Learning with Massive Data

Academic Year 2022/2023

Nicola Aggio 880008

Index

1 Introduction

1.1 Course structure

The **exam** is composed of the following tests:

- a **written exam**, containing about 6 questions or exercises and providing the most of the score of the final grade. It will be mainly composed of theoretical questions about notions discussed during the course, but it may also contain some exercise;
- **3 assignments**, mainly asking for implementing and evaluating a parallel algorithm (C++ or Python). The assignments can be delivered:
  - *during the course*, then if the assignment is insufficient, it can be re-submitted;
  - *with the written exam*, but only if the written exam is passed, and in this case if the assignment is insufficient, it cannot be re-submitted.

  In both cases, if the assignment is positively graduated, we get +1, 0 otherwise

1.2 Main topics

The **topics** of the course are:

- parallel programming (cache memory, thread-based and shared memory);
- parallel programming on multiple machines (Map-Reduce, Spark, distributed memory);
- visiting professor (ranking).

For many years, **Moore's Law** has been considered a strong argument against the concept of **parallelism**. Basically, in 1965 Moore said that "*The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. (...) there is no reason to believe it will not remain nearly constant for at least 10 years.*" However, the increase in power consumption of the machines, the overall DRAM access latency and the diminishing returns of more instruction-level parallelism resulted in denying the Moore's Law as a good argument against parallel programming. For this reason, in the last years we faced an **increase of the parallelism** (see also the birth of Deep Learning etc..). The main **advantages** of parallelism, or multi-core machines, are:

- *power*: many simple cores offer higher performances per unit area for parallel codes than a comparable design employing smaller numbers of complex cores;
- *design cost*: the behavior of a smaller, simpler processing element is much easier to design and to predict;
- *defect tolerance*: smaller processing elements provide an economical way to improve defect tolerance by providing redundancy.

In general, we can distinguish two types of computing:

- **sequential computing**: in this case the problem is solved with an algorithm whose instructions are executed *in sequence*, so the corresponding computational model is characterized by a *single processor*;
- **parallel computing**: in this case the problem is solved with an algorithm whose instructions are executed *in parallel*, so the corresponding computational model is characterized by *multiple processors* with a specific *cooperation mechanism*.

On the one hand, parallelism can be exploited with the goal of making the execution faster, but on the other it causes some issues, depending on the level at which it is applied:

- in multi-cores we have the problems of memory hierarchies, false sharing and synchronization;
- in distributed systems we have the problems of data distribution and fault tolerance;
- in GPUs we have the problems of explicit memory management and the impossibility of executing recursive algorithms.

An example of application in which parallel computing can be used is in the **PageRank** algorithm. This algorithm computes the relevance of a web page based on the link structure of the Web. Let W be the adjacency matrix of the Web graph, then $W[i,j]$ is the probability of a random user of going from page j to i, and it is define as $W[i,j] = \frac{1}{o(j)}$, where $o(j)$ is the number of outgoing links from j. In general, the PageRank π is the stable state distribution of the transition probability matrix W, and it can be computed as: $\pi_{t+1} = W\pi_t$. After a certain number of iterations, usually 50, the importance of a page becomes steady. Assuming that the number of pages is $N = 10^{10}$, then the calculation of the PageRank requires $10^{20} * 50$ floating point operations (each iteration requires N^2 multiplications). Assuming that a modern processor (10^{12} floating point operations per second) is used, then the total running time is $\frac{5*10^{21}}{10^{12}} = 5 * 10^9$ seconds, which is clearly unfeasible. Moreover, if the matrix W is stored in a sparse format, we can assume that an entire web graph requires 800GB, and reading 800GB at 200MB/s takes more than 1h, just for one iteration. In this sense, PageRank is unfeasible if parallelism is not implemented both on CPU and on disk storage.

2 Cache

2.1 Introduction

A significant characteristic of the hardware development during the last decades has been the increasing **gap** between **processor cycle time** and **main memory access time**. Example 1 (Memory latency): let's consider a processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns, and if we assume that processor has two ALU units and it is capable of executing two instructions in each cycle of 1 ns, then the peak processor rating is 2 GFLOPS (Giga Float.Pt. Ops per Sec). However, since the memory latency is equal to 100 cycles every time a memory request is made, the processor must wait 100 cycles before it can process the data.
To use processor cycles efficiently, a **memory hierarchy** is typically used, consisting of multiple levels of memories with different sizes and access times. Only the main memory on the top of the hierarchy is built from DRAM, the other levels use **SRAM** (static random access memory), and the resulting memories are often called **caches**. The goal in the design of a memory hierarchy is to be able to access a **large percentage of the data from a fast memory**, and only a small fraction of the data from the slow main memory, thus leading to a small average memory access time. The simplest form of a memory hierarchy is the use of a single cache between the processor and main memory (one-level cache, L1 cache), but nowadays two or three levels of cache are used for each processor. Note that for multiprocessor systems, there exists an additional problem, the so-called **cache coherence problem**, i.e. it must be ensured that a processor accessing a data element always accesses the most recently written value.
In general, the access times of caches are 0.5-2.5 ns, while for the DRAM are 50-70 ns. We can consider a cache system as the following:

Figure 1: Cache system

- the data needed by the processor is first fetched into the cache (cache hit or miss), then all subsequent accesses to data items residing in the cache are serviced by the cache. The **cache hit ratio** is defined as the fraction of memory references that are resolved by the cache memory;
- performance improves in presence of *high locality*:
  - *temporal locality*: a data item is re-used after a short amount of time;
  - *spatial locality*: data with close memory addresses is used within a short amount of time.
- data transport between cache and main memory is done by the transfer of blocks comprising several words, whereas the processor accesses single words in the cache.

Example 2 (Memory latency with cache): let's consider a processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns and suppose that we have a cache of 32KB, with a latency of 1 ns per word, and we consider the problem of multiplying two matrices A and B of size 32*32 (we suppose that A, B and $A*B$ fit in the cache). Then, the time needed to load A and B into the cache is $32*32*2*100ns = 205\mu s$. Then, multiplying two matrices of size n*n takes $2n^3$ multiply-adds, in our case $2*32^3 = 66K$ multiply-adds, which implies $66\mu s$. Finally, the total time is $205 + 66 = 271\mu s$, so the throughput is $\frac{66K*2}{271} = 488$ MFLOPS ($>$ 10 MFLOPS of the Example 1) (still $<$ 2 GFLOPS, which represents the peak processor rating). Moreover, we underline the fact that the locality is exploited by observing that the computational complexity of these operations is n^3, but here we are computing using n^2 memory locations!
Example 3 (Bandwidth with cache): let's consider the previous example. If the *cache block* has a width of *one single word*, then it takes $32 * 32 * 2 * 100ns = 205\mu s$ to load the two matrices in cache. However, if the width is of *four words*, then the overhead is $32 * 32 * 2/4 * 100ns = 51\mu s$, with a total time (load into cache + computation) of $51+66 = 117\mu s$. In this case, the throughput is $\frac{66K*2}{117} = 1282$ MFLOPS ($>$ 488 MFLOPS of Example 2) ($>$ 10 MFLOPS of Example 1) (still $<$ 2 GFLOPS of the peak processor rating).
Along with cache coherence problem, another important issue for the cache is the **cache-to-memory coherence**, i.e. the problem consisting on when to write to the memory data that have been modified in the cache. There exist two possible alternatives:

- **write-through policy**, i.e. data is immediately written to memory, so a downside is that the write operation is delayed, while the pro is that the memory is always consistent with the content of the cache;
- **write-back policy**, i.e. memory is updated upon eviction. In this case the two major downsides are that the content of the memory may differ from the content of the cache (inconsistency) and that the processor may not need anymore the data that is currently in the cache. An example of inconsistency is represented in the following picture: the processors see different values of u after the event 3.

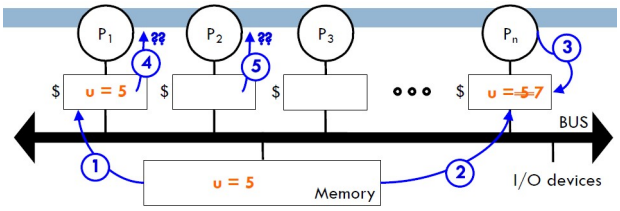


Figure 2: Inconsistency problem

  On the other hand, this policy results in having less writing operations;
- **snooping protocols**, most of which assumes that local caches of the processors use a write-through policy and that all memory accesses are performed via the central bus. In this way, each cache controller can observe all write accesses to perform updates or invalidations

Finally, one last example of important issue concerning cache is the **false sharing**, which happens when two *unrelated variables*, i.e. variables that are logically private to distinct

threads, are allocated in the same block and do not conflict. In this case, from the point of view of the data block the accesses are considered conflicts, even if the two processors access disjoint words of the same block, thus the cache updates are not necessary! In this sense, it is usually a good practice to put on the same block *related variables*.

2.2 Models

So far we've seen that the cache has a significant impact on the performance of modern applications, so now we focus both on the cache models and on some algorithms that exploit the cache use, in particular *matrix multiplication* and *sorting*.

2.2.1 External-Memory Model

Before analyzing the *cache-oblivious model*, we first review the standard model of a two-level memory hierarchy with block transfers. This model is known as **external-memory model**, and it defines the computer as having two levels, as represented in Picture 2.2.1:

- the *cache* which is near to the CPU, cheap to access but limited in space;
- the *disk* which is distant from the CPU, expensive to access but nearly limitless in space.

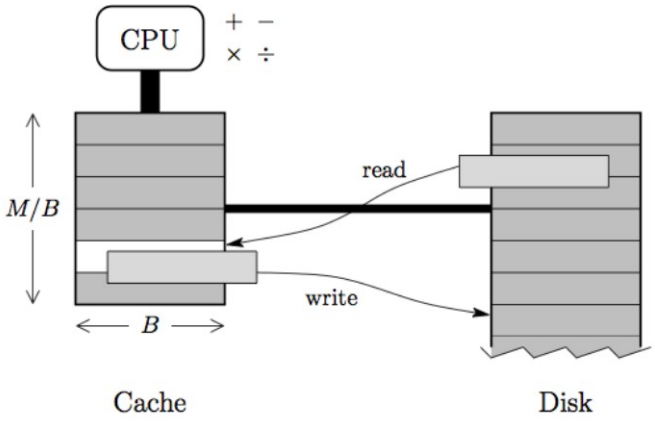


Figure 3: External-Memory model

The central aspect of this model is that transfers between *cache* and *disk* involve *blocks* of data: as we can see, each block has size B, while the *cache* has size $M \geq B^2$ and contains M/B entries. The main properties of this model are:

- it provides a simple description of the relationship between CPU, *cache* and *disk*;
- the complexity of the algorithms we will consider will be based on the number of *disk* accesses;
- the algorithms will be optimized for specific values of M and B;
- we assume that the cache is working in the best possible way, i.e. it always makes the best possible choices.

2.2.2 Cache-Oblivious Model

The other important cache model is the **cache-oblivious model**, and its basic idea is to design external-memory algorithms without knowing B and M. But this simple idea has several surprisingly powerful **consequences**:

- If a cache-oblivious algorithm performs well between two levels of the memory hierarchy (nominally called *cache* and *disk*), then it must automatically work well between any two adjacent levels of the memory hierarchy;
- If the number of memory transfers is optimal up to a constant factor between any two adjacent memory levels, then any weighted combination of these counts (with weights corresponding to the relative speeds of the memory levels) is also within a constant factor of optimal;
- The cache-oblivious model does not require to tune the cache parameters, an operation which make the code portability difficult: using this model, an algorithm should work well on all machines without modification.

In contrast to the external-memory model, algorithms in the cache-oblivious models **cannot explicitly manage the cache**, and this is necessary since both M and B are not known. Moreover, this model is based on several **assumptions**:

1. the *ideal cache model* assumes that the *page replacement* operation is *optimal*: in particular, it specifies that the page replacement strategy knows the future and always evicts the page that will be accessed farthest in the future. Real-world caches do not know the future, and employ more realistic page replacement strategies such as evicting the least-recently-used block (LRU) or evicting the oldest block ($FIFO$);
2. *Full associativity*, i.e. we assume that any block can be stored anywhere in the cache, in contrast with real-world caches that are characterized by *limited associativity*;
3. *Tall-cache assumption*: the cache is assumed to be tall, i.e. $M = \Omega(B^2)$ (usually a weaker condition is sufficient $M = \Omega(B^{1+\gamma})$, for any constant $\gamma > 0$). This property is particularly important in some of the more sophisticated cache-oblivious algorithms and data structures, where it ensures that the cache provides a polynomially large "buffer" for guessing the block size slightly wrong. It is also commonly assumed in external-memory algorithms.

While on the one hand these assumptions are really strong, the following theorems make the model described above more feasible to be applied in order to deal with real-world problems, with the running time that degrades only by a constant factor:

- **Lemma 1**: if an algorithm makes T memory transfers on a cache of size $M/2$ with optimal replacement, then it makes at most $2T$ memory transfers on a cache of size M with LRU or FIFO replacement. This means that LRU and FIFO replacement do just as well as optimal replacement up to a constant factor of memory transfers and up a constant factor wastage of the cache;
- **Lemma 2**: for some constant $\alpha > 0$, an LRU cache of size αM and block size B can be simulated in M space such that an access to a block takes $O(1)$ expected time. This result can be reached by using 2-universal has functions and by knowing both B and M.

By this two theorems we cal conclude that a cache-oblivious model can be translated into a FIFO/LRU cache with 1-associativity paying only constant factors.

2.3 Algorithms

We now analyze some important techniques for designing **cache-oblivious algorithms**.

2.3.1 Scanning

The first problem we address is the problem of *scanning*, in which our goal is to traverse all the elements of a set. On a flat memory hierarchy (uniform-cost RAM), such a procedure requires $\Theta(N)$ time for N elements. In the external-memory model, if we store the elements in $\lceil N/B \rceil$ blocks of size B, then the number of blocks transfers is $\lceil N/B \rceil$.
To achieve a similar bound in the cache-oblivious model, we can lay out the elements of the set in a contiguous segment of memory, in any order, and implement the N-element traversal by scanning the elements one-by-one in the order they are stored: this layout and traversal algorithm do not require knowledge of B (or M), and it is represented in Picture 2.3.1.

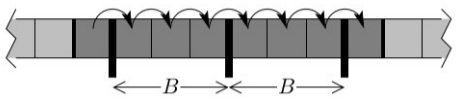


Figure 4: Scanning problem in cache-oblivious model

The overall complexity using the cache-oblivious model is then $O(\lceil N/B \rceil) + 1$: as we can see, the cache-oblivious bound is an additive 1 away from the external-memory bound, and this is ideal, because normally our goal is to match bounds within multiplicative constant factors.

2.3.2 Divide and Conquer

After scanning, the first major technique for designing cache-oblivious algorithms is *divide and conquer*, which often leads to algorithms whose memory-transfers count is optimal within a constant factor.
The basic idea of this approach is that it **repeatedly refines the problem size**, until it will eventually **fit in the cache** (i.e. the size of the problem will be at most M) and, later, in a **single block** (i.e. the size will be at most B). The most important **property** of this approach is that if the number of leaves in the recursion tree is polynomially larger than the divide/merge cost, then the corresponding algorithm will use a number of memory transfers which is optimal within a constant factor.
We will now focus on a specific example of this approach, the *matrix multiplication* problem.
Matrix multiplication
The problem of matrix multiplication consists of computing $C = A \cdot B$, where A and B being $N * N$ matrices. The first issue when considering this problem relies on how to store the matrices, and the two possibilities are (Picture 2.3.2):

- *row-major order*:in this case the elements of the matrix are stored by rows, so it is the preferable organization to store matrix A;
- *column-major order*: in this case the elements of the matrix are stores by columns, so it is preferable organization to store matrix B.

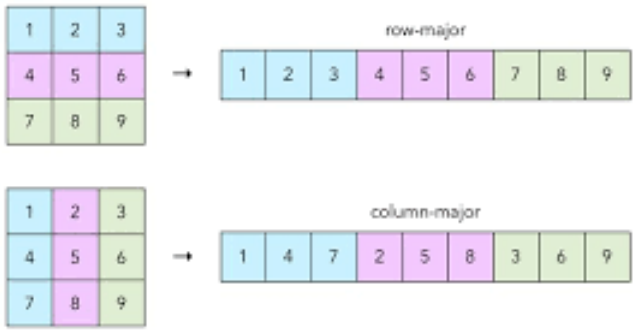


Figure 5: Row-major order and column-major order representation

We can now take into considerations some algorithms to solve this problem.
Algorithm 1 (naive): this is the simplest algorithm, in which each element C_{ij} is computed by scanning the i-th row of A and the j-th column of B, where A is stored in row-major order and B is stored in column-major order. In this case, the **number of memory transfers** is given by:

$$O(N^3/B + N^2)$$

, since each element of C involves a linear scanning of the row of A and the column of B, i.e. $O(N/B + 1)$, and there are N^2 elements in C. A possible approach to reduce the cost of this algorithm consists of storing the row i of A in cache memory (if $M > N$), or to keep the matrix A in cache (if $M > N^2$).
Algorithm 2 (external-memory model): this algorithms is based on the idea of multiplying the sub-matrices of A, B and C, and the **optimal number of transfers** is $O(N^2/B + N^3/(B * \sqrt{M}))$
An important issue about this algorithm is the choice of the block size, and the optimal choice is to choose a block size s such that $3 * s^2 = M$, i.e. such that the cache can contain a block from A and B, and that can store the result block of C.
Algorithm 3 (cache oblivious): this algorithm exploits the *divide-and-conquer* approach by recursively dividing the original matrix into sub-matrices until they fit into the cache, as represented in Picture 2.3.2.
Since we do not know in advance when a sub-matrix will fit into the cache, we need a recursive data layout such that however we recursively split the matrix, at some point all the data will be in (almost) consecutive memory locations that can be easily loaded in cache. An example of this organization is the *Z-order representation*.
The overall **number of transfers** of this algorithm is given by:

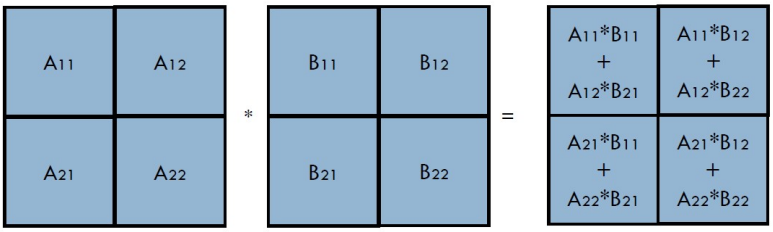


Figure 6: Divide-and-conquer approach for the matrix multiplication problem

$$O(N^2/B + N^3/(B\sqrt{M}))$$

, so it has the same complexity of the external-memory model case.
This particular number of transfers is computed by solving the following recurrence:

$$T(n) = 8T(N/2) + O(1 + N^2/B)$$

, where:

- $8T(N/2)$ represents the number of transfers of each of the eight multiplication sub-problems;
- $O(1 + N^2/B)$ represents the number of transfers for each of the four addition sub-problems

2.3.3 Sorting

We now focus on the sorting problem.
Merge-Sort (external-memory model): this algorithm works by recursively splitting the input vector into smaller sub-vectors (*split phase*), by sorting them and by merging them together into the result sorted vector (*merge phase*). We can easily notice that the main operation of the algorithm is represented by the merging phase.
The best way of implementing this algorithm using an external-memory model is by using the (M/B)-way mergesort. In this case, during the merge each memory block maintains the first B elements of each list, and when a block empties, the next block from that list is loaded. It can be shown this algorithm results in the following number of block transfers:

$$\Theta(N/B \log_{M/B}(N/B))$$

This number is computed by taking into account these informations:

- the recursion tree has $\Theta(N/B)$ leaves;
- the leaf cost is $\Theta(N/B)$;
- the number of levels in the recursion tree is $\log_{M/B} N$

Merge-Sort (cache oblivious): the goal of this algorithm is to run the merge phase, the most expensive one, in cache, for any value of M and B. In particular, the provided solution consists of implementing a 2-way mergesort, whose complexity is:

$$\Theta(N/B \log_2(N/M))$$

Our next goal is then to find a method in order to improve this naive algorithm, in particular by rising the base of the logarithm in order to reach M/B. We will reach this goal by using the **k-funnel** and the **funnel-sort algorithm**.
Funnelsort (cache oblivious): the core of this algorithm is characterized by the **k-funnel**, which is represented in Picture 2.3.3.

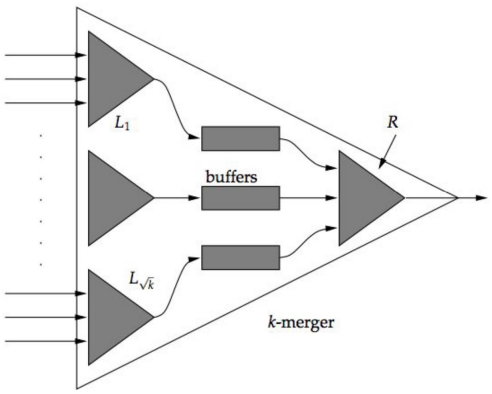


Figure 7: k-funnel

A *k-funnel* is a merger that merges k sorted lists with total size $k^3 = N$ in a space k^2 (note that the cache locality is given by the $k^2 < k^3$ mismatch). More specifically, the *k-funnel* receives in input k sorted lists, that are partitioned into $\sqrt{k}$ groups of size $\sqrt{k}$: each group feeds a $\sqrt{k}$-funnel L_i. The output of each L_i is stored in a buffer (FIFO) of size $2k^{3/2} = 2\sqrt{N}$: finally, the $\sqrt{k}$ buffers feed the $\sqrt{k}$-funnel R whose output is the output of the *k-funnel*. An important property of this merger is that if R has buffer i with less than $k^{3/2}$, it recursively invoke L_i.
The sizes of the *k-funnel* are the following:

- the size of each L_i is $\sqrt{k} * \sqrt{k} = k$;
- R has size k as well;
- buffers has size $2 * k^{3/2}$

, so the total size of the merger is given by $(\sqrt{k}+1)$ (number of funnels + R) + $\sqrt{k}*2*k^{3/2}$, which leads to $\Theta(k^2)$. The *funnelsort* algorithm is the first application of the tall-cache assumption: for simplicity, we assume that $M = \Omega(B^2)$, but the same result can be obtained when $M = \Omega(B^{1+\gamma})$. Clearly, the crucial issue of this algorithm relies on the choice of the value of k: the larger the k, the faster the algorithm; however, k-funnel is fast only if it is fed at least K^3 elements. Thus, $k = N^{1/3}$ is chosen. The algorithm proceeds as follows:

1. Split the array into $k = N^{1/3}$ contiguous segments, each of size $N/k = N^{2/3}$;
2. Recursively sort each segment;
3. Apply the k-funnel to merge the sorted segments.

It can be proven that the **number of transfers** using the funnelsort algorithm is:

$$(N/B \log_{M/B}(N/B))$$

, i.e. it is the same as the mergesort using the external-memory model.
Example: suppose that $N = 4,096 = 2^{12}$, then the operations of the funnelsort algorithm are:

1. Split the array into $k = N^{1/3} = 2^4 = 16$ blocks, each of size $N/k = 2^8 = 256$;
2. Sort each block independently;
3. Merge the blocks with a $2^4 = 16$-funnel.

Picture 2.3.3 represents the structure of the funnels, while Picture 2.3.3 shows the size of each component: note that each buffer has size $2 * k^{3/2} = 2 * 2^6 = 2^7 = 128$, and we recall that if R has buffer i with less than $k^{3/2} = 64$, it recursively invoke L_i.

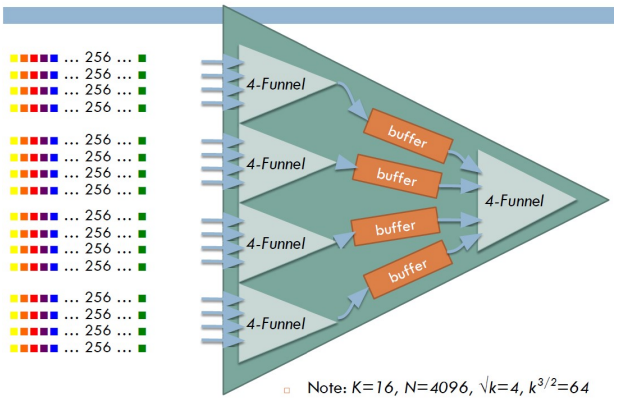


Figure 8: Example of 16-funnel: each buffer has size 128, while R has the same size of the funnel, i.e. 16

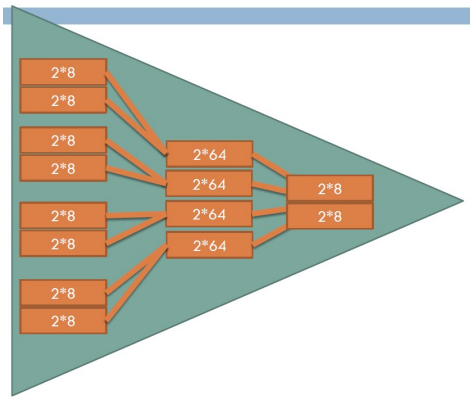


Figure 9: Sizes of the 16-funnel

Finally, another important issue about funnelsort is how to efficiently store a k-funnel. One possible solution could be to use a recursive layout, as represented in Picture 2.3.3.

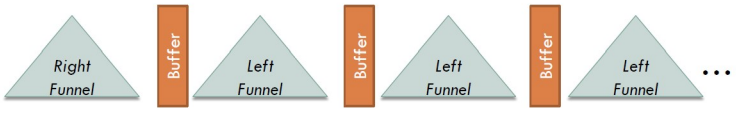


Figure 10: Recursive layout for a k-funnel

However, another possible approach could be to use the so called **lazy k-funnel**, which is represented as a binary tree of buffers (Picture 2.3.3), characterized by the following sizes:

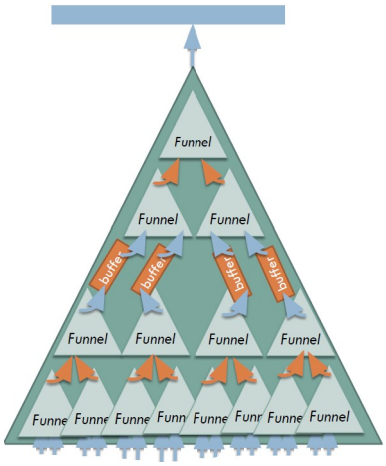


Figure 11: Lazy k-funnel

- the middle layer contains $2^{\lceil \log(k)/2 \rceil}$ buffers, i.e. about $\sqrt{k}$ (in the picture, $k = 16$, so 4 buffers);
- each buffer has size $\lceil k^{3/2} \rceil$ (in the picture, the size is 64);
- the final buffer, containing the sorted elements, has size $k^3 = N$ (in the picture, $N = 2^{12}$).

One peculiarity of this lazy k-funnel is the FILL procedure, through which the buffers are filled, and it is defined as:

1. While the buffer is not full:
   (a) If the left child is empty, then fill it;
   (b) If the right child is empty, then fill it;
   (c) Perform one merge step

2.4 Multicore Hierarchies: Key Challenge

A crucial issue about the theory underlying the ideal cache model is that it falls apart once we introduce **parallelism**, i.e. good performances for any M ans B on a 2-level hierarchy do not imply good performances at all levels of hierarchy. This is mainly due

to the fact that the **caches are not fully shared**, and for this reason the **scheduling** of **parallel threads** has a **large impact** on cache **performances**. Finally, the best technique to deal with caches and threads is to share a largely overlapping working set.

2.5 Other approaches

Among other approaches for hiding the memory latency we can distinguish:

- **multi-threading**, which consists in splitting the problem into multiple sub-problems and in running an independent thread for each sub-problem. Note that when a thread is idle on a miss, another thread can execute computational tasks;
- **pre-fetching**, which consists in anticipating load operations, so that data is already available when needed;
- **drawbacks**, which impact both on bandwidth and on cache pollution.

3 Threads

In general, when we deal with parallel programs, the total CPU time spent by a parallel implementation is larger than a serial one, because parallelism involves an additional overhead caused by:

- Synchronization and communication between threads;
- Exchange of data between threads;
- Load imbalance, i.e. threads that deal with smaller problems remain idle until other threads deal with bigger ones.

3.1 Evaluation metrics

Moreover, in general, evaluating a parallel algorithm is difficult, since its performances may depend on the architecture, on the network etc.. Some simple measures are:

- The **parallel runtime** $T_p(n)$, where p is the number of parallel units (cores), and n is the problem size;
- The **cost** $C_p(n) = pT_p(n)$, which represents the total amount of work that is performed.

A parallel algorithm is said to be **cost optimal** if

$$C_p(n) = pT^*(n)$$

, where $T^*(n)$ is the runtime of the fastest sequential algorithm for the given problem. In this sense, a parallel algorithm is cost-optimal if its cost has the same asymptotic growth as the fastest serial algorithm (as a function of the input size).
Another important metric for comparing sequential and parallel algorithms is the **speedup**, which is defined as:

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

Theoretically, $S \leq p$, but in practice $S < p$, i.e. we have a *sublinear speedup*, mainly due to the overheads we discussed before. Picture 3.1 represents the speedup w.r.t to the number of processors.
The goal when we implement a parallel algorithm is to measure a *linear speedup*, i.e. $S = p$, and it is even possible to have *superlinear speedup* because of cache sharing.
An alternative measure is **efficiency**:

$$E_p(n) = \frac{S(n)}{p} = \frac{T^*(n)}{pT_p(n)}$$

, and it is a measure of resource usage: it represents the fraction of time the processors are fully used to execute a fraction $1/p$ of the best sequential algorithm. We have that $E = 1$ for linear speedup.
When we deal with the evaluation of parallel algorithms, we have to consider two very important laws: the Amdahl's law and the Gustafson's law. The **Amdahl's law** regards

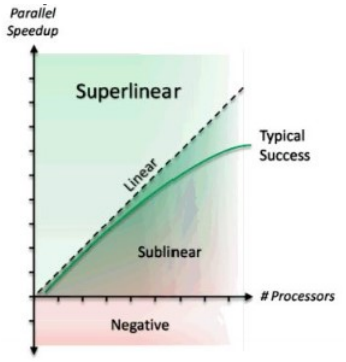


Figure 12: Speedup vs number of processors

speedup and says that the parallel execution time T_p cannot be arbitrarily reduced by increasing p, i.e. the number of parallel units. If we suppose that a fraction f of the computation cannot be executed in parallel (for example because of some dependencies), the have that:

$$S_p(n) = \frac{T_{\text{seq}}}{T_{\text{par}}} = \frac{T_{\text{seq}}}{(f * T_{\text{seq}} + (1-f) * \frac{T_{\text{seq}}}{p})} = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

, i.e. the sequential fraction f is an upper bound of the speedup $S_p(n)$.
The **Gustafson's law** on the other hand regards scalability, and in particular it studies the behaviour of the algorithm when $n \to \infty$. In general, when we have a large amounts of data we would like to use a large number of processors. Suppose that the sequential part c of an algorithm is constant w.r.t. to n. Then, let $T(n, p)$ be the execution time of the parallelizable part over p processors: we define the **scaled speedup** as:

$$zS_p(n) = \frac{c + T_1(n)}{c + T_p(n)} = \frac{c + T_1(n)}{c + \frac{T_1(n)}{p}} = \frac{\frac{c}{T_1(n)} + 1}{\frac{c}{T_1(n)} + \frac{1}{p}}$$

When $n \to \infty$, we have:

$$\lim_{n \to \infty} S_p(n) = p$$

In general, the **scalability** of a parallel system is its ability to increase the speedup in proportion to the number of processors: scalable algorithms are characterized by a constant efficiency when increasing both the number of processors and the problem size.

3.2 Shared-memory programming models

- **Process based models** assume that all data associated with a process is private by default, unless otherwise specified. Moreover, distinct page tables exist for each process in order to map the virtual addresses to distinct physical memory addresses. In this case, processes are units of resource ownership, i.e. they have their own program counter, heap memory etc.. Note that process can include more threads;
- **Thread based models** assume that all memory is global, i.e. the threads share the same address space (same page table), and there's at least one thread per process.

Note that in this case the communication between the threads is far more easy than in the process based model;

- **Directive based models**: in this case the concurrency is specified in terms of high-level compiler directives, resulting in a sort of "annotated" source code. An example is given by the OpenMP library.

3.3 pthreads

In general, each thread has a separate execution flow, and it is the owner of a private program counter, stack memory, stack pointer etc.. , and they're also known as *lightweight processes*. In this course we analyze the **pthreads** library (POSIX threads).
An example of the usage of the threads can be viewed in the matrix multiplication problem, as represented in Picture 3.3.

```
for (row = 0; row < n; row++)
  for (column = 0; column < n; column++)
     c[row][column] =
        new_thread( dot_product( get_row(a, row),
                                 get_col(b, col)) );
```

Figure 13: Matrix multiplication with threads

In this case, we can consider a thread as the parallel asynchronous execution of a function with its parameters. In the *pthreads* library, all the threads in a process are peers, and there's no explicit parent-child model, with the exception of the "main thread" that holds the process information. The main advantages over the processes are that threads can read/write to shared variables for communication, and that the context-switch operation is faster. The life cycle of a thread is represented in Picture 3.3.

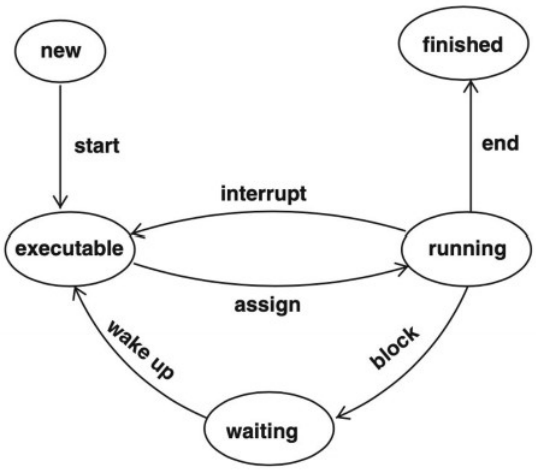


Figure 14: Life cycle of a thread

An example of code that exploits the threads is shown in Picture 3.3.

- A vector of *std::thread* objects is created. Notice that each thread object must be associated to a single actual thread;

```cpp
#include <thread>
#include <iostream>
#include <vector>

void hello(void) {
    std::cout << "Hello World!" << std::endl;
}

int main(int argc, char **argv) {
    if (argc!=2) {
      std::cout << " Usage: " << argv[0]
                << " <num_threads>" << std::endl;
      std::exit(EXIT_FAILURE);
     }

    const int NUM_THREADS = atoi(argv[1]);

    std::vector<std::thread> threads(NUM_THREADS);

    // start threads
    for (int i = 0; i < NUM_THREADS; i++)
      threads[i] = std::thread(hello);
    // wait for completion
    for (int i = 0; i < NUM_THREADS; i++)
      threads[i].join();
}
```

Figure 15: Example of usage of threads

- In the first for loop, each actual thread is created by passing the function *hello*, which has no parameters;
- The second for loop is used to wait each thread to complete the execution, and this is done by the *join* function.

In general, the class *std::thread* is used to represent individual threads of execution: a thread is joinable, and it has a unique thread id. As we said before, a non-initialized thread does not represent a thread. The typical contructor for a thread is *thread (function, arg1, arg2, .., argn)*, where the new thread of execution calls *function* passing *arg1, arg2, .., argn* as arguments. As we said before, no two *std::threads* objects may represent the same thread of execution!
Other important functions are:

- *void join()*, which returns when the thread execution has completed, and after a call to this function, the thread object becomes non-joinable;
- *void detach()*, which separates the thread of execution from the thread object, while the underlying thread executes independently;
- *hardware_concurrency()* returns the number of hardware thread contexts;
- *void this_thread::yield()*, which suggests the OS to re-schedule so as to allow other threads to be executed;
- *void this_thread::sleep_for(duration)*, which stops the execution of the thread for duration, and the actual restart depends on OS scheduling decisions;
- *void this_thread_sleep_until(time_point)*, which stops the execution until time_point is reached. As before, the actual restart depends on OS scheduling decisions.

Example (PI computation): the formula for deriving the value of π can be approximated as:

$$\pi = \frac{1}{n}\sum_{i=1}^{n}\frac{4}{1+(\frac{i-0.5}{n})^2}$$

, the larger the n, the better the precision in the calculation of π. The idea to parallelize this computation is to split the work among the threads, in particular by assigning each thread a term of the summation, and then sum them up. Picture 3.3 shows the function that is called for each thread, while the others show the main.

```
void PIworker(int id, int n, int skip, double* result) {
   double d = 1.0/(double)n;
   double s = 0.0;
   double x = 0.0;
   for (int i=id+1; i<=n; i+=skip) {
      x = (i-0.5)*d;
      s += 4.0/(1.0+x*x);
   }
   *result = d*s;
}
```

Figure 16: Function than computes the π

```
int main(int argc, char **argv) {
   if (argc!=3) {
      std::cout << " Usage: " << argv[0]
         << " <num_threads> <num_steps>"
         << std::endl;
      std::exit(EXIT_FAILURE);
   }
   std::cout << "No. of available cores : "
      << std::thread::hardware_concurrency()
      << std::endl;

   const int NUM_THREADS = atoi(argv[1]);
   const int STEPS = atoi(argv[2]);


   std::vector<std::thread> threads(NUM_THREADS);
   std::vector<double> partial_pi(NUM_THREADS);

   std::cout << "Starting..." << std::endl;
   auto start =
std::chrono::high_resolution_clock::now();

   for (int i = 0; i < NUM_THREADS; i++) {
      threads[i] = std::thread( PIworker, i, STEPS,
            NUM_THREADS, &partial_pi[i]);
   }



std::cout << "Waiting..." << std::endl;
for (int i = 0; i < NUM_THREADS; i++)
    threads[i].join();

double pi = std::accumulate(partial_pi.begin(), partial_pi.end(),
                0.0);

auto end = std::chrono::high_resolution_clock::now();
auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>
                        (end - start);

std::cout << " PI: " << pi << std::endl;
std::cout << "Elapsed: " << elapsed.count() << " ms." << std::endl;
return 0;
}
```

As regards the *PIworker* function:

- the *int skip* parameter determines the distance between the work of a thread and the work of another thread, since the work of the threads is interleaved;

- the result is stored into a variable, so it is not returned.

As regards the main:

- we read from the terminal the number of threads and the value for the *skip* parameter;
- a vector of threads is created: each thread writes in different part of the memory;
- a vector of double is created, to store the partial results.

3.4 Mutex

When multiple threads are manipulating the same data, results can be incoherent: we call *critical section* the portion of code where a race condition occurs, and the idea of the *mutual exclusion* is that at any point of time, only one thread can be in the critical section. In this sense, the *mutex* is an object that prevents other threads with the same protection from executing concurrently and access the same memory locations. The main functions are:

- *void lock()*, to get the lock;
- *void unlock()*, to release the lock;
- *bool try_lock()*, returns True if the lock was acquired;
- *std::recursive_mutex*, allows a thread to lock the same mutex multiple times;
- *std::timed_mutex*.

Example (Consumer/Producer): the scheme of the consumer/producer problem is shown in Picture 3.4. IN particular, the producer puts an item in the queue for 10 times, while the producer gets an item from the queue only it is not empty, again for 10 times. As we can see, the queue represents a critical section, since both the consumer and the producer access to it.

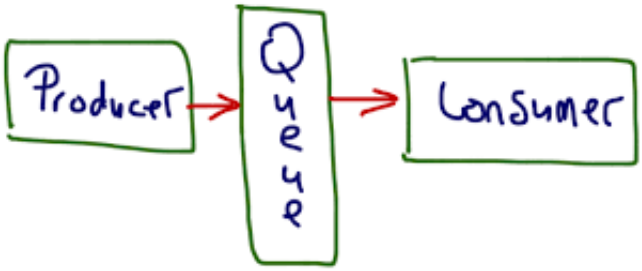


Figure 17: Consumer/Producer

Picture 3.4 shows the main function to resolve the Consumer/Producer problem, while Picture 3.4 and 3.4 shows the consumer and producer functions.
For what regards the main:

```
std::queue<std::thread::id> shared_queue;
std::mutex shared_queue_mutex;

int main(int argc, char **argv) {
    ..........
    std::vector<std::thread> producers(n_prod);
    std::vector<std::thread> consumers(n_cons);
    for (int i = 0; i < n_prod; i++)
        producers[i] = std::thread(producer, rand()%1000, 10);
    for (int i = 0; i < n_cons; i++)
        consumers[i] = std::thread(consumer, rand()%1000, 10);
    for (int i = 0; i < n_prod; i++)
        producers[i].join();
    for (int i = 0; i < n_cons; i++)
        consumers[i].join();
    ..........
```

```
void consumer(int time_msec, int runs) {
    std::thread::id my_id = std::this_thread::get_id();
    for (int i=0; i<runs; i++) {
        std::thread::id task;
        bool task_taken = false;
        // access the queue and add an item to it
        while (!task_taken) {
            shared_queue_mutex.lock();
            if (shared_queue.size()>0) {
                task = shared_queue.front();
                shared_queue.pop();
                task_taken = true;
                std::cout << "Consuming " << task << std::endl;
            }
            shared_queue_mutex.unlock();
        }
        // consume
        std::this_thread::sleep_for(std::chrono::milliseconds(time_msec));
    }
}
```

```
    void producer(int time_msec, int runs) {
        std::thread::id my_id = std::this_thread::get_id();
        for (int i=0; i<runs; i++) {
            // produce
            std::this_thread::sleep_for(
                std::chrono::milliseconds(time_msec)   );
            // access the queue and add an item to it
            shared_queue_mutex.lock();
            shared_queue.push(my_id);
            std::cout << "PROD " << my_id
                << " added an item to the queue" << std::endl;
            shared_queue_mutex.unlock();
        }
    }
```

- we see that both a queue of thread IDs and a mutex are defined;
- a vector of threads is created, both for consumer and for producer: each consumer and each producer executes the respective function, then they're joined.

For what regards the producer, we see that before adding an item to the queue, the mutex is locked; then, after the writing, the mutex is unlocked, otherwise a deadlock would result. For what regards the consumer, we see that before reading the content of the queue, the mutex is locked, then if the queue is not empty, the value is read. Finally, the mutex is unlocked.
However, usually the mutexes are not used directly, but through:

- *lock_ guard(mutex)*, which allows to acquire a given mutex when created, and to release it when the end of the scope is reached;

- *scoped_ lock*, which is similar to the previous one, but the locks are acquired and released on multiple mutexes.

, which allows easier handling and error free in case of exceptions or other issues. Moreover, the function *std::call_ once(flag, callable, arg1..)* is used to make sure that a given function is executed only once even if invoked by multiple threads. The arguments are:

- *flag*, which should be an instance of *std::once_flag*;
- *callable*, which is a callable object, i.e. a function or anything supporting the () operator;
- *arg1,..* are the arguments passed to the *callable*.

Picture 3.4 and 3.4 shows the revised implementation of the consumer and producer functions adopting the functions defined above.

```
void producer(int time_msec, int runs) {
    std::thread::id my_id = std::this_thread::get_id();
    for (int i=0; i<runs; i++) {
        // produce
        std::this_thread::sleep_for(
            std::chrono::milliseconds(time_msec)   );
        // access the queue and add an item to it
        std::lock_guard<std::mutex> guard(shared_queue_mutex);
        // shared_queue_mutex.lock();
        shared_queue.push(my_id);
        std::cout << "PROD " << my_id
            << " added an item to the queue" << std::endl;
        // shared_queue_mutex.unlock();
    }
}
```

```
void consumer(int time_msec, int runs) {
    std::thread::id my_id = std::this_thread::get_id();
    for (int i=0; i<runs; i++) {
        std::thread::id task;
        bool task_taken = false;
        // access the queue and add an item to it
        while (!task_taken) {
        std::lock_guard<std::mutex> guard(shared_queue_mutex);
        // shared_queue_mutex.lock();
            if (shared_queue.size()>0) {
                task = shared_queue.front();
                shared_queue.pop();
                task_taken = true;
                std::cout << "Consuming " << task << std::endl;
            }
            // shared_queue_mutex.unlock();
        }
        // consume
        std::this_thread::sleep_for(std::chrono::milliseconds(time_msec));
    }
}
```

As we can see, the producer initialize a *lock_ guard* passing the mutex as a parameter, and in this way the mutex is automatically locked and unlocked, so the *lock()* and *unlock()*

functions are not needed anymore. On the other hand, the consumer does the same thing, so again the *lock()* and *unlock()* functions are not used.
Another useful tool that can be used are the **condition variables**, which allow a thread to block itself until specified data reaches a predefined state, which is checked by a predicate. In this sense, a condition variable can be thought as a notification system on the status of a variable, and it always has a mutex associated to it. The functioning is the following:

1. A thread checks some data: if the data is ok it does some work, otherwise it waits;
2. At some point the thread is awakened: it must re-check the data, since other threads may have been waiting for the same event and they may have already changed the status of the data.
3. When a different thread modifies the data and reaches the desired status, it signals all the waiting threads.

Notice that the shared data is accessed in mutual exclusion. If we consider the consumer/producer problem, we have:

- the producer acquires a mutex, modifies the data and *notify_ one* or *notify_ all* on the condition variable, to wake up one or all threads on waiting;
- the consumer acquires s unique lock and executes the *wait*, *wait_ for* or *wait_ until*. Then, the thread is awakened, it checks the condition and accesses the data.

Picture 3.4 and 3.4 shows the functions of the consumer and producer using the condition variable.

```
std::queue<std::thread::id> shared_queue;
std::condition_variable shared_queue_cv;
std::mutex shared_queue_mutex;

void producer(int time_msec, int runs) {
   std::thread::id my_id = std::this_thread::get_id();
   for (int i=0; i<runs; i++) {    // produce
   std::this_thread::sleep_for(std::chrono::milliseconds(time_msec));

    // access the queue and add an item to it
    std::lock_guard<std::mutex> guard(shared_queue_mutex);

    shared_queue.push(my_id);
    std::cout << "PROD " << my_id
          << " added an item to the queue" << std::endl;

    // notify other thread data is ready
    shared_queue_cv.notify_all();
   }
}
```

As we can see, the condition variable *shared_ queue_ cv* is created, and the producer notifies all the treads after writing in the queue. On the other hand, the consumer waits until the queue is not empty, and then it consumes the produced data. Notice that the while loop is very useful since many consumers may be waken up, so the loop is used to check whether some other consumer consumed the produced data. Moreover, we notice that the consumer waits on a unique lock.
Finally, other important tools are **atomic** and **promise/future**. The **atomic** creates a new variable whose modification and access does not cause data races. The **promise/future** tool allows to implement the communication between threads in a convenient way. An example is showed in Picture 3.4 and 3.4.

```
void consumer(int time_msec, int runs) {
   std::thread::id my_id = std::this_thread::get_id();
   for (int i=0; i<runs; i++) {
    std::thread::id task;

    // access the queue and add an item to it      std::unique_lock<std::mutex>
   guard(shared_queue_mutex);
    while (shared_queue.size()==0)
        shared_queue_cv.wait(guard);

    task = shared_queue.front();
    shared_queue.pop();

    std::cout << "CONS " << my_id
        << " removed item "<< task <<" from the queue." << std::endl;

    // consume
   std::this_thread::sleep_for(std::chrono::milliseconds(time_msec));
   }
}
```

```
void PIworker(int id, int n, int skip,
        std::promise<double> sum) {
   double d = 1.0/(double)n;
   double s = 0.0;
   double x = 0.0;
   for (int i=id+1; i<=n; i+=skip) {
      x = (i-0.5)*d;
      s += 4.0/(1.0+x*x);
   }
   sum.set_value( d*s );
}

std::vector<std::thread> threads(NUM_THREADS);
std::vector<std::future<double>> partial_pi(NUM_THREADS);

for (int i = 0; i < NUM_THREADS; i++) {
    std::promise<double> promise;
    partial_pi[i] = promise.get_future();
    threads[i] = std::thread(PIworker, i, STEPS, NUM_THREADS,
                  std::move(promise) );
}
double pi = 0;
for (int i=0; i<NUM_THREADS; i++)
    pi += partial_pi[i].get();

for (int i=0; i<NUM_THREADS; i++)
    threads[i].join();
```

Another way to launch a thread is by using the *std::threads(std::launch policy, Function f, arg1,..)*, which runs the given function f asynchronously and returns a *std::future* that will eventually hold the result of that function call. The *policy* can be:

- *async*, i.e. it creates a new thread that executes f immediately;
- *deferred*, i.e. the task is executed on the calling thread the first time its result (future) is requested

3.5 OpenMP

OpenMP is a very useful library for parallelizing some common patterns, e.g. for-loop, and it works by writing some instructions for the compiler, that automatically parallelizes the operations we indicate. In this sense, it provides the programmer a higher level of abstraction than pthreads, and it supports:

- Parallel execution;
- Parallel loops;
- Critical sections;
- etc..

The OpenMP **directives** are based on the #**pragma** compiler directives, and they consist of a directive name followed by some clauses, i.e. #**pragma omp** ***directive*** **[clause list]**. The OpenMP programs execute serially until they encounter the *parallel* directive, which creates a team of threads that execute in parallel the given block. The main thread that encounters the parallel directive becomes the *master* of the team of threads, and it is assigned with the thread id 0 within the group. We notice that despite executing in parallel the portion of code, no information is provided about the first thread that executes, the second etc.., i.e. the execution flow of each thread is not known.

3.5.1 Basic clauses

The **basis clauses** of the OpenMP library are:

- *num_threads (int)*, that specifies the degree of concurrency, i.e. the number of threads that are created;
- *if (scalar expression)*, that specifies the conditional parallelization, i.e. whether the parallel construct results in creation of not. If the expression evaluates false, the only one existing thread executes the following instruction block.

An example of the usage of this bases clauses is provided in Picture 3.5.1.

```
bool go_parallel = true;
#pragma omp parallel if(go_parallel) num_threads(10)
{
   cout << "Hello World!" << endl;
}
```

3.5.2 Variable sharing clauses

In OpenMP we can specify the level at which the variables of the code are shared among the threads that are created (in the following list, x refers to a variable outside the block which can be accessed by the threads):

- **private(x)**: in this case each thread has his own copy of x; x is not initialized;

- **shared(x)**: in this case every thread accesses the same memory location, so it introduces race condition;
- **firstprivate(x)**: in this case each thread has his own copy of x, and x is initialized with the current value of x before the various threads start;
- **default (shared/none)**: affects all the variables not specified in other clauses

An example of the functioning of these clauses is provide in Picture 3.5.2: as we can see, the number of threads is 10, and the variable a is accessed as private, b and d are shared, while c is firstprivate. In this sense, we see that the value of a and c at the end of the block are 1, since the increments are done in the private copies of the threads, while the values of b and d are 11, since they're incremented by each of the 10 threads.

Source:

```
bool go_parallel = true;
int a=1, b=1, c=1, d=1;
#pragma omp parallel if(go_parallel) num_threads(10) \
        private(a) shared(b) firstprivate(c)
{
    cout << "Hello World!" << endl;
    a++;
    b++;
    c++;
    d++;
}

cout << "a: " << a << endl;
cout << "b: " << b << endl;
cout << "c: " << c << endl;
cout << "d: " << d << endl;
```

Output:

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
a: 1
b: 11
c: 1
d: 11
```

It is important to notice that there's no guarantee that for the *shared* variables the increments are atomic, so the result of the previous code could also be different from 11 in some cases, due to the race condition described above.

3.5.3 Reduction clause

The **reduction clause** specifies how multiple **local copies** of a variable at different threads are **combined** into a single copy at the master when threads exit. The clause is defined as *reduction (operator: variable list)*, where:

- each variable of the list is accessed as *private* by each thread, and the variable is initialized as the value which is neutral w.r.t. the *operator* (if $+$, then it is initialized as 0, if * as 1 etc..);
- the *operator* can be one of $+$, *, -, , |, etc..

An example of the functioning of reduction clause is provided in Picture 3.5.3. As we can see, b, c and d are shared, so their values are 11, while the value of a is incremented in each thread to 2, then it is reduced using the * operator, so the overall computation is $2 * 2 * .. * 2 = 2^{10} = 1024$.

Source:

```
bool parallel = true;
int a=1, b=1, c=1, d=1;
#pragma omp parallel if(parallel) num_threads(10) \
        reduction(*:a)
{
    cout << "Hello World!" << endl;
    a++;
    b++;
    c++;
    d++;
}

cout << "a: " << a << endl;
cout << "b: " << b << endl;
cout << "c: " << c << endl;
cout << "d: " << d << endl;
```

Output:

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
a: 1024
b: 11
c: 11
d: 11
```

3.5.4 *for* directive

OpenMP provides a directive *for* to split iterations of the subsequent for loop among available threads: an example is provided in Picture 3.5.4. As we can see, the code computes the sum of the first 10 squared numbers, and the variable *add* is reduces according to the operator +.

Source:

```
int add = 0;
#pragma omp parallel reduction(+:add)
{
    #pragma omp for
    for (int i=0; i<10; i++) {
      add += i*i;
    }
}
cout << "squares sum: " << add << endl;
```

Output:

```
squares sum: 285
```

An additional clause can be attached to the *for* directive, and it is the *schedule* clause. The *schedule* is defined as *schedule (policy [, param])*, and it deals with the assignment of iterations to each thread. The policies can be:

- **schedule(static)**: in this case the loop is statically split into chunks and each chunk is statically assigned to a thread. Notice that we still do not know in which order the chunks will be executed;
- **schedule(dynamic)**: in this case the loop is statically split into chunks, and each thread asks for the next chunk to be executed. We do not know how much chunks each thread will execute, but this policy is useful to balance the load;
- **schedule(guided)**: in this case the chunk size decreases exponentially in time. Long tasks are assigned soon to minimize any overhead, while short tasks are assigned at the end to avoid idle threads and to have threads that complete their chunks at the same time;
- the *param* specifies the chunk size; if the policy is *guided*, then it specifies the minimum chunk size.

Other additional clauses are the *nowait* clause and the *ordered* directive. The *nowait* clause enables the thread that completed the for loop execution to proceed the parallel execution of the rest of the code, while the *ordered* directive forces a piece of code to be executed according to the natural order of the for loop.
Finally, there are some restrictions in order to apply the *for* directive:

- for loops must not have break statements;
- loop control variable must be integer, as its initialization expression;
- the logical expression must be of $<, \leq, >, \geq$;
- the increments/decrements must have integers.

3.5.5 Sections

OpenMP supports non-iterative parallel task assignment using the *sections* directive: in this case, each section is executed by only one thread (in parallel). There exist some synchronization directives:

- **barrier**: all the threads must wait for each other to reach the barrier;
- **single[nowait]**: the following structures block enclosed is executed by only one thread in the team. Threads in the team that are not executing the single single block have to wait at the end of the block unless *nowait* is specified;
- **master**: only the master thread of the team executes the block enclosed by this directive, the others skip it and continue;
- **atomic**, which ensures atomicity of expressions like $x++, ++x, x--, --x$ etc..;
- **critical[(name)]**, which restricts access to the structured block to only one thread at a time. The optional name argument identifies the critical region: no two threads can enter the critical section with the same name.

3.5.6 Synchronization issues

OpenMP is able to synchronize shared variable, but the **compiler** may unpredictably optimize the code, and in general it uses a minimum effort approach on synchronization. The *flush(var list)* directive can be used to synchronize a set of shared variables among the threads in the team.

3.5.7 OpenMP vs explicit thread management

- Directives layered on top of threads **facilitate a variety of thread-related tasks**, and the programmer is **rid of the tasks** of initializing attributes objects, setting up arguments to threads, partitioning iteration spaces, etc..;
- However, using explicit threading the **data exchange is more apparent**, and this helps in alleviating some of the overheads from data movement, false sharing, and contention. Explicit threading also provides a **richer API** in the form of condition waits, locks of different types, and increased flexibility for building composite synchronization operations;

- Finally, since explicit threading is used more widely than OpenMP, **tools and support** for Pthreads programs are **easier** to find.

Example (π computation): Picture 3.5.7 shows the code for the parallel computation of π. We recall that the formula is

$$\pi = \frac{1}{n}\sum_{i=1}^{n}\frac{4}{1+(\frac{i-0.5}{n})^2}$$

Some comments:

- we see that the *parallel* and *for* directive are put together;
- the variable *pi* is reduced using the $+$ operator (see the formula);
- x and d are private, and d is initialized as $\frac{1}{n}$

```
void computePi(int n_points, int n_threads) {
    cout << "Starting..." << endl;

    double d = 1.0/(double)n_points;
    double pi = 0.0, x;

    #pragma omp parallel for \
      num_threads(n_threads) \
      reduction(+:pi) \
      private(x) \
      firstprivate(d)

    for (int i=1; i<=n_points; i++) {
      x = (i-0.5)*d;
      pi += 4.0/(1.0+x*x);
    }
    pi *= d;

    cout << "PI = " << pi << endl;
}
```

Example (Mandelbrot set): the Mandelbrot set is the set c of complex numbers such that the following function does not diverge:

$$\begin{cases} z_0 = 0 \\ z_{k+1} = z_k^2 + c \end{cases}$$

The issues about this example are that the number of iterations k is unknown, and the overall computation could lead to potential load imbalance, i.e. it can create very short or very long tasks! Picture 3.5.7 shows the parallel computation of the Mandelbrot set. As we can see:

- the schedule is *dynamic*, since the function could have some load imbalance;
- the *ordered* directive is used for the output of the points, since we have some constraints in their order.

```
void mandelbrot(int maxiter, int n_threads)
{
    const int width = 50, height = 50, num_pixels = width*height;
    const complex<double> center(-.7, 0), span(2.7, -(4/3.0)*2.7*height/width);
    const complex<double> begin = center-span/2.0, end = center+span/2.0;

#pragma omp parallel for schedule(dynamic) num_threads(n_threads) ordered
for(int pix=0; pix<num_pixels; ++pix)
{
    const int x = pix%width, y = pix/width;
    complex<double> c = begin + complex<double>(x * span.real() / (width +1.0),
            y * span.imag() / (height+1.0));
    int n = MandelbrotCalculate(c, maxiter);
    if(n == maxiter) n = 0;

                                #pragma omp ordered
}                                   {
                                        char c = ' ';
                                        if(n > 0) {
                                          static const char charset[] = ".,c8M@jawrpogOQEPGJ";
                                          c = charset[n % (sizeof(charset)-1)];
                                        }
                                        std::putchar(c);
                                        if(x+1 == width) std::puts("|");
                                    }
```

3.6 OpenMP and Cache

Once we introduced the OpenMP library, we can now analyze in detail some of its results when used for solving some classic problems. For example, if we consider again the **matrix multiplication problem**, we can compare the sequential and the parallel computation. If we consider the sequential computation, showed in Picture 3.6, we see that each of the three for loops can be parallelized.

```
// ---------------------------
// SEQUENTIAL ALGORITHM
// note: b has col-wise layout
// ---------------------------
for (long row=0; row<rows; row++) {
  for (long col=0; col<rows; col++) {
    // compute c[row,col]
    for (long i=0; i<rows; i++) {
      c[row*rows + col] +=
          a[row*rows + i] * b[col*rows + i];
    }
  }
}
```

Figure 18: Sequential algorithm

Intuitively, if we parallelize the most external loop, then 10 threads are created; if we parallelize the central loop, then 10 threads are created for each external loop, so $10 * N$

threads, where N is the number of rows; finally, if we parallelize the most internal loop, $10 * N^2$ threads are created.
More specifically, by parallelizing the external loop, a thread is created for each row of the matrix A, and it scans the full B before writng a new row in C: from the point of view of the cache this is quite bad, since the matrix B usually does not fit entirely in the memory.
On the other hand, if we parallelize the central loop, a thread is created for each column of matrix B, and it writes a column in C. The advantages of this approach are:

- differently from the previous case, not the entire matrix A needs to be stored in memory, but only one row, since all the threads will access the same row for computing a column of C. Thus, this approach is very nice since it is characterized by a great cache locality;
- each thread computes one column at at time, so we only need to store one column for each thread, which is better for the cache.

On the other hand, the main disadvantage of this approach is that the threads are created and destroyed multiple times, resulting in a quite high overhead. Finally, the last approach is to parallelize the internal loop: the code is provided in Picture 3.6.

```
for (long row=0; row<rows; row++) {
  for (long col=0; col<rows; col++) {
    // compute c[row,col]
    double c_ij = 0;
    #pragma omp parallel for \
                num_threads(num_threads) \
                reduction(+:c_ij)
    for (long i=0; i<rows; i++) {
      c_ij += a[row*rows + i] * b[col*rows + i];
    }
    c[row*rows + col] = c_ij;
  }
}
```

Figure 19: Parallel algorithm: third strategy

As we can see, the count variable c_{ij} is used to store the partial sums, and it is updated by each thread independently. In this sense, the parallelization is applied at single row and single column level. From the point of view of the cache, we only need to store one row and one column at a time, and this is very nice, but on the other hand the threads are created and destroyed N^2 times, so there's a huge overhead due to thread management.
The resulting execution times were:

- 11,500ms for the sequential computation;
- 800-1,200ms for the first version of the parallel computation;
- 700-6,000ms for the second version of the parallel computation;

- 153,923ms for the last version of the parallel computation.

In general, we notice a sort of **instability** in the timings, and this phenomenon is due to the fact that threads can be rescheduled to a different core/processor, and in this case the thread is moved to a different core (or CPU) along with all the data. This phenomenon leads to many cache misses and to a significant decay of the performances, and it can be solved by forcing the threads to not move along the cores, by using the command *export OMP_PROC_BIND = true*.
However, we notice that the second strategy is the one that better performs for our task. Now the goal is to **reduce the thread management overhead**, since, as we underlined before, the threads are created and destroyed multiple times (in particular, once for each row). A solution for this problem is given by switching the first and the second loop: in this case, each thread of the column is destroyed after all the rows of A are scanned, but in terms of the cache this is worse, since we need now to store the matrix A entirely. The code is showed in Picture 3.6.

```
#pragma omp parallel for \
            num_threads(num_threads)
for (long col=0; col<rows; col++) {
  for (long row=0; row<rows; row++) {
    // compute c[row,col]
    for (long i=0; i<rows; i++) {
      c[row*rows + col] +=
          a[row*rows + i] * b[col*rows + i];
    }
  }
}
```

Figure 20: Parallel algorithm: inverted strategy

A possible improvement to this implementation is showed in Picture 3.6.

```
#pragma omp parallel num_threads(num_threads)
for (long row=0; row<rows; row++) {
  #pragma omp for schedule(static) nowait
  for (long col=0; col<rows; col++) {
    // compute c[row,col]
    for (long i=0; i<rows; i++) {
      c[row*rows + col] +=
        a[row*rows + i] * b[col*rows + i];
    }
  }
  if (row%100==0) {
    #pragma omp barrier
  }
}
```

Figure 21: Parallel algorithm: synchronized strategy

As we can see, first of all the loop is statically split into chunks, and the *nowait* directive means that once a thread has completed the row, it moves to the following one, resulting in an increased number of rows that are processed simultaneously. Moreover, this approach results in less synchronization overhead and, potentially, in a better use of the cache; however, on the other hand, this reduction of synchronization could result in no cooperation between threads (so in less cache hits) and could lead to situations in which the data do not fit into the cache. For these reasons, a thread synchronization is performed after 100 rows of A (note that this quantity depends on the cache memory that is available).
The timings for the strategy we described above are:

- 810ms for the inverted strategy;
- 348ms for the synchronized strategy.

In particular, recalling that the time for the sequential implementation was 11,500ms, we notice that using the synchronized strategy we reach a speedup of $\frac{11{,}500}{348} = 33$, which is a great result.

3.7 *perf*

The Linux OS provides the *perf* command for profiling the execution of a given program, and in particular it allows to measure the number of misses for each level of the cache, the percentage of misses etc..

4 Patterns of Parallelism

In general, there exist some frequent **patterns of parallelism**, but we may need to design something quite specific for our problem/algorithm. We might identify two main techniques:

- **decomposition techniques** (like *divide-and-conquer*) that are used to generate (possibly) **independent sub-problems** that can be run in **parallel**. Notice that usually this is not a simple task, and also the merging phase of the outputs could be a problem;
- **mapping techniques** to decide who is going to execute what. These techniques depend on the **dependencies** among the tasks to be performed and their goal is to **maximize the load balance**.

4.1 Task dependency graph (TDG)

In general, the decomposition can be modelled with a **Task Dependency Graph** or **TDG**, which is a direct acyclic graph (DAG) where each **node** corresponds to a **task**, the **edges** represent task/data **dependencies** and the labels on the nodes measure the task computational cost.

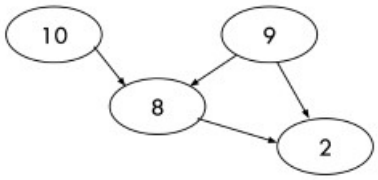


Figure 22: Example of TDG

In this sense, the TDG is used to show the relationships between the tasks resulting from a decomposition, and is not unique for a given problem: each problem could have different decompositions, task size etc.., i.e. the TDG depends on the specific decomposition of the problem. Moreover, TDG can be used for discovering load balance and the order of execution of the tasks.
Now, we define **parallelism degree** the number of tasks that can be executed in parallel. Notice that this number may change during the execution of an application, but in general it increases with a fine-grained decomposition. Obviously, our goal is to obtain an high parallelism degree, in order to better exploit threads and parallel computations.
A **directed path in a TDG** is a sequence of tasks in the TDG (which cannot be executed in parallel) linked by a dependency relation. The length of the path is given by the sum of the weights (i.e. labels) of its nodes.
A **critical path** is the longest directed path in the TDG, and it represents the **bottleneck** of the application. In this sense, in represents the minimum execution time, i.e. total running time $\geq$ critical path.
Finally, the **average parallelism degree** is defined as $\frac{\text{Total work}}{\text{Length of the critical path}}$, where Total work represents the sum of all the labels of the graph. This measure provides an estimation

of how much work we can run in parallel, so the larger the result, the more work we can run in parallel. Our goal is to define a TDG with the shortest possible critical path, in order to maximize the work that can be done in parallel.

4.2 Task interaction graph (TIG)

In this case, the nodes represent the tasks, the edges represent interaction/data exchange, the node labels represent the computational cost and the edge labels represent the amount of data that is exchanged.

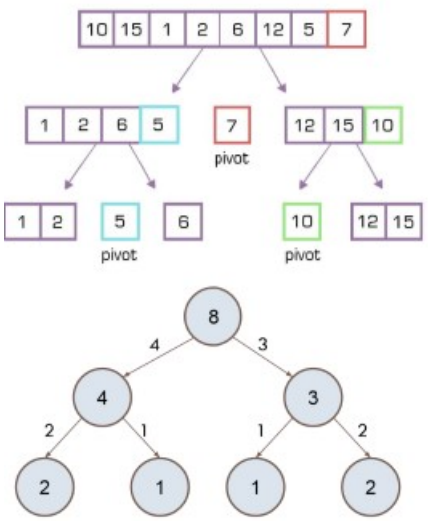


Figure 23: Example of TIG

Our goal is to find the minimum cut (i.e. the cut with minimum weight) s.t. each partition has the same load.

4.3 Mapping

In general, the number of tasks exceeds the number of processors, so we have a **mapping problem**, i.e. we have to assign the tasks to the processors. Usually, this mapping is planned on the basis of TDG and TIG:

- TDG helps in achieving load balance and minimizing waiting times (each processor receives the same load);
- TIG minimizes the interactions/communications between tasks (useful in a distributed environment).

Clearly, there can be a conflict between these two goals, but in general the **guidelines** are the following:

- Assign independent tasks to different processors;
- Tasks on the critical path must be assigned as early as possible;

- Minimize the interaction/communication costs by scheduling "dense" sub-graphs of the TIG to the same processor.

We will now focus on some common patterns of parallelism:

- Pipeline;
- Single Program Multiple Data / Data Parallel;
- Task Pool;
- Dynamic Task Creation;
- Distributed Load Balancing.

4.3.1 Pipeline

A **pipeline** is a special kind of task-parallelism, where the **computation** is **partitioned** into stages that are executed sequentially: the output of a stage provides the input of the following one.

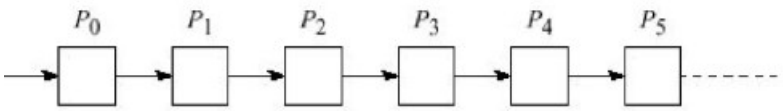


Figure 24: Pipeline

Asymptotically, a pipeline achieves a speed-up equal to the number of stages. If we have p processors, an input stream of n elements and a cost of each task of t, then:

- $(p-1)$ steps are used to fill the pipeline, in $(p-1)t$ time;
- n steps are used to produce the output, in nt time (each stage of the pipeline must be processed);
- The total speedup is $\frac{\text{sequential time}}{\text{parallel time}} = \frac{ptn}{nt+(p-1)t} = \frac{p}{1+\frac{p-1}{n}} \to p$

In this sense, if each stage has the same load of work, then we reach a **linear speedup**.

4.3.2 Single Program Multiple Data

This strategy addresses to problems that can be solved with a large set of completely independent sub-tasks, i.e. in case of a completely disconnected TDG. Clearly, this is not a common scenario, but for example involves the Mandelbrot problem.

4.3.3 Task Pool

In this case the task list is stored in a shared data structure, we have a fixed number of threads and each thread dynamically picks a task and executes it. Clearly, this technique is characterized by a synchronization overhead, and a thread may possibly generate a new task too.
In the Mandelbrot problem, there's a static partitioning of the input, assigning one pixel per task, then each thread processes a portion of the input, with dynamic assignment.

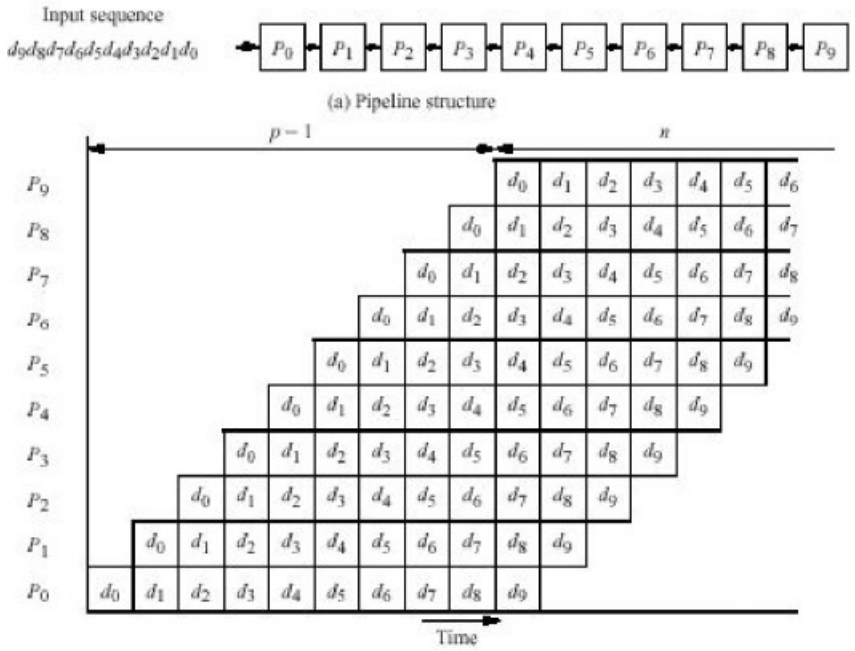


Figure 25: Execution of a pipeline

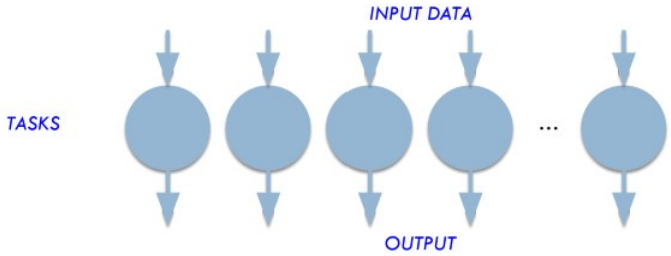


Figure 26: Single Program Multiple Data

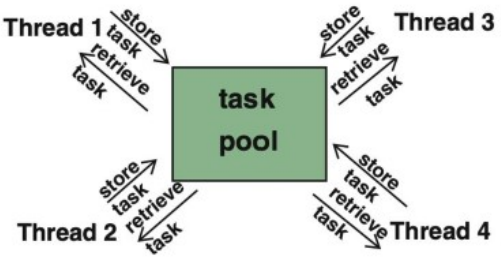


Figure 27: Task Pool

4.3.4 Dynamic Task Creation

In this technique, each task can dynamically create new tasks, and it is useful to:

- Improve the **parallelism degree**;
- Adapt to task of unknown costs, since it can split a long task into smaller ones;
- Adapt to non uniform resources, since it can exploit multiple machines with different

computing power.

Notice that a **task queue** must be shared among a pool of threads. This technique is very common, since it allows to deal with very different tasks.

4.3.5 Distributed Load Balancing

The goal of this approach is to remove centralization, i.e. remove the concept of *master node*, and to favor data exchange among neighbors. There exist two different variant for this technique:

- Push/Sender-Initiated, in which the worker that generates a new task sends it to another worker, instead of putting it in the task list;
- Pull/Receiver-Initiated, in which when a worker is idle, it asks other workers for a job to execute (work stealing)

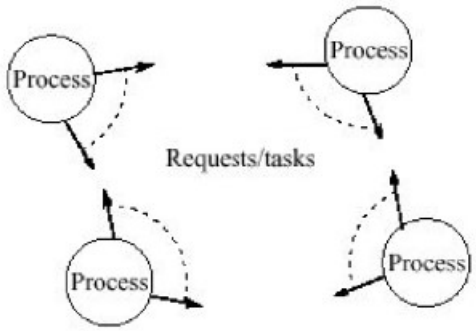


Figure 28: Distributed Load Balancing

In the Push/Sender-Initiated, the worker to which the task is sent can be selected:

- At **random**, among all the workers;
- **Global Round Robin (GRR)** in which a global random variable/marker points to the "next" worker, and a worker needing a partner reads and increments the global variable;
- **Local Round Robin (LRR)**, in which each processor keeps a private pointer to the next available worker, so there's no overhead due to sharing a global variable.

In general, the choice among **static mapping** (pipeline, single program multiple data) and **dynamic mapping** (task pool, dynamic task creation) can be made according to the following schema:
In general, the **static mapping** is preferable, since is does not need any synchronization.

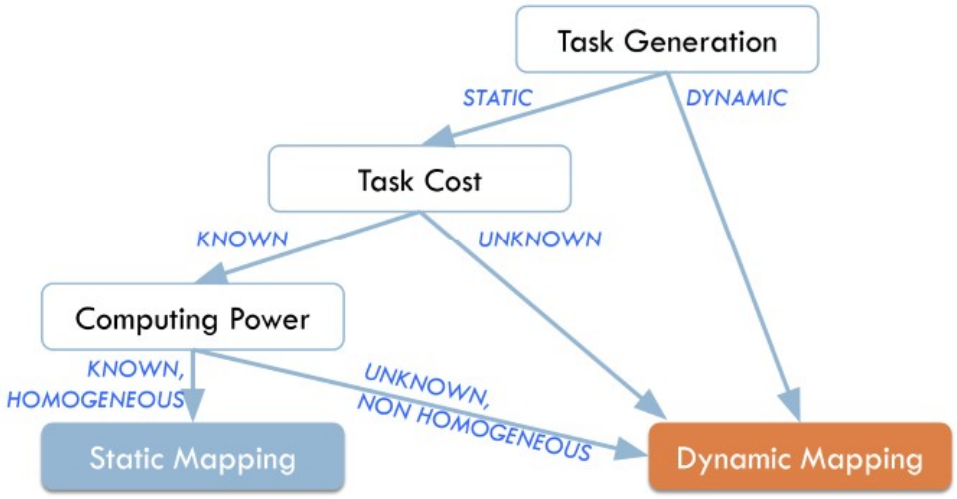


Figure 29: Static vs dynamic mapping

4.4 Prefix sum

Given a list of integers $x_0, x_1, .., x_{n-1}$, compute the partial sum up to index i, for each i. The optimal sequential algorithm has complexity $O(n)$: for $(i = 1;\ i < n;\ i++)$: $x[i] += x[i-1]$.
However, this algorithm is difficult to be parallelized, since each iteration depends on the previous one. For example, we cannot split the vector x into two parts T_1 and T_2, since T_2 must wait T_1 to perform its computations. For this reason, the naive parallel algorithm works as follows:

- Break the dependencies through a step-wise algorithm;
- One thread computes one element at each step.

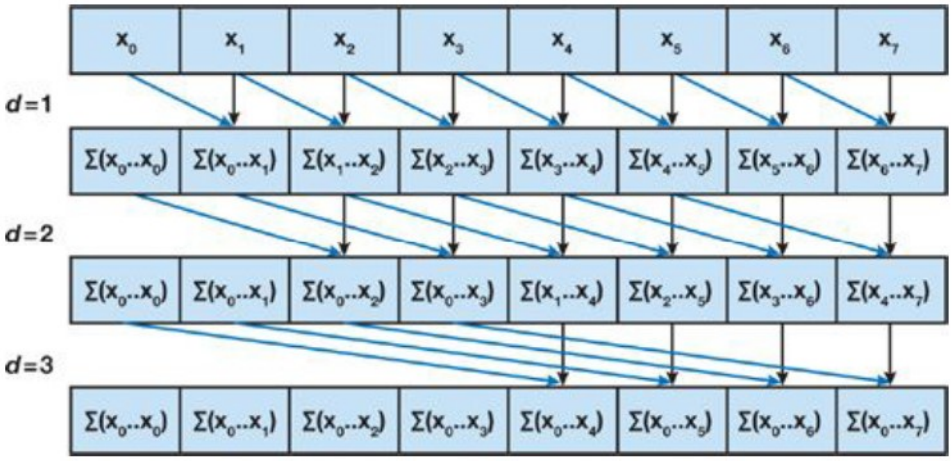


Figure 30: Parallel prefix sum

In this case:

- The total amount of work is $O(N \log N)$, where $\log N$ is the number of steps of the algorithm, and N is the length of the vector. In the sequential version, the amount of work was $O(N)$;
- The complexity is $O(\log N)$, while in the sequential algorithm we had $O(N)$.

As we can see, in this case the parallel version of the algorithm is quite different from the sequential one, and we also notice that the parallel version results in more work to be done, but the execution time is faster. For this reason, in this case the benefits of the parallel algorithm can be obtained only if enough processors are available, otherwise it is better to execute the sequential algorithm.

```
for (int j=0; j<std::log2(n); j++) {
    #pragma omp parallel for
    for (int i=std::exp2(j); i<n; i++)
        xx_tmp[i] = xxx[i]+xxx[i-std::exp2(j)];
    xx_tmp.swap(xxx);
}
```

Figure 31: Implementation of the parallel algorithm

For this reason, now our goal is to **reduce the amount of work**, i.e. number of sums, in order to reduce the number of processors that are needed for the parallelization.
The idea is to focus on an **exclusive prefix sum**, i.e. a prefix sum that discards the last element, which is composed on two phases: the **Up-sweep** phase and the **Down-sweep** phase.

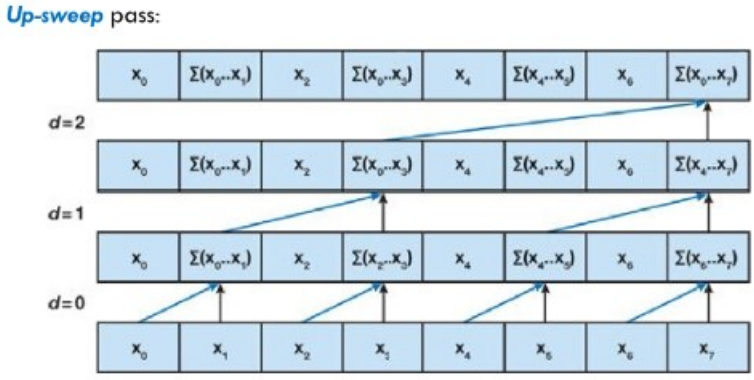


Figure 32: Up-sweep phase

On the one hand, the goal of the **Up-sweep** phase is to store in a vector some important partial sums. In particular, it performs $N-1$ summations in $\log N$ steps, and the resulting vector contains several **correct partial sums**, together with elements for which the prefix sum is not computed. In this sense, an important property of the resulting vector is that each node that contains the correct partial sums holds the **sum of its children**.
On the other hand, the idea of the **Down-sweep** is to enforce the property that **each node should have the sum of all the leaves preceding it**, i.e. before its leftmost child. This means that:

- The root has value 0;

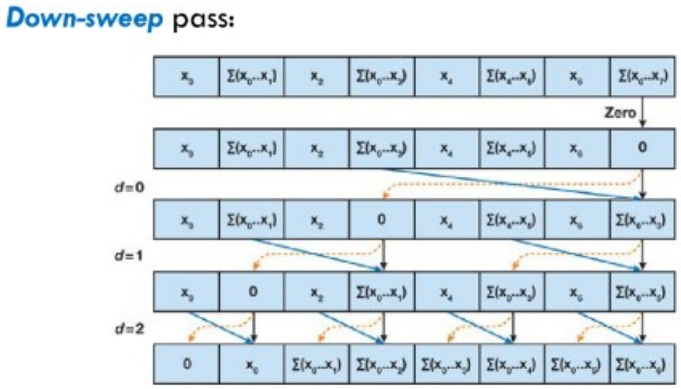


Figure 33: Down-sweep phase

- A **left** child has the same predecessors of its parent, so it has the **same value as it's parent**;
- A **right** child is preceded by its parent predecessors plus the sibling's children, so its value should be equal to its **parent plus the up-sweep value of its sibling**.

This phase takes $\log N$ steps and $N - 1$ summations, so the overall algorithm:

- Has complexity of $O(\log N)$ (i.e. number of steps);
- Performs $O(N)$ summations (amount of work), which is much less that the previous parallel version of the algorithm. Moreover, notice that a large parallelism degree can be exploited for most steps.

Picture 4.4 shows the performances of the sequential algorithm (CPU), the naive parallel algorithm (Naive) and a version of the algorithm running on graphic processors (CUDA).

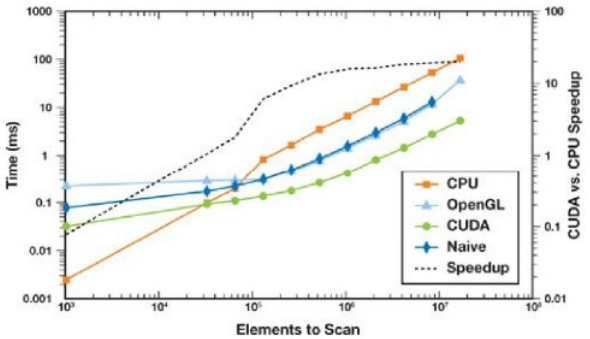


Figure 34: GPU performance

4.5 QuickSort

We now focus on the problem of parallelizing the **QuickSort** algorithm.
A naive approach would be of sunning step (a) as 1 sequential task, step (b) as 2 parallel tasks, step (c) as 4 parallel tasks, and so on.., until the number of tasks is equal to

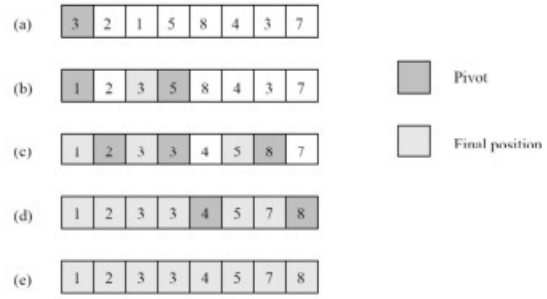


Figure 35: QuickSort algorithm

available parallelism. However, this approach, and in general this algorithm, could lead to two important **issues**:

- **Load imbalance**, since the pivot does not split the array in a balanced way, so the recursion in one side could complete the computation before the recursion in another side. In other words, the array subsequences depend on the pivot selection, and different lengths lead to different workloads and imbalance.

We now focus on different techniques for parallelizing the QuickSort algorithm.

4.5.1 Local and global arrangement

The first method works as follows:

1. Split the input array into several sub-arrays, and assign each of them to a processor;
2. Pick a pivot;
3. **Local arrangement**: separate each sub-array according to the chosen pivot;
4. **Global arrangement**: separate the global array according to the chosen pivot;
5. Recursively on the two sub-arrays until each processor can use a sequential algorithm to sort its sub-array.

This method suffers from the following **disadvantages**:

- It requires a very high **parallelism degree**;
- The **load balance** is still sensitive to the pivot selection.

4.5.2 Odd-Even transposition

This method works as follows:

1. Repeat $\frac{n}{2}$ times:
   (a) Compare-exchange the **odd** elements with their immediate neighbor;
   (b) Compare-exchange the **even** elements with their immediate neighbor.

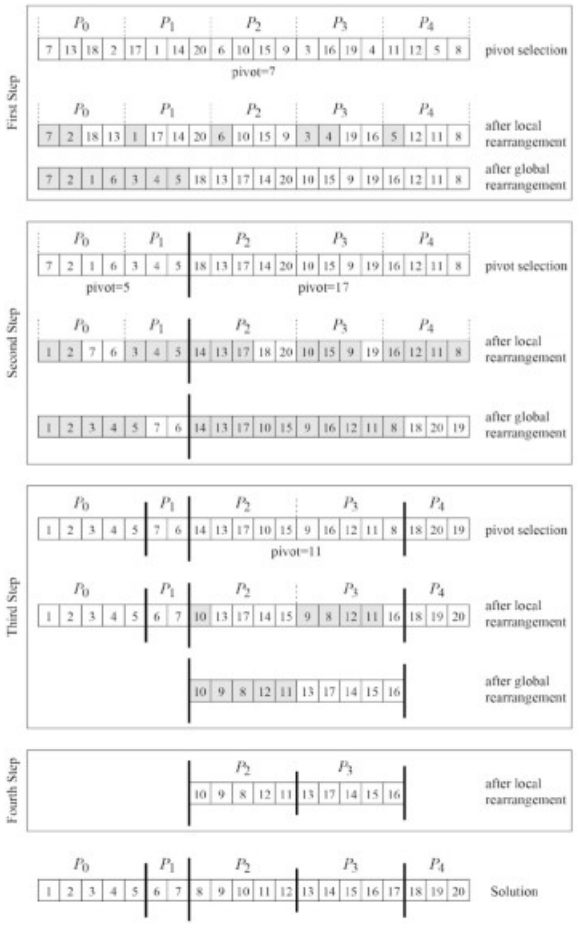


Figure 36: Parallel QuickSort algorithm

Clearly, the **sequential** complexity is $O(n^2)$, so it is a bad sequential algorithm. On the other hand, each phase can be easily **parallelized**, and if the number of processors $p = \frac{n}{2}$, then each phase (compare odd or compare even) takes $O(1)$, so in total $O(n)$, so the overall complexity is $\frac{n}{2} * n = O(n^2)$. Since the complexity of sorting is $O(n \log n)$, this algorithm is not cost optimal.
However, this algorithm can be generalized by assigning batches of n/p elements to each processor. In this case, the local sort on n/p elements (i.e. the sort of each batch) takes $p(n/p \log(n/p) = O(n \log(n/p))$, so in order to produce the result we need a linear scan of all the batches for each phase, so the cost is $p(p * n/p) = O(np)$. If $p = O(n)$, then the parallel algorithm is cost optimal with cost $O(n \log n)$. However, this strategy has the drawback that when increasing p, we must exponentially increase n in order to have the same scalability, otherwise the algorithm would not be cost optimal anymore.

4.6 Bitonic sort

We now focus on another type of sort, and we will discuss a sequential algorithm which is not optimal, but easily to be parallelized (with an high parallel degree too).
A sequence $X = x_1, x_2, .., x_N$ is said to be **bitonic** if $x_1, .., x_j$ is **monotonically increasing** and $x_{j+1}, .., x_N$ is **monotonically decreasing**, or this holds for a cyclic shift of the

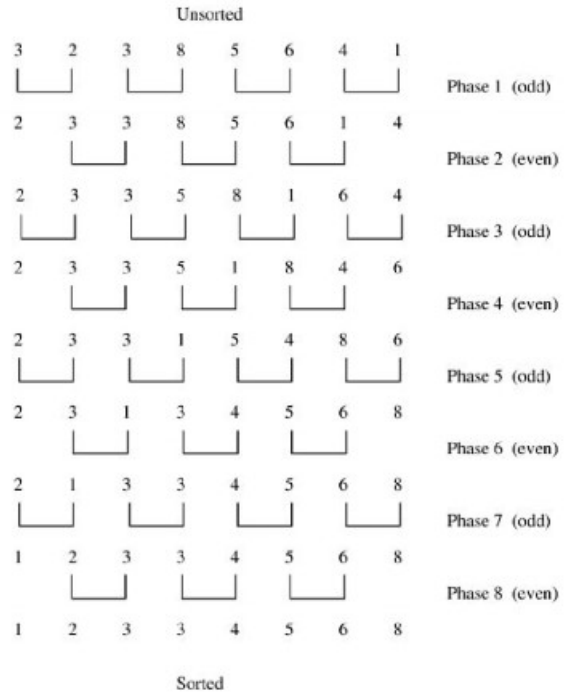


Figure 37: Odd-even transposition

input sequence.

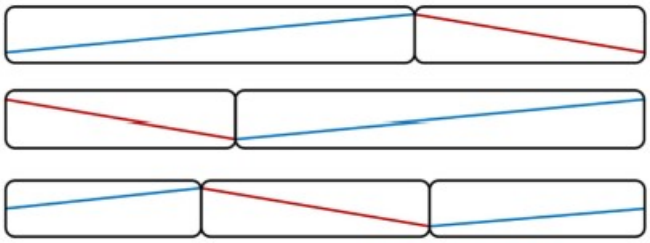


Figure 38: Examples of bitonic sequences

We now address two problems:

1. How to sort a bitonic sequence?;
2. Supposing that we can sort a bitonic sequence, how can we make a sequence bitonic?

4.6.1 Bitonic sort

A **comparator** $[i : j]$ sorts the i-th and the j-th element of a sequence into a non-decreasing order. A **Bitonic split** is the result of applying N/2 comparators of the kind $[i : i + N/2]$ for $1 \leq i \leq N/2$ to a **bitonic sequence**. Picture 4.6.1 shows the functioning of a bitonic split.
The Bitonic split ensures two properties:

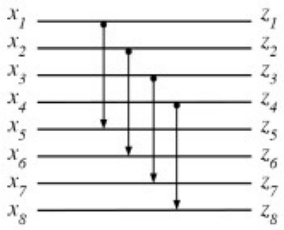


Figure 39: Functioning of a bitonic split

- All the elements of the first half are less or equal the elements of the second half: if $Z = BS(X)$, i.e. Z is the result of the Bitonic split of the input X, then:

$$Z_1, .., Z_{N/2} \leq Z_{N/2+1}, .., Z_N$$

- Each of the halves is a bitonic sequence, i.e. $Z_1, .., Z_{N/2}$ and $Z_{N/2+1}, .., Z_N$ are bitonic.

Both the properties hold since the input sequence is bitonic.
Now, the idea of the algorithm is to **recursively** apply a **Bitonic split** to **each of the sequences**. Thus, a **Bitonic merge** performs recursive Bitonic splits until the resulting sequence is sorted in increasing order.

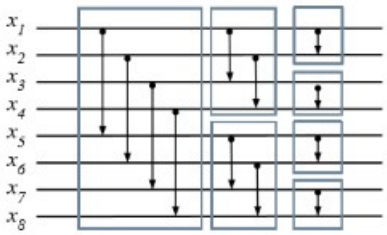


Figure 40: Functioning of a bitonic merge

This method involves $\log N$ steps (where N is the length of the input sequence), and each step performs $N/2$ compare/swap operations (i.e. bitonic splits). Thus, the **complexity of the sequential algorithm** is $O(N \log N)$.
If we have $N/2$ processors, than each comparison can be executed in parallel, so the **complexity of the parallel algorithm** is $O(\log N)$.
Before focusing on how we can make a sequence bitonic, we now discuss an interesting **property** of this algorithm. Indeed, in this method the **operations do not depend on the input data**, so the algorithm works for every input, provided that it is bitonic. Notice that this is a very good advantage, since it makes the algorithm easily parallelizable, exploiting an high parallelism degree. Notice also that this property does not hold for every algorithm, e.g. the operations of the QuickSort algorithm depend on the choice of the pivot.

4.6.2 Constructing a Bitonic sequence

In order to feed the final Bitonic Merge step with a bitonic sequence, the two halves of the input are sorted ascendingly/descendingly by recursively invoking Bitonic sort.

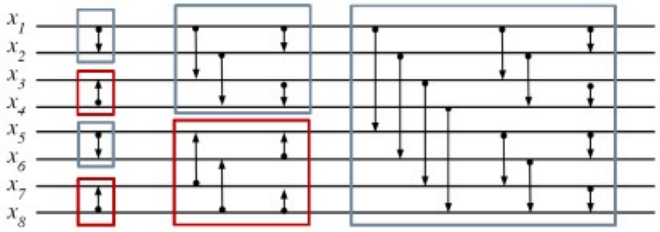


Figure 41: Functioning of a bitonic merge

As we can see, there are $\log N$ Bitonic merge phases, each having complexity $N \log N$, so the overall complexity id $O(N \log^2 N)$. Notice that the Bitonic sort can be parallelized very well, and it can exploit a large parallelism degree. The parallel cost is $O(p \log^2 N)$, so it is not cost optimal, and this algorithm is very popular for GPUs and for medium sized inputs.

5 Large-scale data processing with MapReduce

When we have a lot of data, we must exploit multiple machines, so larger architectures that support parallelization, both on the computation and on the read/write operations. In this sense, some standard architectures are **commodity clusters**, which are standard servers that scale the computation up on different machines, and **Gigabit Ethernet interconnections.**

5.1 Commodity clusters

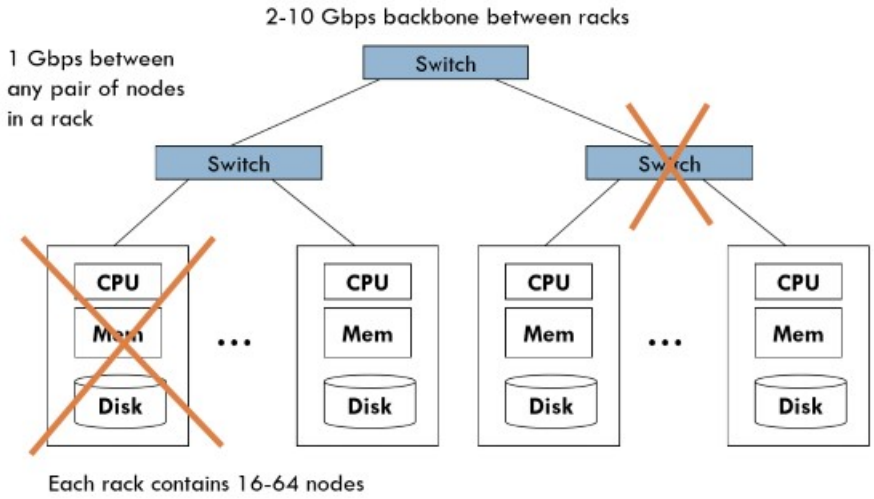


Figure 42: Cluster architecture

As we can see, two main failures may happen:

- A failure in the **network**, after which some nodes are not reachable anymore;
- A failure on a **machine**.

In general, handling failures is difficult, since they're extremely frequent, so an important property of the cluster is to be **failure resilient**. In particular, if we address the problem of a *machine failure*, we have a solution represented by the **Distributed File System**, which provides a global file namespace and it supports huge files (100 of GBs to TBs): in this case, data is rarely updated in place, and **reads** and **appends** are common. Some examples of such DFS are *Google GFS* and *Hadoop HDFS*
In particular, the Hadoop DFS is composed of the following entities:

- **Chunk servers**, which contain the splitted file and serve the *master node*. Each chunk is typically 16-64MB, and it is replicated 2x or 3x, so the **replication** is at **chunk level**. Note that the HDFS tries to keep the replicas in different racks;
- **Master node** (or **Name node**), which stores all the metadata and takes care of giving the user a unique FS, and it keeps track of all the chunk servers. The mater node might be replicated;

- Client library for file access, which talks to the master node to find the chunk servers and connects directly to chunk servers to access data.

The operations that are supported by the HDFS are **concurrent reads** and **single write** operations, which are implemented as append operations.
Two very important files in the HDFS are:

- The **sequence file**, which is persistent data structure for binary key-value pairs. This file can be written only by using the *append()* method, and can be read only by using the *next()* method;
- The **map file**, which represents a higher-level file and it is a **sorted sequence file** with an **index** to support lookups by key. Thus, in this case the file can be read by using *next()* or *get()* (by key). Notice that the lookup by key is implemented with two sequence files: *Index* and *Data*, where the *Index* file must be loaded in memory for random lookups.

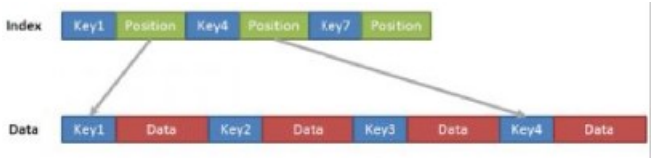


Figure 43: *Index* and *Data* files

Finally, notice that the **replication of data three times** is a robust guard against loss of data due to uncorrelated node failures: for a large cluster, the probability of losing a block during one year is less than 0.005. The key understanding is that about 0.8 percent of nodes fail each month, so for the sample of large clusters, a node or two is lost everyday.

5.2 MapReduce

MapReduce represents a "novel" programming paradigm, which is based on top of *<key,value>* pairs, where the *keys* and the *values* are **user-defined**, so they can have any type. This framework is essentially based on two functions:

- **Map**, which takes as input a key-value pair (k_1, v_1), and produces as output an intermediate data v_2 labeled with a key k_2:

$$\text{Map}(k_1, v_1) = \text{list}(k_2, v_2)$$

- **Reduce**, which given every data $\text{list}(v_2)$ associated with a key k_2 (so we suppose that we can execute multiple *Map* operations on the same data), it produces the output of the algorithm $\text{list}(k_3, v_3)$, so its functioning is pretty similar to a GroupBy.

$$\text{Reduce}(k_2, \text{list}(v_2)) = \text{list}(k_3, v_3)$$

If we put all in parallel, we have a set of **mappers** and a set of **reducers**:

1. A **mapper** processes only a split of the input, which may be distributed across several machines;

$$\text{Map}(k_1, v_1) = \text{list}(k_2, v_2)$$

2. A **shuffle** phase transfers the data associated with a given key from the *mappers* to the proper *reducer*, so that the reducer will receive data sorted by key;
3. A **reducer** produces only a portion of the output associated with a given set of keys. Notice that *reducers* can run in parallel.

$$\text{Reduce}(k_2, \text{list}(v_2)) = \text{list}(k_3, v_3)$$

Example: WordCount using MapReduce
Picture 5.2 show the code of the Map and Reduce functions for the word count problem.

```
map(key, value):
// key: document name; value: text of document
   for each word w in value:
     emit(w, 1)
```

```
reduce(key, values):
// key: a word; value: an iterator over counts
     result = 0
     for each count v in values:
          result += v
     emit(key,result)
```

Figure 44: Map and Reduce functions for word count problem

As we can see, the Map function produces a (word,1) pair for each of the word in the document, while the Reduce function receives in input a word and a list of 1's, and produce in output a (word,count) pair.
Picture 5.2 shows the execution of the MapReduce framework for word counting.
As we can see, the text is splitted into 3 chunks: the Map function of each chunk produces a (word,1) pair for each word of the input. Then, the Shuffle phase implements a GroupBy over the output of the Map functions, and produces a pair composed by a word and a list of 1's, which represents the input of the Reduce function. Finally, the output provides the count for each word of the input. Notice that the only operations that the user has to implement are the one from the Split phase to the Map phase, and from the Shuffle phase to the Reduce phase, since all the others belong to the MapReduce framework.

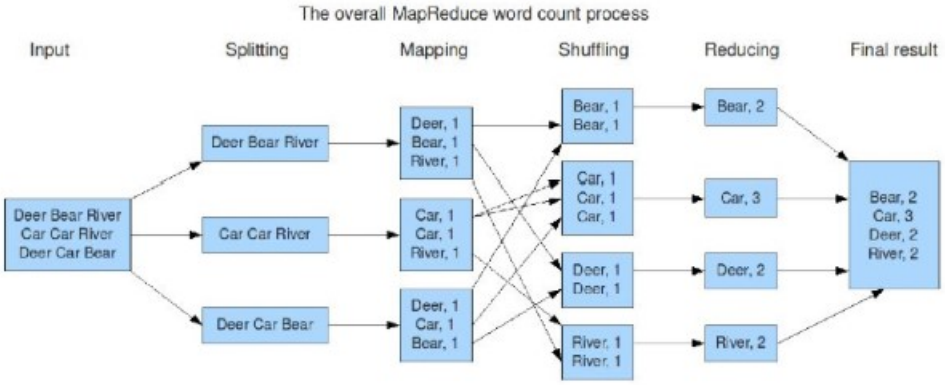


Figure 45: Execution scheme of MapReduce for word counting

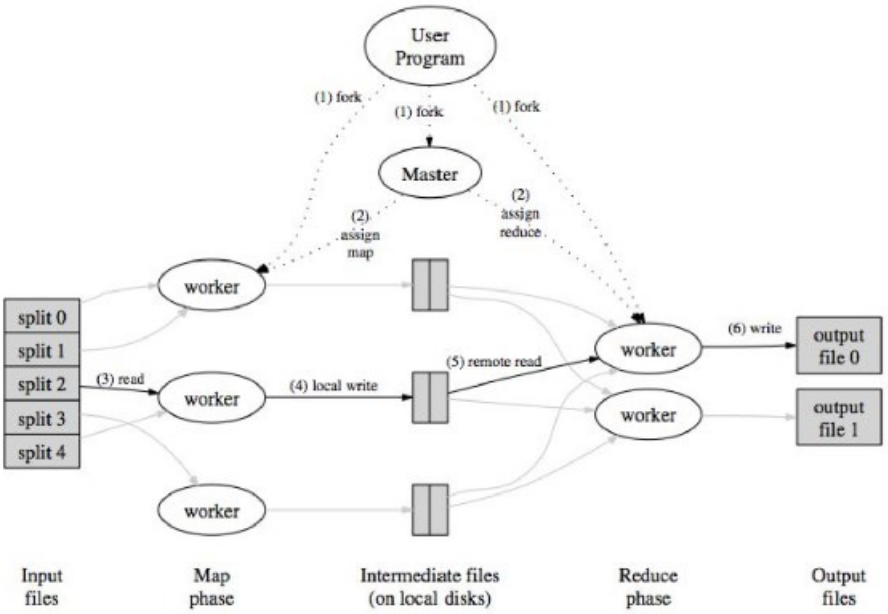


Figure 46: MapReduce architecture

5.2.1 Architecture

Picture 5.2.1 shows the general architecture of the MapReduce framework.

1. The *user program* invokes the MapReduce;
2. The *master node* coordinates the work. In particular, it assigns the *workers* the splitting to which performing the Map function, and to other *workers* the splitting to which performing the Reduce function and what intermediate files to read. The assignment of the chunks is usually based on the machine in which the chunk itself resides;
3. The *Map workers* read the split of the input files and they execute the Map function, writing the (key,value) output pairs into intermediate files in the local File System;
4. The *Reduce workers* read the intermediate files from the FS and execute the Reduce

function, writing the final result into output files, which in turn can represent the input of another Map function.

5.2.2 Coordination and failures

The coordination between all the workers is ensured by the **master node**. This data structure maintains the *status* for each of the tasks (idle, in-progress, completed), and idle tasks get scheduled as workers become available. Moreover, when a Map task completes, it sends the master the location and size of its R intermediate files, one for each reducer, and the master pushes these info to the reducers. Finally, another task of the master is to periodically ping the workers to detect failures.
A **failure** may happen on Map workers, on Reduce workers and on the Master node:

- If a **Map worker failure** happens, then the Map tasks that are completed or in-progress are reset to idle, and the Reduce workers are notified when the task is rescheduled on another worker, in order to delete his local copy of the failed Map's data;
- If a **Reduce worker failure** happens, then only the in-progress tasks are reset to idle, since the results of the completed tasks are already saved in the distributed FS;
- If the **Master node** fails, then the MapReduce task is aborted, and the client is notified.

5.2.3 Number of Map and Reduce jobs

In general, we number M of map tasks and R of Reduce tasks are parameters to be tuned. A rule of thumb suggests that M and R should be larger than the number of nodes in the cluster, otherwise we would not exploit the parallelization. In general:

- M is determined by the number of input splits;
- R is determined by the number of distinct key-value pairs, so on the data;
- Having several mappers and reducers may improve load balancing and it speeds the recovery from work failure, but on the other hand it increases the overhead given by the coordination between the different workers;
- Usually $R < M$, because the system has a maximum capacity of parallel reducers, so the reducers are executed in waves. The shuffling of the first wave is done in parallel with mappers, i.e. very early in the computation, but the shuffling of other waves is done later, so there's less communication/computation overlap.

5.2.4 Some improvements

Some improvements over the classic architecture of Picture 5.2.1 can be done:

- We know that Map tasks on the same node will produce many pairs with the same key (an example is provided in Picture 5.2, where we have a node containing two

pairs *(Car, 1)*), so the idea is to save some network time by pre-aggregating the pairs at Map level by using the **combiners**. In this sense:

$$\text{combine}(k_2, \text{list}(v_2)) = \text{list}(k_3, v_3)$$

In the example, the result of the use of the *combine* function is to have a pair *(Car, 2)* before the shuffling phase. Notice that the *combine* functionality usually exploits the Reduce function, if the Reduce function is commutative and associative.

- We know that the input of the Map tasks are created by contiguous splits of the input file, and for the Reducer we need to ensure that intermediate *(k,v)* pairs with the same key end up at the same reducer. In this sense, the system uses a default **partition function**, e.g. hash(key)mod$R$, in order to ensure both a faster computation of the Reduce functions and load balance among the reducers. Notice that the partition functions may be overridden and customized.
- In order to shorten the total execution time and to ensure fault tolerant property of the system, the system enables the **back-up tasks**, i.e. tasks that are executed twice, in two different machines, and the result of the faster one is kept. This solution dramatically shortens the job completion time.

In general, it is useful to underline that the MapReduce framework is **not** adopted for **performances**, since for example the writes of the mapper to the disk produce a huge overhead, and in general the shuffle step represents the bottleneck of the system. The **advantage** of this framework is that the user is asked only to override two functions (sometimes, other functions are needed to be implemented, such as combiners, partitioners, etc..), but on the other hand **not everything can be implemented in terms of MapReduce**. For example, the *all pairs similarity problem* requires to scan N^2 candidates, where N is the number of documents in the collection, and experiments showed that for 60MB worth of documents, the framework would generate 40GB of intermediate data to be shuffled. Graph mining problems such as community detection and PageRank computation are hard problem for MapReduce too.

5.2.5 Implementations of MapReduce

There exist many methods for implementing a MapReduce framework:

- **Hadoop**, which is an Apache project providing a Java implementation of MapReduce and relying on the HDFS (Hadoop Distributed File System), which is efficient and reliable. Hadoop only needs to copy Java libraries into each machine of the cluster;
- **Dryad DryadLINQ**;
- **Pig**, which provides a rich scripting language, which is then transformed into a chain of MapReduce jobs;
- **Hive**, which is a open-source data warehousing solution built on top of Hadoop, supporting queries in a SQL-like declarative language;

- **Spark**, which is a fast and general-purpose cluster computing system, that provides a high-level APIs in Scala, Java and Python, that make parallel jobs easy to write. It resolves around the concept of a Resilient Distributed Dataset (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel

5.3 Documents all pairs similarity with MapReduce

The problem is defined as follows: given a collection of documents and a minimum required similarity threshold, the goal is to retrieve from the collection the number of pairs of documents that have a similarity greater or equal to the threshold value. In this case, each document is represented as a vector d of N elements, where N is the size of the lexicon, and each $d[i]$ stores the *tf-idf* value of term i in the corpus. Moreover, the similarity is computed using the cosine similarity:

$$s(a, b) = \sum_{i=1,..,N} a[i] \cdot b[i]$$

We now discuss a method for solving this problem exploiting the MapReduce framework.

5.3.1 First solution

First of all, we can observe that the minimum requirement for two documents to have a similarity greater than 0 is to have at least one term in common, so the idea is to skip the computation of the similarity for two documents that do not have any term in common. In this sense, if we store, for each term in the lexicon, the list of documents containing that term, we can easily compute the similarity between these documents.

$$(\text{term}_1, [\text{doc}_1, \text{doc}_2, ..])$$

Thus, the **Reduce function** works like this: it receives in input a key-value pair, where:

- the key is represented by the term;
- the value is represented by the list of documents in which the term appears.

, and for each pair of documents in which each term in the lexicon appears, it computes the similarity and it compares it with the threshold value. Thus, it never computes the cosine similarity of documents that do not contain any term in common.
Once we designed the Reduce function, we must design the Map and the Shuffle functions. The **Map function** produces, for each term in the lexicon, a key-value pair where:

- the key is represented by the term;
- the value is represented by a tuple (docID,document), where document represents the full document;

Then, the **Shuffle function** simply performs a GroupBy operations by key, i.e. by term, and it produces the key-value pairs that are given in input to the Reduce function. Picture 5.3.1 shows the Map and the Reduce functions for this problem.

```
Map(doc_id, document):
    for each term t in document:
        emit(t,[doc_id,document])


Reduce(t, list([doc_id,document])):
    for id1,d1 in list([doc_id,document]):
        for id2,d2 in list([doc_id,document]):
            if sim(d1,d2)>=threshold:
                emit([id1, id2, sim(d1,d2)])
```

Figure 47: Map and Reduce functions for all pairs similarity

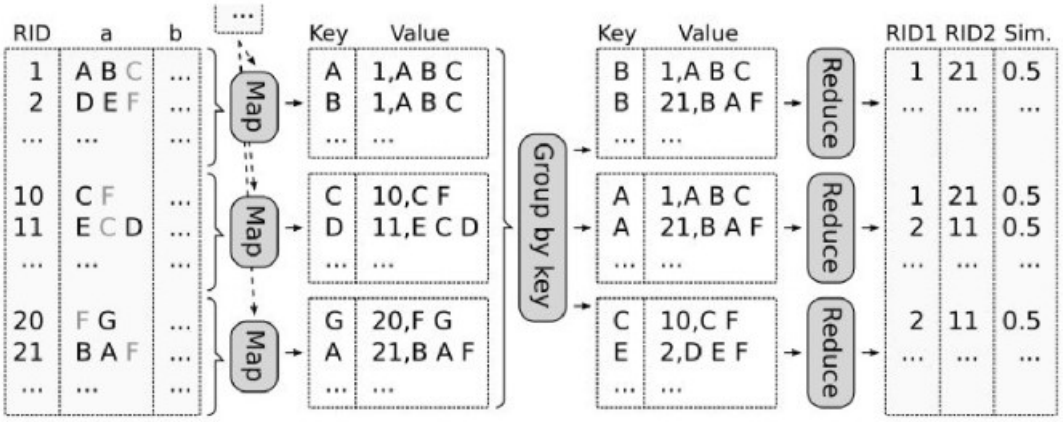


Figure 48: Example of MapReduce functioning for all pairs similarity

Picture 5.3.1 shows an example of the functioning of the MapReduce framework for this problem.
However, a **first problem** is the following one: if a document has 1,000 terms, then the document is replicated 1,000 times after the Map execution.

5.3.2 Second solution

In order to solve the previous problem, we implement the so called **prefix filtering**. The idea is to reduce the number of terms we consider, in order to reduce the number of pair for which the similarity is computed.

1. First of all, if we consider the term-document matrix, we sort the terms by decreasing score in the corpus, i.e. $d[0]$ corresponds to the term having the largest score of *tf-idf*;
2. Then, we create the *maximum document* d*, i.e. $d^*[i] = \max d[i]$ for each d in the collection. In other words, d^* stores the maximum score of terms in any document. Now we can have **two observations**:

- If $\text{sim}(d, d^*) < \text{threshold}$, then d has no similar documents in the corpus. This is true because d^* represents the maximum document, so for sure it does not exist another document d_j s.t. $\text{sim}(d, d_j) \geq \text{threshold}$;
- Suppose we arrive at a certain term, and $\text{sim}(d, d^*) < \text{threshold}$: then, we can find another document d_j similar to d_i only if they have a term in common after the term we're now considering, otherwise we now that it does not exist another document d_j s.t. $\text{sim}(d, d_j) \geq \text{threshold}$ form the above observation. In this sense, all the terms up to the current one are useless, so we only consider the other in the Map function. In other words, the Map function only considers the terms $t > b(d)$, where $b(d)$ is the *boundary* of a document s.t.:

$$\sum_{0 \leq t \leq b(t)} d[t] \cdot d^*[t] < \text{threshold}$$

, i.e. the boundary of each document is represented by the largest term for which we have $\text{sim}(doc, d^*) < \text{threshold}$

Picture 5.3.2 shows the Map and Reduce functions for this version of the solution.

```
Map(doc_id, document):
    for each term t in sorted(document):
        if t>b(document):
            emit(t,[doc_id,document])


Reduce(t, list([doc_id,document])):
    for id1,d1 in list([doc_id,document]):
        for id2,d2 in list([doc_id,document]):
            if sim(d1,d2)>=threshold:
                emit([id1, id2, sim(d1,d2)])
```

Figure 49: Map and Reduce functions for all pairs similarity with prefix filtering

However, we have another **problem**: if two documents share 10 terms after filtering, then their similarity is computed and produced in output 10 times.

5.3.3 Third solution

A possible solution for this new problem could be sort the whole output and remove the duplicates, but we can easily notice that it is quite expensive.
For this reason, another approach is based on the following idea: only one Reducer is forced to compute the similarity. However, if the two documents are replicated 10 times with 10 different keys, they may end up in 10 different Reducers, so how do we tell the Reducers whether to compute or not the similarity? A selection mechanism can be

```
Map(doc_id, document):
   for each term t in sorted(document):
      if t>b(document):
         emit(t,[doc_id,document])


Reduce(t, list([doc_id,document])):
   for id1,d1 in list([doc_id,document]):
      for id2,d2 in list([doc_id,document]):
         if (t= max(d1 ∩ d2)):
            if sim(d1,d2)>=threshold:
               emit([id1, id2, sim(d1,d2)])
```

Figure 50: Map and Reduce functions for all pairs similarity with improved prefix filtering

exploited, e.g. if the reducer has the key equivalent to the largest item, it can compute the similarity.
Picture 5.3.3 shows the Map and Reduce functions for this version of the solution.

5.4 Graph mining problem with MapReduce

We consider an undirected graph, and our goal is to find the **number of connected components**, possibly exploiting the MapReduce framework. In general, graph mining problems are difficult to be parallelized, because, for example, if we think of the "divide-et-impera" approach, we have to partition the graph (i.e. breaking connected components), run the algorithm on each sub-graph and merge the partial results. In this sense, we now exploit a different paradigm of computation, the **vertex centric paradigm**.
The idea of this paradigm is to write an algorithm for a node rather than a graph, i.e. we specify what each node has to do. In particular, the sketch of the algorithm is the following:

- Each node knows about its neighbors and may hold some additional custom information;
- The same computational kernel is executed for each node;
- Repeat until convergence.

The **advantages** of of this paradigm are:

- It is easier to design a "local" algorithm for a single node, rather than for the entire graph;
- We can exploit a large parallelism degree, since each node is independent from the others;
- It can be implemented over MapReduce, and there exist some specific frameworks like Spark GraphX.

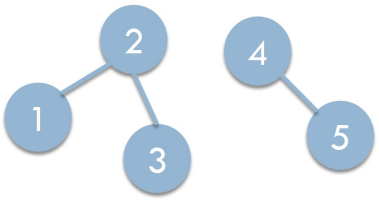
However, a **disadvantage** is that the local view may lead to sub-optimal results.

5.4.1 Label propagation

We now describe the algorithm we introduced in the previous section.
The **input** of the algorithm is a list of $(X, C_{\min})$ pairs, meaning that X belongs to the connected component with id $C_{\min}$: clearly, when the algorithm starts, node X belongs to component with id X. Moreover, each node knows its neighbors $N(X)$.
The algorithm works as follows:

1. Node X sends its own component id $C_{\min}$ to all its neighbors $N(X)$, along with itself. In this sense, each node outputs a $(X, C_{\min})$ pair;
2. Node X receives a list of component ids and computes the minimum among them, i.e. the new $C_{\min}$;
3. Repeat until convergence, i.e. when no updates occur.

As we said before, the idea of the algorithm is at each step, each node shares its own information to the neighbors, and the information shared by a node may eventually reach all the nodes of the connected component.
Picture 5.4.1 shows an example of the execution of Label Propagation.



	Node 1	Node 2	Node 3	Node 4	Node 5
Iter 0	1	2	3	4	5
Iter 1	(1) 2 1	(2) 1 3 1	(3) 2 2	(4) 5 4	(5) 4 4
Iter 2	(1) 1 1	(1) 1 2 1	(2) 1 1	(4) 4 4	(4) 4 4
Iter 3	(1) 1 **1**	(1) 1 1 **1**	(1) 1 **1**	(4) 4 **4**	(4) 4 **4**

Figure 51: Example of Label Propagation execution

As we can see, the initial value of $C_{\min}$ is X itself. In the first iteration, node 1 receives the component id of node 2, while node 2 receives the component ids of both node 1 and 3, and so on. Now, the minimum of such received component ids in computed, and the spread operation is repeated until convergence. In the end, the two connected components are retrieved.
Notice that this algorithm well adapt the MapReduce framework, since:

- The Map function produces a list of (destination, message) pairs, where the *destination* is the destination node of the *message* that is spread at each iteration;

- The Shuffle function simply performs a GroupBy operation by key, in this case by *destination*, so all the messages for each destination node are collected;
- The Reduce function produces a list of (destination, minimum value).

Notice that the **complexity** of the algorithm depends on the size of the largest connected component(s).

5.4.2 Hash-To-Min

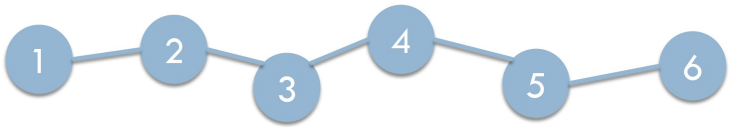
We now discuss another algorithm for collecting the connected components of a directed graph.
In this case the **input** of the algorithm is given by a list of (X, C) pairs, meaning that X knows about nodes in the set C: clearly, when the algorithm starts, the set C is composed of the node X and its neighbors.
The algorithm works as follows:

1. Node X computes $C_{\min}$, the smallest node in C;
2. Node X sends C to node $C_{\min}$;
3. Node X sends $C_{\min}$ to any other node in C;
4. Node X creates a new C by merging all the received messages;
5. Repeat until convergence.

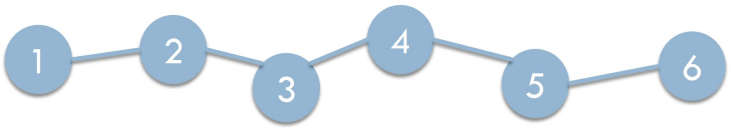
Picture 5.4.2 shows an example of the execution of Hash-To-Min. As we can see:

- When the algorithm starts, each node knows about itself and its neighbors;
- At iteration 1, each node computes $C_{\min}$, and sends C to $C_{\min}$: node 1 sends (1,2) to itself, while it receives (1,2,3) from node 2. Then, each node sends $C_{\min}$ to any other node in C: node 1 sends 1 to node 2, while node 2 sends 1 to node 3 etc..;
- At the end of each iteration, each node merges the received messages, and the operation are repeated until convergence.

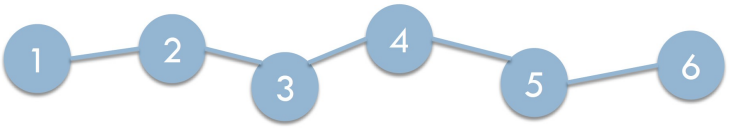
Note that in this case the complexity is $O(\log d)$, where d is the diameter of the graph: the idea is that at each iteration the information of each node nearly doubles.



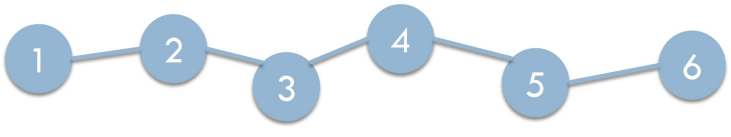
	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6
Iter 0	1,2	1,2,3	2,3,4	3,4,5	4,5,6	5,6
Iter 1	(1,2)	1				
	(1,2,3)	1	1			
		(2,3,4)	2	2		
			(3,4,5)	3	3	
				(4,5,6)	4	4
					5,6	5
	1,2,3	1,2,3,4	1,2,3,4,5	2,3,4,5,6	3,4,5,6	4,5



	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6
Iter 1	1,2,3	1,2,3,4	1,2,3,4,5	2,3,4,5,6	3,4,5,6	4,5
Iter 2	(1,2,3)	1	1			
	(1,2,3,4)	1	1	1		
	(1,2,3,4,5)	1	1	1	1	
		(2,3,4,5,6)	2	2	2	2
			(3,4,5,6)	3	3	3
				(4,5)	4	
	1,2,3,4,5	1,2,3,4,5,6	1,2,3,4,5,6	1,2,3,4,5	1,2,3,4	2,3



	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6
Iter 2	1,2,3,4,5	1,2,3,4,5,6	1,2,3,4,5,6	1,2,3,4,5	1,2,3,4	2,3
Iter 3	(1,2,3,4,5)	1	1	1	1	
	(1,2,3,4,5,6)	1	1	1	1	1
	(1,2,3,4,5,6)	1	1	1	1	1
	(1,2,3,4,5)	1	1	1	1	
	(1,2,3,4)	1	1	1		
		(2,3)	2			
	1,2,3,4,5,6	1,2,3	1,2	1	1	1



	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6
Iter 3	1,2,3,4,5,6	1,2,3	1,2	1	1	1
Iter 4	(1,2,3,4,5,6)	1	1	1	1	1
	(1,2,3)	1	1			
	(1,2)	1				
	1					
	1					
	1					
	1,2,3,4,5,6	**1**	**1**	**1**	**1**	**1**

Figure 52: Example of Hash-To-Min execution

6 Ranking

The topic of **ranking** is central in the usage of massive data, for example is used in:

- **Search**: a search engine takes a query and provides the most relevant pages on the web in an ordered list;
- **Question answering**: find a resource that contains the answer to a question. QA in turns can be divided into *Factoid QA*, whose goal is to find the answer to a factual question (e.g. "The heigth of Monte Bianco"), and *Community QA*, whose goal is to provide similar questions that other asked in the past;
- **Recommendation**: find new content I might be interested in;
- etc..

To tackle the problem of document retrieval, many heuristic ranking models have been proposed and used in IR literature. Recently, given the amount of potential training data available, it has become possible to leverage **Machine Learning (ML)** technologies to build **effective ranking models**. Specifically, we call those methods that learn how to combine predefined features for ranking by means of discriminative learning **"learning-to-rank" methods**.

6.1 What is ranking

In general, the problem of ranking cam be formalized as follows: given a query $q \in Q$ and a (dynamic) collection of documents D, ranking finds a **ordered k-subset** that **maximizes** a function $f : Q \times D^k \to \mathbb{R}^k$.

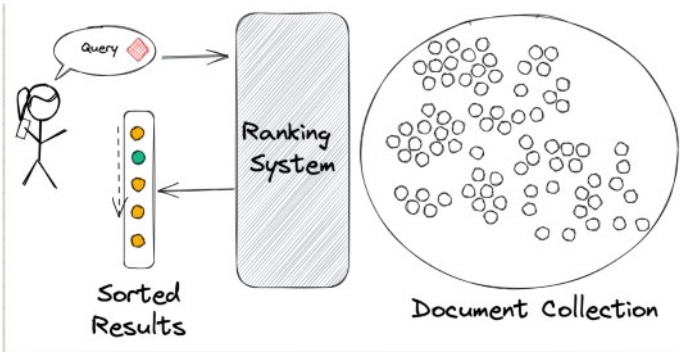


Figure 53: Sketch of the ranking problem

In general, the main challenges when we talk about ranking are:

- How do we evaluate a ranking system?
- How do we represent Q and D?
- What utility do we care about? How do we formulate f?
- How do we find the k-subset from a large collection D as efficiently as possible?

Over a decade ago, machine learning transformed how we approach the document ranking problem and answer the questions above. That wave resulted in a paradigm shift from early statistical methods, heuristics, and hand-crafted rules to determine the relevance of documents to a query, to what would later be called **Learning to Rank** (**LtR**), where the relevance of a document to a query is estimated by a learnt function, hence "learning" to rank. More specifically, this framework comprises of two distinct algorithms, depicted in Picture 6.1: **top-k retrieval**, which finds a subset of k documents that are more relevant to a query, followed by **ranking** which orders the documents in the top-k set.

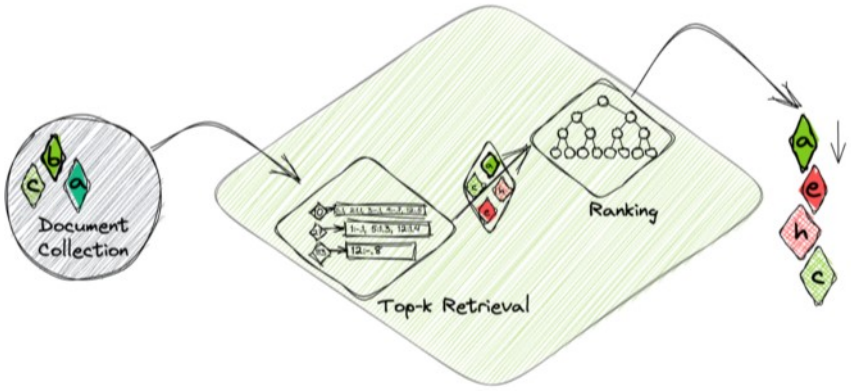


Figure 54: Top-k retrieval and ranking

Usually, the **ranking stage** uses an often **expensive function** that was trained using supervised or online learning methods, while the **retrieval algorithm** solves a form of the **maximum inner product search (MIPS)** problem among the documents and the queries.

6.2 LtR framework and approaches

In general, the following elements are needed to learn a function:

- Ψ, a *dataset* of labeled examples;
- $U(.,.)$, an *evaluation metric*;
- $l(.,.)$, a *loss function*;
- $\phi(.)$, a *representation function*;
- $f(.,\theta)$, the *hypothesis class*, i.e. the relationship between the data and the labels.

Moreover, there exist many approaches for LTR framework:

- **Pointwise**: in this case **each** query-document **pair** is associated with a **score**, and the **objective** of the ML model is to **predict** such **score**. In this sense, it can be considered ad a pure **regression problem** (if the score is "continuous"), or a **multi-class classification problem** (if the score is "discrete"). Notice that this approach does not consider the position of a document into the final ranking;

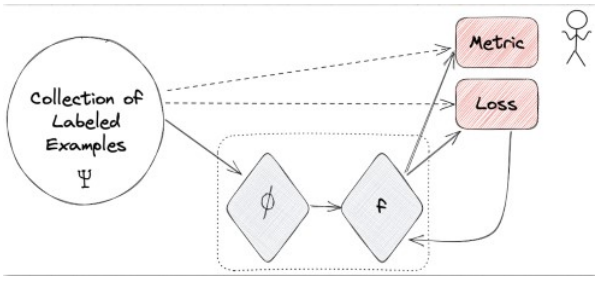


Figure 55: Elements to learn a function

- **Pairwise**: in this case we're given **pairwise preferences**, e.g. d_1 is better than d_2 for a query q, so the **objective** is to **predict** a **score** that **preserves such preferences**. In this sense, it can be considered as a **binary classification problem**. Notice that this approach does not consider the relevance of a document into the final ranking;
- **Listwise**: finally, in this case we're given the **ideal ranking** of results for each query, so the **objective** is to **maximize the quality of the whole resulting ranked list** by exploiting the whole list at training time. This approach is also used in recommendation systems.

In general, a modern ranking architecture should be:

- **Effective**, i.e. the users should be happy of the results they receive;
- **Efficient**, i.e. it should have a low response time (<0.1 sec);
- **Easy to adapt**, i.e. it should perform a continuous crawling of the Web, exploiting users' feedback.

6.3 Review of Supervised Learning

The first topic of supervised learning we cover is **linear regression**. Given m data points $X \in \mathbb{R}^{m \times n}$ and their real values $Y \in \mathbb{R}^m$, we can predict the value of an *unseen* example $x \in \mathbb{R}^n$ using a linear function:

$$f(x) = w^T x + b$$

In this case, the **quality** of the regression function is computed using the **distance** between the **predicted value** and the **actual value**. In particular, many measures exist:

- *L1 distance*: $l(f, y) = |f(x) - y|$;
- *L2 distance*: $l(f, y) = (f(x) - y)^2$.

By exploiting this concept of quality of a function, we can **learn** a regression function by adjusting the parameters of the function, $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$ until the error is minimal. Mathematically, this is translated into the following problem:

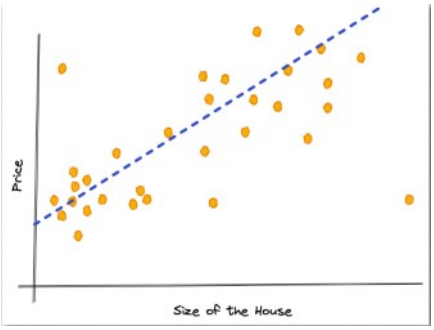


Figure 56: Linear regression

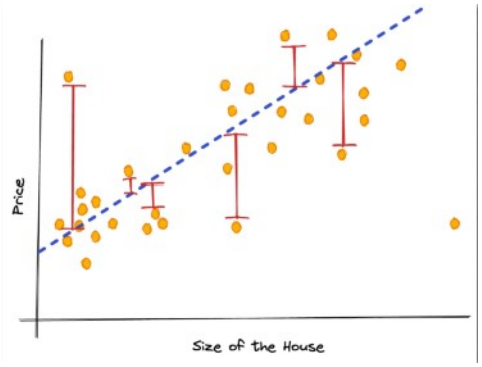


Figure 57: Quality of a linear regression function

$$w^*, b^* = \arg\min_{w,b} \quad L(f, y) = \arg\min_{w,b} \quad \frac{1}{m} \sum_{i=1}^{m} l(f(x_i), y_i)$$

In particular, the *gradient descent* approach exploits the gradients in order to update the parameters, in particular:

1. w and b are initialized randomly;
2. $$w \leftarrow w - \eta \nabla_w L(f, y)$$
   and
   $$b \leftarrow b - \eta \nabla_b L(f, y)$$
3. Return w and b.

In general, if we have a parameterized function $f(.; \Theta) : \mathbb{R}^n \rightarrow \mathbb{R}$, then:

1. Initialize Θ randomly;
2. Update until convergence:
   $$\Theta \leftarrow \Theta - \eta \nabla_\Theta L(f(.; \Theta), y)$$
3. Return Θ.

The other important task in Supervised Learning is **classification**. In this case, the *gradient descent* approach is applied to learn a function $f(.;\Theta) : \mathbb{R} \to \{\pm 1\}$. In this case, there exist many different methods for evaluating the accuracy of the predicted class:

1. *0-1 loss*:
$$l(f, y) = \begin{cases} 0 & \text{if } y = f(x) \\ 1 & \text{otherwise} \end{cases}$$
Notice that the problem with this loss is that it cannot be derived!
2. *Hinge loss*:
$$l(f, y) = \max(1 - y(f(x)), 0)$$

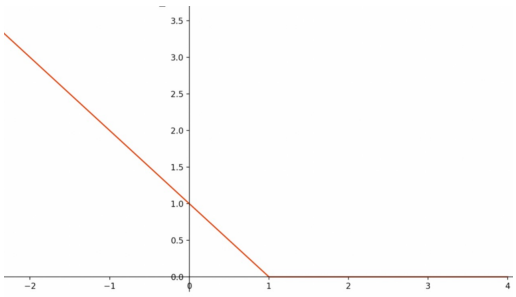


Figure 58: Hinge loss

3. *Logistic loss*, if $y \in \{0, 1\}$:
$$l(f, y) = -y \log \frac{1}{1 + \exp -f(x)} - (1 - y) \log(1 - \frac{1}{1 + \exp -f(x)})$$

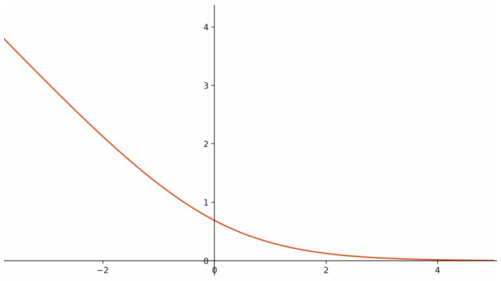


Figure 59: Logistic loss

6.4 Datasets

A typical LtR dataset comprises a set of triplets (q, d, y), where:

- $q \in Q$ represents a **query** ;
- $d \in D$ represents a set of k **documents** ;
- $y \in Y^k$ represents the **labels** (i.e. the relevance) of the documents.

As we can see, this task is more complicated than a classification/regression, since we have multidimensional vectors, and in general the evaluation function is more complicated.

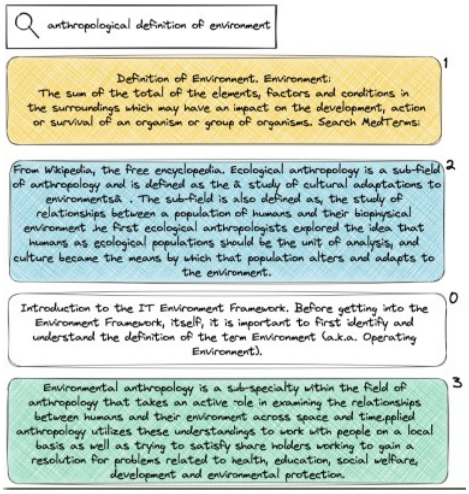


Figure 60: Ranking datasets

6.4.1 Representation

One of the main **challenges** of ranking datasets is the **representation of queries and documents**. Suppose for now that there exists a function $\phi : Q \times D \to \mathbb{R}^{k \times m}$, that turns a query and a set of k documents into m-dimensional vectors, as represented in Picture 6.4.1.

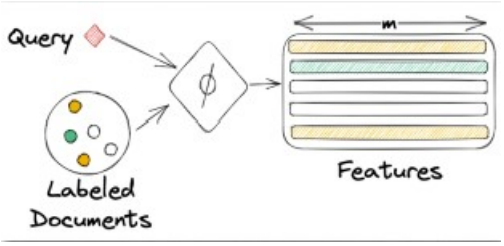


Figure 61: Representation of queries and documents

However, we distinguish three methods for selecting the features that represent queries and documents as vectors:

- **query-only features**, for example *query type, length, topic* etc.. These kind of features are only considered for queries;
- **query-independent features**, for example *PageRank, URL, length, in- and out-links, click count* etc.. These kind of features are only considered for documents;
- **query-dependent features**, for example *query terms, TF-IDF score, BM25 score* etc.. These features are considered for both queries and documents.

6.4.2 Relevance labels

Regarding the entities involved in the ranking task, the labels may come from:

- *explicit feedback*, i.e. human assessors are presented with a query and a ranked list, and they're asked to grade each document w.r.t. the query. Clearly, this approach

is not scalable when the collection of documents becomes very large. Moreover, this approach is characterized by several issues: for example, the cost of annotating queries and ranked lists, or the possibility that a new ranking function returns a document which is not judged etc..;

- *implicit feedback*, i.e. the users interacting with the ranking system collect signals that are indicative of the relevance of the documents. One the one hand, this approach is scalable, but on the other it may suffer from some biases, e.g. the position bias (users more likely click on the first results).

6.5 Evaluation metrics

Another important issue about ranking are the **ranking metrics**, which are used to evaluate the quality of a ranking. As represented in Picture 6.5, after applying the function f to the vector representation of the documents and the query, for each document we obtain a vector of **scores** $s \in \mathbb{R}^k$, which in turn is sorted to obtain the ranking π_s, i.e. the **actual ranking**. Finally, the actual ranking is compared with the **ideal ranking** (given by the sorting of the documents in decreasing order of their relevance labels) by using the ranking metrics.

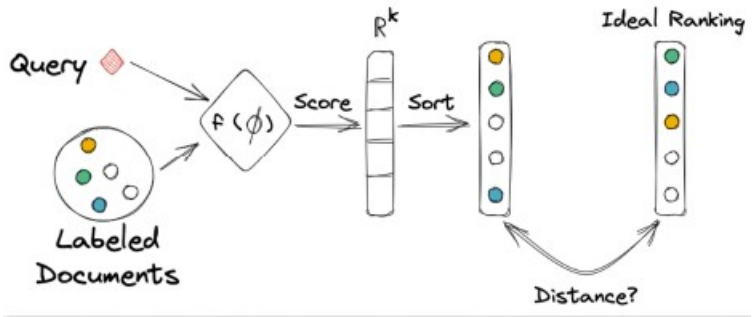


Figure 62: Evaluation of a ranking function

It is helpful to consider the important factors in the way users interact with a ranked list. First, **users expect** the **relative ordering of documents to be correct**. That is, documents that are placed higher in the ranked list (i.e., towards the top of the list) should satisfy the information needs of a user better than documents lower on the list, and that as the user goes down the list, documents become less relevant to the query. Second, user **attention** has a **skewed distribution** with much of it focused on the top of the ranked list. In other words, users typically do not examine all documents at every position with equal probability or care. As such, higher positions carry more weight.
Some examples of ranking metrics are:

- *0-1 loss*, i.e.

$$l(f, y) = \begin{cases} 0 & \text{if } y = f(x) \\ 1 & \text{otherwise} \end{cases}$$

  As we can see, this metric returns 0 if the sorted scores of the actual ranking ($\pi_{f(q,d)}$) are the same of the ones of the ideal ranking (π_y), 1 otherwise;

- *Regression loss*, i.e.
$$l(f, y) = ||f(q, d) - y||_2^2$$
As we can see, this metric measures the squared distance between the actual scores and the ideal scores.

However, these two metrics have some disadvantages. The *0-1 loss*, for example, tells us if a ranked list is correct, in its entirety, or not. That's often **too coarse**. We often don't care how non-relevant documents are ranked with respect to one another, only that relevant documents are above non-relevant ones. That is, if document A is relevant, B, C, and D are not, then A-B-C-D is correct and so is A-C-B-D, but the 0-1 metric considers one of these correct and the other incorrect. On the other hand, a regression loss evaluates whether the ranking function predicted the label of a document (0, 1, 2, etc.). But labels are only used in an ordinal sense: 1 does not mean integer 1, it just indicates a document labeled as 1 is more relevant than a document labeled as 0. So predicting the labels makes little sense.
In this sense, we can introduce some other metrics:

- *Kendall's* τ, i.e.:
$$\tau = \frac{2}{n(n-1)} \sum_{i<j} sign(y_i - y_j) sign(f_i - f_j)$$
, which can also be reformulated as:
$$\tau = \frac{\text{(number of concordant pairs) - (number of discordant pairs)}}{\text{number of pairs}}$$
In this sense, the *Kendall's* τ captures the misclassification rate. If we consider, for example, the actual ranking and the ideal ranking in Picture 6.5, then $\tau = \frac{5-5}{10}$, since the concordant and discordant pairs are the one represented in Picture 6.5.

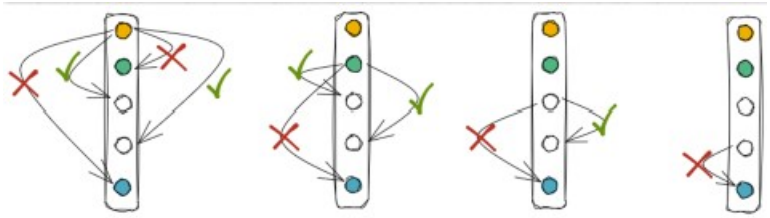


Figure 63: Concordant and discordant pairs of Kendall's τ coefficient

The main drawback of this metric is that the top and the bottom elements of the ranking are treated equally, while we know that they have different importance.

- *ReciprocalRank@k* or *RR@k*, which is defined as the inverse of the rank of the first positive document, i.e. the first document among the top-k of the ranking which has the same ranking in the ideal ranking. If none of the top-k ranked documents is contained in the ideal ranking, the reciprocal rank is 0.

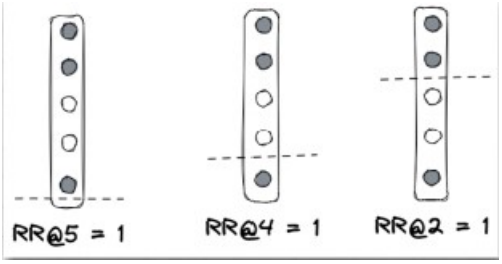


Figure 64: RR@k

$$RR@k = \frac{1}{\text{rank}_i}$$

The main drawback of this metric is that we only care about the first relevant document, and this is showed in the Picture 6.5, where the RR has the same values for all the values of k, since the ranking of the first relevant document is 1.

- *Precision@k* or *P@K*, which is defined as the fraction of documents x_i in the top k set for which $y_i > 0$, i.e. the fraction of documents in the top k that are relevant.

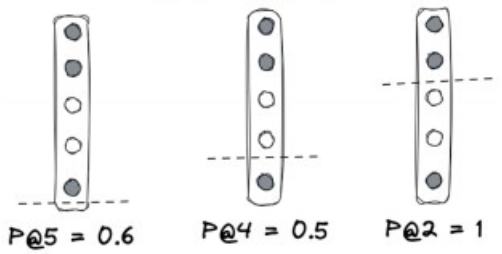


Figure 65: P@k

The two main drawbacks of this metric are that it does not take into account the order of the documents, but it only considers the fraction of relevant doc, and tho other issue is that when $k \to \infty$, then $P@k \to 0$, which in general is not a good behaviour.

- *Recall@k* or *R@k*, which is defined as the proportion of relevant documents that are retrieved in the first k positions.

  This metric is quite useful, since it can be combined together with the *P@k* metric to produce the *Precision-Recall curve*, for which we can compute AUC. An example is provided in Picture 6.5.

- *AveragePrecision@k* or *AP@k*, which combines *R@k* and *P@k*:

$$AP@k = \frac{1}{\sum_i 1_{y_{\pi_i}>0}} \sum_{i=1}^{k} 1_{y_{\pi_i}>0} P@i$$

- *DiscountedCumulativeGain@k* or *DCG@k*, which tries to focus on the behaviour of a user when a ranked list is presented (Picture 6.5), and

Figure 66: Precision-Recall curve

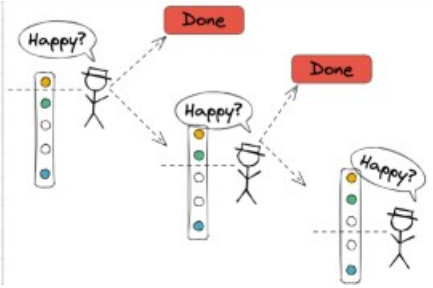


Figure 67: Usual behaviour of a user

it is defined as:

$$DCG@k = \sum_{r=1}^{k} Gain(y_{\pi_r}) \cdot Discount(r)$$

, where

$$Gain(y) = 2^y - 1$$

and

$$Discount(r) = \frac{1}{\log_2(r+1)}$$

We can notice that:

- the input of the *Gain* function is the label of the document in position r of the ranking; if the label is 0, then $Gain = 0$, if the label is 1, then $Gain = 1$;
- the *Discount* function is correlated to the probability of a document of being visited.

- *NDCG@k*, i.e. the normalization of *DCG@k* by its maximum value.

In general, by now we've considered only metrics that are related to a single query: given a test set of N ranking samples $\Psi = \{(q_i, D_i, Y_i)\}_{i==1}^{N}$ we can consider the mean of the metrics of interest over the entire set:

$$MAP@k = \frac{1}{N} \sum_{i} AP@k(\pi_{f(q_i, D_i)}, Y_i)$$

$$MeanNDCG@k = \frac{1}{N} \sum_i NDCG@k(\pi_{f(q_i,D_i)}, Y_i)$$

, where $\pi_{f(q_i,D_i)}$ denotes the ranked list induced by evaluating $f(q, D)$.

6.6 Loss functions

After considering how the dataset is produced and the most important metrics that are used to evaluate a ranking system, we now focus on the **loss functions**. If we consider again the Picture 6.6, we notice that the loss is used to train our ranking function, and it is used in the training phase.

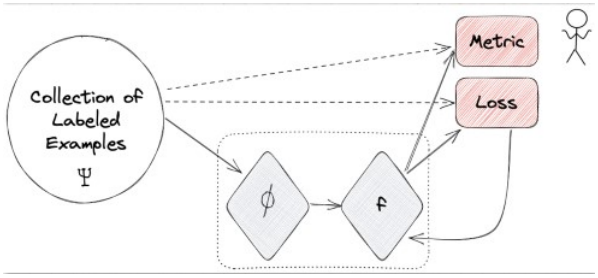


Figure 68: Elements to learn a function

Depending on the objective of our problem, we may consider many different losses:

- If we consider **ranking** as a **classification** or **regression** problem, i.e. each document has a score of relevance (0/1 in case of classification, other scores in case of regression), then we may consider **pointwise losses**. In this case **each** query-document **pair** is associated with a **score**, and the **objective** of the ML model is to **predict** such **score**. In this sense, it can be considered ad a pure **regression problem** (if the score is "continuous"), or a **multi-class classification problem** (if the score is "discrete"). Notice that this approach does not consider the position of a document into the final ranking;
- If we consider **ranking** as a **preference learning** problem, then we may consider **pairwise losses**. In this case we're given **pairwise preferences**, e.g. d_1 is better than d_2 for a query q, so the **objective** is to **predict** a **score** that **preserves such preferences**. In this sense, it can be considered as a **binary classification problem**. Notice that this approach does not consider the relevance of a document into the final ranking;
- If we consider **ranking** as a **permutation** or **probabilistic** problem, then we may consider **listwise losses**. Finally, in this case we're given the **ideal ranking** of results for each query, so the **objective** is to **maximize the quality of the whole resulting ranked list** by exploiting the whole list at training time. This approach is also used in recommendation systems.

6.6.1 Pointwise losses

We begin our discussion with the **regression loss**. In this case:

$$l(f, y) = ||f(q, d) - y||_2^2$$

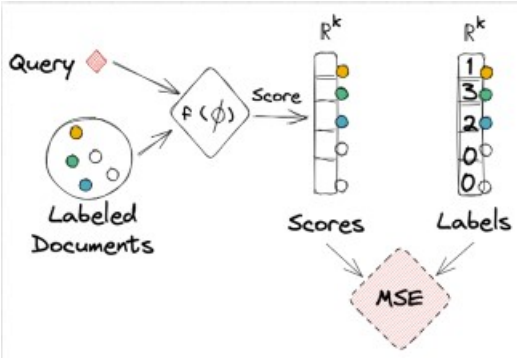


Figure 69: Regression loss

If we consider **classification loss**, we could either use the *0-1 loss*:

$$l(f, y) = \begin{cases} 0 & \text{if } y = f(x) \\ 1 & \text{otherwise} \end{cases}$$

or if we assume $f(q, d) = [g(q, d_1), g(q, d_2), .., g(q, d_k)]$ where $g(q, d) : Q \times D \to \mathbb{R}$, and given a set of training examples $\Psi = \{(q_i, d_i, y_i)\}_{i=1}^N$, then the learning objective becomes the cross entropy:

$$L(f) = \frac{1}{N} \sum_{(q,d,y)\in\Psi} \sum_{i=1}^{k} -y_i \log \frac{1}{1 + e^{-g(q,d_i)}} - (1 - y_i) \log[1 - \frac{1}{1 + e^{-g(q,d_i)}}]$$

, where $\frac{1}{1+e^{-g(q,d_i)}} = sigmoid(g(q, d_i))$. In this case the gradient of L w.r.t. g is:

$$\frac{\partial}{\partial g} L(f) = -\frac{1}{N} \sum_{(q,d,y)\in\Psi} \sum_{i=1}^{k} (y_i - \frac{1}{1 + e^{-g(q,d_i)}})$$

Since the **input** object in the **pointwise approach** is a **single document**, the **relative orde**r between documents **cannot** be naturally **considered** in the learning process, although **ranking** is **more about predicting relative order** than accurate relevance degree. Moreover, the **position** of **each document** in the ranked list is **invisible** to the **pointwise loss functions**. Therefore, the pointwise loss function may unconsciously overemphasize those unimportant documents (which are ranked low in the final ranked list and thus do not affect user experiences).
Given the above problems, the pointwise approach can only be a **sub-optimal solution to ranking**. To tackle the problem, people have made attempts at regarding a document pair, or the entire group of documents associated with the same query, as the input object. This results in the *pairwise* and *listwise* approaches of learning to rank. With the pairwise approach, the relative order among documents can be better modeled. With the listwise approach, the position information can be visible to the learning-to-rank process.

6.6.2 Pairwise loss

If we consider **pairwise loss**, we could consider Kendall's τ:

$$\tau = \frac{2}{n(n-1)} \sum_{i<j} sign(y_i - y_j) sign(f_i - f_j)$$

, which can also be reformulated as:

$$\tau = \frac{\text{(number of concordant pairs) - (number of discordant pairs)}}{\text{number of pairs}}$$

Another possibility is to consider **RankNet's loss function**.
Assume $f(q,d) = [g(q,d_1), g(q,d_2), .., g(q,d_k)]$ where $g(q,d) : Q \times D \to \mathbb{R}$, and given a set of training examples $\Psi = \{(q_i, d_i, y_i)\}_{i=1}^{N}$, then *RankNet*'s learning objective is:

$$L(f) = \frac{1}{N} \sum_{(q,d,y)\in\Psi} \sum_{i,j} -p_{ij} \log \frac{1}{1+e^{-(g_i - g_j)}} - p_{ij} \log \frac{1}{1+e^{-(g_j - g_i)}}$$

, where $g_i \triangleq g(q, d_i)$,

$$p_{ij} = \begin{cases} 1 & \text{if } y_i > y_j \\ 0.5 & \text{if } y_i = y_j \\ 0 & \text{if } y_i < y_j \end{cases}$$

and

$$-p_{ij} \log \frac{1}{1+e^{-(g_i - g_j)}} - p_{ij} \log \frac{1}{1+e^{-(g_j - g_i)}} = l_{ij}$$

In this case the gradient is:

$$\frac{\partial}{\partial g_i} l_{ij} = -p_{ij} + \frac{1}{1+e^{-(g_i - g_j)}} = -\frac{\partial}{\partial g_j} l_{ij}$$

If we denote with $l = \sum_{i,j} l_{ij}$ and let Θ be the parameters of g, we can write:

$$\begin{aligned} \nabla_\Theta l &= \sum_{i,j} \frac{\partial l_{ij}}{\partial g_i} \nabla_\Theta g_i + \frac{\partial l_{ij}}{\partial g_j} \nabla_{\Theta g_j} \\ &= \sum_{i,j} \frac{\partial l_{ij}}{\partial g_i} (\nabla_\Theta g_i - \nabla_{Theta} g_j) \\ &= \sum_i \left(\sum_{j:y_i > y_j} \frac{\partial l_{ij}}{\partial g_i} - \sum_{j:y_i < y_j} \frac{\partial l_{ij}}{\partial g_i} \right) \nabla_\Theta g_i \end{aligned}$$

, where

$$\sum_{j:y_i > y_j} \frac{\partial l_{ij}}{\partial g_i} - \sum_{j:y_i < y_j} \frac{\partial l_{ij}}{\partial g_i} = \lambda_i$$

Each λ_i can be thought as a force that "pulls" a document down the list of that "pushes" it up the list. We could amplify this force based on the rank of document d_i in the following way:

$$\nabla_{\Theta l} = \sum_i \left(\sum_{j:y_i>y_j} \frac{\partial l_{ij}}{\partial g_i} |\Delta Z_{ij}| - \sum_{j:y_i<y_j} \frac{\partial l_{ij}}{\partial g_i} |\Delta Z_{ij}| \right)$$

, where $|\Delta Z_{ij}|$ is the amount of change in the ranking metric Z is d_i and d_j were to trade ranks.
It seems that the pairwise approach has its advantages as compared to the pointwise approach, since it can model the relative order between documents. However, in some cases, it faces even larger challenges than the pointwise approach, for example the problem of the pointwise approach when documents are distributed in an imbalanced manner across queries. Here this issue becomes even more serious in the pairwise approach. Considering that every two documents associated with the same query can create a document pair if their relevance degrees are different, in the worse case, the pair number can be quadratic to the document number. As a result, the difference in the numbers of document pairs is usually significantly larger than the difference in the numbers of documents.

6.6.3 Listwise loss

Finally, we can now consider **listwise loss**: since we cannot directly optimize a *DCG@k* susing gradient descent since it is a discontinuous function, the idea is to replace it with a smooth surrogate, i.e.

$$DCG@k = \sum_{r \leq k} \frac{2^{y_r} - 1}{\log_2(\pi^{-1}(d_r) + 1)}$$

and we can approximate the rank of document d by writing

$$\pi^{-1}(d_r) = 1 + \sum_{i \neq r} 1_{f_r < f_i} \approx 1 + \sum i \neq r \frac{1}{1 + e^{-\alpha(f_i - f_r)}}$$

Another possibility, provided by the **ListNet loss function**, is to apply the *softmax* function to both the set of labels Y and to the scores $f(q, d)$. In general, we have that $softmax(x_1) = \frac{e^x_1}{\sum_i e^{x_i}}$, so for $y \in \mathbb{R}^n$ and $f(q, d) \in \mathbb{R}^n$ we have that the *ground truth probability* is:

$$p_i = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

, while the *predicted probability* is:

$$q_i = \frac{e^{f_i}}{\sum_j e^{f_j}}$$

Finally, we can compute the *cross-entropy* as:

$$-\sum_i p_i \log q_i$$

Intuitively speaking, they model the ranking problem in a more natural way than the pointwise and pairwise approaches, and thus can address some problems that these two approaches have encountered. As we have discussed in the previous sections, for the pointwise and pairwise approaches, the position information is invisible to their loss functions, and they ignore the fact that some documents (or document pairs) are associated with the same query. Comparatively speaking, the listwise approach takes all the documents associated with the same query as the input and their ranked list (or their relevance degrees) as the output. In this way, it has the potential to distinguish documents from different queries, and to consider the position information in the output ranked list in its learning process.

6.7 Ranking functions

After discussing the loss functions, we now focus on **ranking functions**. We will mainly focus on three different families of ranking functions:

- Gradient Boosting Decision Trees (GBDTs);
- Neural Networks and Pre-Trained Transformers;
- Inner Product of Dense or Sparse Vectors (or both).

6.7.1 Gradient Boosting Decision Trees (GBDTs)

In this family of ranking functions we wish to learn an additive function

$$F^{(M)}(x) = \sum_{t=1}^{M} f_t(x)$$

, where each $f_t(x)$ is a weak learner, that at iteration m learns to predict the residual error:

$$r_m(x) = -\frac{\partial L(F)}{\partial F}|_{F=F^{m-1}(x)}$$

The weak learner of GDBTs are the **regression trees**, which are models that recursively partition the dataset into two subsets by thresholding on a *feature value*: when no further splits are possible, the node of the tree is a *leaf* and it is assigned with the output of the regression. An example is provided in Picture 6.7.1.

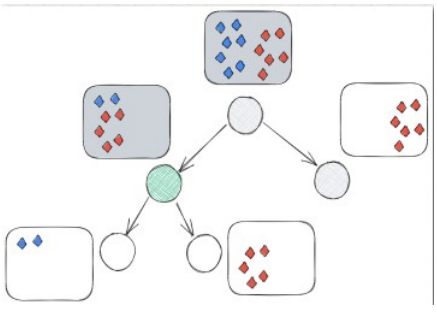


Figure 70: Regression tree

In general, the more leaves, i.e. the more splittings, a regression tree contains, the less generalization in the prediction we have.
Now, we can consider the **empirical error** of a regression tree f made up of leaves L_j with values C_j:

$$L(f) = \sum_i (y_i - f(x_i))^2 = \sum_{L_j} \sum_{i:x_i \in L_j} (y_i - C_j)^2$$

, i.e. for each leaf, we compute the squared difference between the actual value and the predicted one.
The **splitting criterion** of a node $L*$, according to a feature $k*$ and a threshold $\theta*$ is to choose:

$$L*, k*, \theta* = \arg\min_{L_j,k,\theta} (\min_l \sum_{i:L_j \ni x_{ik} < \theta} (y_i - l)^2 + \min_r \sum_{i:L_j \ni x_{ik} \geq \theta} (y_i - r)^2)$$

In other words, we want to split the node $L*$, according to a feature $k*$ and a threshold $\theta*$ in a way that minimizes the sum between the empirical error of the left partition and the one of the right partition. The splitting is performed until a stopping criterion is reached, e.g. the number of data points in a leaf is below a certain limit etc..

6.7.2 Neural Networks and pre-trained Transformers

One problem of *GBDTs* is that we have to provide ourselves the features to the model, and in general the feature engineering operation is costly. Thus, in this section we focus on the following problem: can we learn $\phi(q, d)$, i.e. the representation of the queries and the documents (possibly in conjunction with f)?
For example, *Skip-gram* learns to predict neighboring terms given a current term, or *Continuous Bag of Words* learns to predict a term given its neighboring terms: these methods produce **static embeddings**. However, there exist some methods that exploit the contextual information in order to represent the words, like **BERT**.

6.7.3 Complexity of ranking functions

In general, the complexity of a ranking system is given by the following costs:

- the cost of **model training**;
- the cost of **feature extraction**;
- the cost of **model inference**, which may involve hundreds to thousands of decision nodes to traverse or large tensor multiplication.

In this section we focus on methods for reducing the inference complexity.
Knowledge distillation The first method we consider is **knowledge distillation**, which can be described with the following sentence: *given a large model, find a smaller, more efficient model that is just as effective.* In other words, the idea of *knowledge distillation* is to shrink the model in order to produce a more efficient model that is as effective as the original one.

When considering GDBTs, this technique could be implemented by **pruning the nodes in the trees**, in order to generate shallower and more balanced trees. For example, we could stop generating nodes when the number of nodes $n \geq \alpha(2^{d+1}-1)$, where d indicates the depth of the tree. Another techinque could be computing the impact scores of the trees, **discarding the one with lowest impact**, repeating this operation until p percent of the trees are removed. At each tree discard, the other trees are re-weighted. Finally, one last method could be **learning a neural network from a tree**, since NNs are more efficient and more predictable.

Early exit algorithm Here, the idea is to perform an approximate inference by stopping the model before its conclusion.
In the case of GBDTs, we have that:

$$F(x) = \sum_i f_i(x)$$

, i.e. the output is the sum of the inference of each decision tree f_i, where $x = \phi(q, d) \in \mathbb{R}^m$, as represented in Picture 6.7.3.

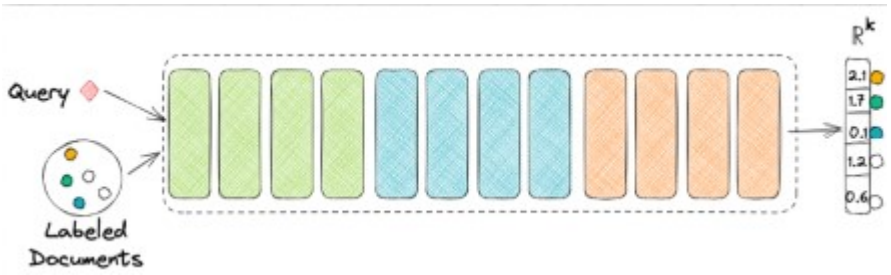


Figure 71: GBDT for ranking

In this sense, the idea of early exit is to compute a "partial" output using a certain number of trees, and then train a classifier that decides whether a document has to be discarded or not, as represented in Picture 6.7.3. Notice that the decision of discarding a document can rely either on a certain score threshold or on a rank threshold.

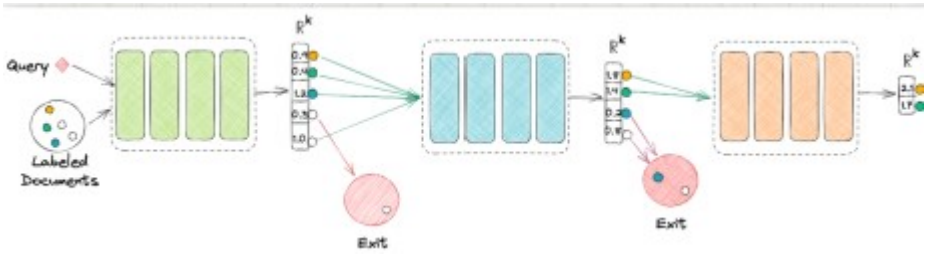


Figure 72: GBDT for ranking with early exit

A similar method applies for NNs, where a classifier can be inserted in each exit point in order to discard some documents by comparing the current approximate ranking with the ideal one. In this sense, the classifier becomes part of the ranking model, as in the case of GDBT.

Ranking cascades Thus far we have not considered the case in which a set of documents is provided as input to the ranking model. Due to the dimensionality of the set

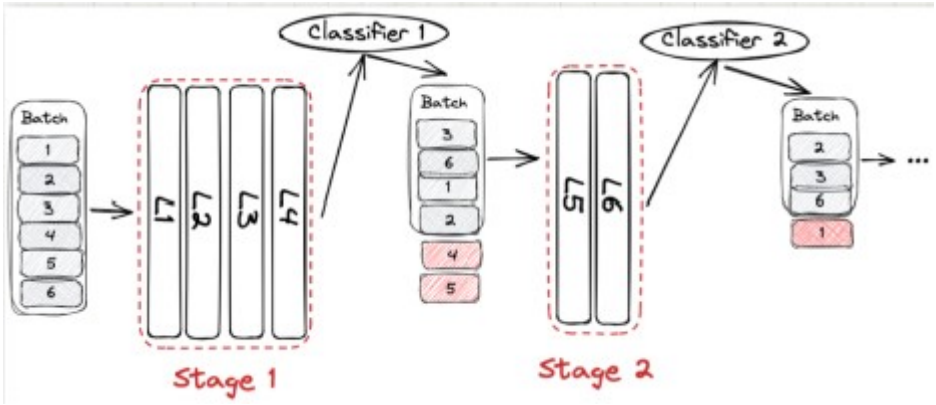


Figure 73: NN with exit points

of documents, applying the same sophisticated ranking function to each of the document becomes very expensive (Picture 6.7.3): for this reason, the idea of **ranking cascades** is to apply **progressively more complex models** to **progressively smaller** set of **documents**, as represented in Picture 6.7.3.

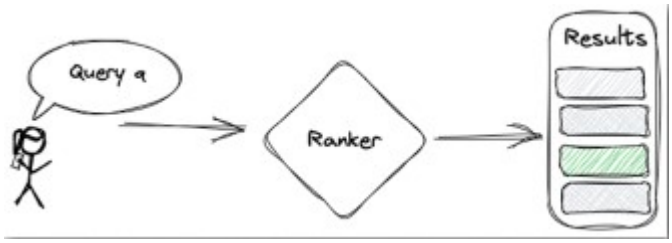


Figure 74: Normal ranking model

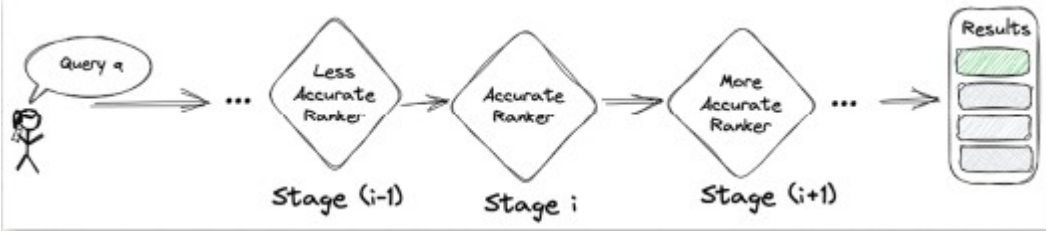


Figure 75: Ranking cascade model

In this sense, the idea of *ranking cascade* is to use less sophisticated ranking functions to provide faster and less accurate evaluatations on the majority of the documents, and use more sophisticated ranking functions are used to provide more accurate evaluations on few documents. Again, the main idea is that at each evaluation stage some documents are pruned in order to make the evaluation of more sophisticated models feasible in terms of computation time.
More specifically,

1. Stage i evaluates $f_i(q, d^{(i)})$ on the set of documents $d^{(i)}$;

2. The set $d^{(i)}$ is pruned and reduced to $d^{(i+1)}$ where $|d^{(i+1)}| \leq |d^{(i)}|$;
3. $d^{(i)}$ is passed to stage $i + 1$.

In this sense, we will see that **linear functions**, for example **dot product**, can be used to get rid of the irrelevant documents, in an operation called **first stage retrieval**.

7 Retrieval with MIPS

In Section 6.7.3 we introduced the notion of **first stage retrieval**, which is an operation that takes as input the vector representations of documents and queries and produces a ranking of the documents w.r.t. the queries. The main features of this operation are:

- it is **cheap**, since in involves linear function, such as the *inner product*, so it can be implemented for very large collections of documents and queries;
- it is **effective**, i.e. it allows to discard all the non relevant documents, in order to reduce the size of the corpus for the following stages of the cascade.

In this section we address the first stage retrieval as a **MIPS** (or **Maximum Inner Product Search**) problem, which can be described as follows: given a query vector $q \in \mathbb{R}^n$ and a set of document vectors $D \subset \mathbb{R}^n$, the goal is to find the **top-k documents** with maximal inner product with q, i.e.:

$$\varsigma = \max_{d \in D}^{(k)} \langle q, d \rangle$$

In general, the solution of this problem scales well for very large collections since the documents representation ϕ_d can be pre-computed, so the only operation that are done at query time are:

- representation of the queries ϕ_q;
- computation of the dot product.

There exist some algorithms for solving the *MIPS* problem when the queries and the documents are provided as sparse and dense vector, but in general the brute-force approach has a complexity of $O(nm + n \log k)$, where:

- nm is given by the dot product;
- $n \log k$ is given by the sorting operation of the top-k results, using HeapSort.

In Section 7.1 and 7.2 we discuss two different methods for representing the queries and the documents, while in Section 7.3 and 7.4 we address some algorithms for solving the MIPS problem when the documents and the queries are either sparse or dense.

7.1 Sparse representation

In general, when we're asked to measure the relevance of a document w.r.t. an input query, there exist two main indicator of relevance:

- the **frequency** of a query term in the document;
- the **propensity** of a term in the document, i.e. how much information the term provides to the document. Intuitively, the more common a term is, the less information it conveys.

The first indicator is measured by the **term frequency (TF)**: for a query term $q_t \in q$, the term frequency w.r.t. a given document d is defined as:

$$TF(q_t, d) = |\{d_i | q_t = d_i, d_i \in d\}|$$

, while the second one is measured by the **inverse document frequency (IDF)**: for a query term $q_t \in q$, the inverse document frequency w.r.t. the collection of documents D is defined as:

$$IDF(q_t) = \log(1 + \frac{|D| - DF(q_t) + 0.5}{DF(q_t) + 0.5})$$

, where:

- $DF(q_t) = |\{d | q_t \in d, d \in D\}|$ is the **collection frequency** of q_t, i.e. the number of documents containing the query term q_t;
- 0.5 is a quantity that ensures that we do not divide by 0.

Another possible measure for the frequency of a term is provided by the $\mathbf{TF_{BM25}}$, which is define as:

$$TF_{\text{BM25}}(q_t, d) = \frac{TF(q_t, d)(k_1 + 1)}{TF(q_t, d) + k_1(1 - b + b\frac{|D|}{l})}$$

, where k_1 and b are hyperparameters and l is the average lenghts of the documents in the collection. Notice that in this case, $0 \leq TF_{\text{BM25}} \leq 1$.
In this sense, we can compute the impact score of query-document pair in the following sum:

$$\text{score}(q, d) = \sum_{q_t \in q} TF(q_t, d) \cdot IDF(q_t)$$

or

$$\text{score}_{\text{BM25}}(q, d) = \sum_{q_t \in q} TF_{\text{BM25}}(q_t, d) \cdot IDF(q_t)$$

From a mathematical point of view, the *sparse* score of a document w.r.t. a given query is basically represented by the dot product between the TF (or TF_{BM25}) and the IDF vectors. Let V be a vocabulary, $\vec{d} \in \mathbb{R}_+^{|V|}$ be a vector whose i-th coordinate records the importance of the i-th term in V, i.e. the TF (in the case of BM25, the TF_{BM25}), and $\vec{q} \in \mathbb{R}_+^{|V|}$ be a vector whose i-th coordinate records the count of the i-th term of V in the query, i.e. the IDF, then:

$$\text{score}(q, d) = \langle \vec{q}, \vec{d} \rangle$$

,or

$$\text{score}_{\text{BM25}}(q, d) = \langle \vec{q}, \vec{d} \rangle$$

Notice that in this case:

- $\vec{q}$ and $\vec{d}$ are **sparse** vectors, i.e. very few coordinates are non-zero in each vector, and every coordinate is non-zero in very few vectors;
- $\vec{q}$ and $\vec{d}$ are **non-negative** vectors.

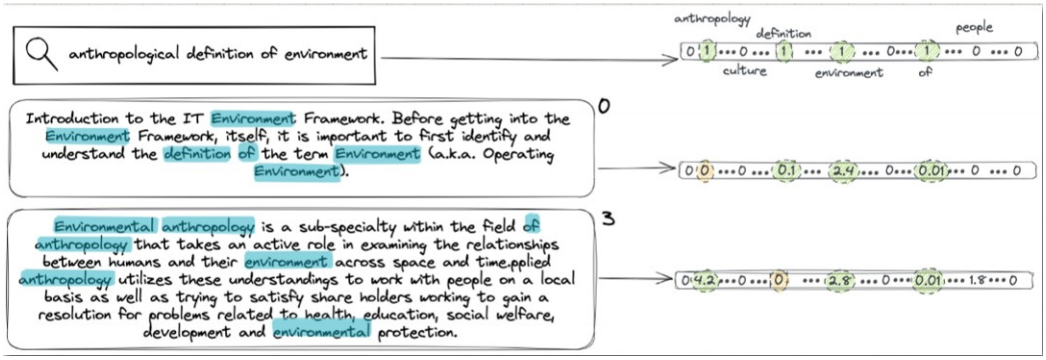


Figure 76: Sparse representation of documents w.r.t. an input query

7.1.1 Learning term importance

We now address the problem of learning the *term frequency*, and in particular we consider the **Deep Contextualized Term Weighting** (or **DeepCT**). In this case each term frequency is defined as:

$$TF(q_t, d) = \begin{cases} w(q_t; d), & q_t \in d \\ 0, & \text{otherwise} \end{cases}$$

, where w is a regression function that predicts a value given a term in the context of a document. In this case the **training set** is represented by the collection D of query-document pairs, and the objective is the **MSE** associated with the following metric:

$$QTR(t, d) = \frac{|\{q|(q, d) \in D \wedge \ t \in d\}|}{|\{q|(q, d) \in D\}|}$$

, i.e. the fraction of queries that contain the term t.
Another important concept in ranking is the **query expansion**, that is represented in Picture 7.1.1.

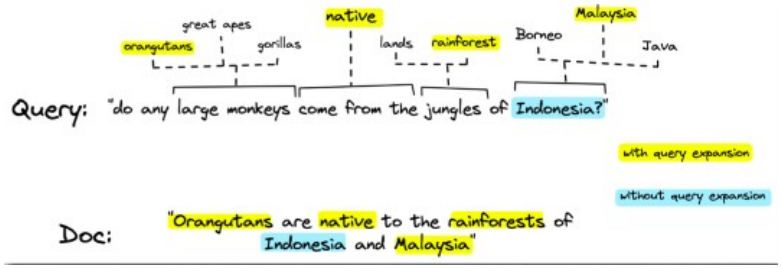


Figure 77: Example of query expansion

As we can see, the term overlap between the document and the query without query expansion is represented only by the term *Indonesia* (**vocabulary mismatch** problem), while exploiting the query expansion method we notice that the document results to be very relevant w.r.t. the query. A famous model for query/document sparse expansion is **SPLADE**, which exploits the BERT MLM (Masked Language Model) for encouraging sparsity via regularization. The main features of this model are that it produces sparse expansion which are also semantically similar to the original terms of query/document.
Finally, one last important concept is **query prediction**, i.e. predicting a possible query for an input document. This operation works as follows: the query predictor receives as input a document d, and it produces as output the top-k query terms with highest probability. Then, the term with highest probability is sampled, and it is provided as input again to the query predictor: as scheme of the functioning is provided in Picture 7.1.1.

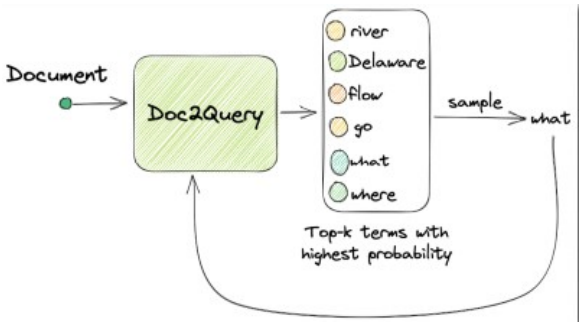


Figure 78: Example of query prediction

A famous model for query prediction is **Doc2Query**, which is characterized by the following advantages:

- All the operations are done offline, since the query does not need any type on transformation;
- It helps the vocabulary mismatch problem, since it performs an expansion using terms that are not present in the document;
- It optimizes log-likelihood: $-\sum_{i=1}^{|q|} \log P[q_i|q_{<1}, d]$

7.2 Dense representation

The idea of **dense representation** is that instead of learning a function $\phi(q, d)$ to represent a query-document pair (Picture 7.2), we can separately learn ϕ_q and ϕ_d to represent query and documents, respectively (Picture 7.2). In this way, we can perform the dot product (linear operation) between the two representations, which have a good quality, since the similarity of original queries/documents is preserved in the dense vectors.
We recall that we introduced *Skip-gram* and *Continuous Bag of Words* for representing **words** as vectors, but there exist some models for representing **sentences** as vectors, such as **Sentence BERT**. This model is trained on pairs of similar sentences: it takes as input two sentences and produces in output the probability that the sentences are similar, as showed in Picture 7.2.

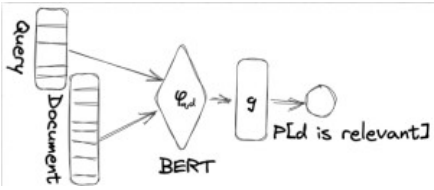


Figure 79: Original query-document representation

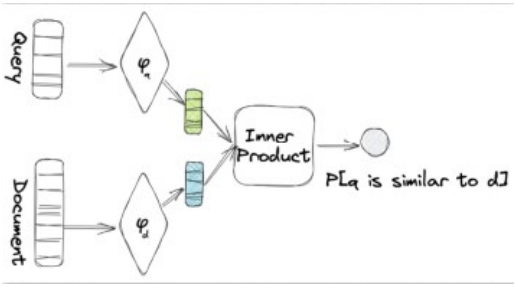


Figure 80: Dense representation

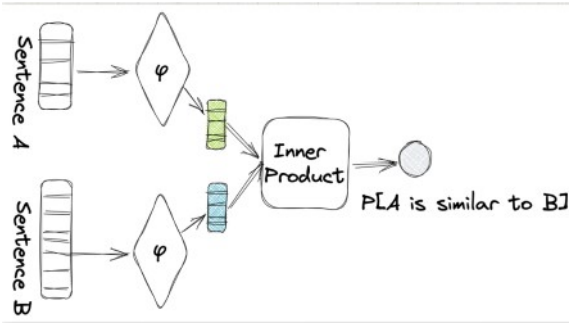


Figure 81: Sentence BERT

However, since our goal is to assess the relevance of a (long) document w.r.t. a (short) query, if a term is rare (whether in query or document), its embedding less accurately captures its semantics. That's because the embedding model doesn't see all the term's usage examples and contexts. So its model of the semantics of a rare term in a sentence is more noisy.
Finally, a document can also be split up into several passages, so we can consider a model that represents **passages** as vectors, such as **Deep Passage Retrieval (DPR)**. The goal of this model is to learn two different representations ϕ_q and ϕ_d, and it is trained on a dataset composed of a query, a positive document and many negative documents. Then, the model optimizes an inner product ranker's loss to learn better representations ϕ_q and ϕ_d.
Clearly, the peculiarity of this model is represented by the presence of **negative documents** in the training set, but where do these negative docs come from? Intuitively, any non-positive example is good, so which one do we pick?
A possible strategy could be the following one:

1. Compute the BM25 score between the query and each of the document in the collection;

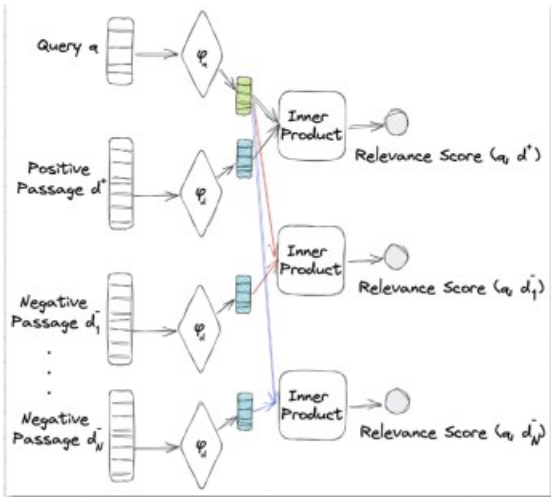


Figure 82: Deep Passage Retrieval (DPR)

2. Select another document randomly form the corpus (d^+ in the Picture);
3. Perform in-batch negative sampling among the non-relevant documents, i.e. all the documents $d \neq d^+$. Usually, the in-batch negative sampling is performed by providing a batch as input to a model, then propagate the error of the loss function back to the model, in order to change its parameters.

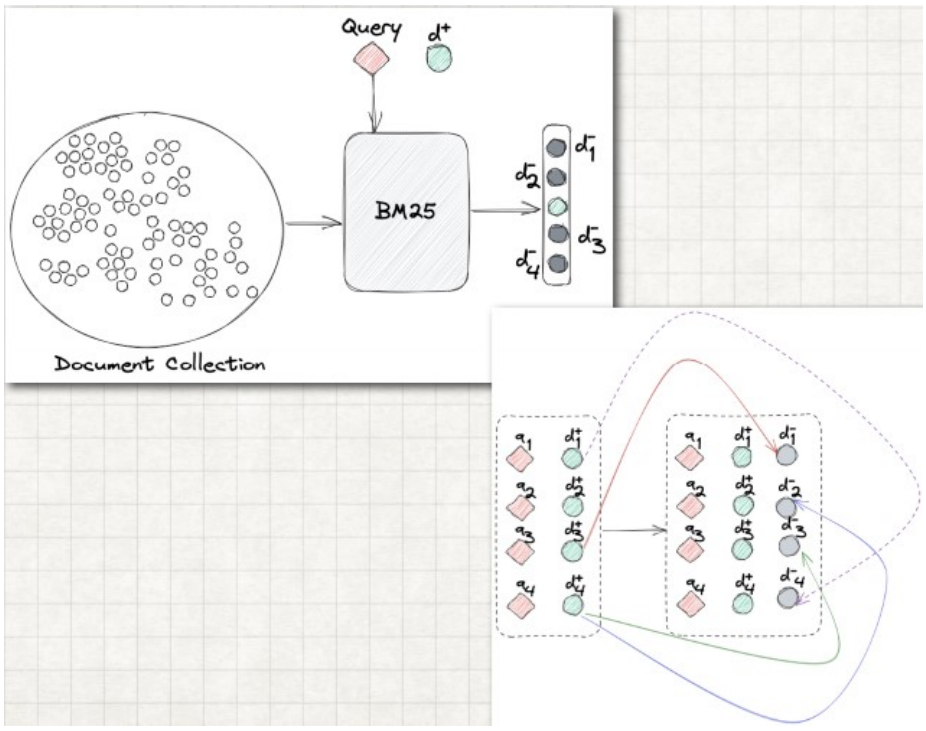


Figure 83: A possible strategy for negative sampling

The **advantage** of this method is that the non-relevant documents retrieved by *BM25* are syntactically relevant w.r.t. the query, but not semantically, so choosing them as negative

examples for the model is very useful for improving its accuracy. However, this method suffers of two **disadvantages**:

- If the queries are very similar, then the documents are very similar too, so the negative examples that are retrieved by the method are not significant;
- On the other hand, if the queries are very dissimilar, the negative examples are trivial to be learned.

Another possibile method is **ANCE**, which finds negative examples by retrieving over the current learnt representations, and it works as follows:

1. Computes the embedding for each document of the corpus using the representation function ϕ;
2. Performs MIPS to retrieve the non-relevant documents;
3. The non-relevant documents are used for training the representation function ϕ.

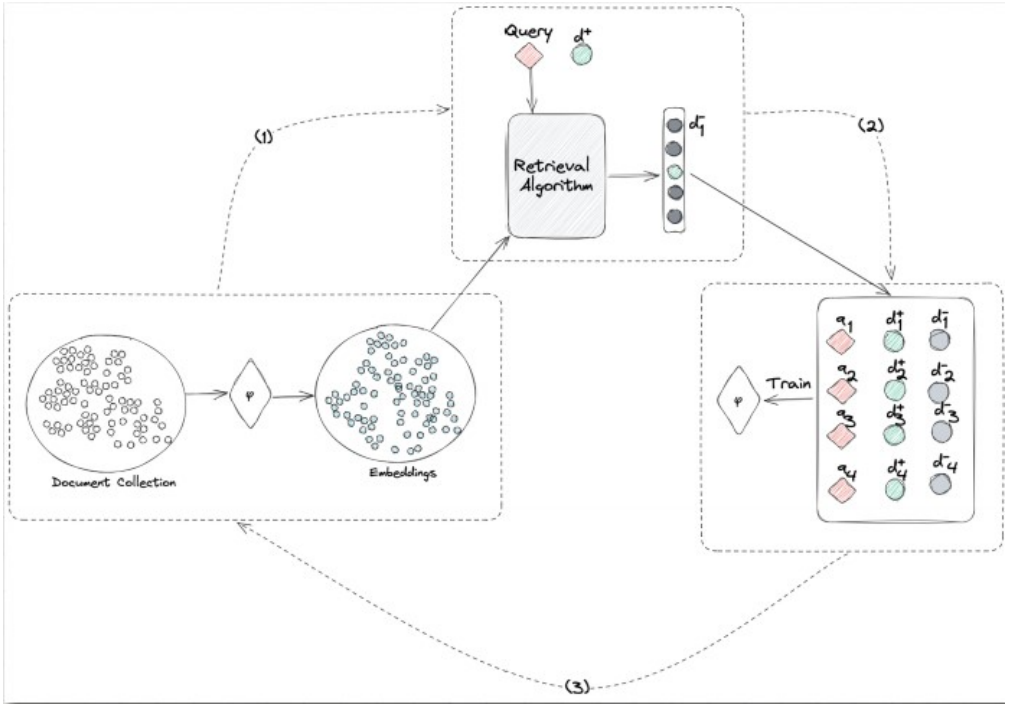


Figure 84: ANCE method for negative sampling

The **advantage** of this method is that it is quite accurate, but on the other hand it is very expensive, since it asks every time to compute $\phi(d)$, for each $d \in D$, so it suffers of a very slow tuning phase.
Once we discussed the main methods for representing queries and documents, we can now discuss the possible solution for solving the MIPS problem:

$$\varsigma = \max_{d \in D}^{(k)} \langle \phi(q), \phi(d) \rangle$$

7.3 MIPS algorithms for sparse vectors

We now discuss some algorithms for solving the MIPS problem when both the documents and the queries are represented as **sparse vectors**.
The first issue we have to solve is how to represent sparse vectors: one naive representation could be a $|D| \times m$ matrix, as represented in Picture 7.3.

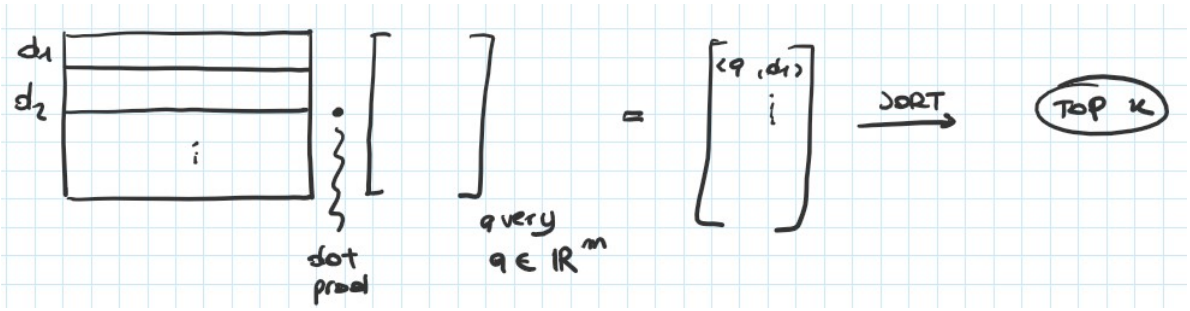


Figure 85: Matrix for sparse vectors

However, this representation is quite expensive, both in space and time (the sort operation takes $O(|D| \log k)$ time).
In this sense, we can exploit the sparse nature of the vectors by removing all the 0s and represent documents and queries as an **inverted index**. An inverted index is composed of:

- A **term identifier**;
- An **inverted list**, that consists of pairs (docID, score), where the score represents the impact score of the term for that document, for each docID that contains the term. Formally, each entry of the inverted list is $L[i] = \{(j, d_j[i] | d_j[i] \neq 0)\}$

An example of inverted index is provided in Picture 7.3.

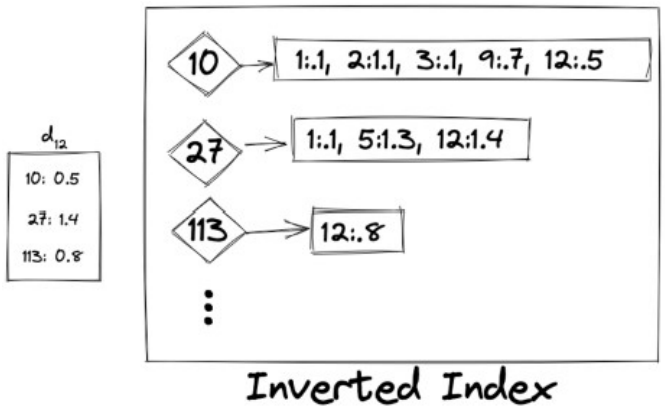


Figure 86: Example of inverted index

Analogously, a query is then represented as a list of terms.
We now discuss two possible methods for query processing, i.e. for retrieving the top-k sparse documents w.r.t. a sparse query.

7.3.1 TAAT query processing

The first method we discuss is defined as **TAAT** query processing, and the idea is that for each term of the query, we build a vector $v \in \mathbb{R}^{|D|}$, s.t. $v[i]$ contains the product between the impact score of the query term and the impact score of document d_i. Clearly, if a document does not contain a query term, the corresponding impact score is 0. A visual representation of this method is provided in Picture 7.3.1.

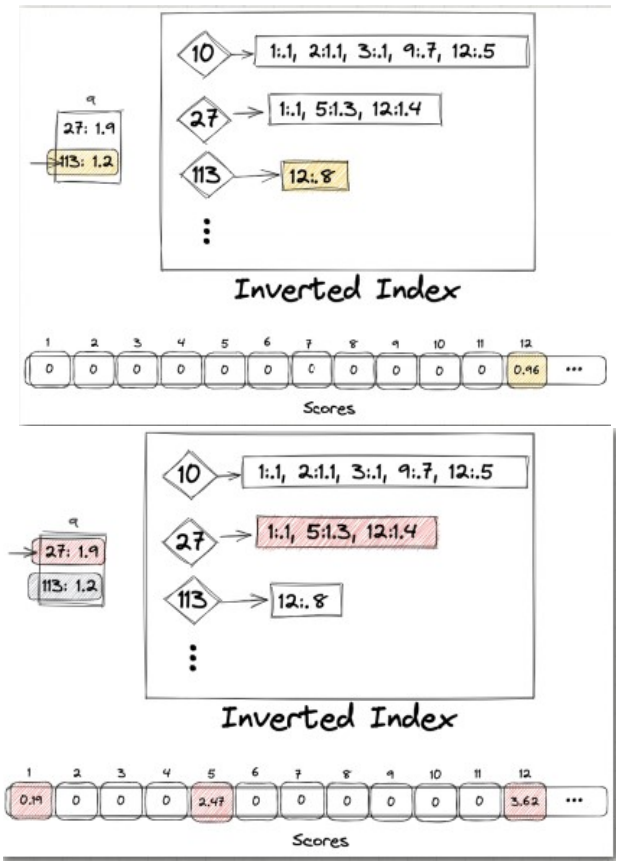


Figure 87: Example of TAAT query processing

More formally, the algorithm takes as input an inverted index L and a query vector q, and it is described in 1.

7.3.2 DAAT query processing

Here the algorithm receives as input an inverted index L where the inverted lists are sorted by docID, and a query vector q, and it is described in 2.

7.3.3 Dynamic pruning algorithms

Since users are mostly interested in the top few pages of results for a query, the complete scoring of every document that contains at least one query term results in high latency. However, not all of these documents will make the top-K retrieved set of documents that the user will see. Two main general approaches have been exploited to **increase the efficiency** of **query processing** in IR systems in the top-K ranked retrieval scenario:

Data: L, q
Result: Top-k document vectors with scores
$Scores[1:|D|] \leftarrow 0$;
for i *s.t.* $q[i] \neq 0$ **do**
  for $(j, d_j[i]) \in L[i]$ **do**
    $Scores[j] \leftarrow Scores[j] + q[i]d_j[i]$
  end
end
Sort $\{1, 2, .., |D|\}$ by $Scores[]$ in descending order;
Return top-k identifiers with corresponding scores;

Algorithm 1: TAAT query processing

Data: L, q
Result: Top-k document vectors with scores
$Q \leftarrow$ PriorityQueue;
$p[i] \leftarrow 0 \quad \forall_{i:q[i]\neq 0}$;
$j \leftarrow \min\{j|(j, .) \in L[i], \forall i\}$;
while $p[i] < |L[i]| \quad \forall i$ **do**
  $Score \leftarrow 0$;
  next $\leftarrow \infty$;
  for i *s.t.* $q[i] \neq 0$ **do**
    if $j = L[i][p[i]].id$ **then**
      $Score \leftarrow Score + q[i] \times L[i][p[i]]$.weight;
      $p[i] \leftarrow p[i] + 1$
    end
    if $L[i][p[i]] < next$ **then**
      next $\leftarrow L[i][p[i]]$.docID
    end
  end
  Q.Push($(j, Score)$);
  if $|Q| > k$ **then**
    Q.Pop()
  end
  $j \leftarrow$ next;
end
Return Q;

Algorithm 2: DAAT query processing

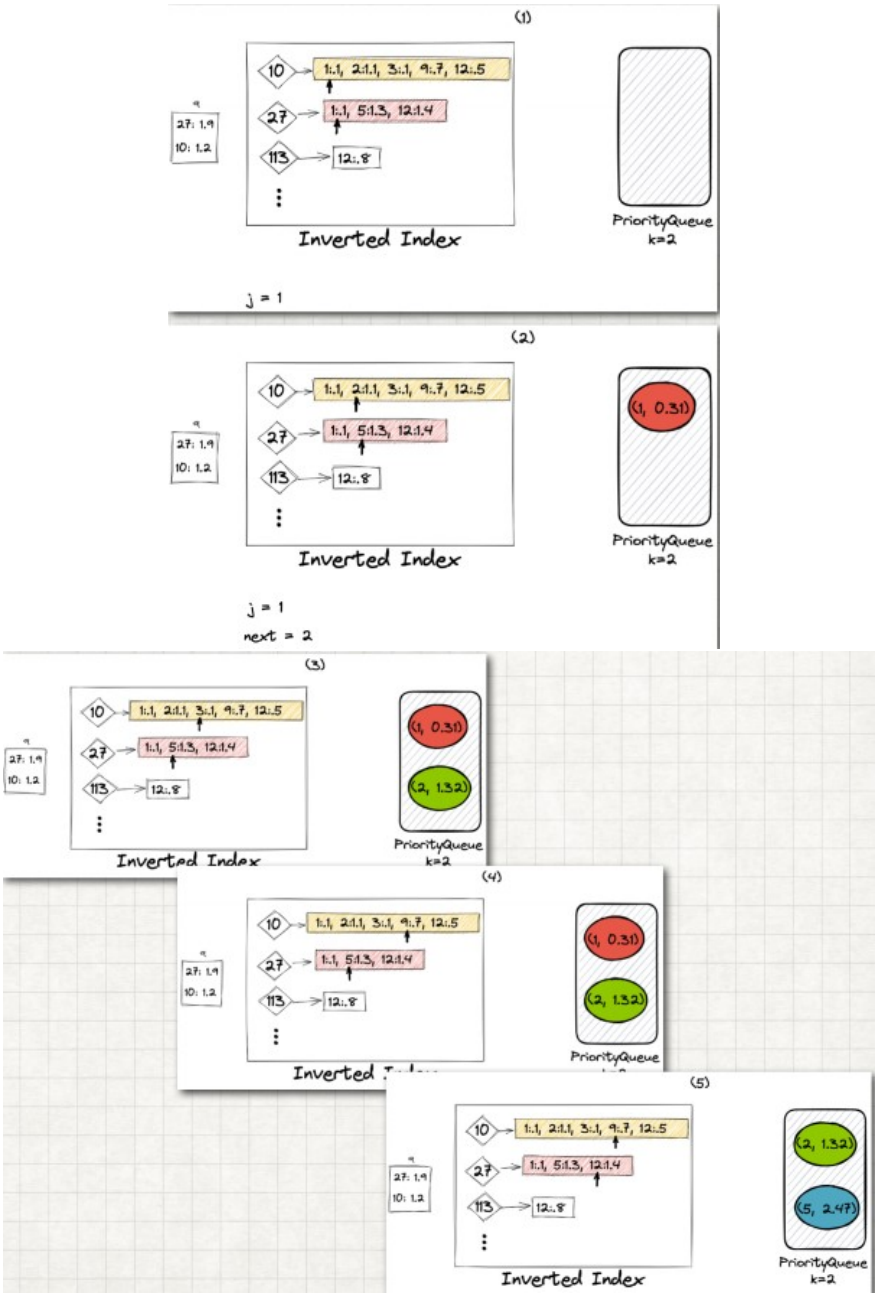


Figure 88: Example of DAAT query processing

1. Avoid wasting time in processing portions of the inverted index containing documents that are unlikely to be relevant;
2. Improve the efficiency of algorithms when processing portions of the inverted index containing relevant documents

One of the main design solutions for dealing with these two approaches can be implemented at the query processing level, i.e., by modifying the behaviour of the retrieval algorithms to try to prune documents that will not be retrieved in the top-K . In this section, we summarise the growing literature on dynamic pruning optimisation techniques that improve the query processing efficiency for both the TAAT and DAAT retrieval

strategies.
While static pruning strategies alter the index structure at index construction time, **dynamic pruning** aims to **alter query processing** in such a way that potentially **non-relevant documents**, for a given query, **can be efficiently ignored**. Although different dynamic pruning strategies have been proposed for TAAT and DAAT strategies through the years, all of these optimisations rely on a common observation, which is that as soon as it can be determined that a document will never be able to enter in the final top-K results, we can ignore it during processing, or stop its current processing. This observation, sometimes referred to as the **early termination condition**, can be stated more clearly by introducing the following definitions:

- *early termination*: during the processing of a query, a document evaluation is early terminated if all or some of its postings, as defined by the terms of the query, are not fetched from the inverted index or not scored by the ranking function;
- *term upper bound*: for each term t in the vocabulary, we compute a term upper bound (also known as *max score*) $\sigma_t(q)$ such that, for all documents d in the posting list of term t:

  $$\sigma_t(q) \geq s_t(q, d)$$

  , where $s_t(q, d)$ is the score. The term upper bounds can be computed offline by taking the maximum score value of the highest scoring document for each term in the vocabulary and storing the observed score in the vocabulary. Alternatively, for some similarity measures, term upper bounds can be quickly estimated at runtime;
- *document upper bound*: given a similarity function, for a query q and a document d, we can compute a document upper bound $\sigma_d(q)$ based on the terms occurring in the document, by summing up the n term upper bounds:

  $$\sigma_d(q) = \sum_{t \in q} \sigma_t(q)$$

- *thresholds*: during query processing, the top-K full or partial scores computed so far, together with the corresponding docIDs, are organised in a separate data structure, and the smallest value of these (partial) scores is called threshold θ. If there are not at least K scores, the threshold value is assumed to be 0. This data structure can be implemented as a priority queue queue with capacity K (also known as max-heap), supporting the operations of *push* (adding the (docID, score) pair to the queue if the score is greater than the current threshold), *min* (returning the value of the current threshold θ) and *pop* (removing the top scoring (docID, score) pair from the queue). Notice that the threshold has the fundamental property of **non-negative monotonicity**, i.e. during query processing its value always increases.

At this point, we can formulate the **pruning condition**: for a query q and a document d, if the **document upper bound** $\sigma_d(q)$, computed by using partial scores, if any, and term upper bounds, is **less than or equal to the current threshold** θ, the **document processing can be early terminated**, i.e., if the condition

$$\sigma_d(q) \leq \theta$$

evaluates to false.
In essence, all dynamic pruning strategies aim to process queries conjunctively when possible, and disjunctively otherwise. Dynamic pruning strategies work well when queries are composed of highly discriminative (i.e., rare) terms as well as poorly discriminative (i.e., common) terms.
In general, this pruning condition and how it is used can have further consequences on the particular query processing strategy adopted, but we can classify the various optimizations obtained from this pruning condition into 4 classes:

- **safe / un-optimized**: in this case, all the documents, not just the top-k, are ranked correctly;
- **safe up to** k **/ rank safe**: in this case the top-k documents are ranked correctly, but the document scores are not guaranteed to coincide with the scores produced by an un-optimized strategy. These are the most interesting dynamic pruning optimizations, since they do not negatively impact the effectiveness of the documents returned to the users while introducing efficiency gains. Indeed, relevance evaluation metrics are typically computed over the top K = 10, 20 documents, such as *MAP@10* or *NDCG@20*;
- **unordered safe up to** k **/ set safe**: in this case the documents returned by this optimization coincide with the top-k documents computed by a full strategy, but their ranking could be different;
- **approximate / unsafe**

Dynamic pruning for TAAT
In this case a very effective technique is the **early termination**. An example is provided in Picture 7.3.3.

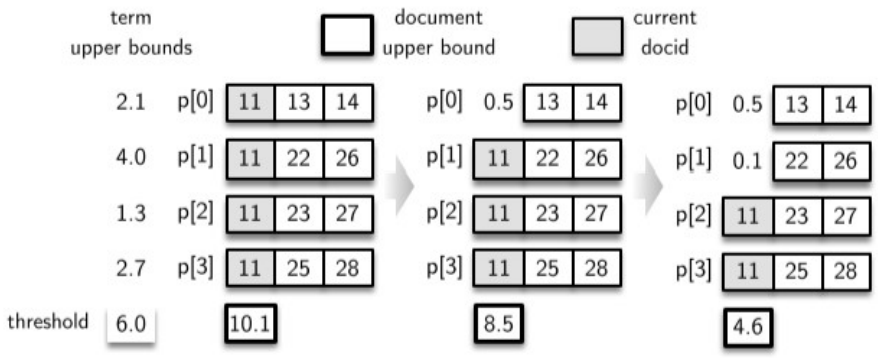


Figure 89: Example of early termination

In the example, the current threshold value θ is 6.0, resulting from documents in the current top results. While scoring docID 11, the document upper bound is initially set to 10.1, the sum of the 4 term upper bounds. While postings are processed, the document upper bound is adjusted with the actual scores computed for each term. Hence, it

decreases to 8.5 after the posting from the first posting list is processed. Since $8.5 > 6.0$, we must continue to process the other posting lists. After the posting from the second list is processed, the document upper bound becomes 4.6. At this point we are sure that document 11 will never obtain a final score greater than 4.6 and since it is less than the current threshold, the postings of the last two lists can be ignored, and the processing of docID 11 can be terminated, proceeding to the next docID.

Dynamic pruning for DAAT

We now focus on some dynamic pruning algorithms for the DAAT strategy, **MaxScore** and **WAND**, which both maintain a term upper bound in the vocabulary for each posting list, which is used at runtime to take decisions on the early termination and/or skipping of certain postings/documents.

The **MaxScore** algorithm is a **safe up to k strategy** aiming to boost the efficiency of the DAAT algorithm through the following observation. At some point during query processing, we can expect that the threshold will be large enough to prune documents appearing only in the posting list with the smallest term upper bound contribution. When this happens, the algorithm can **safely skip over documents** appearing only in that **posting list**, and can consider as top-K candidate documents only those appearing in the remaining posting lists. Once a new candidate document must be fully scored, that posting list can be traversed in an AND mode to look for the candidate docID only. This observation can be applied to the remaining posting lists as the query processing proceeds and the threshold increases.

A possible implementation is reported in Picture 7.3.3.

The algorithm takes as input two arrays of size n: the posting lists p to be processed and the corresponding term upper bounds σ. Both arrays are sorted in increasing order of max score. At runtime, the posting lists are kept separated into two sub-lists by a pivot index, running from 0 to $n-1$. The posting lists indexed from the pivot up to n form the *essential lists*, while the remaining posting lists, if any, are the *non-essential lists*. At any time during query processing, no document can be returned as a top-K result if it only appears in the non-essential lists, i.e., at least one of the terms corresponding to the essential lists must occur in any top-K document.

To update the pivot, we compute n document upper bounds ub (line 5). The entry $ub[0]$ contains the document upper bound for documents appearing just in $p[0]$, the entry $ub[1]$ contains the upper bound for documents appearing only in $p[0]$ and $p[1]$, and so on. While there is at least an essential list and there are documents to process (line 9), the MaxScore algorithm first processes the essential lists selecting the candidate docID as in DAAT, while storing the next docID to process (lines 12–17). Then, it proceeds by processing the non-essential lists by skipping to the candidate docID (lines 18-23). As soon as the pruning condition holds (line 19), we are sure that the candidate docID cannot be in the final top-k documents, and the remaining posting lists can be skipped completely. If all the non-essential posting lists are processed, we check if the final score is high enough to enter.

Picture 7.3.3 shows the MaxScore pruning with 4 posting lists.

In the example, the current threshold value θ is 6.0, resulting from documents in the current top results. The posting lists are sorted by increasing term upper bound, and we have used the term upper bounds to compute the values of the array ub. Since $ub[1] = 3.4$ and $ub[2] = 7.1$, the pivot is set to 2, $p[0]$ and $p[1]$ are the *non-essential lists*, while $p[2]$

Algorithm 3.1: The MaxScore algorithm

```
Input  : An array p of n posting lists, one per query term,
         sorted in increasing order of max score contribution
         An array σ of n max score contributions, one per query term,
         sorted in increasing order
Output : A priority queue q of (at most) the top K ⟨docid, score⟩ pairs,
         in decreasing order of score
MaxScore(p,σ):
1   q ← a priority queue of (at most) K ⟨docid, score⟩ pairs,
        sorted in decreasing order of score
2   ub ← an array of n document upper bounds, one per posting list,
        all entries initialised to 0
3   ub[0] ← σ[0]
4   for i ← 1 to n − 1 do
5       ub[i] ← ub[i − 1] + σ[i]
6   θ ← 0
7   pivot ← 0
8   current ← MinimumDocid(p)
9   while pivot < n and current ≠ ⊥ do
10      score ← 0
11      next ← +∞
12      for i ← pivot to n − 1 do                          // Essential lists
13          if p[i].docid() = current then
14              score ← score + p[i].score()
15              p[i].next()
16          if p[i].docid() < next then
17              next ← p[i].docid()
18      for i ← pivot − 1 to 0 do                          // Non-essential lists
19          if score + ub[i] ≤ θ then
20              break
21          p[i].next(current)
22          if p[i].docid() = current then
23              score ← score + p[i].score()
24      if q.push(⟨current, score⟩) then                   // List pivot update
25          θ ← q.min()
26          while pivot < n and ub[pivot] ≤ θ do
27              pivot ← pivot + 1
28      current ← next
29  return q
```

Figure 90: Implementation of MaxScore algorithm

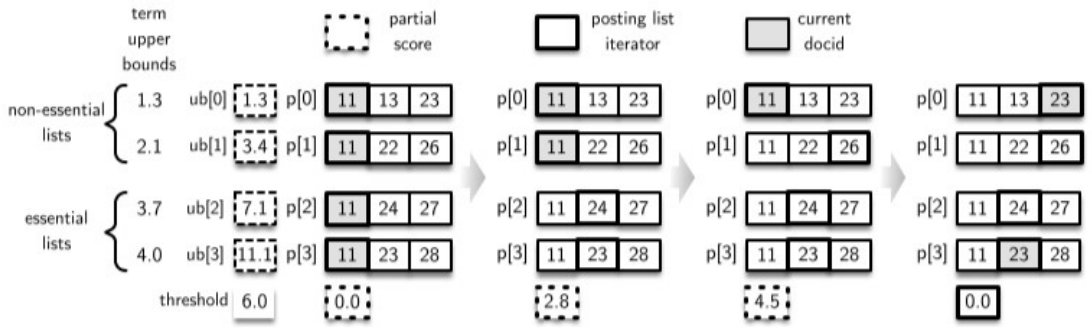


Figure 91: Example of MaxScore algorithm

and $p[3]$ are the *essential lists*. We are processing the document with docID 11. Firstly, we process the *essential lists* and compute the partial score of the document, keeping track of the next docID to process while advancing the posting list iterator of these lists by one step. Since, given the partial score computed so far, it is possible that docID 11 could exceed the current threshold (e.g., $2.8 + ub[1] = 2.8 + 3.4 = 6.2 > 6.0 = \theta$), we start processing the *non-essential lists* for that document. After we process $p[1]$, skipping

directly to the next docID greater than or equal to the next docID to process (e.g., 23), we get an updated partial score of 4.5. Now, since $ub[0] = 1.3$, we can safely ignore the first posting list (e.g., $4.5 + 1.3 = 5.8 < 6.0$), and skip directly to docID 23 using the $p.next(d)$ operator. Note that in the example we do not include the priority queue and pivot updates for simplicity.
The other algorithm is **WAND**. This algorithm is based on the *WAND*, or *Weak AND* operator, which takes as input a list of n boolean variables $X_0, ..., X_{n1}$, a list of n associated weights $w_0, ..., w_{n1}$ and a threshold θ. By definition, $WAND\ (X_0, w_0, ..., X_{n1}, w_{n1}, \theta)$ is true if and only if:

$$\sum_{i=0}^{n-1} w_i x_i \geq \theta$$

, where x_i is equal to 1 if X_i is true, and 0 otherwise. Note that, with unary weights and a threshold equal to n or 1, the *WAND* operator implements the boolean *AND* or *OR*, respectively.
Given a query $q = \{t_0, ..., t_{n1}\}$ and a document d, we can apply the *WAND* operator in the following way. We assume that X_i is true if and only if the term t_i appears in document d, and we take the term upper bound $\sigma_{t_i}(d)$ as weight w_i. The threshold θ has the usual meaning, i.e., the smallest score among the top K documents scored thus far during query processing. Hence the condition $WAND\ (X_0, \sigma_{t_0}(d), ..., X_{n1}, \sigma_{t_{n1}}(d), \theta)$ evaluates to true if and only if:

$$\sum_{i=0}^{n-1} \sigma_{t_i}(d) \geq 0$$

Assuming that all terms t appear in document d, this inequality corresponds to the pruning condition $\sigma_d(q) \leq \theta$.
This algorithm exploits the *WAND* operator to prune the documents whose *WAND* evaluation is false, and then perform a full evaluation to compute the actual scores. Picture 7.3.3 represents a possible implementation of the algorithm.

- INPUT: two arrays of size n: the posting lists p to be processed and the corresponding term upper bounds σ. Note that they are not required to be sorted since they will be kept sorted though increasing the docIDs by the algorithm itself (lines 3 and 25). Indeed, the $SortByDocid(p, \sigma)$ procedure guarantees that the array of posting lists are sorted by increasing docID and that the term upper bound $\sigma[i]$ always corresponds to the posting list $p[i]$;
- The core idea of the algorithm is to evaluate the *WAND* operator (i.e., the pruning condition) one posting list at a time, accumulating the score of the candidate document in σ_d (line 10);
- As soon as this value exceeds the current threshold (line 11), we have potentially identified a pivot docID (pivot_id) that could enter the top K documents. The pivot docID undergoes a full evaluation (lines 17–22) and might be included in the current top K results (lines 23–24) only if it is present in all posting lists up to, and including, the list containing the pivot docID. Since the posting lists are sorted by

Algorithm 3.2: The WAND algorithm

```
Input  : An array p of n posting lists, one per query term
         An array σ of n max score contributions, one per query term
Output : A priority queue q of (at most) the top K ⟨docid, score⟩ pairs,
         in decreasing order of score
WAND(p,σ):
1   q ← a priority queue of (at most) K ⟨docid, score⟩ pairs,
        sorted in decreasing order of score
2   θ ← 0
3   SORTBYDOCID(p,σ)
4   while true do
5       σ_d ← 0
6       pivot ← 0
7       for pivot ← 0 to n − 1 do                          // Find list pivot
8           if p[pivot].docid() = ⊥ then
9               break
10          σ_d ← σ_d + σ[pivot]
11          if σ_d > θ then
12              break
13      if σ_d ≤ θ then                                    // No list pivot found
14          break
15      pivot_id ← p[pivot].docid()
16      if pivot_id = p[0].docid() then                    // If matching doc pivot
17          score ← 0
18          for i ← 0 to n − 1 do
19              if p[i].docid() ≠ pivot_id then
20                  break
21              score ← score + p[i].score()
22              p[i].next()
23          q.push(⟨pivot_id, score⟩)
24          θ ← q.min()
25          SORTBYDOCID(p,σ)
26      else                                               // Else move list up to the pivot
27          while p[pivot].docid() = pivot_id do
28              pivot ← pivot − 1
29          p[pivot].next(pivot_id)
30          SWAPDOWN(p,σ,pivot)
31  return q
```

Figure 92: WAND algorithm

docID, it is sufficient to test the pivot docID with the current posting's docID of the first list p[0] (line 16);

- Otherwise, since the posting lists are sorted by docID, we can safely affirm that the pivot docID is the smallest docID among all posting lists from 0 to pivot that could enter the top K results. The docID smaller than pivot_id will never be able to accumulate enough term upper bounds to have a chance to beat the current threshold;
- Unfortunately, we cannot be sure that pivot_id will be a candidate, since we do not know yet in which posting lists it appears. Thus, we "backtrack" to the first posting list whose iterator is not on the pivot docID (lines 27–28), and we move its iterator to the pivot_id (or further) (line 29) using the p.next(d) operator. The

SwapDown$(p, \sigma, pivot)$ procedure on line 30 restores the docID-sorting of the posting lists by moving the pivot posting list and the associated term upper bound "down" to the correct position. Using the p.next(d) operator means that skipping occurs, and hence the reading and decompression of skipped postings can be avoided;

- To conclude, note that if no pivot docID can be found (line 13), we are sure that no new document can beat the current threshold, and hence the algorithm can safely terminate.

Picture 7.3.3 illustrates a few *WAND* iterations with 4 posting lists. In the example, the current threshold value θ is 6.0, resulting from documents in the current top results.

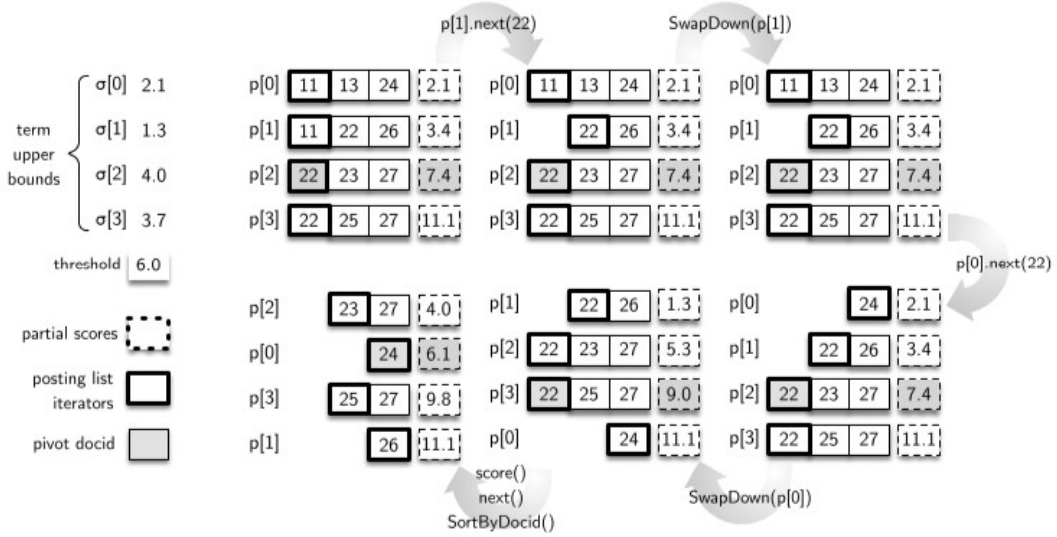


Figure 93: Example of WAND algorithm

The posting list iterators are sorted by current docID. The next pivot id is 22 since neither 2.1 nor $2.1 + 1.3 = 3.4$ are greater than the current threshold, while $2.1 + 1.3 + 4.0 = 7.4$ does exceed the threshold. Since the $p[0]$ iterator is not 22, the only knowledge we have so far is that no docID smaller than 22 will have a score greater than the current threshold. Hence, we select $p[1]$ and we advance its iterator to 22, with no reordering of the posting lists since they are already sorted by docID. At the next iteration, our pivot docID is again 22, but we cannot fully score it since we do not know if $p[0]$ will actually contribute to the approximate score of the pivot docID, i.e., we do not know if $p[0]$ contains docID 22. We try to advance the $p[0]$ iterator to 22, but it skips to docID 24, forcing the move of $p[0]$ to the end of the array of iterators. At the next iteration, we are again considering docID 22: its approximate score now is $1.3 + 4.0 + 3.7 = 9.0$, enough for a full processing, and since all posting lists up to the pivot docID include it, docID 22 undergoes a full evaluation, and a potential threshold update. During the full evaluation, the iterators of the involved posting lists are advanced to the next docID (line 22), hence a full reordering of the four lists is mandatory, to correctly select the next pivot docID (that, with the current threshold $\theta = 6.0$, will be 24, since $\sigma[2] + \sigma[0] = 4.0 + 2.1 = 6.1 > \theta$).
Note that the *WAND* optimisation is **safe up to** K, since the pruning condition is used to select the candidate docIDs. Nevertheless, aggressive (i.e., approximate) versions of

WAND have been proposed forcing the new candidate documents to beat the current threshold by a larger quantity.

7.4 MIPS algorithms for dense vectors

We now focus on the solving methods for the MIPS problem when the documents and the queries are **dense vectors**, i.e., vectors characterized by very few zero entries.
First of all, we try to reason about some of the techniques we used for sparse vectors, for example inverted indexes: are the suitable for dense vectors? The answer is no, since we recall that the objective of the inverted index data structure is to skip the dot product evaluation in presence of 0 entries. In this sense, the resulting data structure would be a very long list, which is inefficient. Moreover, we also cannot exploit the dynamic pruning algorithm, since the inverted lists are full of elements.
For these reasons, we have to consider new methods for speeding up the computation of the solution of MIPS problem in presence of dense vectors.

7.4.1 Quantization methods

In general, some documents of a collection can be very similar to each other (this is true from the nature of data), so the idea of **quantization methods** is to reduce the complexity of the problem by quantizing very similar documents, which are represented as points in a high-dimensional space, and then solve the MIPS problem considering only the representatives for each quantized region. From a mathematical point of view, we can define a codebook $C = \{c_1, c_2, .., c_m\}$, with $c_i \in \mathbb{R}^n$, and a map $\pi : D \to [m]$. Then, the approximate solution of the MIPS problem is given by all the documents in c_k, where:

$$c_k = \arg\max_{c_{\pi(d)}} \langle c_{\pi(d)}, q \rangle$$

, i.e. the documents contained in the quantized region which has a maximum inner product with the query. A visual representation of this method is provided in Picture 7.4.1.

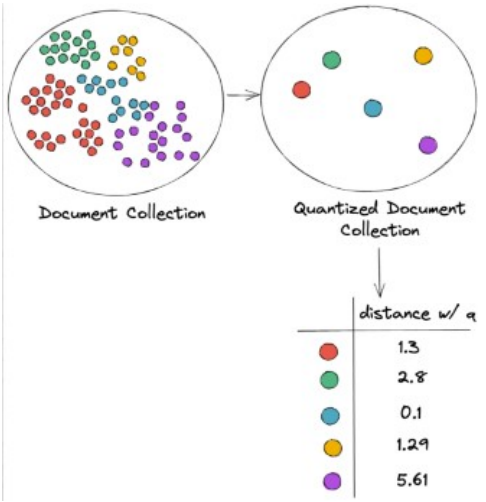


Figure 94: Example of quantization

Finally, we can compare the approximate solution with the exact one by computing the *recall* between the two sets of top-k.
An important **issue** about this methods relies in the so-called **curse of dimensionality**, a phenomenon for which, when the dimensionality of the vectors goes to ∞, the concepts of similarity and distance do not hold anymore.
For this reason, another idea that exploits quantization is called **product quantization**, and it works as follows: since the dot product is linear, we can rewrite is as:

$$\langle q, d \rangle = \sum_{i=1}^{n} q_i d_i = \sum_{i=1}^{n} \sum_{j=1}^{n} q_{ij} d_{ij} = \langle q_1, d_1 \rangle + \langle q_2, d_2 \rangle + .. + \langle q_1, d_1 \rangle = \sum_{i=1}^{M} \langle q^{(i)}, d^{(i)} \rangle$$

, where $d^{(i)} \in \mathbb{R}^{n/M}$. In this sense, the idea is to break up both the vectors q and d into M subvectors, each of which containing a quantized region of points called C_i. Then, for each i, we can compute the distance between $q^{(i)}$ and each of the points in $d^{(i)}$, in order to select the **representative** for each region, i.e. the point with minimum distance. Finally, we compute the dot product between $q^{(i)}$ and each representative C_i and we sum up the scores, in order to get an approximate solution for the dot product between q and d. A visual representation of this method is provided in Picture 7.4.1.

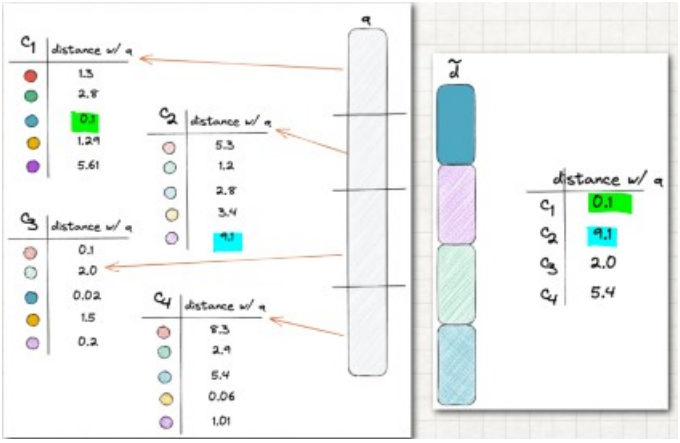


Figure 95: Example of product quantization

One **advantage** of this new method is that it is **efficient** in time, since the only operations that are involved are the **lookup** in the tables containing the distance between each segment of the query and all the points in a segment of the doc and the **summation** of the partial dot product scores. Moreover, this algorithm obtains **perfect results** if each segment contains a number of centroids which is equal to the number of points. On the other hand, this method is **useless** if we have one cluster per segment, and it is still heavy from a space point of view.
For this reason, one last quantization method can be used, called **IVFPQ**, which works as follows: we cluster the collection of documents D and we create an inverted index that maps cluster centers to list of points in that partition. Then, within each cluster we can apply *product quantization* to the residuals $d - u$, where u is the center of the cluster. A visual representation of this method is provided in Picture 7.4.1.

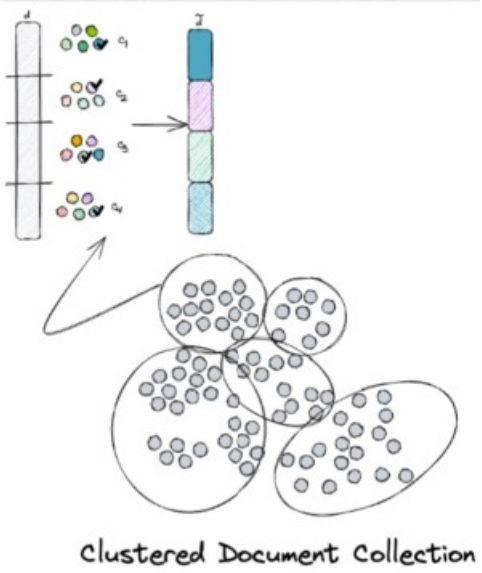


Figure 96: Example of IVFPQ

Finally, Picture 7.4.1 provides a comparison between the complexity, both in time and space, of quantization and product quantization methods.

	Codebook Size	Index Size	Query Time
Quantization	$\mathcal{O}(m)$	$\mathcal{O}(mn + \lvert\mathcal{D}\rvert \log m)$	$\mathcal{O}(mn)$ FLOPs $\mathcal{O}(\lvert\mathcal{D}\rvert)$ lookups
PQ	$\mathcal{O}(m^M)$	$\mathcal{O}(Mmn + \lvert\mathcal{D}\rvert M \log m)$	$\mathcal{O}(mn)$ FLOPs $\mathcal{O}(M\lvert\mathcal{D}\rvert)$ lookups

Figure 97: Comparison of quantization and product quantization

7.4.2 Clustering methods

Another important method is **clustering**, which works as follows:

1. Cluster the documents into m clusters with centroids $C = \{c_1, c_2, .., c_m\}$, with $c_i \in \mathbb{R}^n$
2. Given a query q:
   (a) Compute the distance between the query and each of the centroid: $\langle c_i, q \rangle$
   (b) Solve MIPS over the "closest" clusters.

A visual representation is provided in Picture 7.4.2.
Notice that this method is quite useful and effective when we can exploit distributed that are able to solve MIPS.

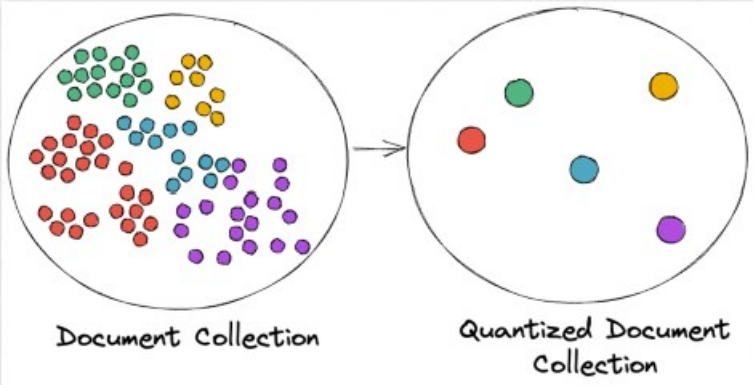


Figure 98: Clustering method

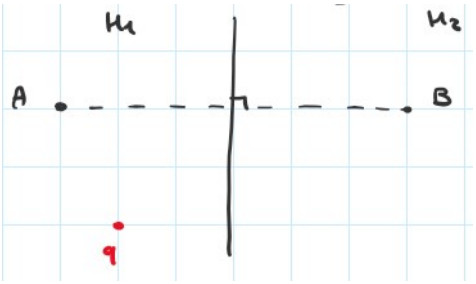
7.4.3 Graph methods

We now discuss the **graph methods**.
Given two points A and B, and a query point q, we can draw the line that is orthogonal to the line connecting A and B, and divide the space into two regions:

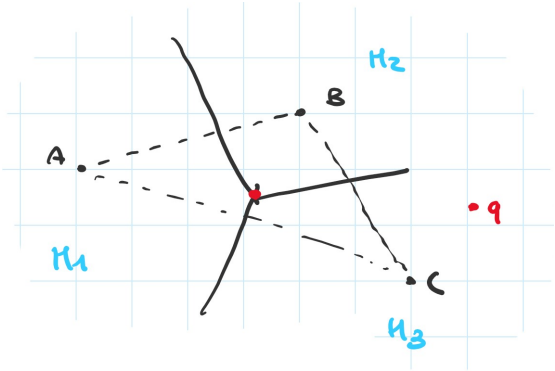
$$H_1 = \{p | d(p, A) < d(p, B)\}$$

$$H_2 = \{p | d(p, A) \geq d(p, B)\}$$

, where $d(x, y) = ||x - y||_2$.



If we have 3 points, the situation is the following one:



Each of the H_i is called **Voronoi region**, and they are formally defined as convex regions formed by the intersection of hyperplanes. A **Voronoi diagram** divides the space into D Voronoi regions s.t. each point d_i lies in exactly one Voronoi region.

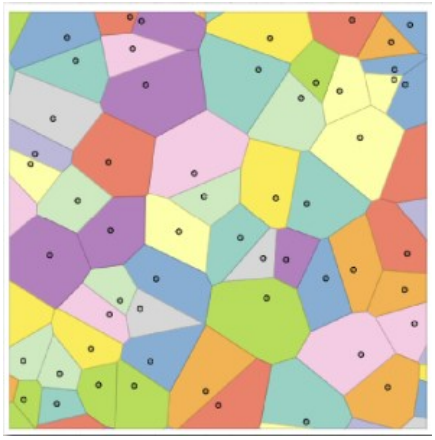


Figure 99: Voronoi diagram

Now we can introduce the **Delaunay graph**, which is a graph $G = (V, E)$ where $V = D$, i.e. the vertices are the documents, and $e_{ij} = 1 \iff H_i \cap H_j \neq \emptyset$.

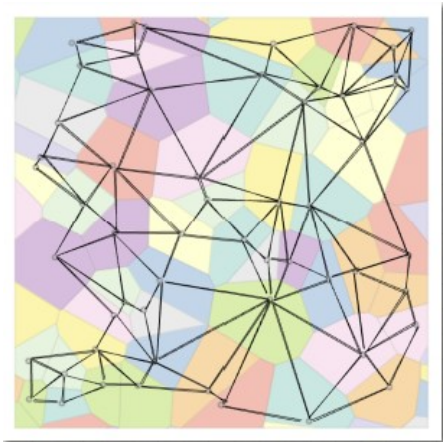


Figure 100: Delaunay graph

How can we exploit this graph to find the document point which is closest to a query point q? We can prove that a **greedy traversal** oh the Delaunay graph guarantees the discovery of the **correct nearest neighbor** to a query point. However, in **high dimensions** the graph is almost **fully connected**, since the average degree in the Euclidean space grows exponentially in dimensions.
For this reason, we can approximate the Delaunay graph by considering a subgraph of the original one in which $e_{ij} = 1 \iff \forall d_k, d(d_i, d_j) \leq \max\{d(d_i, d_k), d(d_j, d_k)\}$.

Figure 101: Approximation of the Delaunay graph