Storing Big Data

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

40 years ago : Hardware expensive
Today : cheap

↳ SQL vs rest

ACID

- Andreas Reuter & Theo Härder in 1983.
- Atomicity → A transaction will either succeed or fail completely
- Consistency → A transaction will move the DB from a valid state to another state
- Isolation → Many transactions can run in parallel and give the same result as a sequential execution.
- Durability → A crash should not lead to data loss

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 (Brewer)

BASE

- ACID is very hard in a distributed environment
- Move to less strict guarantees
- **B**asically **A**vailable → always get an answer, even outdated
- **S**oft-State → without new requests, the state of the DS can change
- **E**ventual **C**onsistency
  ↳ ?

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

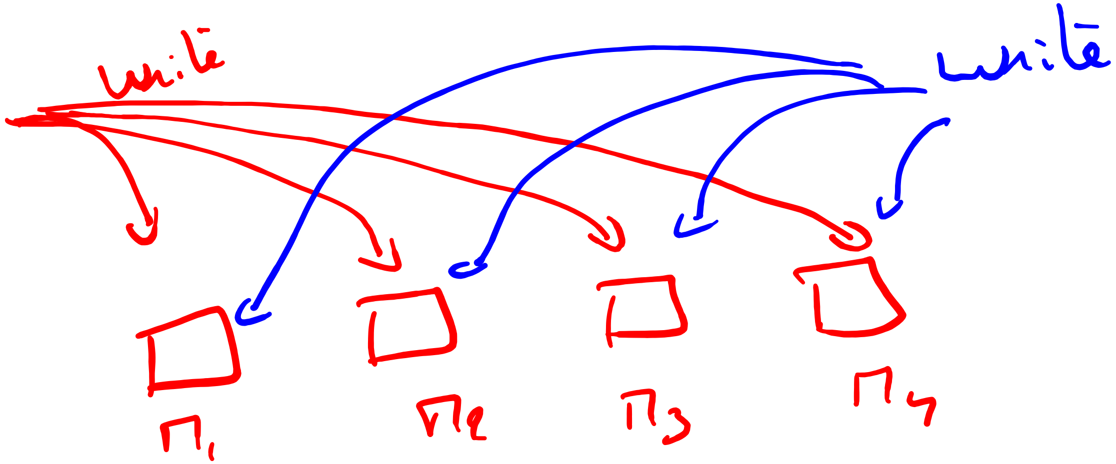
time

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

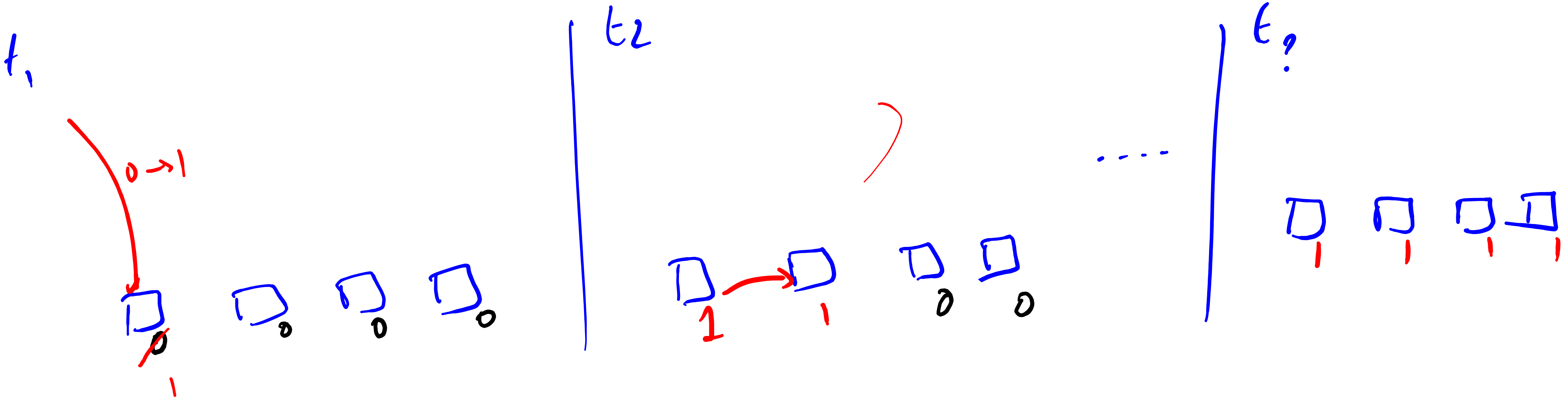


Sequential Consistency

- Weaker than strict consistency
- All write operations by multiple processors 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
  - Given enough time without update, all read access to a variable will return the latest value.



NoSQL databases

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

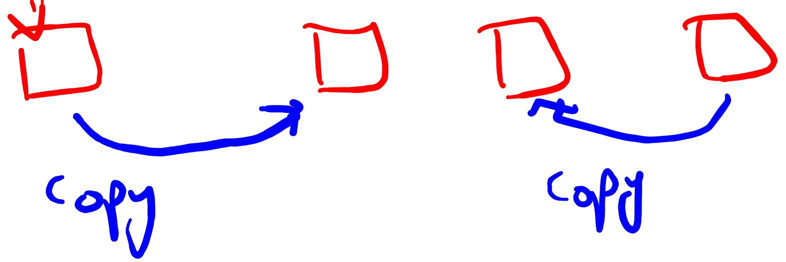
- Data are stored as unique key-value pairs
- Very simple API
  - Get, put, delete
  - Range queries often not supported
- Usually relies on consistent hashing
  - Spread keys among multiples machines
  - Copy pairs for redundancy
- Examples : DynamoDB, Redis, Riak

Keys are in an interval

Divide the key space into intervals. Assign an interval to a machine

put(key$_1$, v$_1$) ↓ h(key$_1$)

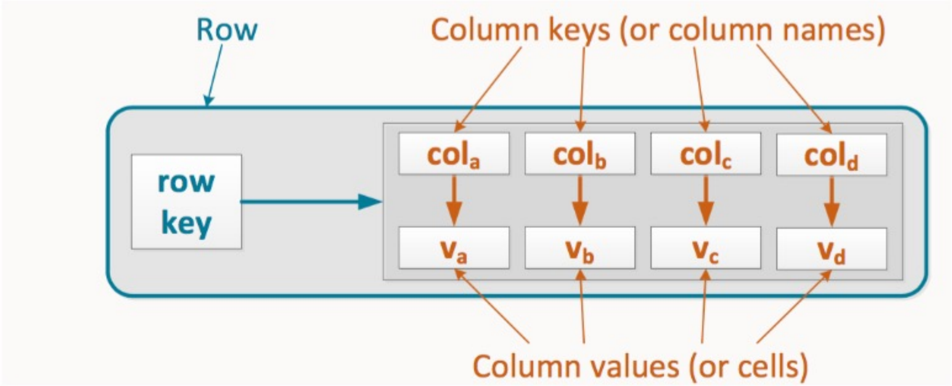
put(key$_2$, v$_2$) ↓ h(key$_2$)



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

  $(key, (key_1, v_1)(key_2, v_2))$
- Examples : Apache Hbase, Cassandra



https://pandaforme.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
  }
)
```

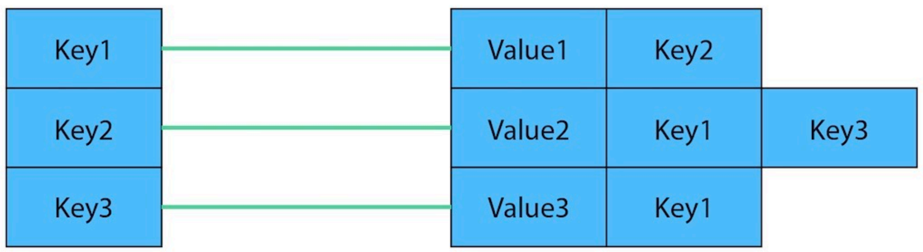
- Examples : CouchDB, MongoDB

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

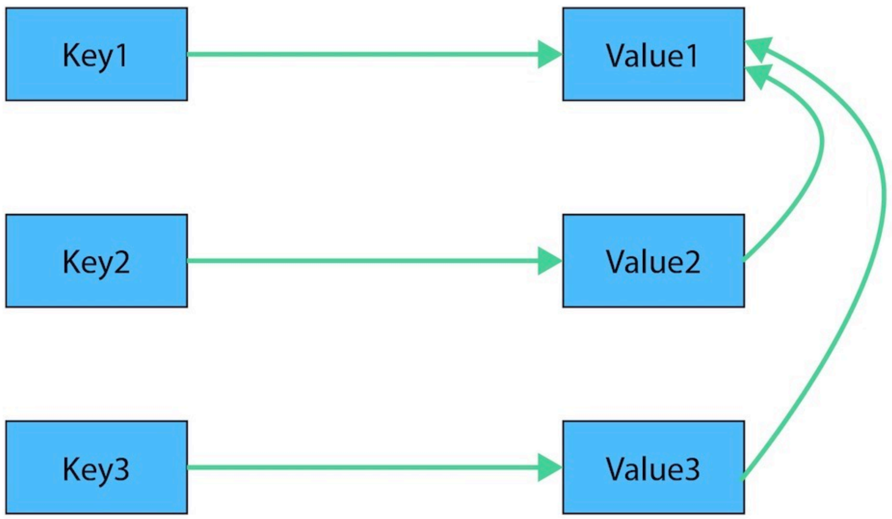
Graph Oriented

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

Key-value



Key-value as graph



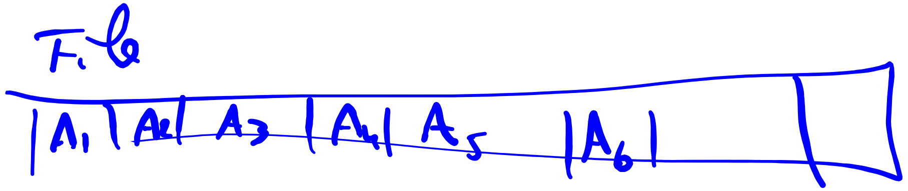
- Examples : Neo4J, RedisGraph

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

Distributed Storage

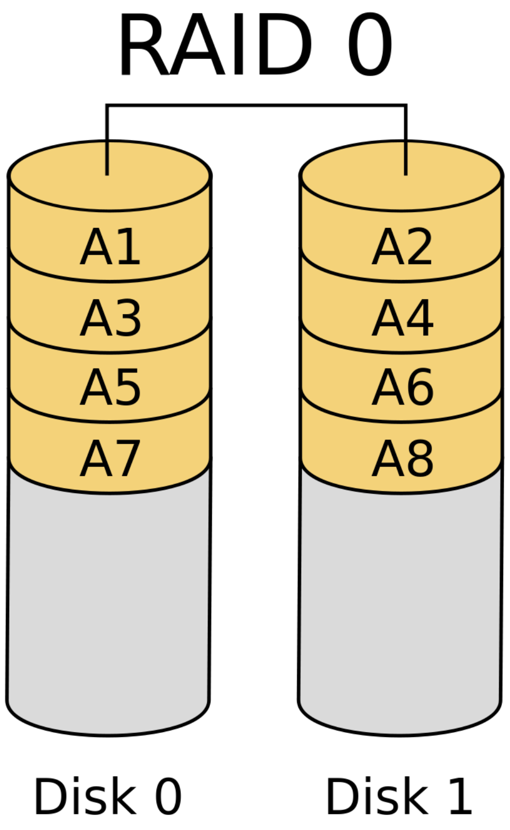
Storing without DB

- Not all files require a DB
- How to store arbitrary data?
  - Hard drive
- A single hard drive is limited
  - So use many!
- Single node or multiple nodes?



Single node

- Add as many HD as possible
  - 24, 36 depending on the hardware
- Users don't like managing independent HD
  - Aggregate them to give the illusion of a single drive
  - At the hardware or OS layer
- 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

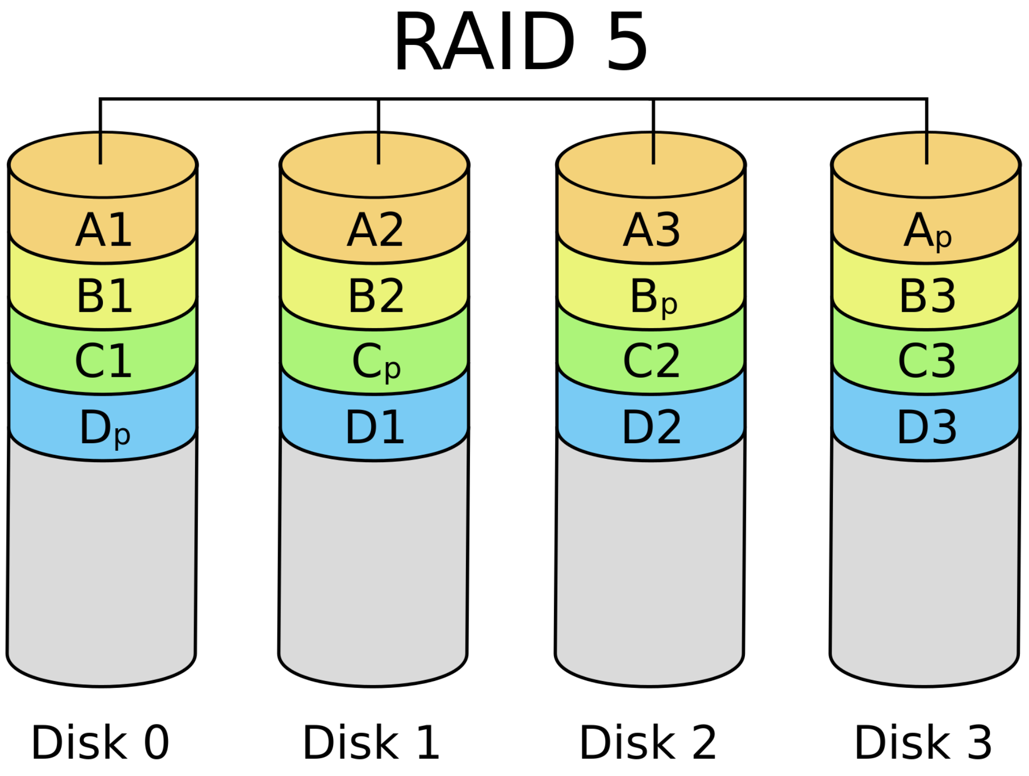
Single node

A

A_1	A_2	A_3	A_4

$A_p : f(A_1, A_2, A_3, A_4)$

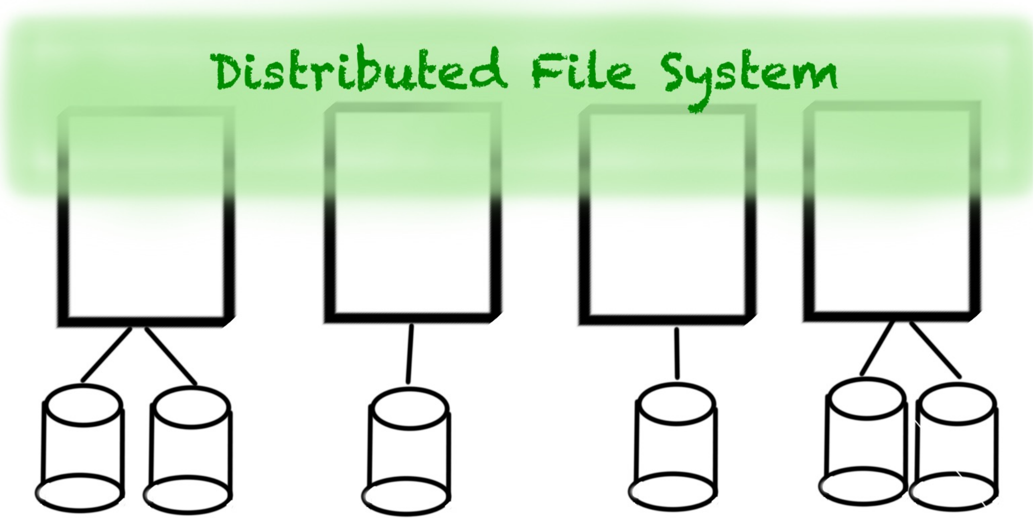
- How to protect against data loss?
  - Add redundancy information to rebuild missing blocks
  - Use Hamming Codes
- RAID 5 or 6
  - Group of blocks + 1 or 2 parity block
  - All blocks spread among all disks
- Pros:
  - User friendly, fast
  - Reliable
- Cons
  - Loss of disk space



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

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

data about data

- 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

Distributed Storage

Case study : The Hadoop File System

Principles

- Distributed filesystem over multiple nodes
- Provides a separate and global filesystem
  - Unix like paths
  - E.g. /home on HDFS is not /home of the machine
- Not part of the OS, added software
  - Written in Java, works on any machine with JVM support
  - Shipped with script for users and administrator

Commands

- All commands are done using bin/hdfs or bin/hadoop
  - hdfs <command> [options]
  - hadoop <command> [options]
- 3 types : client, admin and daemon
- Client commands : use the filesystem
  - *hdfs dfs <args>* or *hadoop fs <args>* : run the command args on the HDFS filesystem
  - Examples : -mkdir, -put, -copyfromlocal, -get, -copytolocal, -rmr

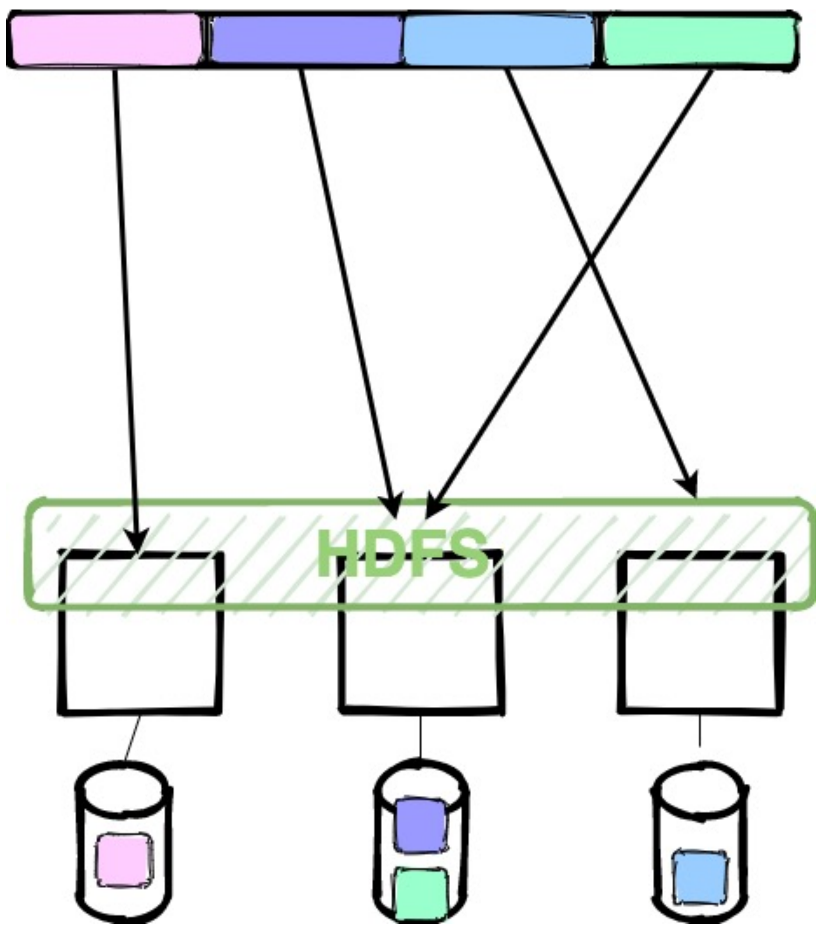
Commands

- Admin commands : manage the filesystem
  - hdfs fsck : check and repair the filesystem
- Daemon commands : manage the infrastructure
  - hdfs balancer : run the balancy utility
  - hdfs datanode : start a new datanode
  - hdfs namenode : start a new namenode

Nodes

Architecture

- Data and metadata are separated
  - Data are stored in DataNodes
  - Metadata are stored in NameNodes
- Data are divided into blocks
  - Default value 64MB or 128MB
- Blocks are stored on different machines

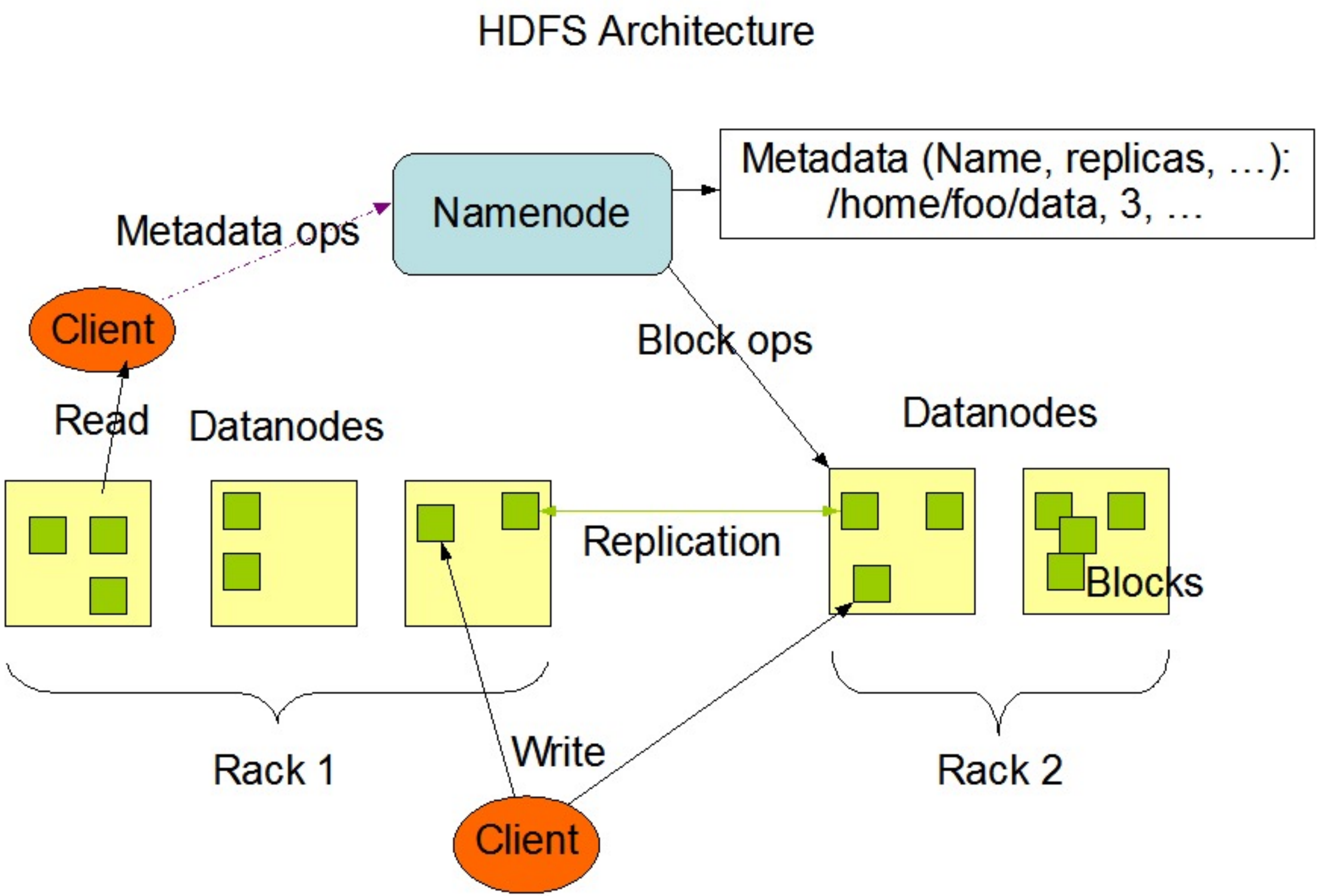


Architecture

- Data can be replicated
  - Replication factor (default 3)
  - More costly (space) than RAID or erasure codes
- Not all operations are supported
  - No random write, only append and truncate

write once
Read Many

- Permission
  - User and group for coarse grained control
    - Set access to owner or group of users
  - Access Control Lists (ACLs) for finer control
    - Set specific permission to specific user
    - Disabled by default



https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

Reading & writing

- All clients read and write requests go through NameNodes
  - Validation and metadata
- Actual data go directly to clients
  - Faster access
  - Blocks can be requested in parallel
- Reading workflow
  - Client send request for file “/test.txt”
  - NameNode checks access rights and returns list of blocks+DataNodes

Reading & writing

- Writing workflow
  - Client contact NameNode
  - If writing allowed, check if file exists
    - If yes, error
  - NameNode returns a list of DataNode
  - Client sends data to DataNodes in round-robin
  - After writing all blocks, the client notifies the NameNode
- Replication is handled by DataNodes while receiving data

HDFS Blocks

- Large blocks
  - Not suited for storing small files
  - Limit overhead of metadata
- Transfer cost of a block
  - Disk access + latency + throughput
  - Large blocks minimize impact of disk access and latency
- Replication for
  - Fault tolerance
    - Tries to put replica on different machine and different racks
  - Faster access
    - Load blocks from node closer to client

Network

Fault tolerance

- NameNode is a single point of failure
  - If down, cannot access data anymore
  - If destroyed (metadata lost), data are lost
- Possibility to use a Secondary NameNode
  - Maintains snapshot of metadata + edit log
  - Periodically apply edit log to metadata and store new state
- In case of crash
  - Restart NameNode
  - Get snapshop + log of Secondary NameNode and rebuilt recent state
- But
  - Rebuilding might be long
  - Might still lose some metadata

Fault tolerance

- *High Availability*
  - Feature introduced in Hadoop 2
- Support 2 NameNode
  - 1 active and 1 passive in standby
  - If active fails, standby takes its place
- How to ensure consistency?
  - Relies on JournalNode
  - Active write edit log to JournalNode
  - Passive regularly get edit log from JournalNode
- Protection against failures of JournalNode
  - Usually use 3 of them but N possible
  - Agreement based on a Quorum algorithm
  - Can tolerate (N-1)/2 failures