

# Performance spectrum

---

>TO DO:

- Finish the problems from exams from previous years
- Look for similar problems on the internet

## Contents

1. Unit prefixes .....	3
2. Data size measurement units: .....	3
2.1 Data size measurements with decimal prefixes: .....	3
2.2 Data size measurements with binary prefixes: .....	3
3. The 8 fallacies of distributed computing.....	5
3.1 Latency and Bandwidth: .....	5
4. Performance limits .....	6
5. Cache .....	7
6. Memory mapped IO (MMIO) .....	8
7. Should we use a Cluster?.....	9
8. Data center architecture .....	11
9. Storage hierarchy .....	12
10. Scale up and Scale out.....	13
11. Modeling communication overhead computation .....	14
11.1 Random access on a single machines example:.....	16
11.1.1. Solving the problem using random access: .....	16
11.1.2. Solve the problem by reading all the records: .....	17
11.2 Top-k algorithm, as example of disadvantages of using random access .....	17
11.2.1 First approach to solving top-k algorithm .....	19
11.2.2 Second approach to solving top-k algorithm .....	21
12. Exam examples of previous years .....	24
12.1 Tasks from 2019/2020 final exam, section performance spectrum .....	24
12.1.1 Task 1.....	24
12.1.2 Task 2.....	24
12.2 Example of 2019/2020 repeat exam (which was online), section performance spectrum .....	25
12.2.1 Task 1, a):.....	25

12.2.2 Task 1, b):..... 27

## 1. Unit prefixes

name	symbol	10 <sup>n</sup>	number size
tera	T	10 <sup>12</sup>	1 000 000 000 000
giga	G	10 <sup>9</sup>	1 000 000 000
mega	M	10 <sup>6</sup>	1 000 000
kilo	k	10 <sup>3</sup>	1 000
mili	m	10 <sup>-3</sup>	0.0001
micro	μ	10 <sup>-6</sup>	0.0000001
nano	μ	10 <sup>-9</sup>	0.0000000001
pico	p	10 <sup>-12</sup>	0.0000000000001

## 2. Data size measurement units:

### 2.1 Data size measurements with decimal prefixes:

unit	symbol	Equivalence	in bytes
byte	B	8 bits	1
kilobyte	kB or KB	10 <sup>3</sup> = 1000 bytes	10 <sup>3</sup>
megabyte	MB	10 <sup>3</sup> = 1000 kilobyte	10 <sup>6</sup>
gigabyte	GB	10 <sup>3</sup> = 1000 megabyte	10 <sup>9</sup>
terabyte	TB	10 <sup>3</sup> = 1000 gigabyte	10 <sup>12</sup>
petabyte	PB	10 <sup>3</sup> = 1000 terabyte	10 <sup>15</sup>
exabyte	EB	10 <sup>3</sup> = 1000 petabyte	10 <sup>18</sup>
zettabyte	ZB	10 <sup>3</sup> = 1000 exabyte	10 <sup>21</sup>
yottabyte	YB	10 <sup>3</sup> = 1000 zettabyte	10 <sup>24</sup>

### 2.2 Data size measurements with binary prefixes:

Unit	symbol	Equivalence	in bytes
byte	B	8 bits	1
kibibyte	KiB	2 <sup>10</sup> = 1024 bytes	2 <sup>10</sup>
mebibyte	MiB	2 <sup>10</sup> = 1024 kilobyte	2 <sup>20</sup>
gibibyte	GiB	2 <sup>10</sup> = 1024 megabyte	2 <sup>30</sup>
tebibyte	TiB	2 <sup>10</sup> = 1024 gigabyte	2 <sup>40</sup>
pebibyte	PiB	2 <sup>10</sup> = 1024 terabyte	2 <sup>50</sup>
exbibyte	EiB	2 <sup>10</sup> = 1024 petabyte	2 <sup>60</sup>
zebibyte	ZiB	2 <sup>10</sup> = 1024 exabyte	2 <sup>70</sup>
yobibyte	YiB	2 <sup>10</sup> = 1024 zettabyte	2 <sup>80</sup>

Notice in the names how „binary“ in short „bi“ is added. For an example, instead of calling something „kilobyte“ (like we would when using decimal prefixes), in case we use binary prefixes, we call it a „kibibyte“.

**NOTE: ON THE EXAM IF THEY WRITE „kB“ (OR „KB“), DO NOT AT ANY CHANCE CONNSIDER IT AS  $10^3$  BYTES, BUT RATHER CONSIDER IT AS  $2^{10}$  BYTES, BECAUSE YOU WILL GET YOUR POINTS DEDUCTED FOR WRITING THIS (ALTHOUGH ACCORDING TO THE TAVLES ABOVE THIS SHOULD BE CONSIDERED AS CORRECT).**

### 3. The 8 fallacies of distributed computing

Eight assumptions architects and designers of distributed systems are likely to make, which prove wrong in the long run - resulting in all sorts of troubles and pains for the solution and architects who made the assumptions. The assumptions are now collectively known as the **"The 8 fallacies of distributed computing"**:

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

#### 3.1 Latency and Bandwidth:

Two of the most important terms for this course (FDE), latency and bandwidth are going to be explained.

**Latency** – how much time it takes for data to move from one place to another

**Bandwidth** – how much data we can transfer (over an internet connection) during time (the latency time), or in other words speed of transfer of data

**Latency can be very good on a LAN, but latency deteriorates (pogoršava se) quickly when you use WAN (or internet). Latency is more problematic than bandwidth.**

**There is even a natural capon latency. The minimum round-trip time (latency) between two points on Earth is determined by the maximum speed of information transmission, which is the speed of light.** (At roughly 300 000 km/s, it will always take at least 30ms to send a ping from Europe to the US and back).

**Taking latency into consideration means you should strive to make as few as possible calls and assuming you have enough bandwidth, you would want to move as much data out in each of these calls.**

Fallacy "bandwidth is infinite", is not as strong fallacy as others, because if there is one thing which is constantly getting better, in relation to network, it is bandwidth. While the bandwidth grows, so does the amount of information we try to squeeze through it.

## 4. Performance limits

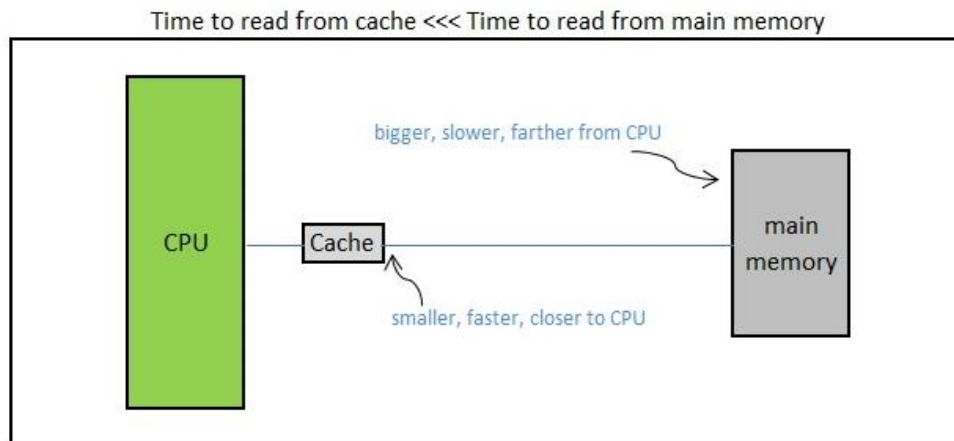
	bandwidth	query time
<b>1GB Ethernet</b> (this is just a notation used for 1 giga bit Ethernet, not 1 giga byte Ethernet)	$1 \frac{Gb}{s} = \frac{1}{8} \frac{GB}{s} = 0.125 \frac{GB}{s} = 125 \frac{MB}{s}$	6s
<b>rotating disk</b>	$200 \frac{MB}{s}$	3.6s
<b>SATA SSD</b>	$500 \frac{MB}{s}$	1.6s
<b>PCIe SSD</b>	$2 \frac{GB}{s}$	0.36s
<b>DRAM (dynamic RAM)</b>	$20 \frac{GB}{s}$	0.04s

In the table above are shown theoretical limits, not what we can achieve in practice (but we should get as close as possible to these limits).

If we use a rotating disk, then the time spent to perform some activity is dominated most likely by the disk speed, but in case of some faster devices we might even get to a point where we have problems with CPU power. In case of using DRAM, at least in theory, we could achieve that time for executing a query is 40ms (i.e. 0.04s), but in practice this is very hard to achieve.

## 5. Cache

The first time a program is executed, data is stored in main memory (i.e. RAM). After that, it gets copied to cache. Afterwards this, it is accessed in a quicker way, making the following (i.e. next) executions faster (and CPU bound, meaning that its speed of execution is limited by the speed of the CPU).



Cache is a small and faster memory, that helps avoid CPU to access main memory (bigger and slower) to save time (cache reads are ~100 x faster than reads from main memory). But this only helps if the data that your program needs has been cached (read from main memory into cache) and is valid. Also, cache gets populated with data over time. So, cache can be:

1. Empty, or
2. can contain *irrelevant* data, or
3. can contain *relevant* data.

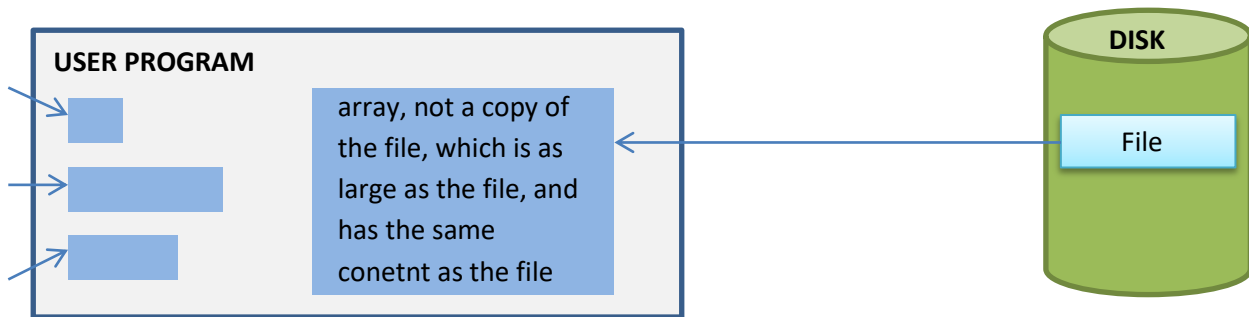
**Cold cache:** When the cache is empty, so that CPU needs to do a slower read from main memory (RAM) for your program data requirement. This is causing cache misses, as the OS is firstly checking the cache to see if it can find the data we are looking for, which might be saved in the cache.

**Hot cache:** When the cache contains relevant data, and all the reads for your program are satisfied from the cache itself. Hot cache leads to cache hits, meaning that when the OS is checking the cache to see if the data we are looking for might be saved in the cache, it finds it, and uses it, instead going all the way to the main memory (RAM).

So, hot caches are desirable, cold caches are not.

## 6. Memory mapped IO (MMIO)

We should avoid copying of a file from the OS to user program, because this may be very expensive. We can avoid this copying by using memory mapped IO. Memory mapped IO is a concept, who's only downside is that we need to use low level functionality for, but there are also libraries which make the use of it very easy. Firstly, we are going to explain what memory mapped IO does:



Memory mapped IO is conceptually a simple idea. The gray box on the left of the image above is a user program, and this program (of course) has a lot of data structures inside it, and the blue rectangles inside the user program present allocated data structures (for an example when you use “new” construct in C++), and all of them can be accessed using some kind of a pointer. Now, where does the memory mapped IO come to play here? We have a disk, and on that disk there is a file. Now, when we tell the OS that we want to use this file from the disk using memory mapped IO, the OS lends this file into the program space (it is mapped in the address space of the program), but an important thing is that it does not do a copy. What happens instead is that **OS internally has a filesystem cache**. We already know that we read a file multiple times, that the reads after the first one are faster, and this is because the operating system keeps the file copied in memory, meaning that it doesn’t really go to disk, but it copies it in memory. **If we use memory mapped IO, then the OS allows you look at the filesystem cache**. So content of the file is now visible in the program. **From our perspective (in the user program), it looks as if we have an array, and that this array is as large as the file, and the content of the array is the content of the file. Additionally, when you access this array, the OS automatically load the part of the file that is needed into memory, and if the file is too large, it throws away the part that is not currently used in the program.** Simply, in short, we can just say that memory mapped IO allows us to view content of a file as if it was a regular array in a program, and no need for copying the file, which is a really expensive thing to do.



## 7. Should we use a Cluster?

Before we conclude should we use a cluster of machines or not, we should pose to questions to ourselves:

### 1. Do we need to **ship data**?

This means that we should ship the data from some central device which is storing the data to the separate machines. This may be a bad idea, because the cost of transferring the data (cost of latency) might be high, and the computation might be more cost efficient to do on a single machine.

### 2. Can we only **ship computation**?

On the other hand, if the data is already distributed across the machines, then we would not have to ship the data, but to ship the computation, meaning that we would only send the program to all the machines, which would afterwards perform some commands on the data which is on the machines.

The time spent on shipping the program, and sending back the result after the computation is performed is roughly 20ms.

Additionally, assuming that:

- “ $|W|$ ” is the size of the data (“W” for “working set”)
- “ $n$ ” number of machines in a cluster
- “ $\frac{|W|}{n}$ ” the amount of the data on each machine
- we are assuming that we can process the data at speed of “ $5 \frac{GB}{s}$ ”

Cost of task execution	Where is the task executed
$\frac{ W }{5 \frac{GB}{s}}$	execution of a task locally
$20ms + \frac{ W }{5 \frac{GB}{s} * n}$	execution of a task across multiple machines

If we have 10 machines (i.e.  $n=10$ ), and we have data which is larger than 113MB (i.e.  $|W| \geq 113MB$ ), then we can calculate that the shipping of the computation is faster than a computation on a single machine. That can be shown from the following calculation:

$$20ms + \frac{|W|}{5 \frac{GB}{s} * n} \leq \frac{|W|}{5 \frac{GB}{s}}$$

$$20ms + \frac{|W|}{5 \frac{GB}{s} * 10} - \frac{|W|}{5 \frac{GB}{s}} \leq 0$$

$$20 * 10^{-3}s + \frac{|W|}{50 \frac{GB}{s}} - \frac{|W|}{5 \frac{GB}{s}} \leq 0$$

$$\frac{20 * 10^{(-3)}s * 50 + |W| - 10 * |W|}{50 \frac{GB}{s}} \leq 0$$

$$\frac{20 * 10^{(-3)}s * 50 \frac{GB}{s} - 9 * |W|}{50 \frac{GB}{s}} \leq 0$$

$$\frac{1GB - 9 * |W|}{50 \frac{GB}{s}} \leq 0$$

$$1GB - 9 * |W| \leq 0$$

$$9 * |W| \geq 1GB$$


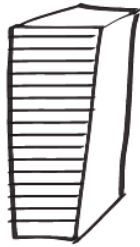
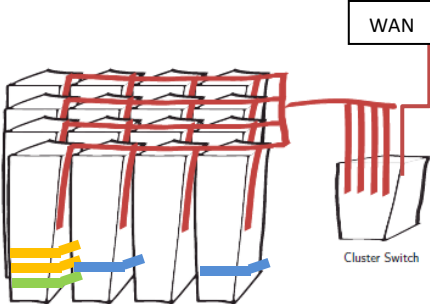
$$9 * |W| \geq 1024MB$$

$$|W| \geq \frac{1024MB}{9}$$

$$|W| \geq 113.77MB$$

This example shows that if the data is large enough, shipping computation might be a good idea.

## 8. Data center architecture

One Sever	Rack of servers	Cluster of racks
 <p>One Server</p>	 <p>Rack of Servers</p>	 <p>Cluster of Racks</p>
<p>A single server, because of its looks is often called “a pizza box”, or “a single box”</p> <p>It is similar to a computer, which can be integrated to be part of a rack. Of course, from a hardware perspective it differs from an ordinary computer, but from a programming perspective, it is a classical computer</p>	<p>Typically in a data center you do not have one server, but many servers, and in that case we form a rack of servers.</p> <p>Note that this rack physically has as network connection (i.e. backplane) between all the server machines which are part of the rack, so that they can talk to each other</p> <p>The network connection speed between servers depends on the rack itself, but we are hopefully going to have a pretty fast network between these servers.</p>	<p>In a data center you most likely do not have a single rack, but a cluster of racks, which are all connected into a network into to a so called cluster switch. Through this cluster switch, one rack may talk to another rack.</p> <p><u>1. Local connection, within same server:</u> If we have multiple processes in one server (let’s say the one noted in light green in the photo), they can of course directly talk to each other, with just some kind of loop-back devise, or something else.</p> <p><u>2. Connection within same rack:</u> If two servers from the same rack want to talk to each other (the ones colored in orange), then they use the backplane of the rack in which they are to talk to each other.</p> <p><u>3. Connection within same cluster:</u> If two servers which are on different racks, but within the same cluster (look at the two servers which are colored in blue), then we have to go outside of the rack, over to the cluster switch, and then to the rack on which is the server we want to talk to is. This is more expensive.</p> <p>Just for completeness, in the photo the cluster switch is connected to the WAN (i.e. the internet), because there might be machines which are even further away, but usually all the machines are within a data center.</p>

## 9. Storage hierarchy

Accessing data on remote machines comes at a price:

		local storage	latency of accessing something in memory	transfer rate from local memory
one (same) server	local DRAM	16 GB	100 ns	$20 \frac{GB}{s}$
	local disk	2 TB	10 ms	$200 \frac{MB}{s}$
same rack	rack local DRAM (80 servers)	1 TB	300 $\mu$ s	$100 \frac{MB}{s}$
	rack local disk (80 servers)	160 TB	11 ms	$100 \frac{MB}{s}$
same cluster	cluster DRAM (30 racks)	30 TB	500 $\mu$ s	$10 \frac{MB}{s}$
	cluster disk (30 racks)	4.8 TB	12 ms	$10 \frac{MB}{s}$

## 10. Scale up and Scale out

In case no single machine is large enough to store all the data, it is necessary to choose between scaling up and out.

**Scale up** - upgrading a cluster by changing small machines with a small number of large and expensive machines

**Scale out** - upgrading a cluster by adding a large number of cheap commodity machines

Nodes need to talk to each other!

If we speak to each other on a single node we need approximately 100ns

If we speak to each other across nodes we need approximately 100 $\mu$ s

## 11. Modeling communication overhead computation

Simple execution cost model:

$$\text{TotalCost} = \text{CostOfComputation} + \text{CostToAccessGlobalData}$$

*CostOfComputation* – the time which is needed for some computation to be done (like for an example computing a sum of  $n$  numbers)

*CostToAccessGlobalData* - data on which operations need to be performed must be accessed, so this term represents the time needed to access the data

Fraction of local access is inversely proportional to size of cluster (number of nodes/machines in a cluster), meaning that " $\frac{1}{n}$ " of the data is stored locally (assuming that the data is evenly distributed across machines, i.e. that every machine has the same amount of data) and " $n$ " represents number of nodes (i.e. machines) in a cluster (ignore cores and multithreading for now).

$$\text{TotalCost} = \frac{1 \text{ ms}}{n} + f * [ 100\text{ns} * \frac{1}{n} + 100\mu\text{s} * (1 - \frac{1}{n}) ]$$

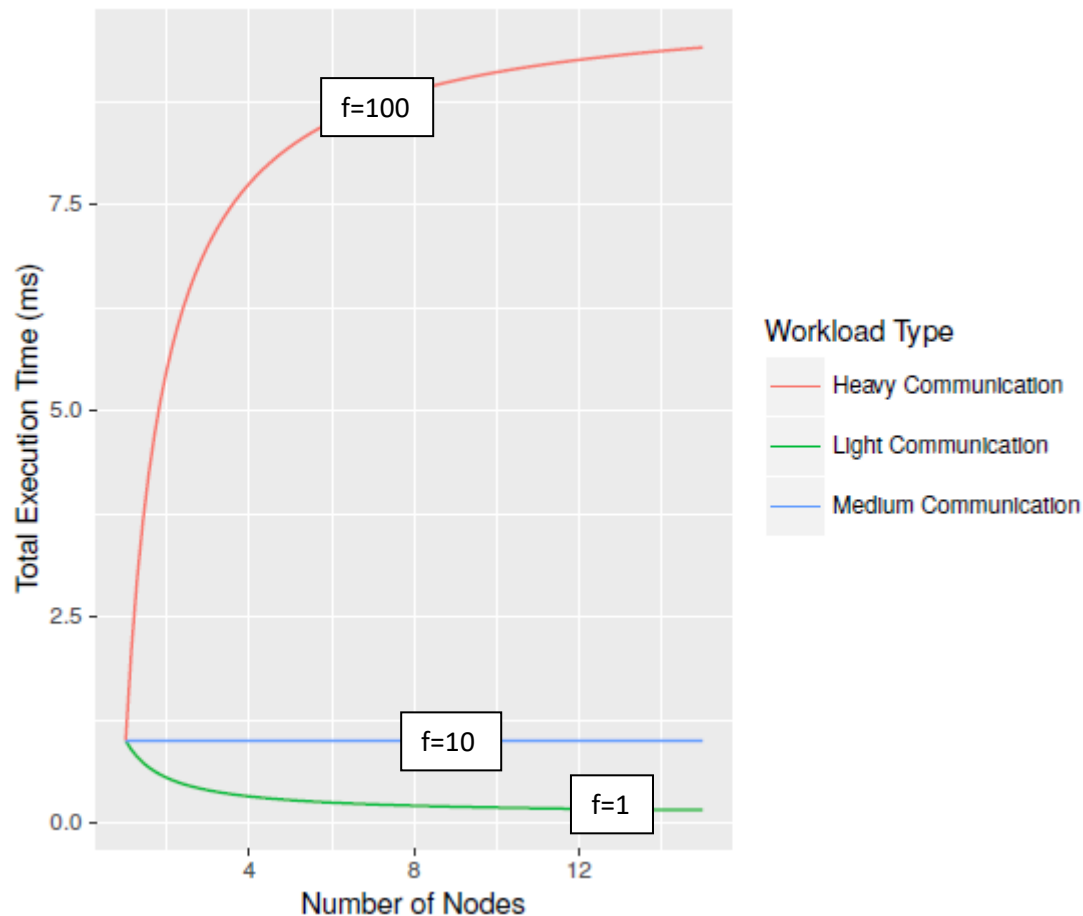
where part of the formula:

1. " $\frac{1 \text{ ms}}{n}$ " represents cost of computation (it is assumed that the computation cost is 1ms). It is divided by " $n$ ", because the more machines that we have, the computation becomes cheaper.
2. " $100\text{ns} * \frac{1}{n}$ " represents local access
3. " $100\mu\text{s} * (1 - \frac{1}{n})$ " represents remote access
4. " $f * [ 100\text{ns} * \frac{1}{n} + 100\mu\text{s} * (1 - \frac{1}{n}) ]$ " represents cost of accessing global data. For each " $f$ " accesses we have a chance of " $\frac{1}{n}$ " that the access is local (which means that the access in this case would cost 100ns), and a chance of " $(1 - \frac{1}{n})$ " that the access is remote (which means that the access in this case would cost 100μs).

Let's now consider three scenarios:

1. Light communication:  $f = 1$
2. Medium communication:  $f = 10$
3. Heavy communication:  $f = 100$

where " $f$ " represents the factor, which tells us how much elements do we have to look at.



The above graph represents the relationship between Total Execution Time in *ms* (i.e. the runtime), and the number of nodes in a cluster  $n$ .

By looking on the graph for the blue line, which represents medium communication, we will see that it represents a flat line. Even though the number of machines is increased, the execution time (i.e. runtime) doesn't become faster any more. The reason for such a behavior is because we are here dominated by network access cost. So, by adding more machines, the *CostOfComputation* is going to decrease, while the *CostToAccessGlobalData* is going to increase.

From the shape of the green line we see that when the communication is light, by increasing the number of nodes, the execution time is decreasing. But even in this case of light communication, the execution time does not go down below a certain point, even if we include more machines and the reason for this is that *CostOfComputation* goes down arbitrary, but the *CostToAccessGlobalData* doesn't go down arbitrarily when adding more nodes.

The red line, representing the heavy communication (i.e. where  $f=100$ , meaning that we will have to perform 100 accesses to get the data that we need), is rising rapidly as the number of nodes/machines in a cluster is increasing. This is due to the fact that the data is more likely to be on another machine, which means that we will need more communication across

machines, and this kind of communication is very expensive, as we have to wait for the network.

In the communication pattern that we have here, using multiple machines doesn't work at all. For an example, if we look at the scenarios of blue and the red curve, we would rather prefer to have more powerful machines (scale-up), than more machines (scale-out), simply because the communication cost is so high.

The main problem for such performance is that we do something that we call "random access", which means that we go to some machine in the cluster and ask them to give us some data item  $x$ , and then we go to some other machine and ask them to give us data item  $y$ . Random access is terrible, so if you want to read something from remote machines at random (i.e. ask for just individual data items) this will lead to very bad performance. Doing it once or twice or so is ok, but this is simply too expensive.

### 11.1 Random access on a single machines example:

Interestingly, the same observation that random access is a bad idea can even be observed by looking at a single machine/node, and it is going to be shown in an example:

Assume we have 1TB of database, with 100 byte records, and we want to update 1% of the records. So, most of the data is going to be unmodified, but only a tiny fraction of the data.

We are going to look at two ways how to solve this problem:

#### 11.1.1. Solving the problem using random access:

By using random access we would basically, jump to the record we want to update, update it, and write it back. Then again, jump to the next record that we want to update, update it, and write it back. And we repeat this procedure until all records that we wanted to update are updated.

In the drawing below the whole dataset is going to be presented:



Assume that the blue rectangle represents the whole dataset, while all the yellow stripes together represent 1% of the data.

What we consider random access is that we go to block of data "B1", update it, and write it back. Then we go to block of data "B2", update it, and then write it back. And so on, until all blocks  $B_i$  (where  $i=1, \dots, 9$ ) which represent the 1% of the data that we want to update, are updated.

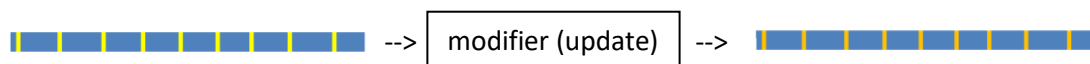


In case of random access each update takes for approximately 30ms on a rotating disk, we would need approximately 35 days to perform this operation.

#### 11.1.2. Solve the problem by reading all the records:

We do not read the individual records, but read everything from the beginning, and only when we see that we are currently at the record that we want to update, we change it in memory, and write everything back. So, we basically use a stream, that is, we read everything, update the needed part, and write everything back

read everything -> update the part you want to update -> write back everything



If we assume that we have  $100 \frac{MB}{s}$  (which in reality is more likely to be  $200 \frac{MB}{s}$ ), for this operation we would need 5.6h.

We see by comparing the time needed to perform the same action using two different approaches, we see that random access approach gives much worse performances than the approach or read and write everything. This leads to a conclusion that using random access is a bad idea, and that we should avoid random access if we can. While using random access is a terrible idea most of the time, this doesn't mean that it is a bad idea always, as there might be situations where using random access is a good approach, and that might be when we are trying to update 1 data instance instead of 1% of the data instances.

**The above explained example considered the case of using a single rotating disk, but the same pattern and observations would be made in cluster computing.** Jumping to one machine and telling it to give us one data instance is a terrible access pattern, and it should not be done this way, because latency is so high, meaning that the **round-trip is too expensive compared to computation itself**, so, asking for a single data item is a terrible idea.

## 11.2 Top-k algorithm, as example of disadvantages of using random access

To present the same conclusions here, the top-k-algorithm is going to be explained:

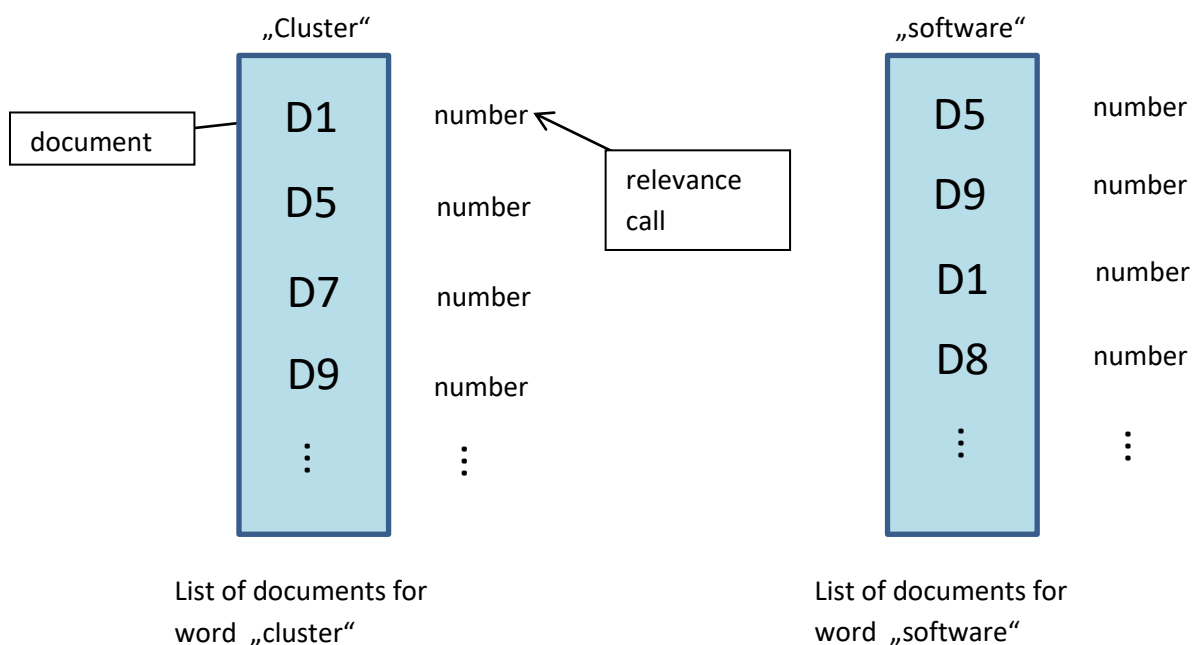
When you search the internet, you get a ranked result. If you are, for an example, for typing in the browser "cluster software", and then you want to get some documents back for this

specific search. In practice, for this purpose page-rank algorithm is used, but to simplify the explanation, we are going to use the top-k-algorithm.

As a result of our search we might get a list of documents which consider the part of the search “cluster”, while the other list that we might have is the list of documents for part of the search “software”.

We use here two lists to make the problem simple, but of course the same problem we are discussing with two might be a problem with multiple lists.

It also could be that you do not have only two lists for two words, as documents for each word can be distributed across multiple machines. This means that one list which is presented here for the part of the search “software”, might be distributed across 5 machines, and therefore it would not be one list, but five lists.



The higher the “relevance call” is, the bigger chance that the document associated with it will be outputted as one of the first for a search. Also, the “relevance call” is the highest for the top element in the list, and as it is going down this number decreases.

What we want now to get is the top (for an example) 2 documents for the search, which have the highest total relevance.

Let’s assume that total relevance is the sum of relevance calls for a single document from the two lists. This would mean that if a document “D5” has a relevance call of “0.7” in the list of documents named “cluster”, and the same document has relevance call of “0.6” in the list of documents named “software”, then then total relevance of document “D5” would be  $0.7+0.6=1.3$ .

Now, to find the top 2 sites with the highest total relevance is the top-k-algorithm. but there are different ways in which we can perform the computations to choose the top sites. Here, we are going to demonstrate two approaches for executing computations for the algorithm. The example on which both of the approaches are going to be explained is given in an illustration below:

„Cluster“		„software“	
D1	0.9	D5	0.7
D5	0.8	D9	0.7
D7	0.6	D1	0.6
D9	0.5	D8	0.4
⋮	⋮	⋮	⋮

### 11.2.1 First approach to solving top-k algorithm

One way to perform the computation is to ask “cluster” node (i.e. “cluster” list of documents) for the most important document, which is in “D1”, and then compute the total relevance by finding the relevance call number for “D1” in the list of documents named “software”. This will give us  $0.9+0.6=1.5$ .

„Cluster“		„software“		Result:
D1	0.9	D5	0.7	
D5	0.8	D9	0.7	1. D1 : $0.9+0.6=1.5$
D7	0.6	D1	0.6	
D9	0.5	D8	0.4	
⋮	⋮	⋮	⋮	

Next, we look at the top document of the list “software”, which is “D5”. Then we look in the list “cluster” for the same document, and afterwards get that the total relevance of the document “D5” is  $0.7+0.8=1.5$ .

„Cluster“		„software“		Result:
D1	0.9	D5	0.7	
D5	0.8	D9	0.7	1. D1 : $0.9+0.6=1.5$ 2. D5 : $0.7+0.8=1.5$
D7	0.6	D1	0.6	
D9	0.5	D8	0.4	
⋮	⋮	⋮	⋮	

We should now look at the second element from the top of the list “cluster”, but we already have the total relevance for the document “D5”, so we skip it, and instead look at the second element of the list “software”.

„Cluster“		„software“		Result:
D1	0.9	D5	0.7	
D5	0.8	D9	0.7	
D7	0.6	D1	0.6	
D9	0.5	D8	0.4	
⋮		⋮		

After looking the second element of list “software”, “D9”, and looking for the same element in the list “cluster”, we conclude that the total relevance of this document is  $0.7+0.5=1.2$ . Because this number is less than the two total relevance that we calculated for the documents “D1” and “D5”, and for our top k algorithm we choose  $k=2$ .

„Cluster“		„software“		Result:
D1	0.9	D5	0.7	
D5	0.8	D9	0.7	
D7	0.6	D1	0.6	
D9	0.5	D8	0.4	
⋮		⋮		

Now, we consider the third document from list “cluster”, which is “D7”, and see that its relevance call is 0.7. Knowing that the last element we looked at in the list “software” had relevance call equal to 0.7, and knowing that documents are sorted from top to bottom in descending ordered by relevance calls in the list, we conclude that the total relevance call of the document that we just looked at (“D9”) cannot be higher than the total relevance documents that are currently in the solution, even if the document “D9” had the biggest possible relevance call in the list “software”, as in that case it would have total relevance call of  $0.7+0.7=1.4$ , and the once in the final result have a total relevance of 1.5.

„Cluster“

D1	0.9
D5	0.8
D7	0.6
D9	0.5
⋮	

„software“

D5	0.7
D9	0.7
D1	0.6
D8	0.4
⋮	

Result:

1. D1 :  $0.9+0.6=1.5$
2. D5 :  $0.7+0.8=1.5$

At this point we stop and terminate the algorithm, even though there might be thousands of documents in the two lists, as at this point we are now sure that none of those remaining documents have a total relevance higher than the total relevance for the documents which are at this point in the result.

In this example we were only reading what we needed to consider, in order to calculate the total relevance for a document. This is a good solution if you want to perform the top-k algorithm on a single machine, but if you do it across multiple machines, it would have terrible performance. The problem is that fetching the relevance call for a corresponding document from another machine is simply too expensive, because the traffic over the network is too expensive.

### 11.2.2 Second approach to solving top-k algorithm

Phase 1:

In this approach we ask both of the lists to give us their top two elements, which is a one round-trip over the network.

„Cluster“

D1	0.9
D5	0.8
D7	0.6
D9	0.5
⋮	

„software“

D5	0.7
D9	0.7
D1	0.6
D8	0.4
⋮	

list „cluster“:

1. D1 : 0.9
2. D5 : 0.8

list „software“:

1. D5 : 0.7
2. D9 : 0.7

Afterwards we were given top two elements from both lists, we have that total relevance for documents “D5”, “D1” and “D9” are 1.5, 0.9, 0.7.

## Phase 2:

At this point we do not know that “D9” is not going to make it to the final result, but just that the final result will have at least 1.5, as we observed it as a total relevance for the document “D5”.

Now we contact the each of the lists again to provide us with all of the documents which have a potential of making it to the final result, meaning that we want to find the documents which have a possibility of having the total relevance equal to 1.5 or higher.

Let’s say we are currently at the list “cluster”, and it is provided with the top relevance calls from list “software”, so that it knows that the relevance calls values for documents in that list, so it can choose which documents in it have potential to make it to the result.

„Cluster“		Relevance calls from „software„ list:	
D1	0.9	1.	0.7
D5	0.8	2.	0.7
D7	0.6	- give documents with potential of having total relevance of 1.5	
D9	0.5		
⋮			

If we had more lists (i.e. nodes), then top elements from all of them would be sent to a list in order to help it choose the documents inside it, which can potentially make it to the result.

List “cluster” doesn’t know what exactly what list “software” has, but it knows that the minimum score from the first phase that a list “software” reported was 0.7, which implies that every document below this one has relevance calls which are 0.7 or lower.

As after the first phase the second best score in the result is 0.9, we will look at “D7”’s relevance call in the list “cluster”, and see that it is equal to 0.7, and we know that the smallest reported relevance call by the list “software” was 0.7, so, we conclude that “D7” from list “cluster” can make it to the result. So, we have to report it in this second phase.

Repeat the same procedure until you reach a value which would not be able to make it to the result. As this example is very small, in this second phase we would have to include all the documents from both of the lists, but in reality if we had a reasonably large lists, the pruning power by such approach would be visible, because it is very high. The reason for this is that you will return just a small part of the lists which could make it to the result, and all other elements you can just ignore.

„Cluster“		„software“	
D1	0.9	D5	0.7
D5	0.8	D9	0.7
D7	0.6	D1	0.6
D9	0.5	D8	0.4
⋮		⋮	

This second strategy for computing the top-k result transfers far more data over the network, as in the first approach we look only at elements which can make it to the result. In the second approach we basically do two queries, where the first query gives the top k results, from each of the lists, and the second query gives everything (all documents) which can still make it to the top k result, given relevance call values from other lists, which are going used as bounds. The total amount of data that is transferred by the second strategy is much higher, but far less round-trips are done, only two round-trips. We talk to every machine once, and then again we talk to every machine once more, and there is no further access to other machines. Transfer volume is higher, but there are far less round-trips, and therefore the second approach is usually better in case of a network scenario. The second approach does a lot of sequential access, it **sends a lot of sequentially data over the network, but this is much faster and also we can buy bandwidth for a modest amount of money, which allows us to transfer reasonable amount of data over the network, while latency we cannot bring down (at least not arbitrary).**

So, we prefer algorithms which send a lot of data sequentially, over algorithms which send a little amount of data at random, assuming that the random access is reasonably large, because by doing a lot of round-trips may cause bad performances.

## 12. Exam examples of previous years

### 12.1 Tasks from 2019/2020 final exam, section performance spectrum

#### 12.1.1 Task 1

Run program locally twice, which reads a file. First execution is 25s and the second execution is 25ms. Please estimate the size of the file using the memory bandwidth and explain with 2 sentences. (No hint about the disk latency, memory latency, if the file is memory mapped)

-----

#### 12.1.2 Task 2

Request a entry at 356 Byte within a 50MB file remotely. Entries are located consecutively in the file. The record can be located using offset and base address. We use 1 Gbit/s Ethernet and request 5 times randomly. (No hint about the CPU, memory, file location, network protocol, network latency, transmission distance). Which following approach is better and explain why with 2 sentences:

1. Get the whole file once and look up record 5 times locally.
  2. Request 5 times separately via network.
-



## 12.2 Example of 2019/2020 repeat exam (which was online), section performance spectrum

### 12.2.1 Task 1, a):

The company NiceBikes Inc. wants to set up a server farm for to keep all their production records in a data lake.

They estimate their overall data volume to be 15PB.

Occasionally, an employee of NiceBikes Inc. searches for specific events in their production data. There are no index data structures in this data lake, so the requests will result in a complete scan of the data.

Estimate how many hard drives and servers are required so that each scan query can finish within 16 minutes.

-----

We choose type of a disk on which we will store the data on, but as in the text of the problem “hard drives” were mentioned, we are going to presume we need to use HDD-s (i.e. rotating disks). Bandwidth (i.e. speed of transfer of data) of a rotating disk is 200 MB/s.

$$200 \frac{MB}{s} = 200 * 2^{(-10)} \frac{GB}{s} = 0.2 \frac{GB}{s}$$

Additionally, from the setting we have 16 minutes to scan all the data. 16min = 960s

From this information we can conclude how much of data can be read from one disk in the given time, and that can be calculated as:

$$0.2 \frac{GB}{s} * 960 s = 192 GB = 192 * 2^{(-10)} GB = 0.1875 TB$$

Now that we have how much data would be read from one disk we can calculate how much disk we would need by simply dividing the total amount of data we will need to read (in a given time frame), which is 15PB, with the amount of data which will be read from a single disk (in a given time frame), which is 0.1875TB. This can be calculated as:

$$\frac{15 PB}{0.1875 TB} = \frac{15 * 2^{10} TB}{0.1875 TB} = \frac{15360}{0.1875} = 81920$$

This means that we would have to have 81920 rotating disks, out of which each has 0.1875TB of data, to be able to read 15PB of data in 16min.

Now, knowing that a server also has a bandwidth, which is equal to the bandwidth of the Ethernet, we can conclude that we can't fit all the 81920 rotating disks on one server, but that we need

multiple servers. This is even true if we take the quickest Ethernet available today to us, which is the 100GB Ethernet ("100 giga bit Ethernet"). If we use this 100GB Ethernet we would calculate what would it's bandwidth by simply applying the following calculation:

$$100\text{GB Ethernet (which is 100 giga bit Ethernet, not 100 giga byte Ethernet)} = \frac{100}{8} \frac{\text{GB}}{\text{s}} = 12.5 \frac{\text{GB}}{\text{s}}$$

This would mean that a server (or equivalently the Ethernet) has bandwidth of 12.5GB/s.

Additionally to this bandwidth we should take into account that in practice we can connect 4 Ethernet cables to one server, which means that we would not have 12.5GB/s as server bandwidth but  $4 * 12.5\text{GB/s} = 50\text{GB/s}$ .

To conclude how much disks per server we could have we come back to the bandwidth of a rotating disk which is 0.2GB/s, and take into account the bandwidth of the server which is 50GB/s. We conclude that multiple rotating disks together have to have maximum bandwidth which is less than the bandwidth of the server which they are a part of (it like having a pipe, and you can only push something through the pipe if its size is smaller or equal to the size of the pipe hole).

$$0.2 \frac{\text{GB}}{\text{s}} * x = 4 * 12.5 \frac{\text{GB}}{\text{s}}$$

$$0.2 \frac{\text{GB}}{\text{s}} * x = 50 \frac{\text{GB}}{\text{s}}$$

$$x = \frac{50}{0.2} \frac{\text{GB}}{\text{s}}$$

$$x = 250$$

This would mean that we can have 250 rotating disks inside one server.

To conclude how much server we will need in total we go back to the total number of rotating disks we will need, which we calculated to be equal to 81920. We will get the total number of needed servers by dividing the total number of rotating disks we will need with the number of rotating disks which will fit one server, i.e.:

$$\frac{81920}{250} = 327.68$$

An finally taking into account that the number of servers has to be an integer, we will conclude that we would need 328 servers. Each of these servers would have 250 rotating disks, and on each of these disks there would be 0.1875TB of data, in order to be able to read 15PB of data in 16min.

### 12.2.2 Task 1, b):

A key-value store uses a radix tree to index the stored data. With the current data the radix tree has 6 levels.

Each level uses one Byte from the key, thus a node of one level contains 256 pointers (of size 8 Bytes) to nodes of the next level.

Overall, the radix tree contains  $10^6$  entries and the whole index structure fits into main memory.

The workload on the radix tree consists of  $10^5$  key containment checks. Each containment check takes key of 6 Bytes and traverses the index structure to check whether there is a value for the given key. The value is not accessed.

Estimate how long a single containment check will take on average in this workload. Assume that the checked keys are uniformly distributed and every checked key is contained in the index.

---