Storing Big Data

File systems

Single node

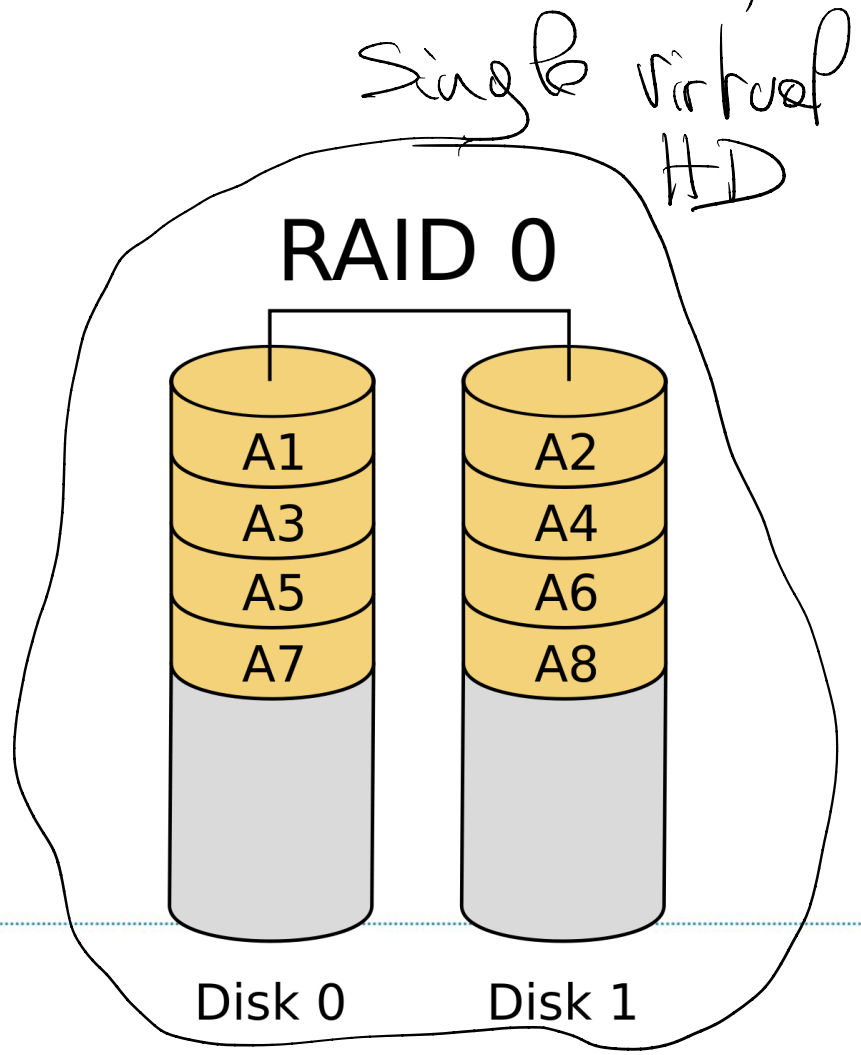
UNIVERSITÉ CÔTE D'AZUR

Storing

- How to store arbitrary data?
  - Hard drive
- A single hard drive is limited
  - So use many!
- But don’t want to clutter users with complexity
  - Windows hard drive hell (C:, D: …)
  - Add a virtualization layer : RAID

Single node

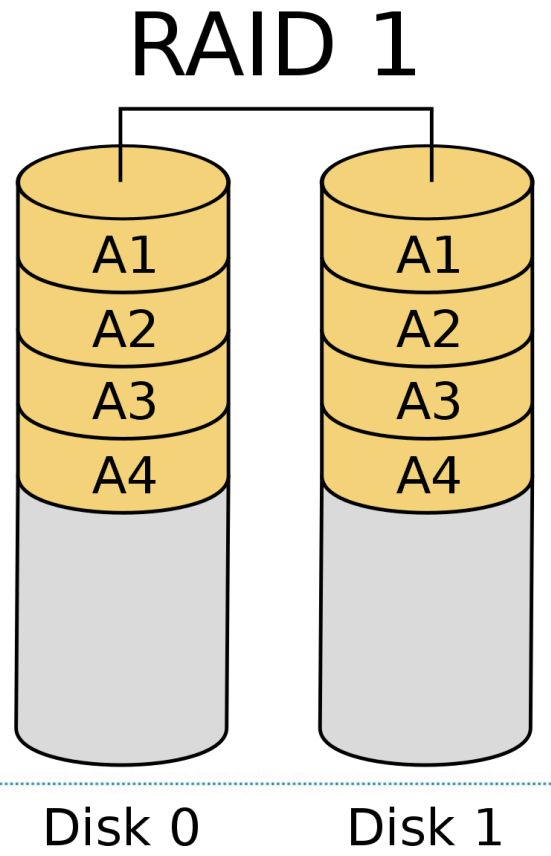
- Add as many HD as possible
  - 24, 36 depending on the hardware
- Data Striping
  - Files are stored as blocks (4MB on recent drives)
  - Spread blocks among disks
  - RAID 0
- Pros
  - User friendly, very fast
- Cons
  - Not reliable



https://en.wikipedia.org/wiki/Standard_RAID_levels

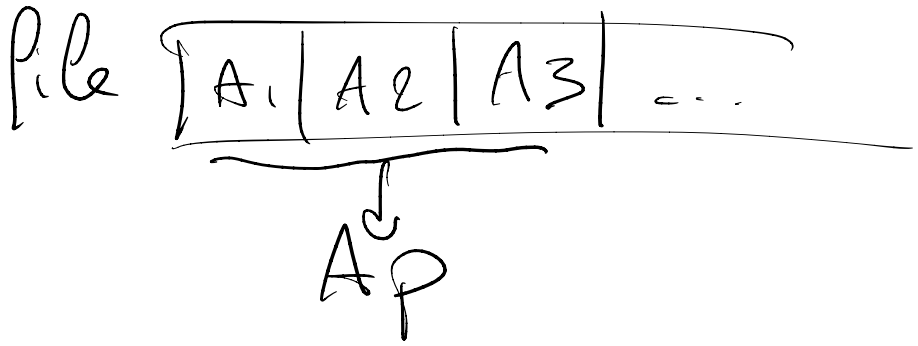
Single node

- Protection against hard drive failure
  - Use mirroring
  - Data is duplicated on another drive, or many
  - Data striping + redundancy
- Pros
  - Safe : only 1 drive needed for accessing data
- Cons
  - High cost

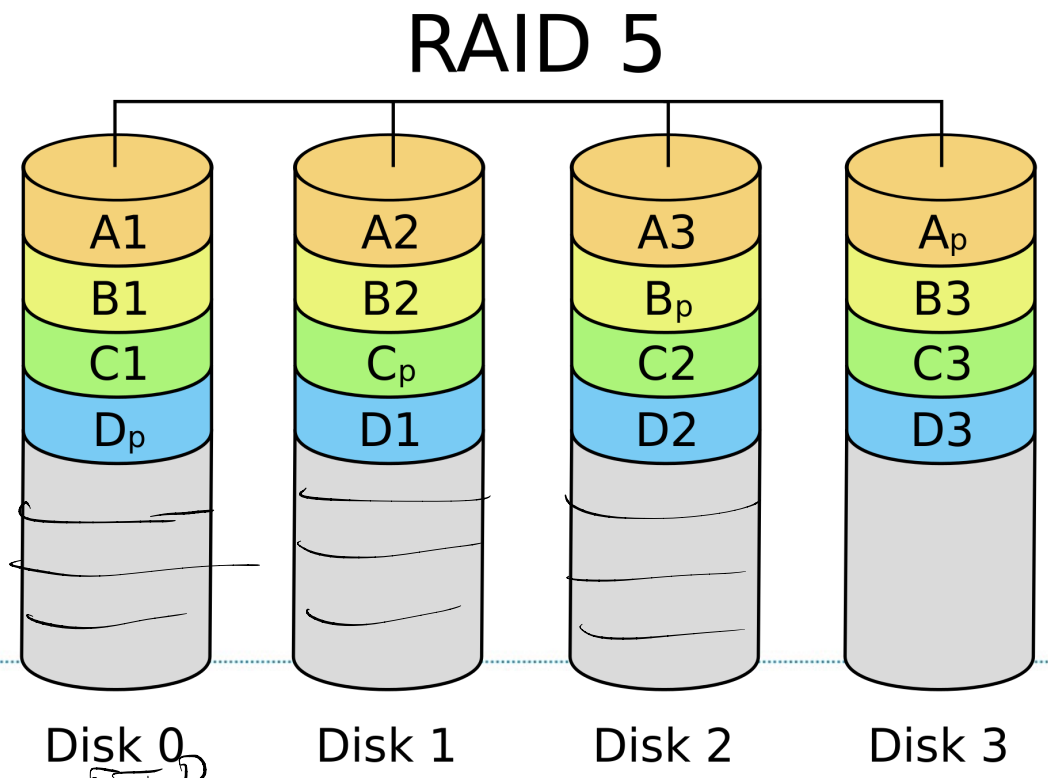


https://en.wikipedia.org/wiki/Standard_RAID_levels

Single node



- How to (better) protect against data loss?
  - Add redundancy information to rebuild missing blocks (Erasure Code)
  - *N* data blocks + *k* parity blocks
  - Data can be rebuilt with any *N* blocks
- RAID 5 or 6
  - 1 or 2 parity block
  - All blocks spread among all disks
    - Parity blocks not on the same disk
- Pros:
  - Increases reliability
- Cons
  - 1-2 disks for parity



UNIVERSITÉ CÔTE D'AZUR

overall ~ 3 TB storage
use full 5 TB

https://en.wikipedia.org/wiki/Standard_RAID_levels

RAID Failure Rates (Simple MTTDL Model)

(Disk failure only, 5yr MTBF, 20x 2TB drives, 50MB/s rebuild rate, Unlimited hotspares, 15TB stored on array)

	Year 1	Year 2	Year 3	Year 4	Year 5	Year 6	Year 7	Year 8	Year 9	Year 10
RAID 0	98.168%	99.966%	99.999%	100.000%	100.000%	100.000%	100.000%	100.000%	100.000%	100.000%
RAID 1	0.078%	0.156%	0.234%	0.311%	0.389%	0.466%	0.544%	0.622%	0.699%	0.776%
RAID 10	0.008%	0.016%	0.023%	0.031%	0.039%	0.047%	0.055%	0.062%	0.070%	0.078%
RAID 4	0.776%	1.547%	2.311%	3.069%	3.822%	4.568%	5.309%	6.044%	6.773%	7.497%
RAID 5	0.776%	1.547%	2.311%	3.069%	3.822%	4.568%	5.309%	6.044%	6.773%	7.497%
RAID 6	0.002%	0.003%	0.005%	0.006%	0.008%	0.010%	0.011%	0.013%	0.014%	0.016%
RAID-Z3	0.000%	0.000%	0.000%	0.000%	0.000%	0.000%	0.000%	0.000%	0.000%	0.000%
RAID 50	0.097%	0.195%	0.292%	0.389%	0.486%	0.583%	0.680%	0.776%	0.873%	0.969%
RAID 60	0.000%	0.000%	0.000%	0.000%	0.001%	0.001%	0.001%	0.001%	0.001%	0.001%

Raid Z3 : ZFS +3 parity drive

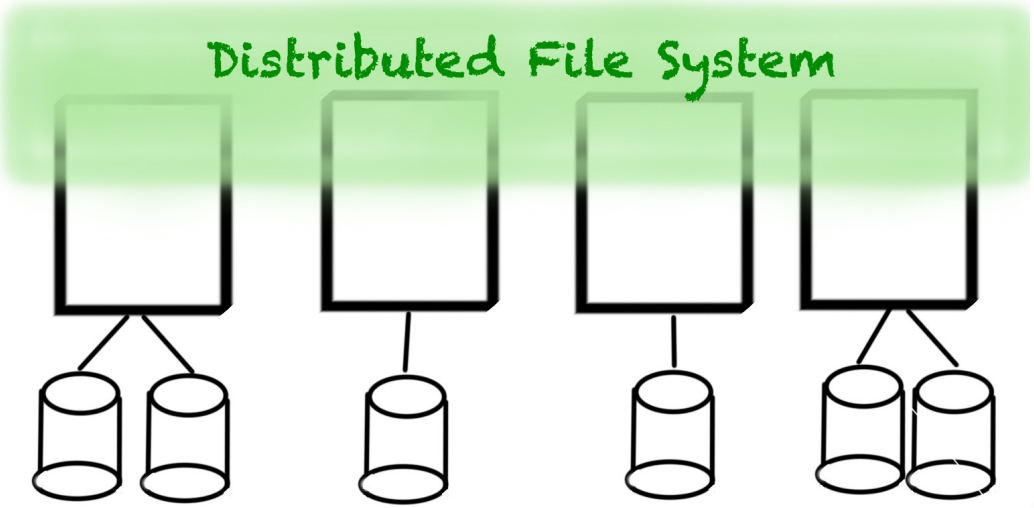
File systems

Distributed file systems

UNIVERSITÉ
CÔTE D'AZUR

Multiple nodes

- Single node
  - Can be expensive
  - Is a Single Point of Failure (network outage, fire…)
- Use multiple machines
  - Each machine can use RAID
  - Add a layer on top of it
- Examples : Ceph, GlusterFS, HDFS…



Storing files on multiple nodes

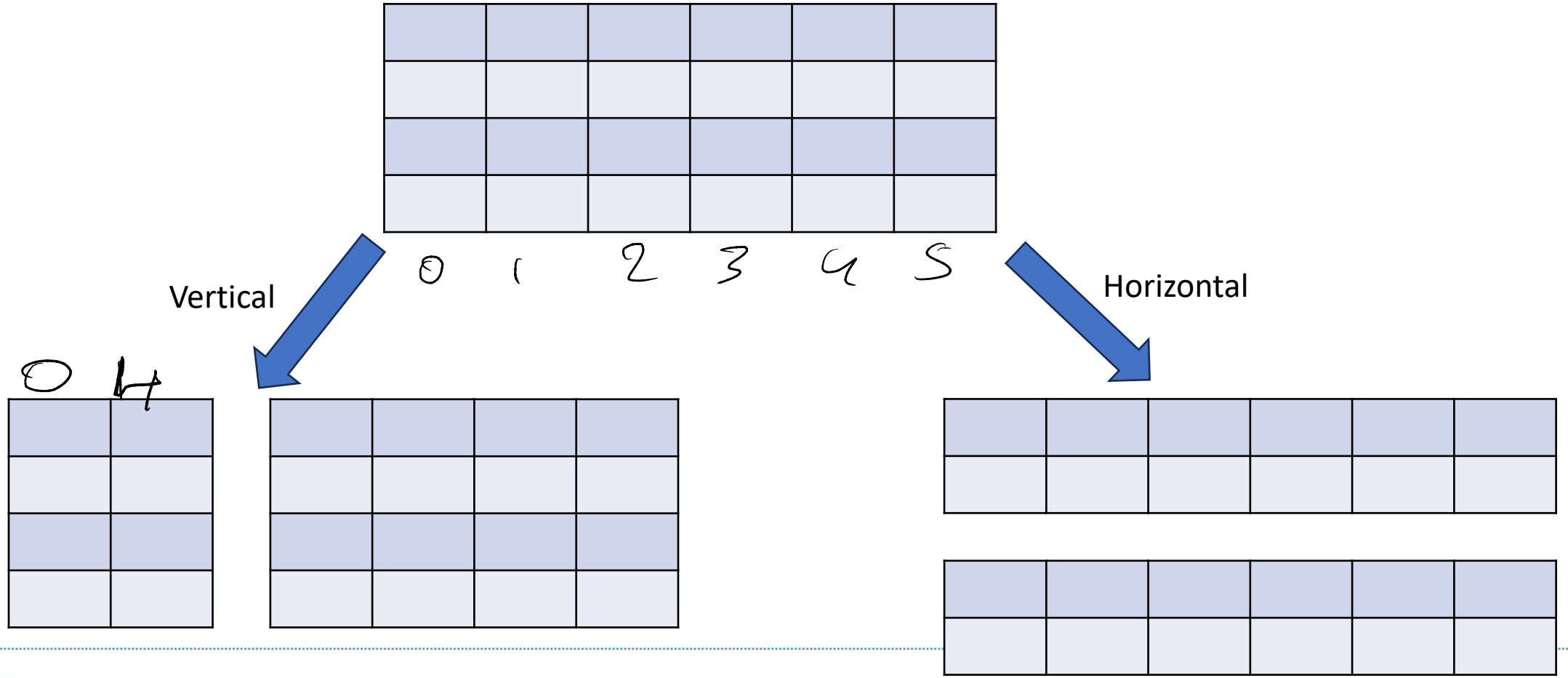
- How to find blocks ?
  - Cannot query all nodes
- Metadata
  - Added data by the OS or the filesystem
  - Date (creation, modification…), ownership…, location of blocks
- Usually “well known” nodes act as metadata servers
  - Extremely important nodes
  - Redundancy is mandatory for reliability

Partitioning data

Introduction

- Files are easy to distribute
  - Naturally stored as set of blocks
  - Just put the blocks on different machines
- What about more structured data ?
  - What is a good partition to distribute them ?
- Data partioning historically from DB
  - Horizontal vs Vertical partitioning

Horizontal vs Vertical Partitioning

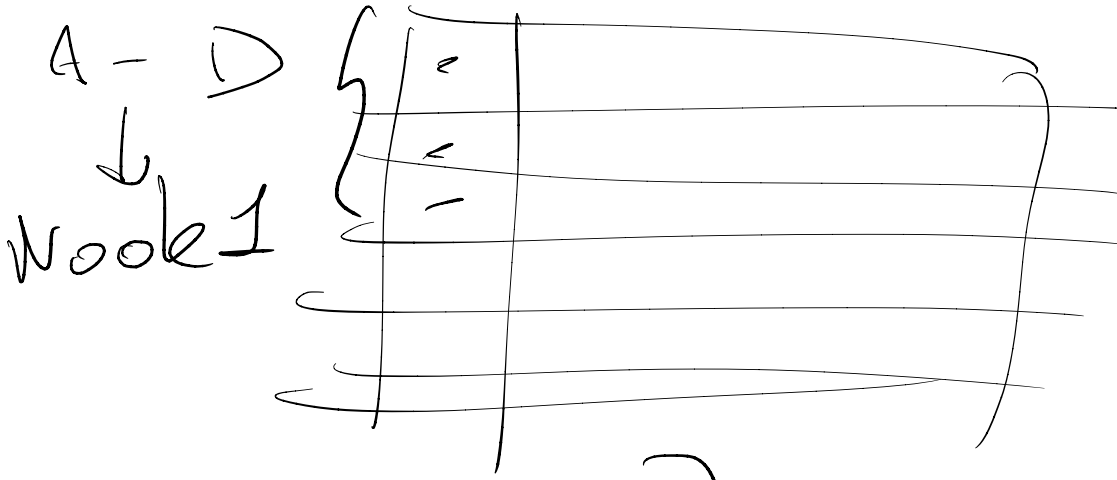


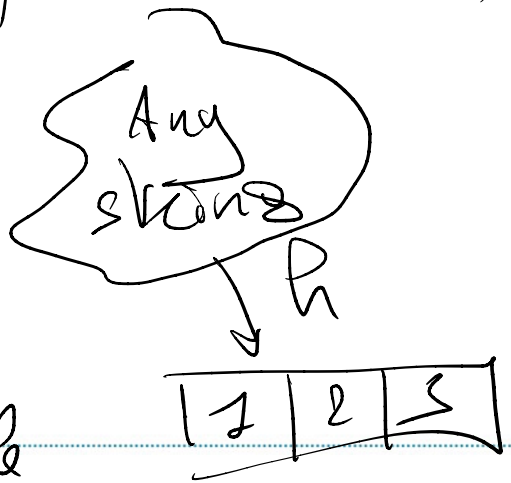
Vertical Partitioning

- Vertical partitioning
  - Split table into tables with fewer columns
- Strategies for grouping columns
  - Functional Dependency-based
    - Columns which functionally dependent (e.g columns for unit price, quantity, and total price)
  - Access Frequency-based
    - Columns that are accessed frequently together
  - Data Type-based
    - Based on the type of data in column
  - Update Frequency-based
    - Columns with frequent access are separated from others
  - Security based
    - Based on access control

Horizontal Partitioning

- Horizontal Partitioning
  - Aka Sharding
  - Split table into multiple smaller tables
- Strategies
  - Range based
    - Values of a specific attribute (e.g. alphabetical order
  - Hash partitioning
    - Hash a specific attribute (called key) and use value for partition
    - If *N* partitions, compute hash(*S*) mod *N*
  - Consistent hashing
    - Use hashing for both partition and nodes





nodes will be responsible for pair of hash space

d_1
d_2
d_3

on 4 machines

d_1 on $Node_0$

$h(d_1) \in [0 \; 1023]$	$h(d_1) \bmod 4 = 0 \in [0;3]$
$h(d_2) \in [0;1023]$	$h(d_2) \bmod 4 = 3 \Leftrightarrow Node_3$
$h(d_3) \in [0;1023]$	

what if we add another node

$\Rightarrow$ Recompute everything ?!

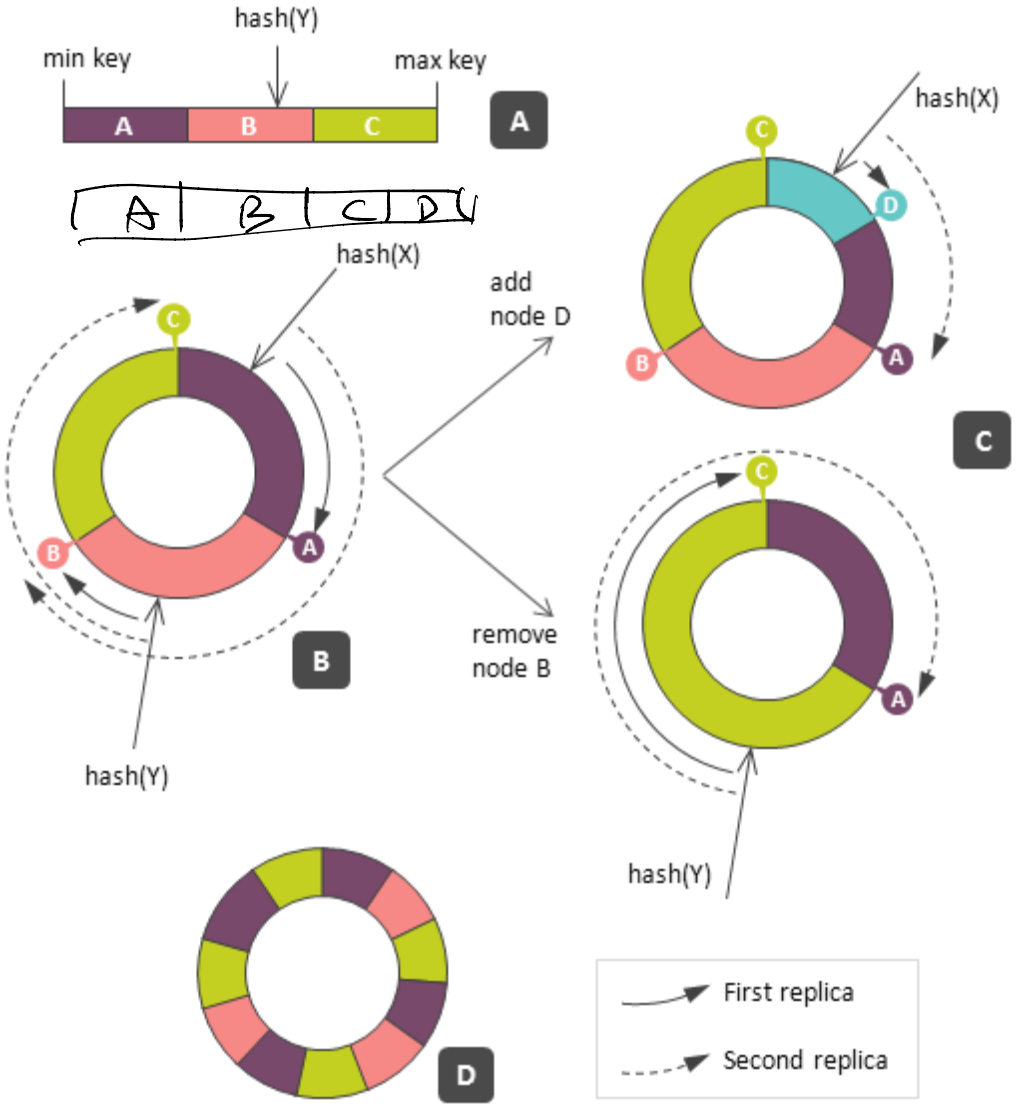
Consistent hashing

- Hash based partitioning has drawbacks
  - Each node gets an equal share of the hashed space
  - Hard to deal with skew data
  - Adding/removing nodes moves a lot of data
- Consistent hashing
  - Hash values space is divided among nodes
    - If hash values are [0-5000] with 5 nodes
      - 1st node gets [0-999], 2nd gets [1000-1999]….
    - Initially evenly distributed
  - Hash value of key determines partition
  - But how is that different from hash partitioning ????

Consistent hashing

Chord (2003?)

- Nodes are organized in a ring
  - Can query any node, which will forward the request
  - Can easily add/remove nodes
- Removing a node
  - The data and hash space are taken by its successor in the ring
- Adding a node
  - Takes part of the hash space of its successor
- Used in Amazon's Dynamo and Facebook's Cassandra (now Apache)



UNIVERSITÉ CÔTE D'AZUR

Hashing

- Advantages
  - No need to maintain mapping of Data->Node
    - Direct (hash partition) or routing (consistent hashing)
  - Help spread data uniformly
  - Can add nodes to deal with hotspots (consistent)

- Disadvantages
  - Adding/removing nodes is costly (hash partition)
  - Hard to do range queries ⟶ all values between x and y
    - Order preserving hash functions
      ↳ $d_1 < d_2 \iff h(d_1) < h(d_2)$

Replicating data

Rational

- Partitioning data has benefits
  - Scalability
  - Performance
- But what about fault tolerance ?
  - Partitioning increases the risk of failure
  - Need data to be available even if some node fails
- Replications stores the data in multiple nodes
  - Nodes are called replicas
  - Needs to be transparent to the end-user or developer
  - These copies must be kept in sync

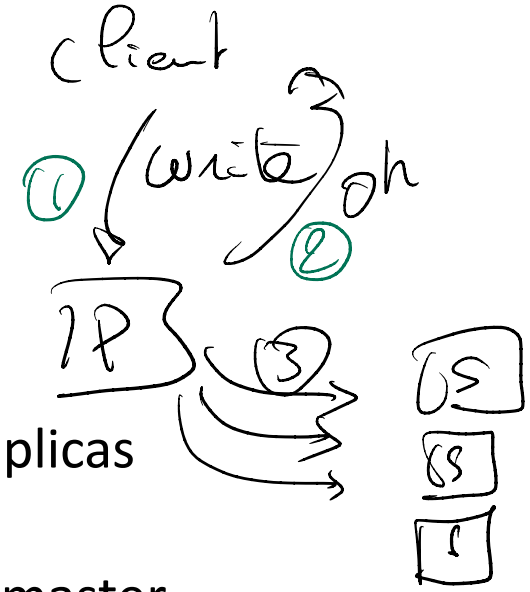
Rational - 2

- Read requests are simple/easy
- Write requests need care
  - Should updates all replicas
  - 2 strategies
- Pessimistic
  - All replicas are identical all the time from the start
- Optimistic
  - Replicas are allowed to diverge
  - They will converge again if the system does not receive any updates for some time

Single Master replication

- A node among replicas is designed *Master* (*primary*)
  - Receives all write requests
  - If crash, a new master can be elected (consensus)
- Other replicas are *slaves* (*secondaries*)
- Upon receiving an update, the master
  - Applied it locally
  - Propagate it to the *secondaries*
- Synchronous replication
  - Updates to secondaries are synchronous
  - Client is sure all replicas are updated (or not)
  - But slow write

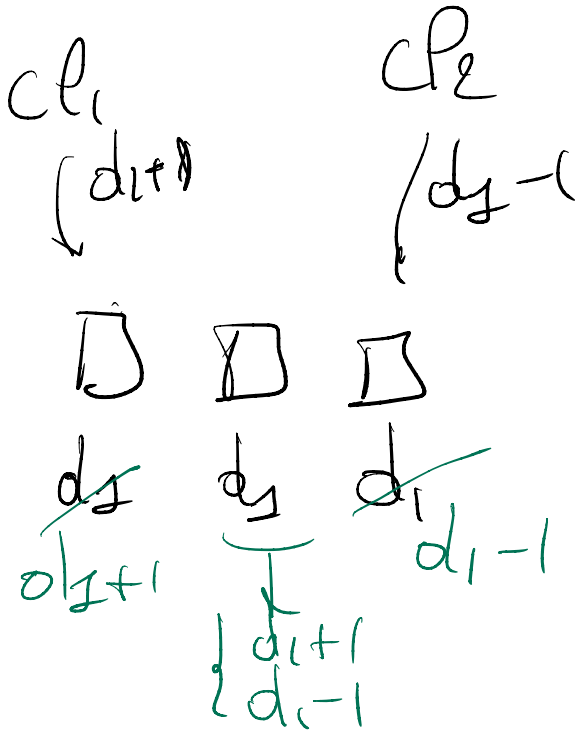
Single Master replication



- Asynchronous replication
  - Primary updates replicas asynchronously
  - Client does not wait for replicas
  - Fast write speed
  - But data might not be consistent across all replicas
- Pros of Single Master
  - Concurrent update requests are serialized at master
  - Reads can scale by adding replicas
- Cons of Single Master
  - No scaling for write
  - Electing new master can take time
  - Master's bandwidth can be saturated if too many replicas
    - Any solution to that ?

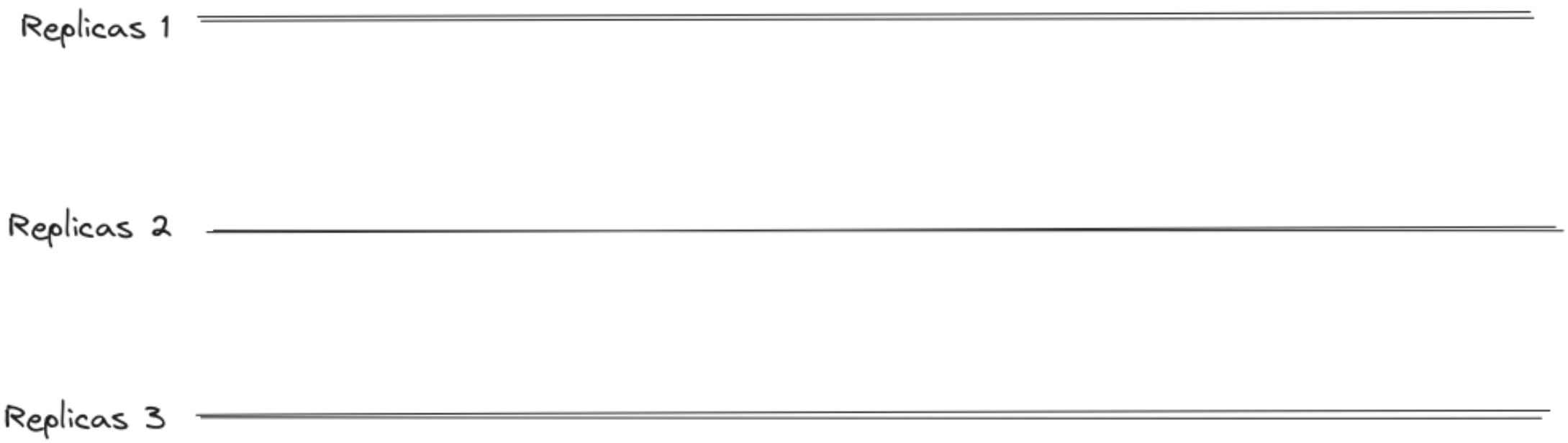
Multi-master replication

- All replicas are considered equals
  - Any can accept a write request
  - Higher availability but lower data consistency
- No serialization of write requests
  - Nodes might not agree on orders
  - Conflicts might happen
- Resolving conflicts
  - Let the client decide
  - last-write-wins : tag updates with **local** clock, on conflict takes the latest timestamp, but might lose causality
  - Causality tracking : track causally related writes to update them in order

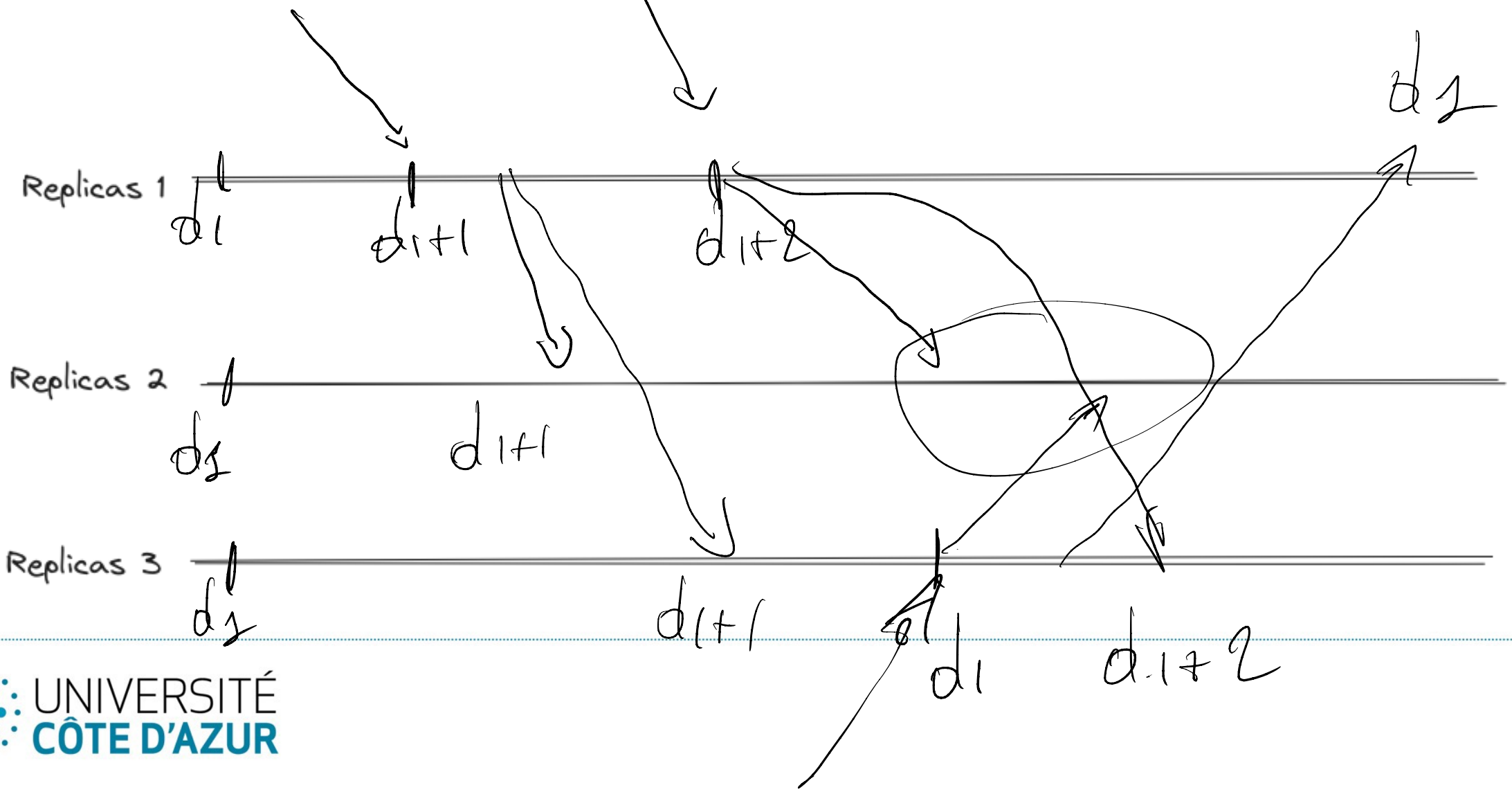


↳ Lamport clock → vector clock → version clock

Conflict free update



Conflicting update



NoSQL databases

Principles

UNIVERSITÉ CÔTE D'AZUR

RDBMS

- Lots of work/research for the past 40 years
- Mostly centralized model
- Different cost model than in the past
- Different paradigm than most programming languages
- Provide a lot of guarantees

ACID

- Andreas Reuter & Theo Härder in 1983.

- Atomicity
  - Transactions either fully succeed or fail
- Consistency
  - Transactions bring the database from a consistent state to a new consistent one
- Isolation
  - Concurrent transactions leave the DB in the same state as if they were executed sequentially
- Durability
  - Once a transaction has been committed, it will remain so even in case of failure

What now ?

- More data (like really more)
  - Not all well structured/organized
- Cheap hardware and not so cheap engineers
  - Many machines, 1 engineer
  - The network is everywhere
  - Machines or network **will** fail
- Is ACID possible in a distributed environment ?
  - Not really (CAP Theorem, Eric Brewer)

BASE

- ACID is very hard in a distributed environment
- Move to less strict guarantees
- **B**asically **A**vailable
  - Any request will be answered, even if the response is an error or incorrect data
- **S**oft-State
  - The state of the system can change in the absence of read or write
- **E**ventual **C**onsistency
  - The data might not be consistent, but will eventually

What is consistency

- A contract between a database and a programmer
  - Follow some rules and your data will be consistent
- Many different models
  - Strict, sequential, causal, eventual…
  - Ordered from strong to weak

Strict consistency

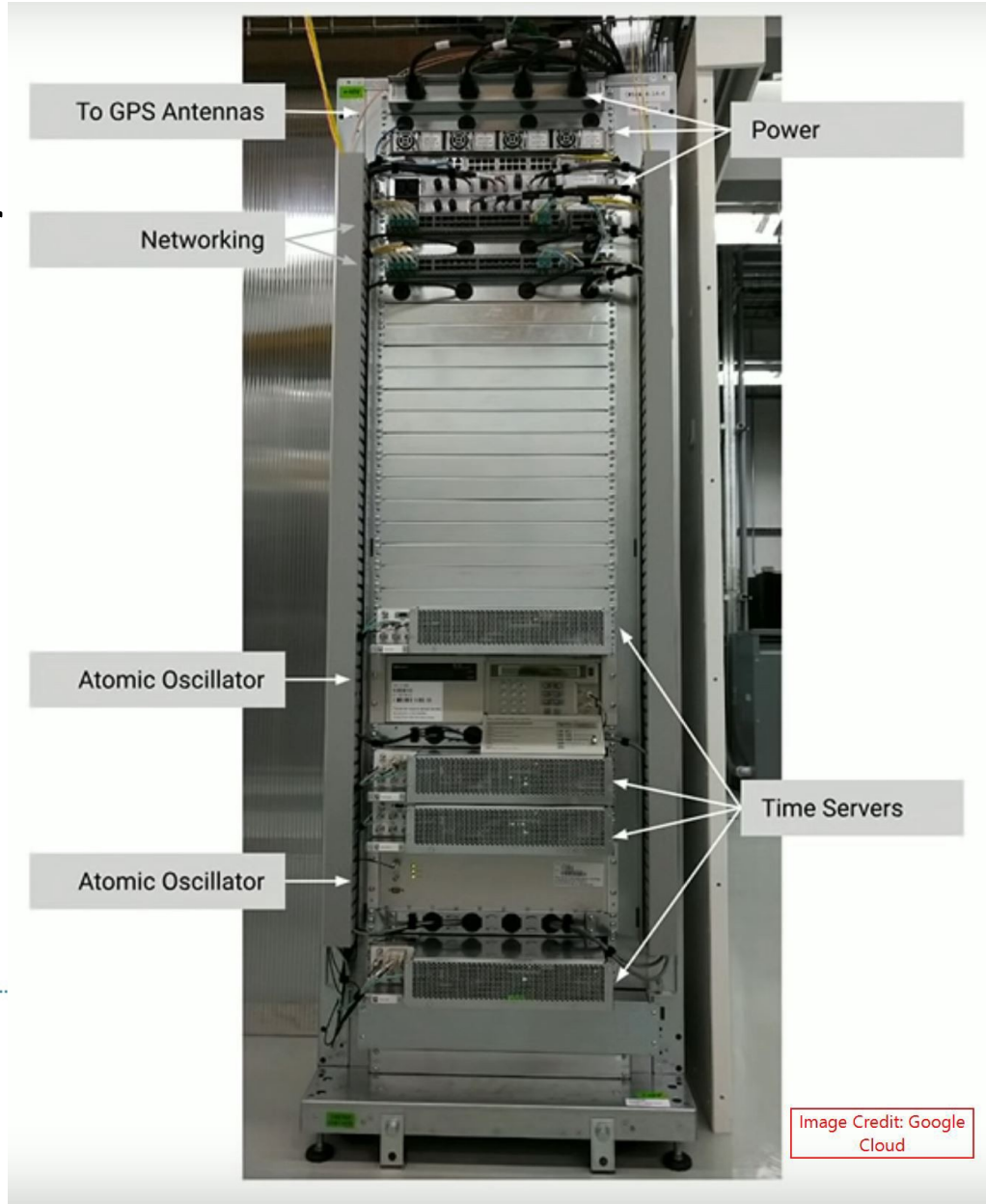
- A write to a variable is instantaneously seen by all processors

Sequence	Strict model		Non-strict model	
	P1	P2	P1	P2
1	$W(x)1$		$W(x)1$	
2		$R(x)1$		$R(x)0$
3				$R(x)1$

https://en.wikipedia.org/wiki/Consistency_model#Strict_consistency

Atomic Consistency

- Operations are executed in the same order on all machines
  - Uses a global clock
  - Same order as they were emitted
- Always deterministic
- But global clock are hard/impossible
  - Unless you have enough money
- Google TrueTime
  - GPS (main time) + atomic clocks (backup)
  - Time as an interval which includes bounded time uncertainty



UNIVERSITÉ CÔTE D'AZUR

Sequential Consistency

- Weaker than strict consistency
- All write operations by multiple nodes have to been seen in the same order
  - No specific order initially
  - Not necessarily consistent between various executions
- Sequential consistency + time => atomic consistency (e.g Google Spanner)

Eventual Consistency

- A form of weak consistency
- Updates can be performed in different orders on different machines
- Reads might not return the latest write
- BUT
  - Given enough time without writes
  - All read will eventually return the same value
- Reaching the same value is called reconciliation
  - Application dependent
- Example of eventual consistency
  - DNS

NoSQL databases

Families of databases

UNIVERSITÉ CÔTE D'AZUR

Principles

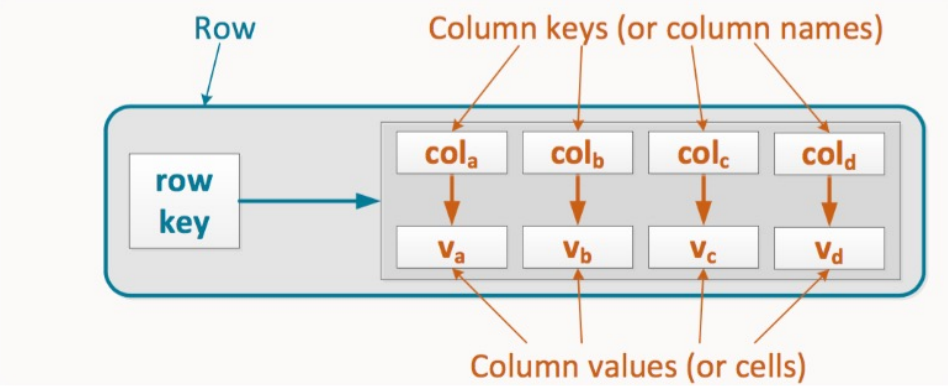
- Not Only SQL
- All follow the BASE principles
- Provides various properties under CAP
- Designed to scale horizontally
- Replication
  - Data is copied on multiples machines
- Various designs

Key-Value

- Data are stored as unique key-value pairs
- Very simple API
  - Get, put, delete
  - Range queries often not supported
- Usually relies on consistent hashing
  - Spread keys among multiples machines
  - Copy pairs for redundancy
- Examples : DynamoDB, Redis, Riak

Wide Column

- Use row/columns to store data
  - Like RDBMS except columns have usually no fixed type
  - Number of columns can vary from row to row
- Can be seen as a 2D key-value store
- Examples : Apache Hbase, Cassandra



https://bamdazonne.gitbooks.io/introduction-to-cassandra/understand_the_cassandra_data_model.html

Document

- Data are stored as documents (XML, JSON…)
  - Rich data structures
  - Support versioning
- An API allows complex queries

```
db.users.insertOne(                    ← collection
   {
      name: "sue",                     ← field: value
      age: 26,                         ← field: value   } document
      status: "pending"                ← field: value
   }
)
```

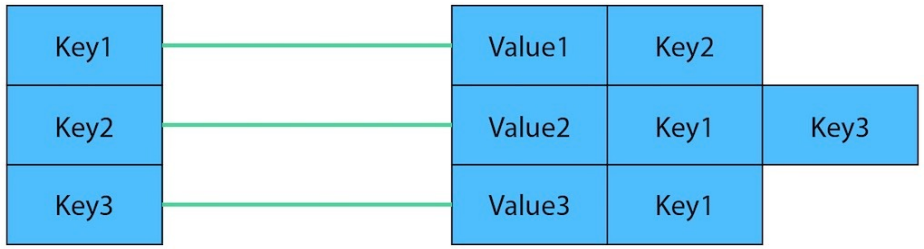
```
db.users.find(                         ← collection
   { age: { $gt: 18 } },               ← query criteria
   { name: 1, address: 1 }             ← projection
).limit(5)                             ← cursor modifier
```

- Examples : CouchDB, MongoDB

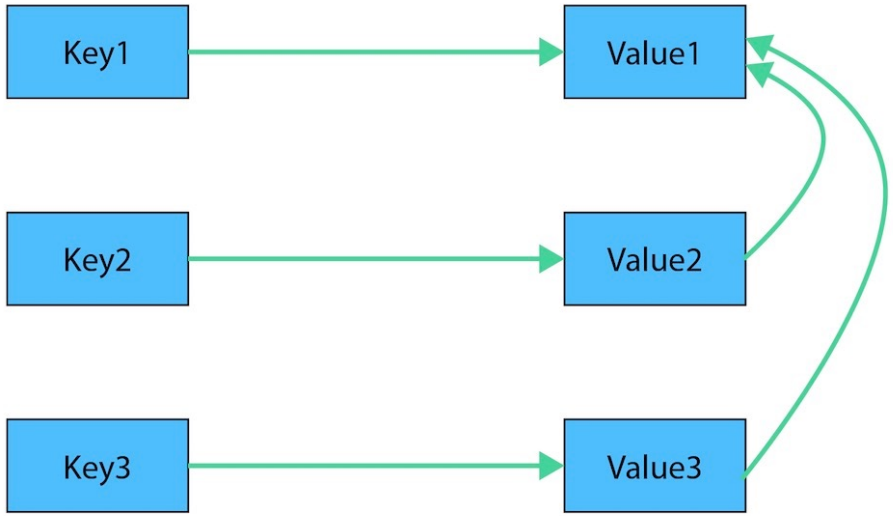
Graph Oriented

- Consider data as graphs
  - Introduce relations more complex than key-value

Key-value



Key-value as graph



- Examples : Neo4J, RedisGraph

https://neo4j.com/developer/graph-db-vs-nosql/