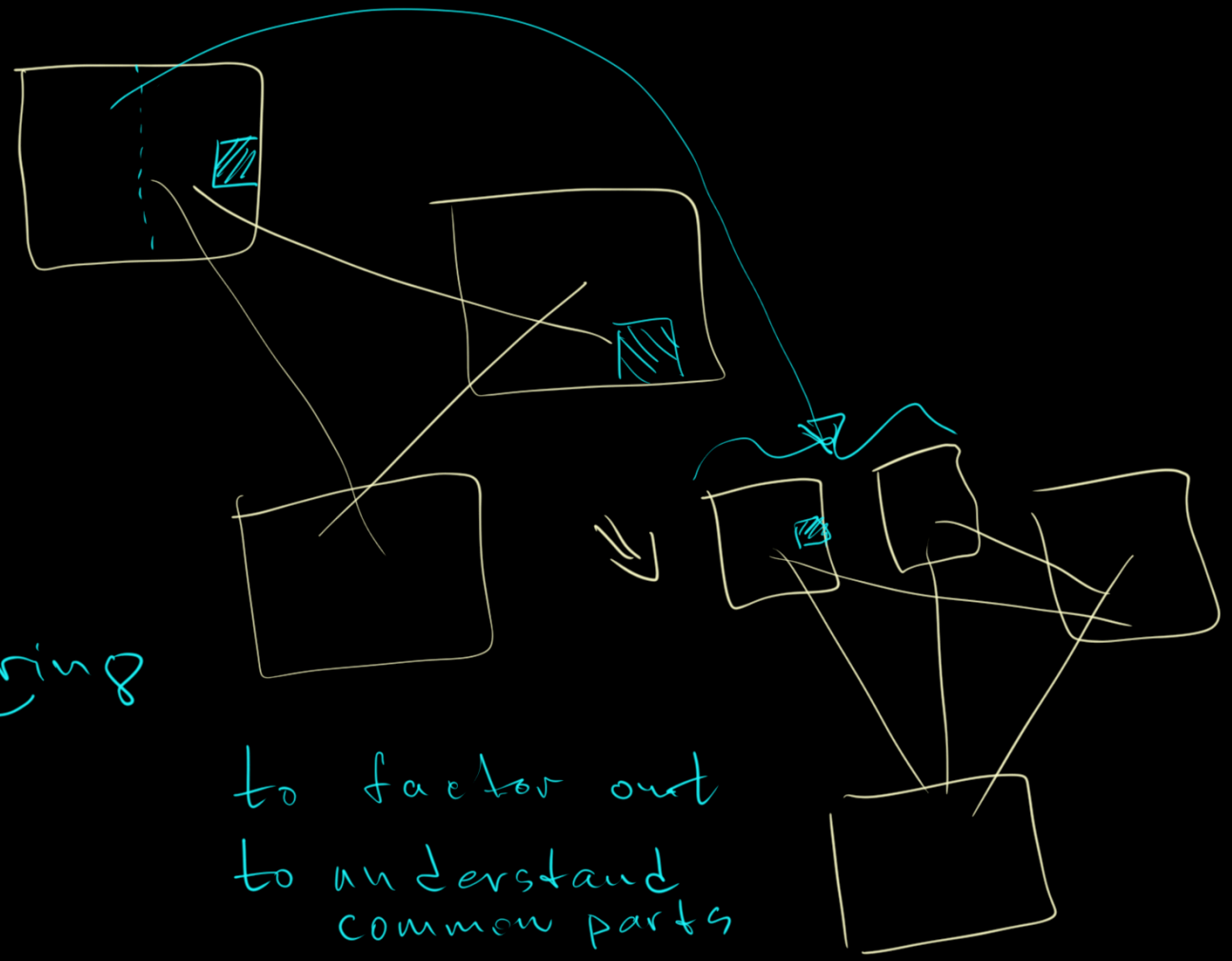
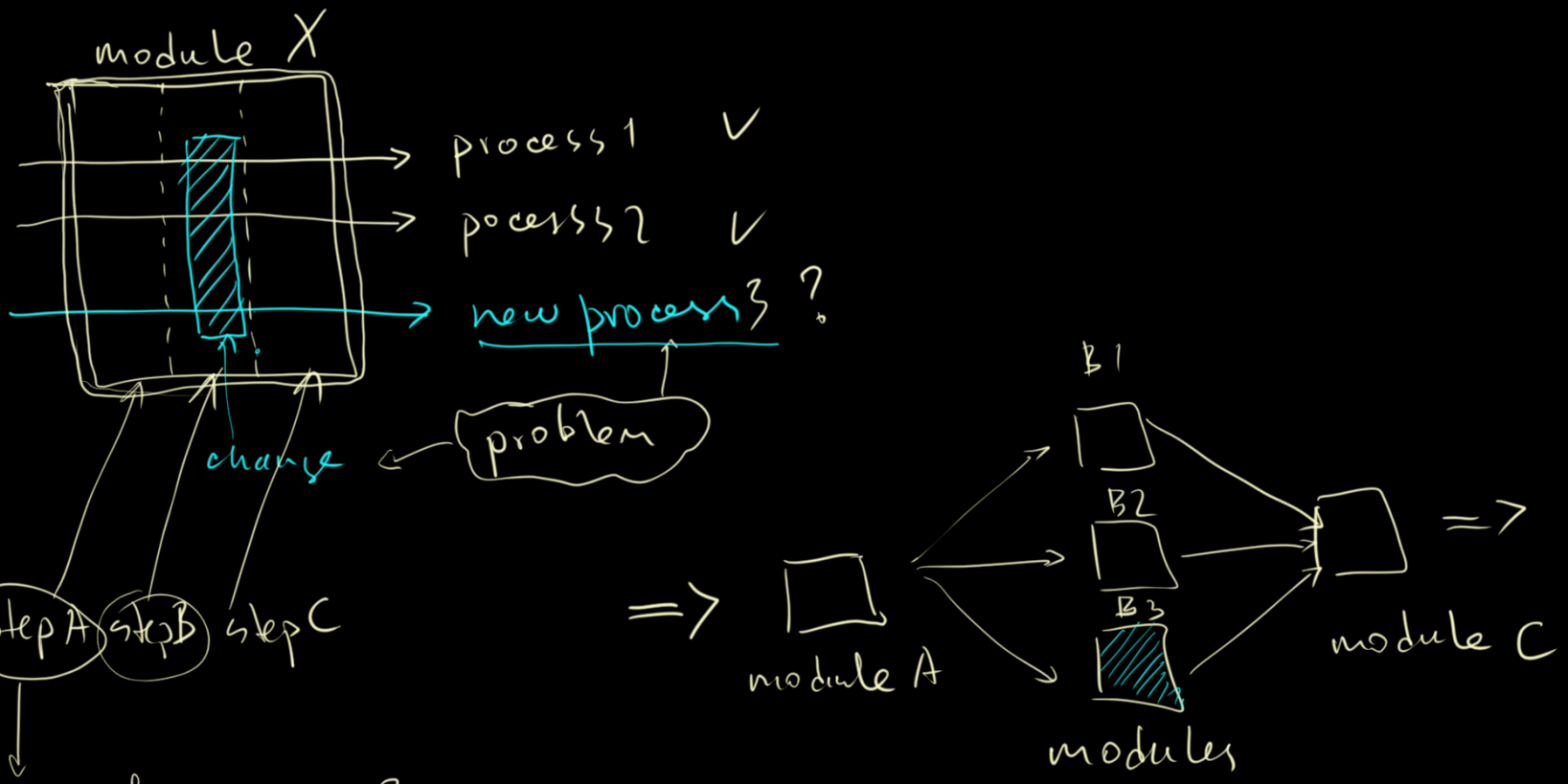
1. Performance
2. Usability
3. Maintainability

redesign
recoding
refactoring



to factor out
to understand
common parts



process 1 : A + B1 + C

2 A + B2 + C

3 A + B3 + C

Architecture Types

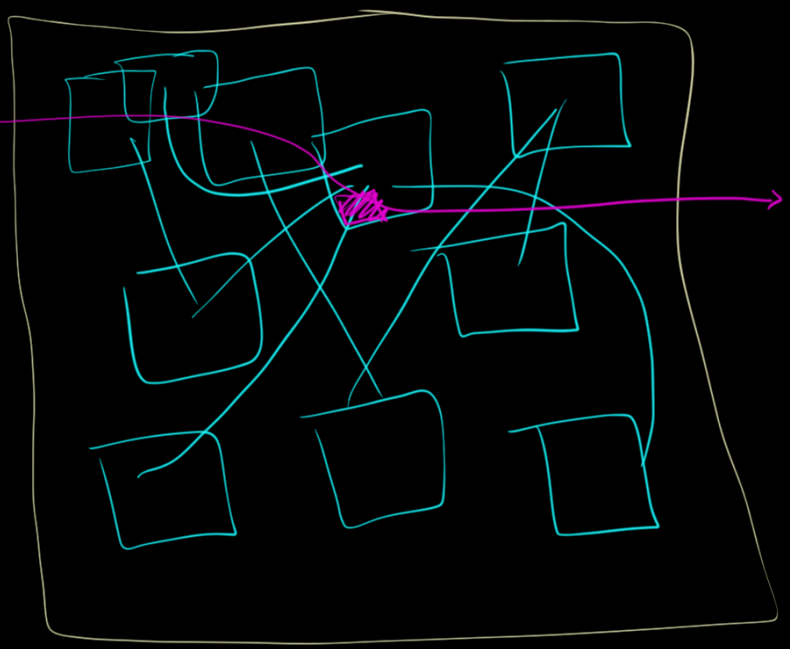
1. Monolith. one big artifact
2. Microservices. many small artifacts
3. N-tier / Hexagonal
4. Streaming.
...

Monolith

1. one artifact (one file to distribute / deploy)
2. one git repository

pros:

1. simplicity of development
2. scope: easy to find what you need
3. easy to deploy
4. easy to find dependencies
5. speed/velocity of development



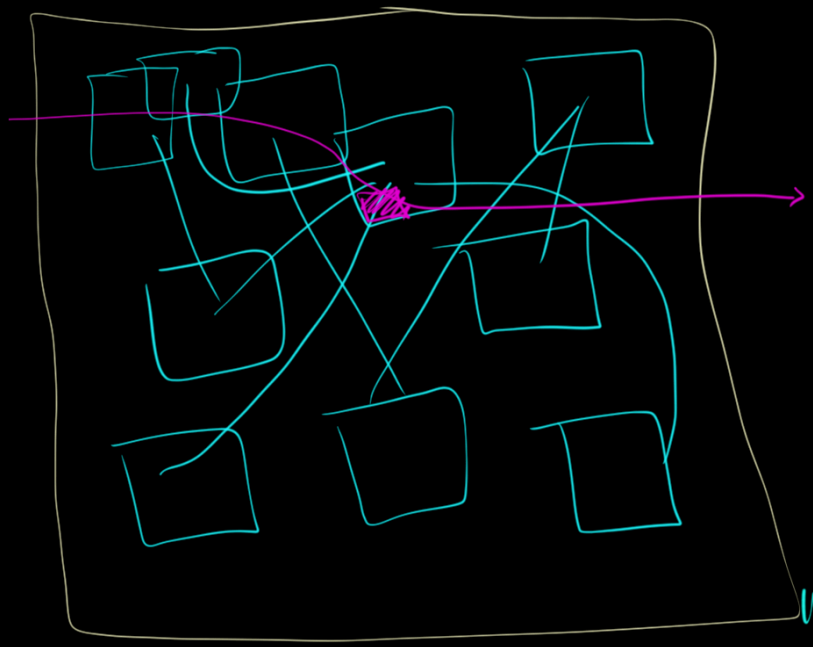
cons:

1. hard to maintain by many developers

1-2-3

10-20

50-1000

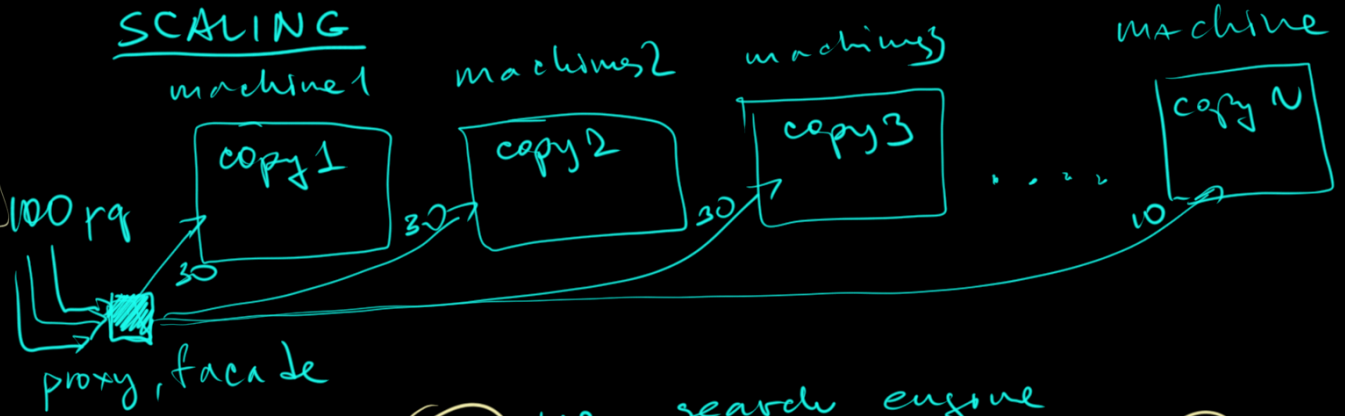


1. search engine
2. payment
3. cart
4. .....
5. warehouse

for ex ONE computer can serve only 30 rq/s

the load is not identical

SCALING



100 use search engine

cart is used only by 10

payment is used by 1-5

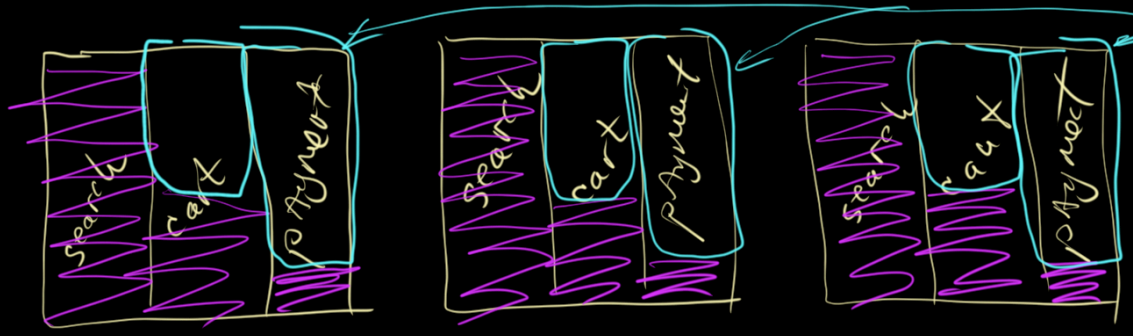
1 req/sec - ok

5 req - ok

20 req - ok

50-100-1000 - who knows!

1000-10000-1M - NO



that resource CAN'T BE USED !!!

we waste these resources

we wasted >50% resources

we lost resources time money ... !!!

1 instance can handle 30 rq/s

we have 100 rq/s

- not all the parts of application need to be scaled to the same scale!

10x search
3x cart
1x payment

- we need to be able to scale them independently
- we need them to be different artifacts
- we need them to be separate pieces of code.

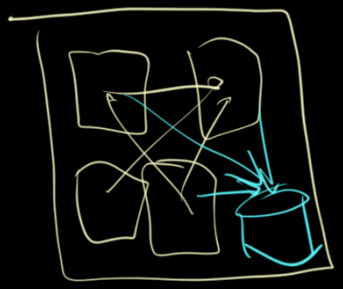
10 x search

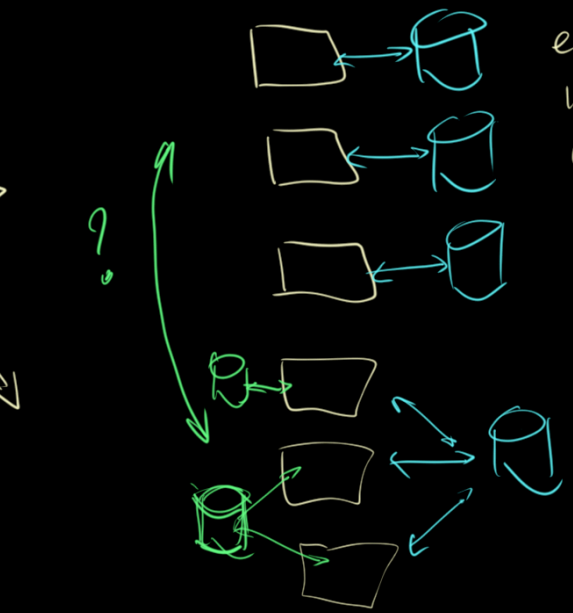
3 x cart

1 x payment

This is the only way to save our ~~money~~.

microservice architecture is a pretty good solution.





each one need to have own db.

more op. work

more stable

no transaction

NO joins

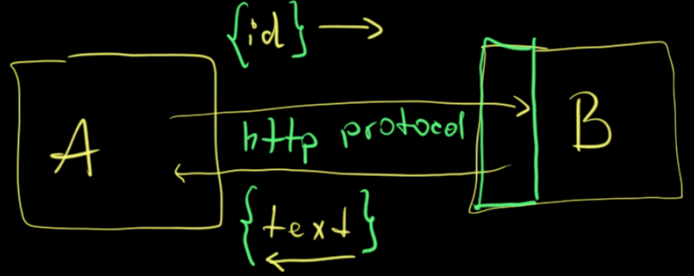
all of them use one common db

less op. work

less stable

transaction

Joins



all interop between microservices is done through HTTP

if contract is changed

microservice A should now a lot about microservice B

A should be aware of all changes in ms B.

→ not an easy need to be managed

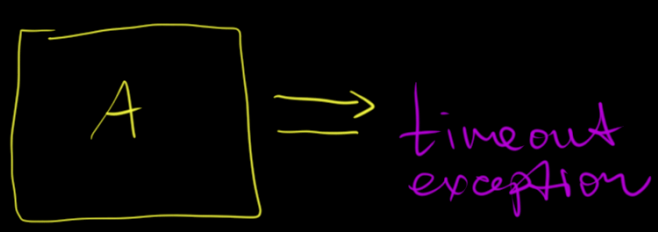
monolith

function call

microservice

http commune.

- bring a lot of complexity due to distributed nature
- we need to deal w/ that.

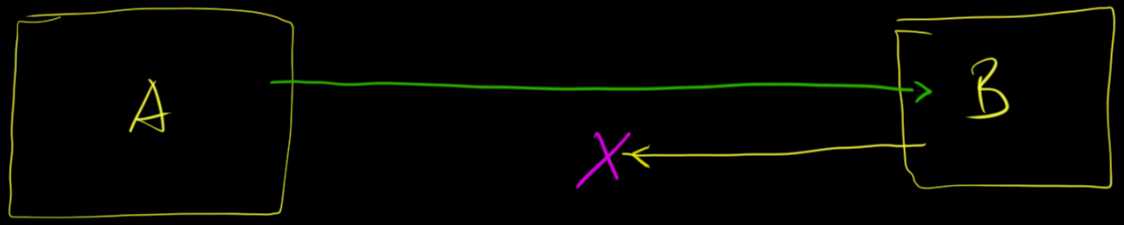


Wikipedia

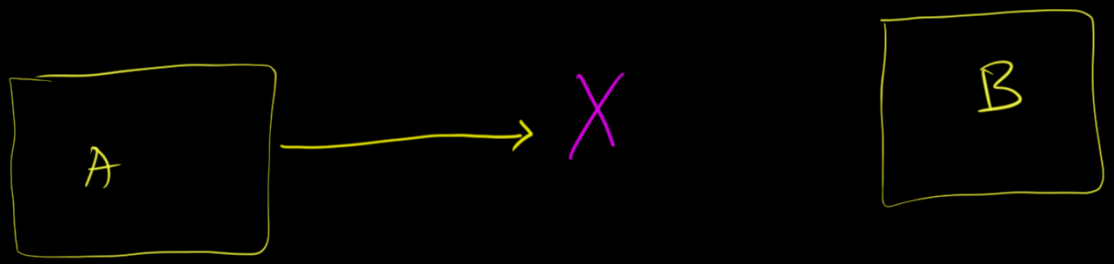
Two Generals

Two Generals' Problem

From Wikipedia, the free encyclopedia

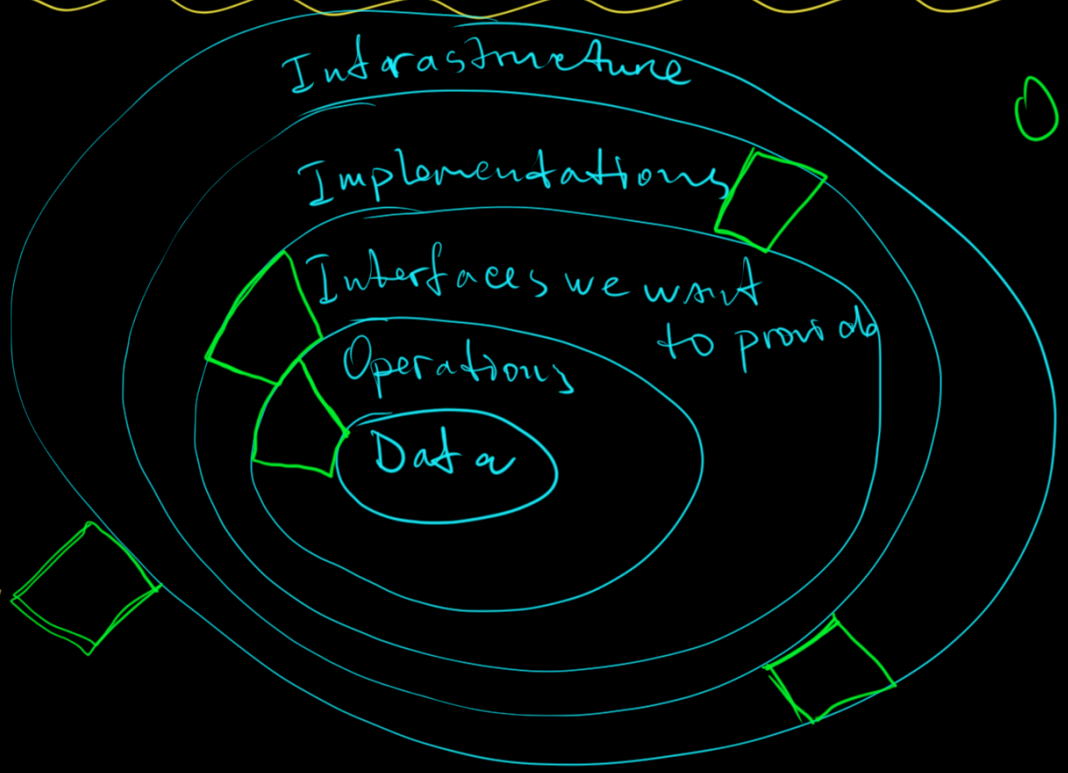


that's about any distributed system



microservices seriously affect the cost of product

Hexagonal
N-Tiers
Data-centric



Onion, Layered

.....

we never modify our code

we just create different implementation

1. Define Data
2. Operations
3. Interfaces
4. Implementation
5. Auth.
   logging
   Tracing
   Deployment, distribution, scalability...

we have an Idea, we need to try
we need to POC
proof of concept
=> Monolith

we don't have many developers
we are limited in time

---

we have something working
we have growing number of users
we are successful company
=> we need to start thinking about microservices

---

we are FAANG
we are about to design Metaverse
with billion users
=> Mircroservices

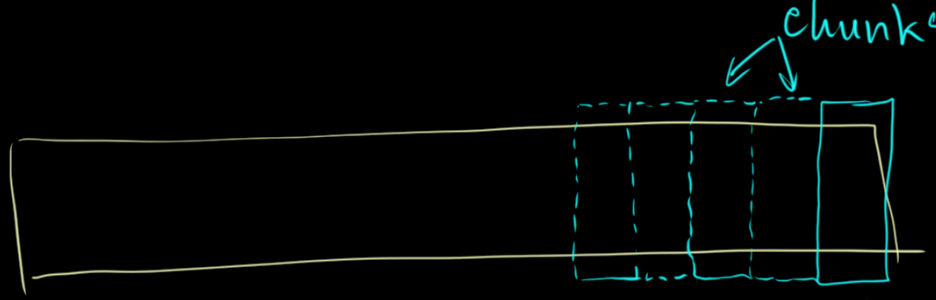
Warehouse, logistic, Banking
Math, Any complicated Domain
w/ a lot of uncertainty
=> N-tier

Streaming, BigData, ML

BigData ≡ DAta doesn't fit to one computer.

You can not work with it in the same manner
load it into memory => it doesn't fit.

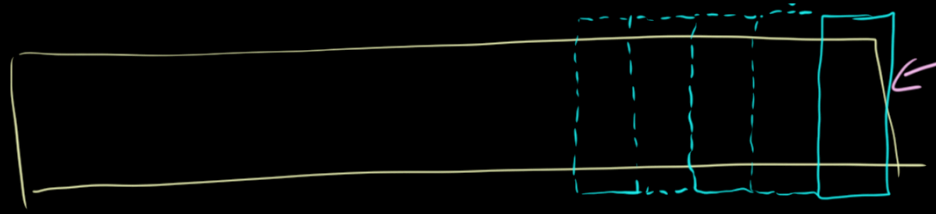
amount of date >> size of computer (computation resources)



we read the source
in chunks
the size of chunk fits to the memory
now I can process ANY amount
the proben is to Represent

huge represent (data) -> chunk[k]
process: (chunk) -> result

data -> chunk[] -> result[]
combine: (result[]) -> result



Streaming is a processing in chunks

any operation w/ Big Data

File
JSON
Text
XML
SQL
⋮
→ chunk

that can be in the Library

result
result
⋮
result
→ result

1. data → chunks[]
2. chunk → result
3. combine result[] → result

the most complicated part

① represent / split

③ result[] combining

bigdata → chunk[] → result[] → result

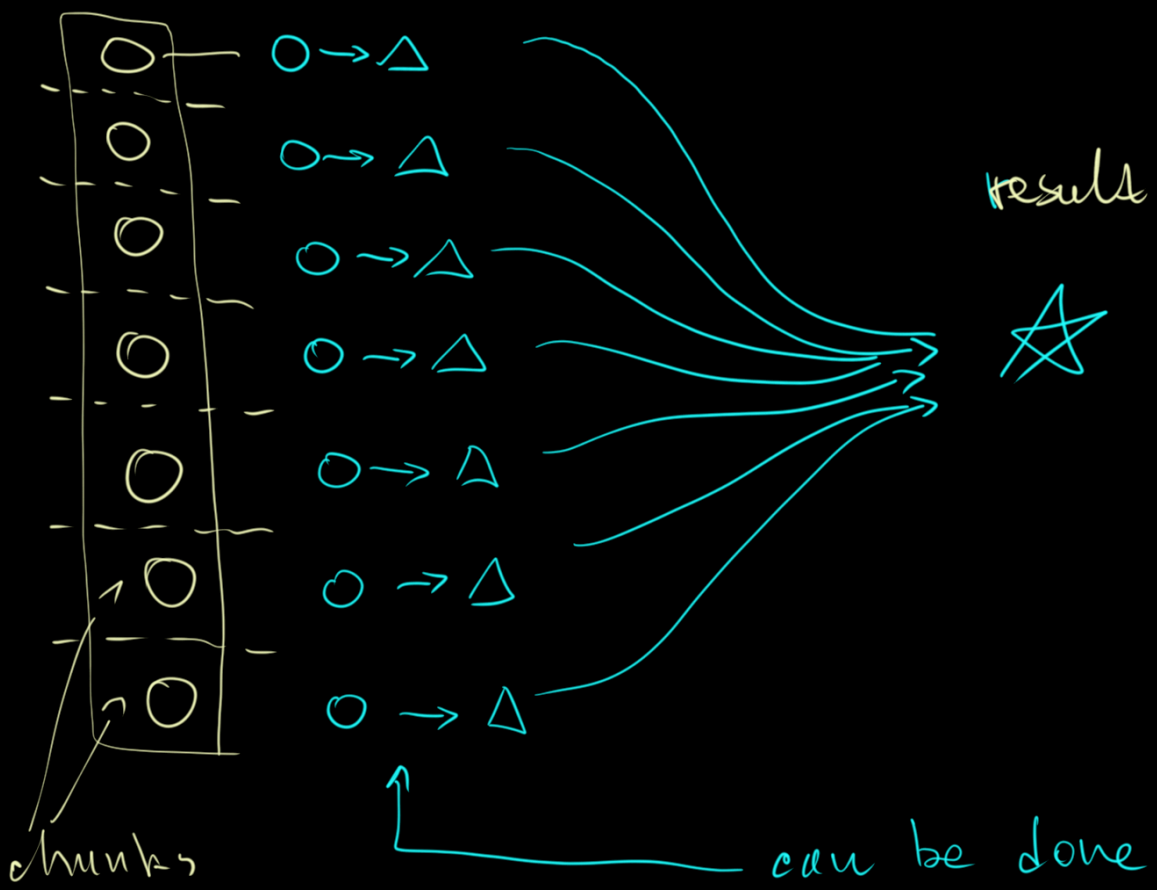
② mapping : chunk → result

mapping fn: ○ → △

split (data) . map (○ → △)
. reduce ((△, △) → ☆)

Map-Reduce frameworks

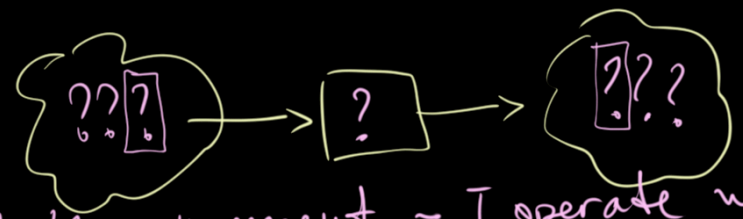
Big Data



can be done in parallel on the different machines

A B

read ⟶ process ⟶ sink (write)



read : (?) => A

process : A => B

write / sink : B => ??

read . andThen (process) . andThen (write)

write (process (read (source)))

chunk

result

in classical streaming
we don't have collect / combine

f: HttpRequest => HttpResponse

HttpServer : Stream < HttpRequest >