

# Unix Command Line Interface (CLI)

分享

operators

按下 Esc 即可結束全螢幕模式

- > > overwrites the file if it exists or creates it if it doesn't exist
- > >> appends to a file or creates the file if it doesn't exist



The last thing for  
us to be mentioned is bash operators.

As you have already know  
a lot of handy tools for

## More operators!



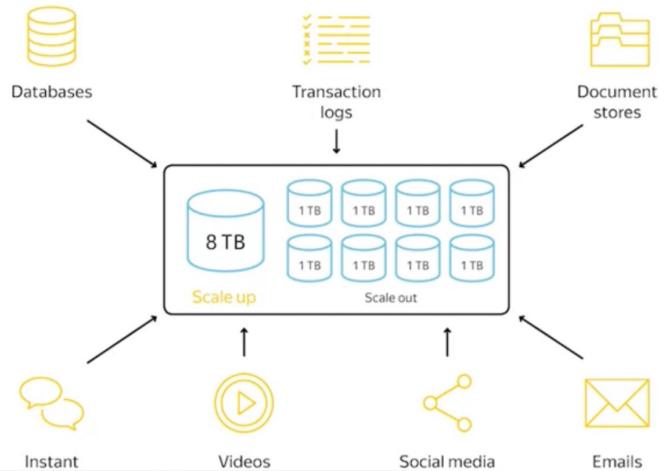
- > & make the command run in background
- > || execute the second command only, if the execution of first command failed
- > && execute second command only if the execution of first command succeeded

You have done a really good  
job with this video and

\* free : 顯示 memory 使用資源

# Distributed File Systems (DFS), HDFS (Hadoop DFS) Architecture and Scalability Problem

- Scaling distributed File System



The first approach is called scale up,  
or vertical scaling.

scale up / Vertical Scaling : 僅存在同一個 node

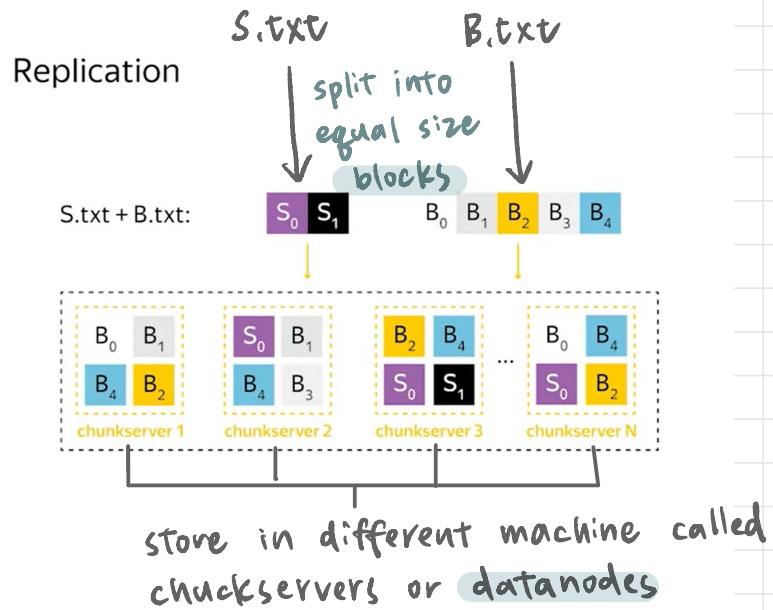
- low latency
- hard to scale

Scale out / Horizontal Scaling : 分開存在不同 node

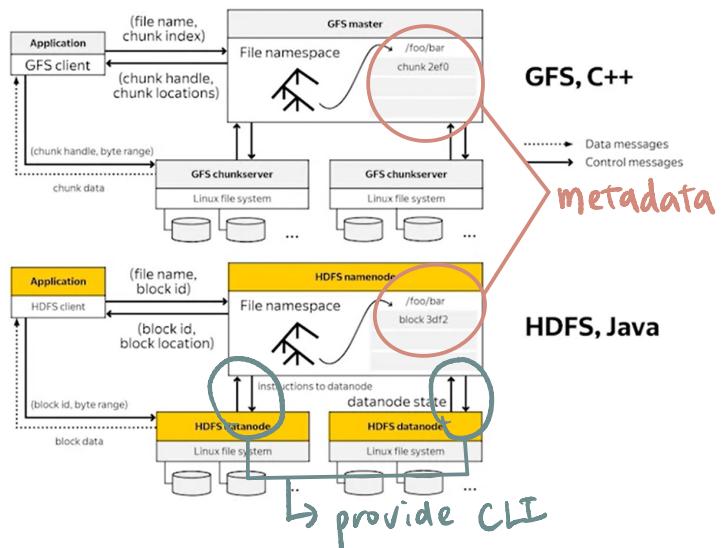
- high latency
- easy to scale

# \* Google File system Components

- components failure are a norm (use replication)
- even space utilisation
- write - once - read - many



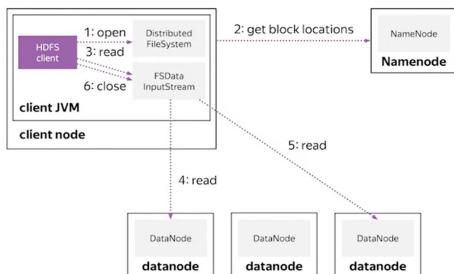
# \*RPC protocol to access data via HTTP protocol



## metadata

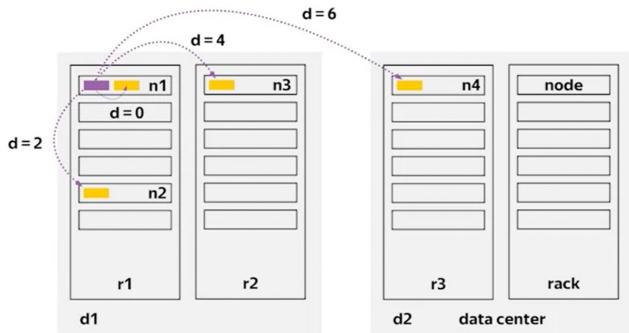
- store in master node with low latency in memory
- contain :
  - administration info.
  - creation time
  - access properties

## Read file from HDFS



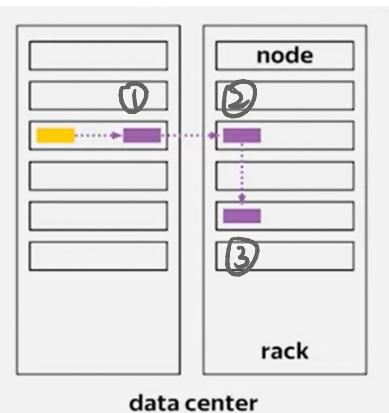
\*get data from  
the closest machine

# Data Center topology



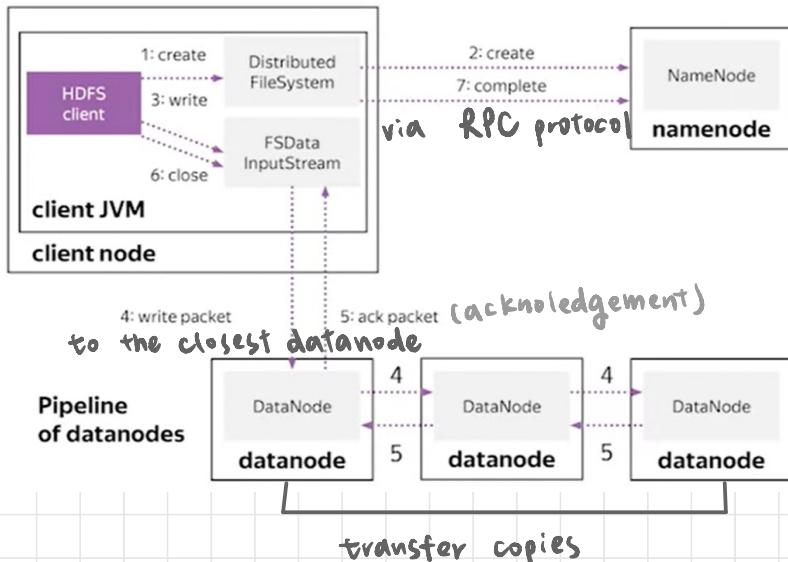
- $d=0$ : same machine use data locality without any extra RPC
- $d=2$ : same rack
- $d=4$ : different rack
- $d=6$ : different data center

\* Write file into HDFS : Redundancy Model

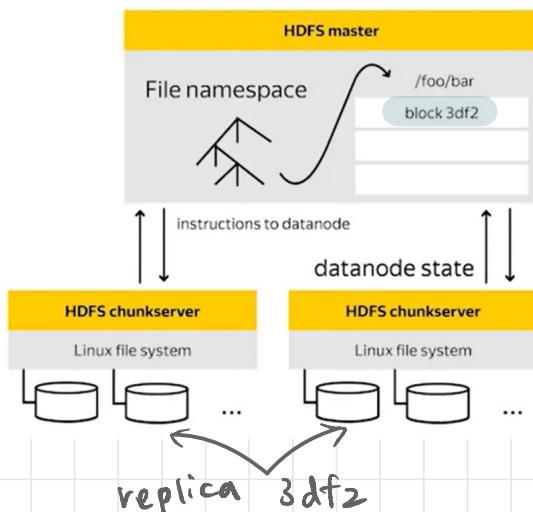


- ① first replica is usually located on the same machine
  - ② second is placed in a different rack.
  - ③ third is placed in different machine, same rack as ②
- ④ further placed random, system tries to avoid placing too many replicas on same rack.

## \* Data flow of Write Application



## • Block and Replica States

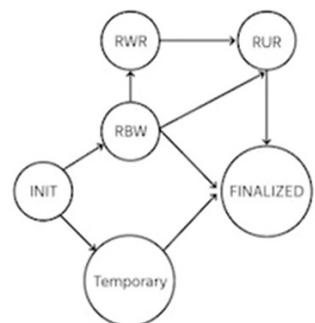
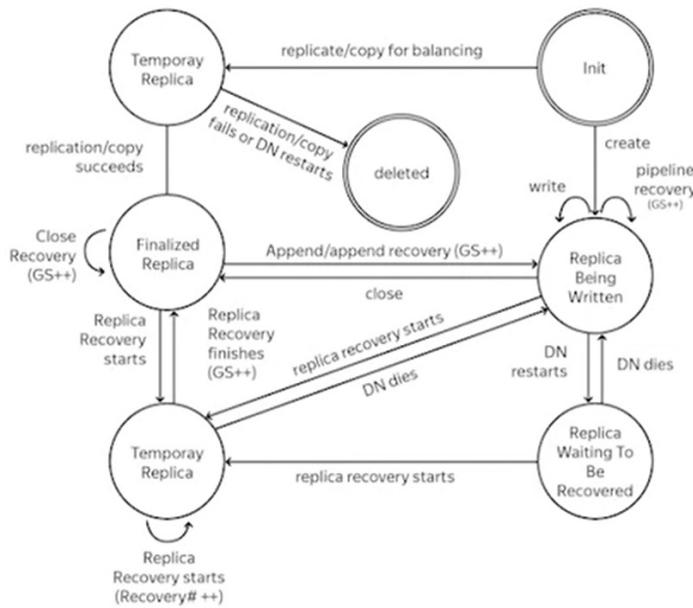


**Replica:** physical storage on datanode  
**Block:** meta-information storage on namenode provides info. about replica's locations and their states

## \* Generation Stamp (GS)

- each block of data has a version number called GS
- only increase over time, happens during error recovery or appending to a block.

# \*Datanode Replica's States



Simplified Replica State Transition

## Finalized State

- content is frozen
- meta-info for this block on namenode is aligned with all replica (read consistency)
- guarantee all replicas have same GS number

## Replica Being Written to (RBW)

- the state of the last block of an open or reopening file
- different datanode can return to use a different set of bytes
- bytes that are acknowledge by the downstream datanode in a pipeline are visible for a reader for this replica.
- datanode and namenode meta-info may not match during this state.

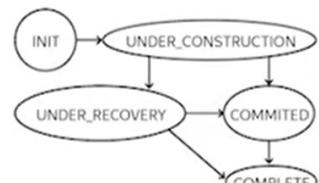
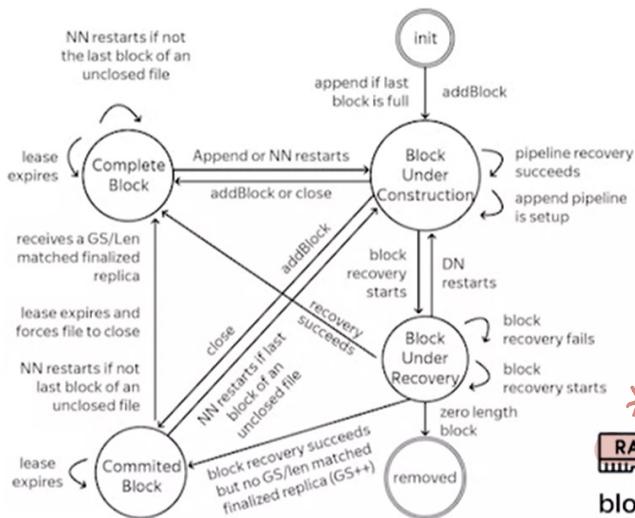
# Replica Under Recovery (RUR)

- HDFS client lease expiration, usually happen during **client's site failure**

## Temporary

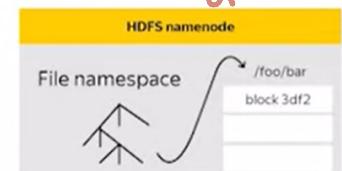
- happens during **data unevenly distributed over the clusters**
- Hadoop admin. can spawn a process of data re-balancing
- Data engineer can request increasing the replication factor
- pretty much same as RBW except this data is **not visible to user unless finalized**

## \* Namenode Block's States



Simplified Block State Transition

\* Store in memory



## under-construction

- when a user opens a file for writing, appending.
- always the last block of file.
- length and GS are mutable.
- contain info. about all RBW, RWR replicas.

## under-recovery

- when replicas transition from **RWR** → **RUR**
- when **client dies**, block from under-construction → under-recovery

## committed

- when client successfully request namenode.
- means already some finalized replicas but not all of them.
- in order to serve a read request, committed block **keep track of RBW replicas**.

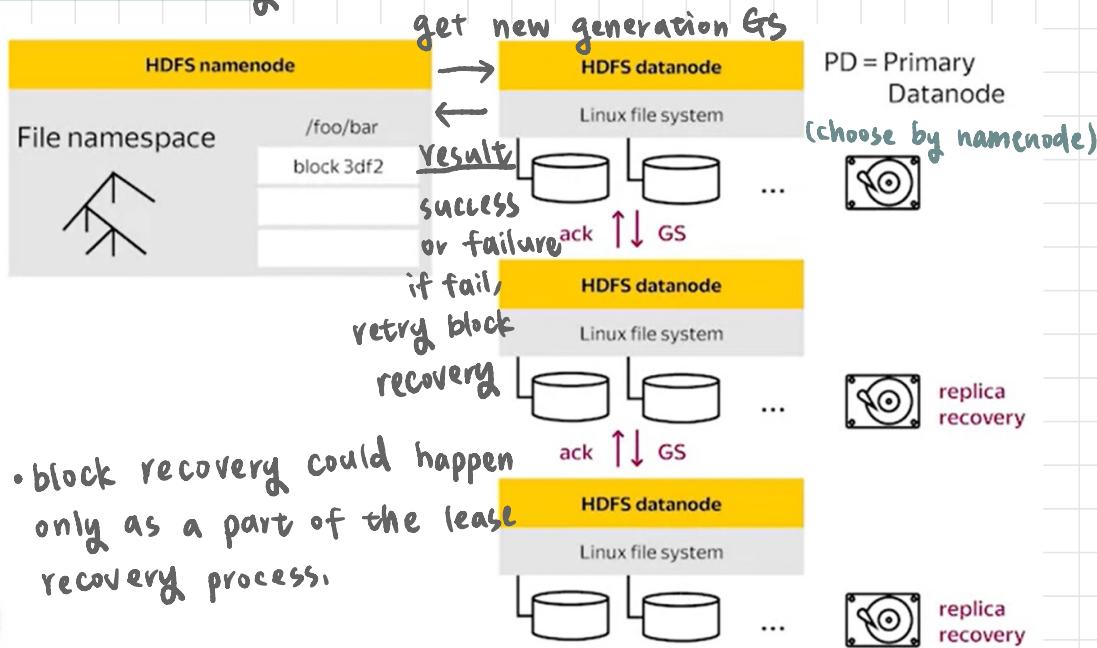
## complete

- all replicas are in the **finalized state**.
- have **identical** visible length and GS.

## \* Recovery Process

- namenode ensure corresponding replicas will transition to common state logically and physically (have the same on disk)

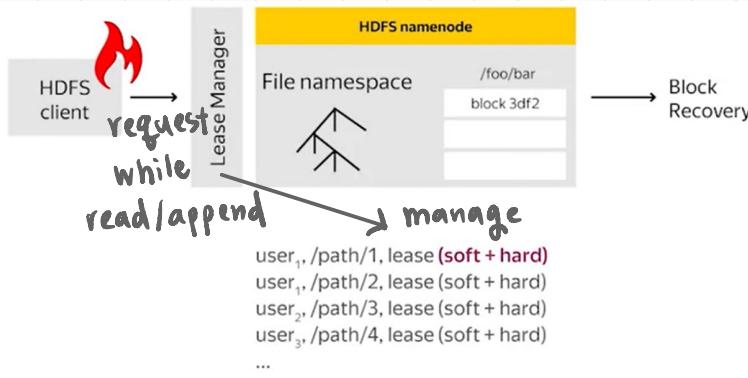
# Block Recovery

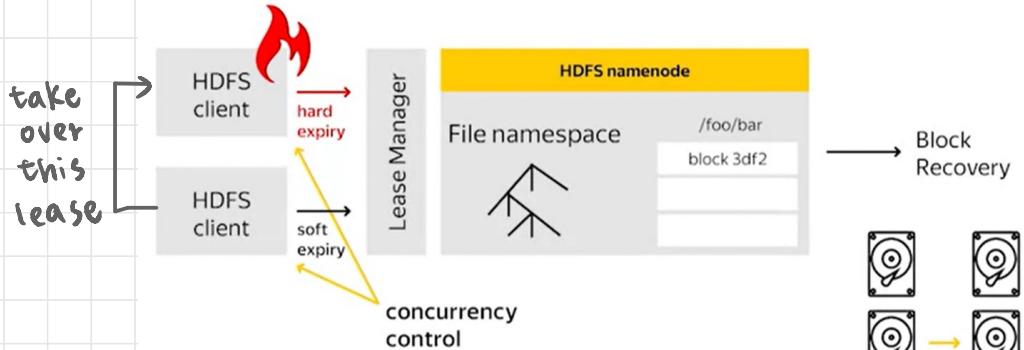


# Replica Recovery

- aborting active clients right into replica.
- aborting previous replica.
- participating in final replica size agreement process.

# Lease Recovery





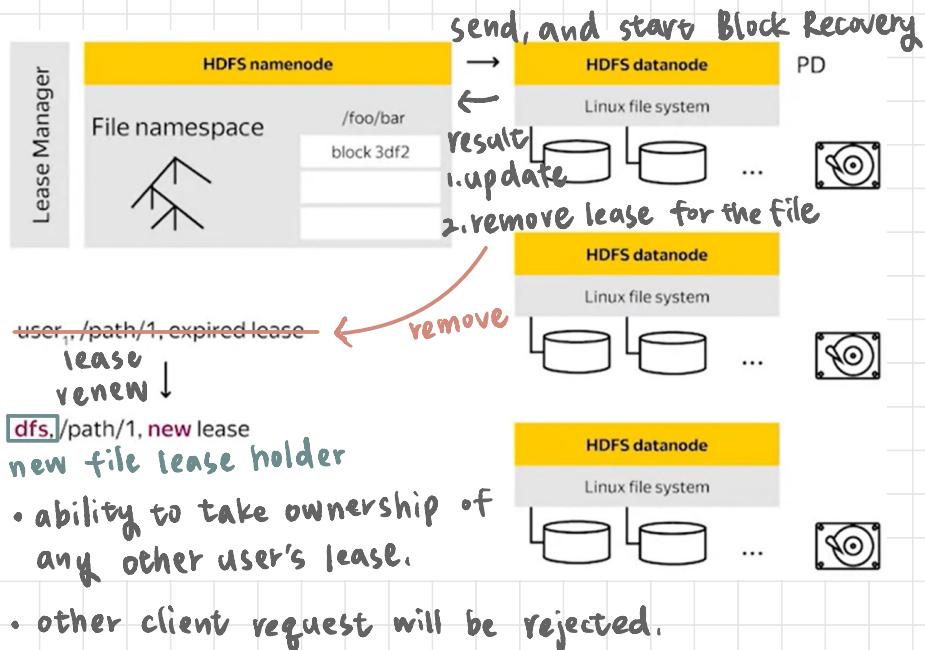
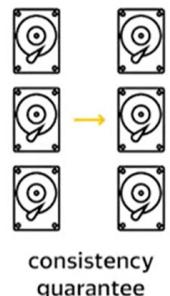
- close open file for the sake of the client.
- guarantee:

### 1. Concurrency control

even if client is still alive, it won't be able to write data.

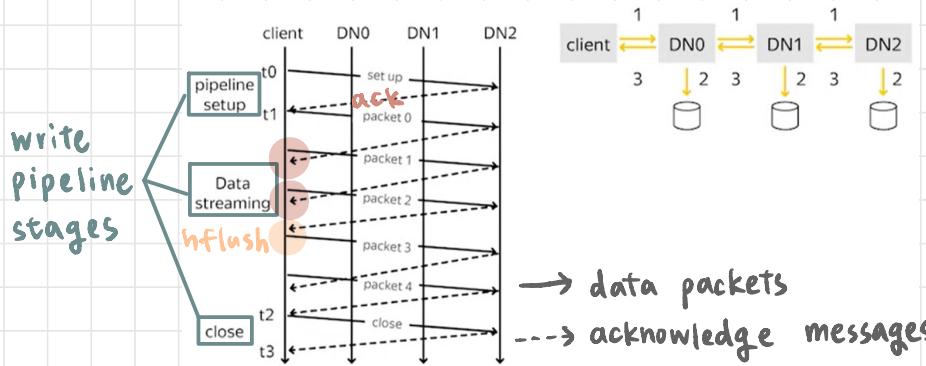
### 2. Consistency guarantee

all replicas should draw back to a consistent state, same on-disk data, GS



# \* Pipeline

- HDFS client writes data block by block.
- block constructed through a pipeline,
- HDFS client breaks down block into packets.
- packets are propagated to datanodes through pipeline.



## 1. pipeline setup stage

client sends setup message to pipeline. Datanode open replica for writing and send ack. message back upstream.

## 2. Data streaming Stage ( $t_1 \sim t_2$ )

$t_1$ : client receive ack. for top stage.

$t_2$ : client receive ack. for all packets.

data is buffered on client site to form packet, then propagated through pipeline.

●: next packet can be sent before the ack of the previous pack.

● **hflush**: Synchronous packet, used as synchronization point for datanode write.

## 3. Close

finalized replicas, shut down pipeline, all datanodes change state to finalized, report state to namenode and send ack.

## Pipeline Recovery

- Failure happens during pipeline setup stage
  - cases: new file / append mode
  - can easily abandon datanode pipeline, request a new one.

## Failure in Data Streaming

- datanode isn't able to process packets.
- allow pipeline about it, but closing the connections, stop sending new packets existing pipeline, request new GS from namenode, rebuild a pipeline from good datanodes.

## Failure in Close

- rebuild pipeline with good datanodes, bumps GS and request to finalized replicas.

## \* Namenode Architecture

- capacity of datanode and namenode

Suppose have 10 PB data

### datanode

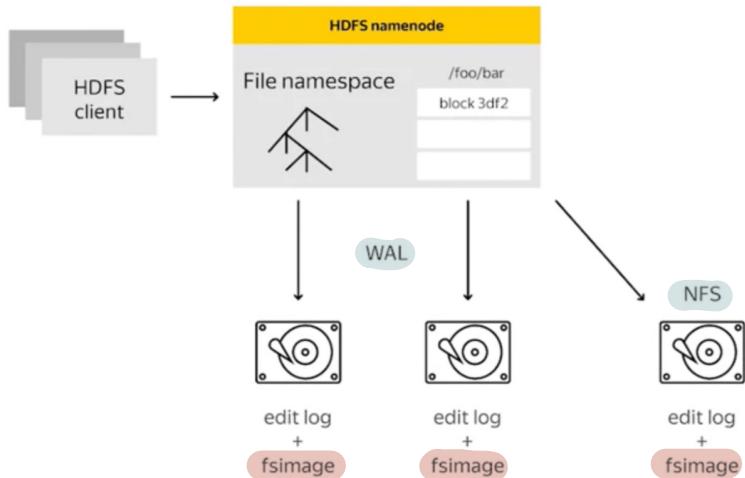
Suppose use 2TB hard drive, replication factor = 3, we need  
 $10\text{PB} / 2\text{TB} * 3 \approx 15\text{K}$  hard drives.

### namenode

Suppose block size = 128 MB, average block size on namenode = 150 B, we need

$10\text{PB} / (128\text{MB} * 3) * 150\text{B} \approx 3.9\text{GB}$  memory to store metadata.

- **Namenode Durability and Speed up Namenode Recovery Process**



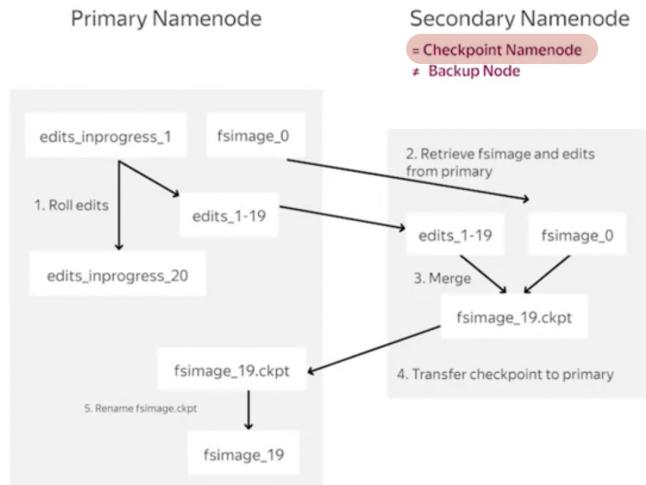
- **WAL (Write-ahead log)**

**persist changes into the storage before applying them**

**fsimage**: snapshot of memory at some point in time from which you can replay transaction stored in edit log.

- **NFS (network file system)**

**overcome node crashes by restore changes from remote storages.**



# Tuning Distributed Storage Platform with File Types.

## \* File Formats

- many file formats
- differ in:
  - space efficiency (determine storage cost)
  - encoding & decoding speed
  - supported data types
  - splittable / monolithic structure (whether can extract subset of data)
  - extensibility (tolerance of schema change)

## \* Text format

- Drawback: you need to parse it, convert it from textual form into programmatic data structure.

### 1. CSV & TSV (comma- & tab-separated values)

- space efficiency: **BAD**, ① contains a single repetitive value
- Speed: **GOOD**
- Data types: **only strings**
- Splittable: Splittable w/o header
- Extensibility: **BAD**.  
not easy to remove or  
reorder fields

①	②
Tickers	Date,Open,HIGH,Low,CLOSE,Adj Close,Volume
"IXIC"	01-01-2010,14160.029785,14160.029785,14131.790039,14131.790039,14131.029824,14143.029824,1738800000
"IXIC"	01-01-2010,14160.029785,14160.029785,14131.790039,14131.790039,14131.029824,14143.029824,1667480000
"IXIC"	01-01-01,014137.029785,0139.779785,0139.779785,0139.779785,0139.779785,0139.779785,2292840000
"IXIC"	01-01-07,01428.569824,0158.180176,0126.479980,0153.180176,0153.180176,0153.180176,2278220000
"IXIC"	01-01-01,014154.279785,0171.750000,0145.000000,0145.609863,0145.609863,0145.609863,2345220000
"IXIC"	01-01-04,014179.040339,0182.740234,0182.740234,0182.740234,0182.740234,0182.740234,2214770000
"IXIC"	01-01-10,014164.939947,0174.680176,0142.209961,0174.680176,0174.680176,0174.680176,22143070000
"IXIC"	01-01-16,014169.040339,0182.740234,0182.740234,0182.740234,0182.740234,0182.740234,2322240000
"IXIC"	01-01-22,014175.040339,0182.740234,0182.740234,0182.740234,0182.740234,0182.740234,0200000000
"IXIC"	01-01-01,014196.529785,0219.760039,0195.079980,0219.760039,0195.079980,0219.760039,0205800000
"IXIC"	01-01-07,014209.589848,0219.779785,0204.160156,0218.699941,0219.689941,0219.689941,2005850000
"IXIC"	01-01-13,0207.819824,0217.240234,0187.310059,0187.580078,0187.580078,0187.580078,2150370000
"IXIC"	01-01-19,0222.797980,0227.930176,0193.169922,0225.759766,0225.759766,0225.759766,2034030000
"IXIC"	01-01-25,0234.580078,0246.549806,0225.520020,0243.000000,0243.000000,0243.000000,2026910000
"IXIC"	01-01-28,0242.4222.797980,0242.4222.797980,0242.4222.797980,0242.4222.797980,0242.4222.797980,0242.4222.797980,0248957000
"IXIC"	01-01-01,014132.220217,0136.459994,0152.629000,0105.610107,0105.610107,0105.610107,0205820000
"IXIC"	01-01-07,014067.860107,0099.810059,0407.689941,0497.559961,0497.559961,0497.559961,2091180000
"IXIC"	01-01-13,04060.610107,0091.270020,0404.760010,0405.129932,0405.129932,02231850000
"IXIC"	01-01-19,04098.810059,0135.839844,0409.4169922,0123.129883,0123.129883,0123.129883,2168410000
"IXIC"	01-01-01,014068.629883,0124.919922,0407.610107,0410.879883,0410.879883,0410.879883,2300570000

## 2. JSON (JavaScript Object Notation)

- space efficiency: **BAD (worse than CSV)**: column name repeated
- Speed: Good enough
- Data types: strings, numbers, booleans, maps, lists.
- Splittable: splittable if 1 document per line.
- Extensibility: YES

"Ticker": "XIC"	"Date": "2014-01-02"	"Adj Close": 4143.069824	"Volume": 1738820000
"Ticker": "XIC"	"Date": "2014-01-03"	"Adj Close": 4131.910156	"Volume": 1667480000
"Ticker": "XIC"	"Date": "2014-01-06"	"Adj Close": 4113.680176	"Volume": 2292840000
"Ticker": "XIC"	"Date": "2014-01-07"	"Adj Close": 4103.75	"Volume": 2178220000
"Ticker": "XIC"	"Date": "2014-01-08"	"Adj Close": 4105.10176	"Volume": 214770000
"Ticker": "XIC"	"Date": "2014-01-09"	"Adj Close": 4156.189941	"Volume": 142070000
"Ticker": "XIC"	"Date": "2014-01-10"	"Adj Close": 4174.669922	"Volume": 322240000
"Ticker": "XIC"	"Date": "2014-01-13"	"Adj Close": 4113.299605	"Volume": 334180000
"Ticker": "XIC"	"Date": "2014-01-14"	"Adj Close": 4183.020024	"Volume": 101870000
"Ticker": "XIC"	"Date": "2014-01-15"	"Adj Close": 4214.879883	"Volume": 205850000
"Ticker": "XIC"	"Date": "2014-01-16"	"Adj Close": 4218.689941	"Volume": 150370000
"Ticker": "XIC"	"Date": "2014-01-17"	"Adj Close": 4197.580078	"Volume": 234030000
"Ticker": "XIC"	"Date": "2014-01-21"	"Adj Close": 4225.79766	"Volume": 191980000
"Ticker": "XIC"	"Date": "2014-01-22"	"Adj Close": 4243.000000	"Volume": 126910000
"Ticker": "XIC"	"Date": "2014-01-23"	"Adj Close": 4218.876883	"Volume": 398280000
"Ticker": "XIC"	"Date": "2014-01-24"	"Adj Close": 4128.169922	"Volume": 391180000
"Ticker": "XIC"	"Date": "2014-01-27"	"Adj Close": 4086.61010	"Volume": 231850000
"Ticker": "XIC"	"Date": "2014-01-28"	"Adj Close": 4097.999696	"Volume": 168410000
"Ticker": "XIC"	"Date": "2014-01-29"	"Adj Close": 42992.3	"Volume": 130057000
"Ticker": "XIC"	"Date": "2014-01-30"	"Adj Close": 4123.129882	"Volume": 14103.879883
"Ticker": "XIC"	"Date": "2014-01-31"	"Adj Close": 4103.879883	"Volume": 130057000

## 3. XML

- space efficiency: **BAD (worse than CSV)**

- Speed: ?

- Datatypes: strings, numbers, booleans, maps, lists.

- Splittable: YES

- Extensibility: YES

```
<?xml version="1.0" encoding="utf-8"?>
<Item>
  <Ticker>XIC</Ticker>
  <Date>2014-01-02</Date>
  <Open>4160.029785</Open>
  <High>4160.959961</High>
  <Low>4131.90039</Low>
  <Close>4143.069824</Close>
  <Adj Close>4143.069824</Adj Close>
  <Volume>1738820000</Volume>
</Item>
<Item>
  <Ticker>XIC</Ticker>
  <Date>2014-01-03</Date>
  <Open>4148.560059</Open>
  <High>4152.959961</High>
```

```
<Low>4124.959961</Low>
<Close>4131.910156</Close>
<Adj Close>4131.910156</Adj Close>
<Volume>1667480000</Volume>
</Item>
<Item>
  <Ticker>XIC</Ticker>
  <Date>2014-01-06</Date>
  <Open>4137.029785</Open>
  <High>4139.779785</High>
  <Low>4103.75</Low>
  <Close>4113.680176</Close>
  <Adj Close>4113.680176</Adj Close>
  <Volume>2292840000</Volume>
</Item>
```

## \* Binary Format.

### 1. Sequence File

- design goal: simplicity and efficiency.

- primary use case: storing the intermediate data in **MapReduce** computation.

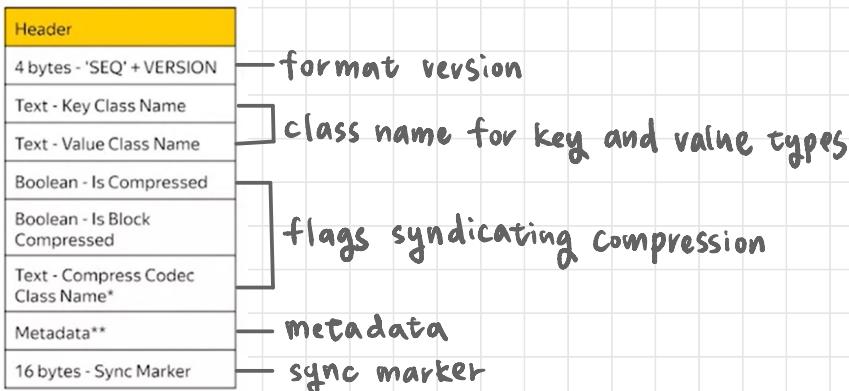
- first binary format implement in Hadoop.

- Stores sequence of **key-value pairs**, key and value are of

arbitrary type.

- Java-specific serialization / deserialization.

- composition of header



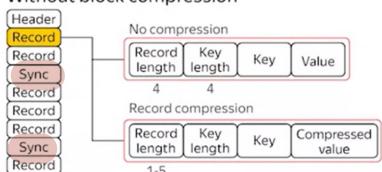
- compression

1. No compression: fixed size header with a record key length and value length.

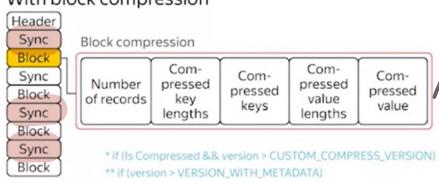
2. Record compression: every value is compressed individually

3. Block compression: a set of keys or values are compressed together (better compression)

Without block compression



With block compression



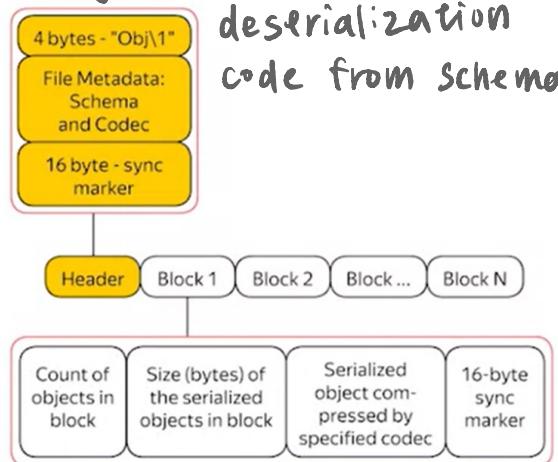
value is compressed with the code specified in the header

sync marker: Similar to the new line character in text file.

- Space efficiency: Moderate ~ Good
- Speed: Good
- Data types: Any with serialization/deserialization code.
- Splittable: Splittable via sync marker.
- Extensibility: NO

## 2. Avro

- design goal: efficient and **flexible format**.
- both format & support library
- stores objects **defined by the schema**
  - specifies field names, types, aliases,
  - defines serialization / deserialization code,
  - allows some schema updates.
- interoperability with many languages
- Space efficiency: Moderate ~ Good
- speed: Good w/ **codegen** (generate serialization and deserialization code from Schema)
- data types: JSON-like
- splittable: Yes, via sync mark
- Extensibility: YES.



# \* Row-based & Column-based formats.

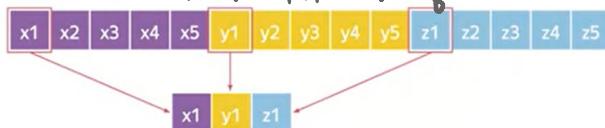
Table representation

X	Y	Z
x1	y1	z1
x2	y2	z2
x3	y3	z3
x4	y4	z4
x5	y5	z5

Row format ex. SequenceFile, Avro

x1 y1 z1 x2 y2 z2 x3 y3 z3 x4 y4 z4 x5 y5 z5

Column format ex. RCFile, Parquet.



## 3. RCFile (Record Columnar)

- first columnar format in Hadoop.
- Horizontal / Vertical partitioning
  - split rows into row groups.
  - transpose values within a row groups
- every RCFile spans multiple HDFS blocks, within every HDFS block, there is at least one row group, every row group contains sync marker, metadata, column data
- metadata used by decoder to read consequent column data.

contains:

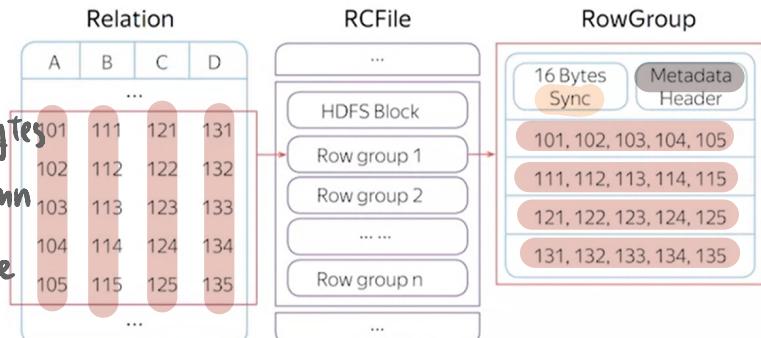
1. # of rows

2. # of columns

3. total # of bytes

4. bytes per column

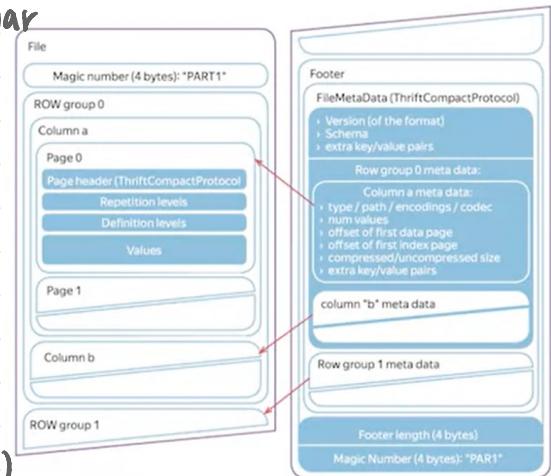
5. bytes per value



- space efficiency: Good
- speed: Moderate ~ Good, less I/O.
- data types: Bytes strings
- splittable: Yes, via sync marker.
- Extensibility: No.

#### 4. Parquet:

- most sophisticated columnar format in Hadoop.
- by Twitter & Cloudera
- supports nested data and repeated data
- exploits many columnar optimization (e.g. predicate pruning, per column codecs)
- optimize write path.



#### \* Compression

- Kinds of compression
- Block-level compression,
  - used in SequenceFiles, RCFiles, Parquet,
  - applied within a block of data.
  - able to navigate through the file quickly. sync marker and metadata could be used to devise splitting for the dataset without compressing entire file.

## • File-level compression

- applied to the file as a whole, ex. ZIP.
- hinders an ability to navigate through file.
- better compression ratios

## • Codecs

codec	compression speed	decompression speed	ratio
Gzip	16~90 MiB/s	250~320 MiB/s	2.77~3.43
Bzip2	12~14 MB/s	38~42 MiB/s	4.02~4.80
LZO	77~150 MiB/s	290~314 MiB/s	2.10~2.48
Snappy	200 MiB/s	475 MiB/s	2.05

## • CPU Bound v.s. I/O Bound

### • CPU Bound



compute speed << I/O speed

\* cannot benefit from compression

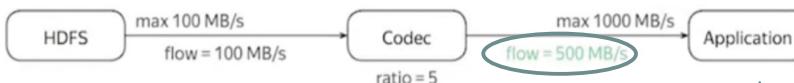
### • I/O Bound



I/O speed << compute speed

\* can benefit from compression.

e.g. compression with ratio = 5



five times more throughout when using compression.

## \*Conclusion

- many application assume relational model.
- file format defines encoding of your data.
  - text format: readable, quick prototyping, but inefficient.
  - binary format: efficient, but more complicated to use.
- file format vary in terms of space efficiency, encoding & decoding speed, supported data types, extensibility.
- when I/O bound, can benefit from compression.
- when CPU bound, compression may increase completion time.