(Dn^{1+\rho} \log_{1/p_2} n)$
- **Space:** $O(Dn + n^{1+\rho} \log_{1/p_2} n)$
- **Query time:** $O(Dn^\rho \log_{1/p_2} n)$

8.7 LSH for Hamming Distance

Let p be a D-dimensional boolean vector and let p_i its i-th bit. Consider the following family of hash functions:

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

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

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

For any two points p, q we have that:

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

$$\Pr_{h \in \mathcal{H}_H}[h(p) = h(q)] = 1 - \frac{d_H(p, q)}{D} \geqslant 1 - \frac{r}{D} = p_1$$

- If $d_H(p, q) > cr$, then

$$\Pr_{h \in \mathcal{H}_H}[h(p) = h(q)] = 1 - \frac{d_H(p, q)}{D} \leqslant 1 - \frac{cr}{D} = p_2$$

Therefore $\mathcal{H}_H$ is $(c, r, 1 - \frac{r}{D}, 1 - \frac{cr}{D})$-locality sensitive, which is effective when $p_1 \gg p_2$.
A parameter used to measure the quality of such a family is the ρ **factor** defined as:

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

which for the bit sampling is equal to $\frac{1}{c}$.

8.7.1 Concatenating Bit Sampling

Concatenating k bit sampling LSH consists in randomly selecting k indexes, and yields the following collision probabilities:

- $Pr_{h \in \mathcal{H}_H}[h(p) = h(q)] \geqslant (1 - r/D)^k$, if $d_H(p, q) \leqslant r$.
- $Pr_{h \in \mathcal{H}_H}[h(p) = h(q)] \geqslant (1 - cr/D)^k$, if $d_H(p, q) \geqslant cr$.

The concatenation does not change the ρ value.

8.8 LSH for Euclidean Distance

Consider points in $\mathbb{R}^D$. For a fixed value $w > 0$ define the family $\mathcal{H}_E$ of hash functions

$$h_{a,b}(p) = \lceil \frac{\langle a, p \rangle + b}{w} \rceil$$

where a and b are sampled as: $a \sim \mathcal{N}^D(0, 1)$ and $b \sim U[0, w]$.
There is a distinct hash function for every pair (a, b).
It can be shown that $\mathcal{H}_E$ is (c, r, p_1, p_2)-locality sensitive with $\rho = O(1/c)$.