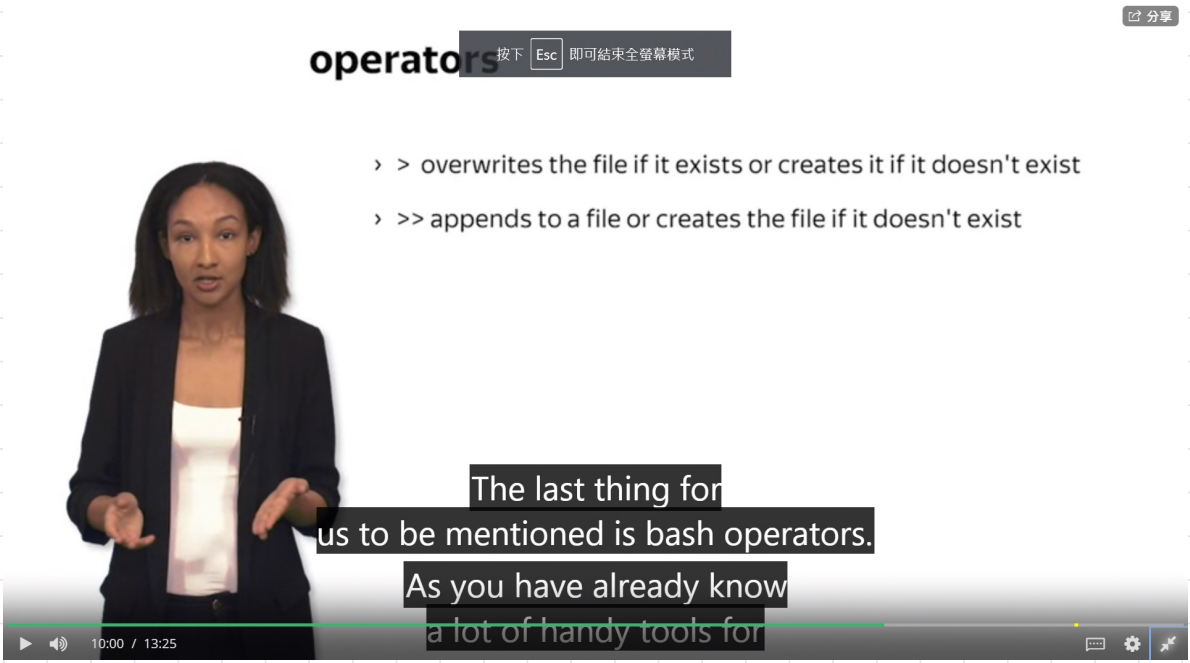
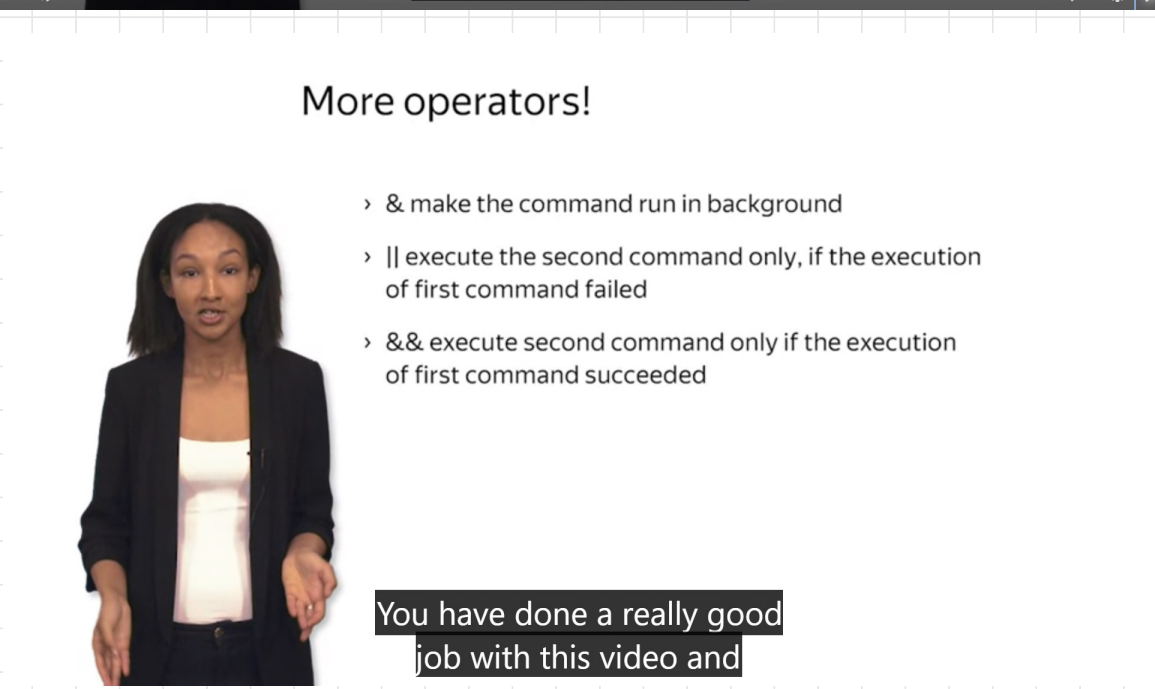
Unix Command Line Interface (CLI)

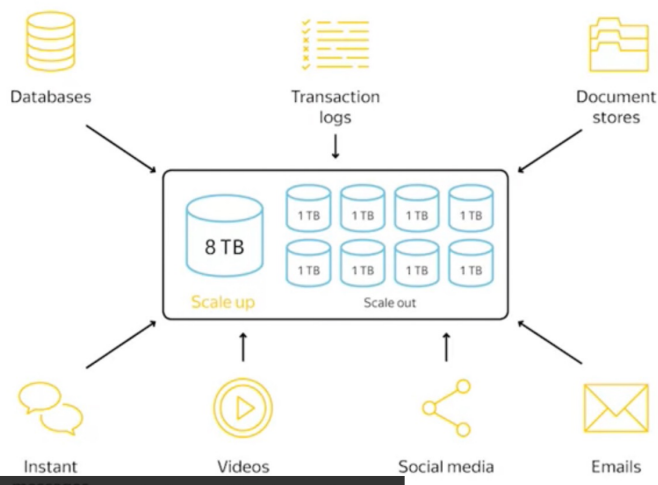




*free：顯示 memory 使用資源

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：存在同一個node

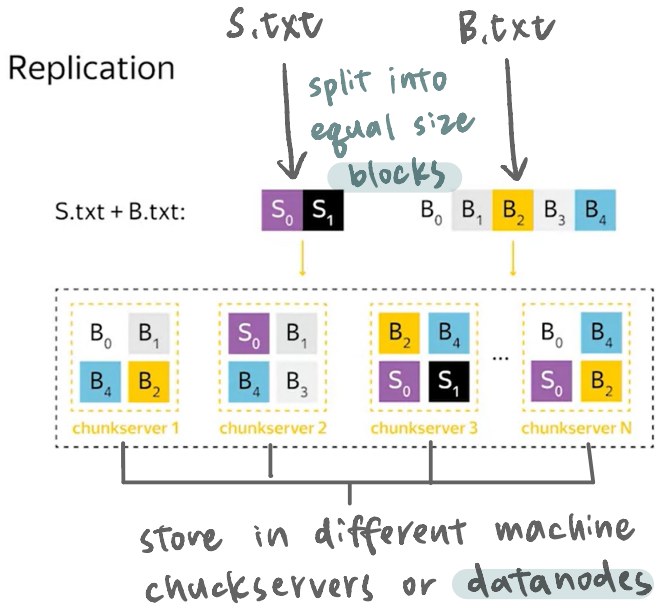
- low latency
- hard to scale

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

- high latency
- easy to scale

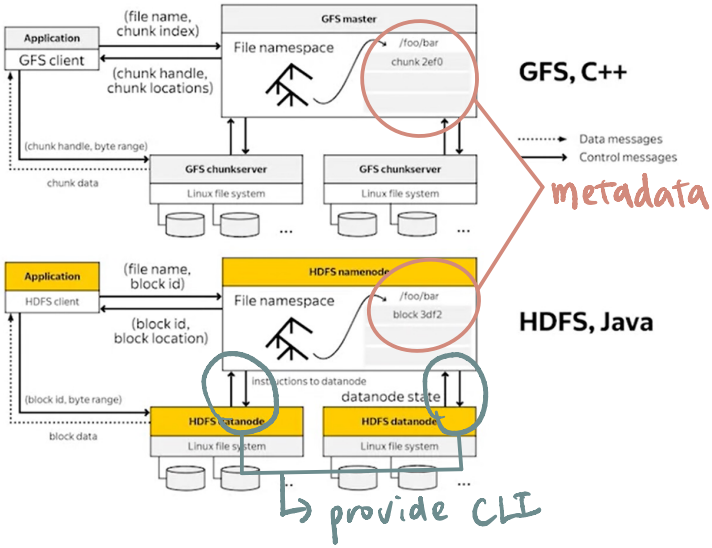
* Google File system Components

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



store in different machine called chuckservers or datanodes

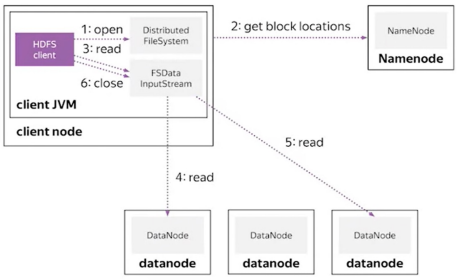
*RPC protocal to access data via HTTP protocal



metadata

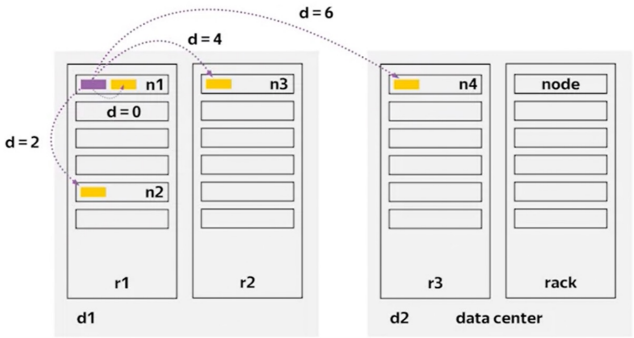
- store in master node with low latency in memory
- contain:
  - administration info.
  - creation time
  - access properties

Read file from HDFS



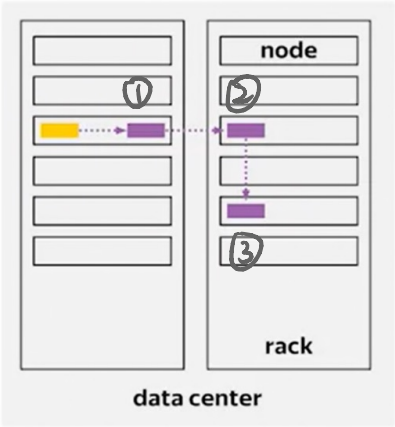
*get data from the closest machine

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



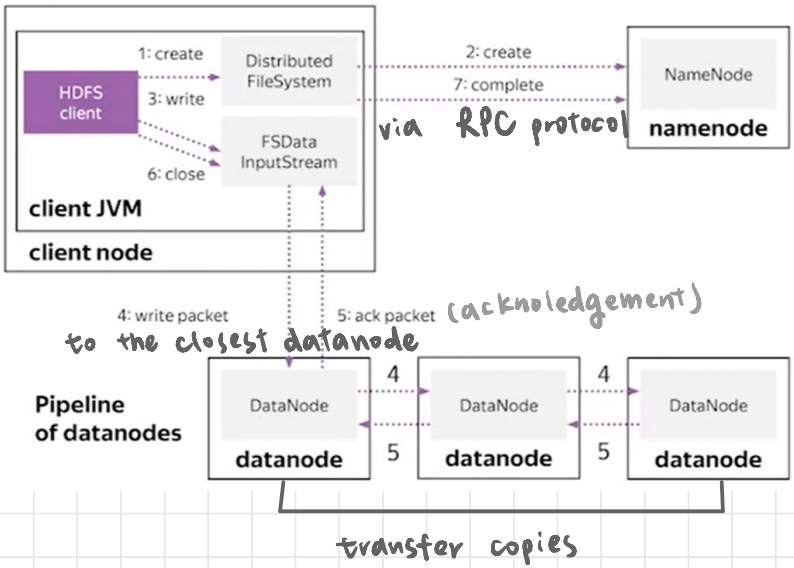
① first replica is usually located on the same machine

② second is placed in a different rack.

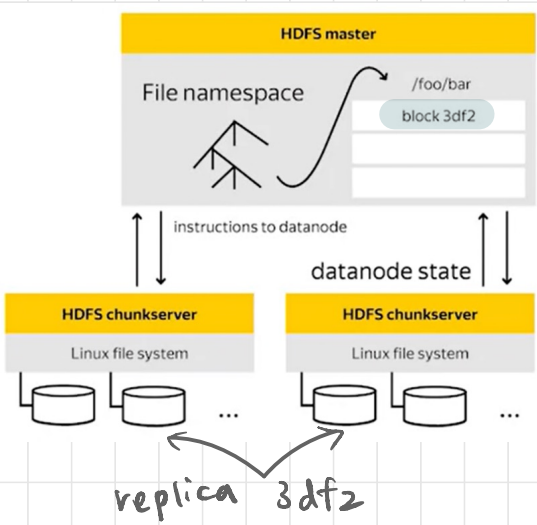
③ third is placed in diferent machine, same rack as ②

④ further placed random, system tries to avoid placing too many replicas on same rack.

* Data flow of Write Application



• Block and Replica States



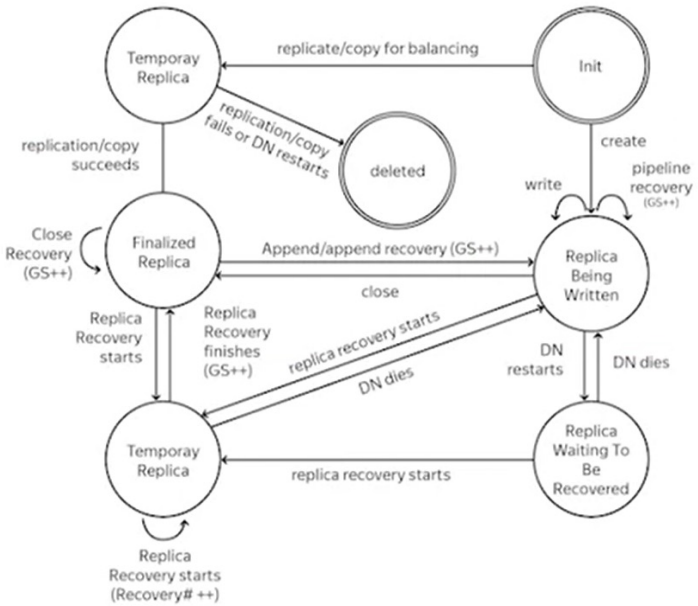
Replica: physical storage on datanode

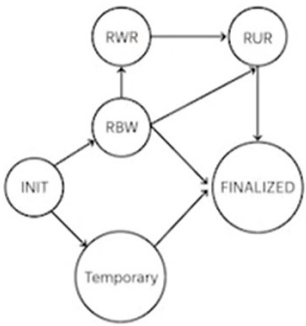
Block: meta-information storage on namenode provides info. about replica's locations and their states

* 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 Writen 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.

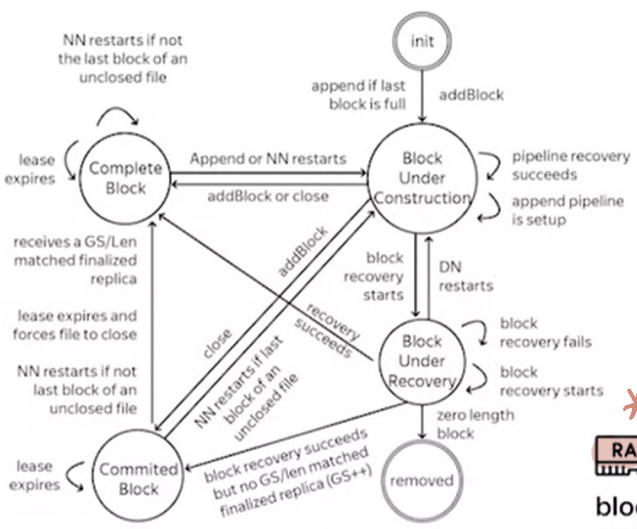
Replica Under Recovery (RUR)

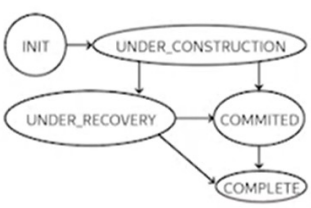
- HDFS client lease expiration, usually happen during client's site failure

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

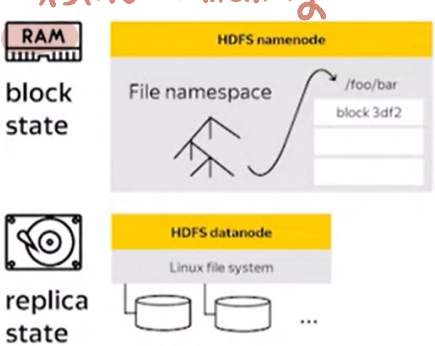
*Namenode Block's States





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 transtion from RWR → RUR
- when client dies, block from under_constnation → under_recovery

commited

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

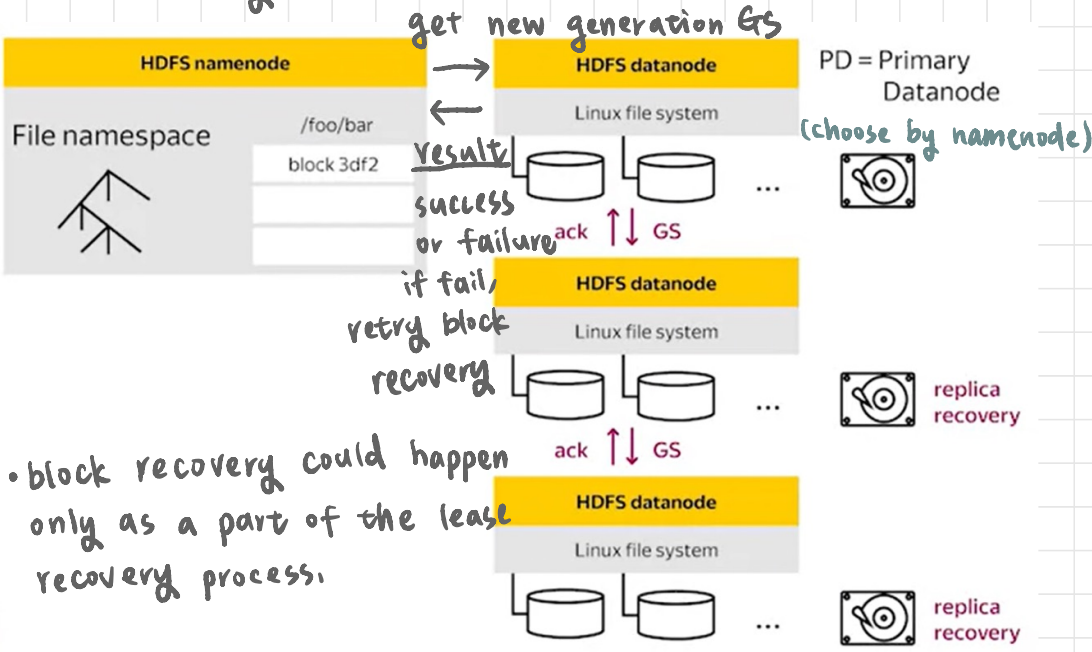
complete

- 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

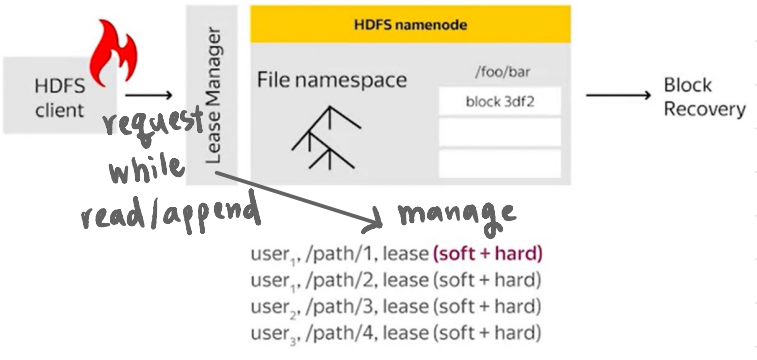


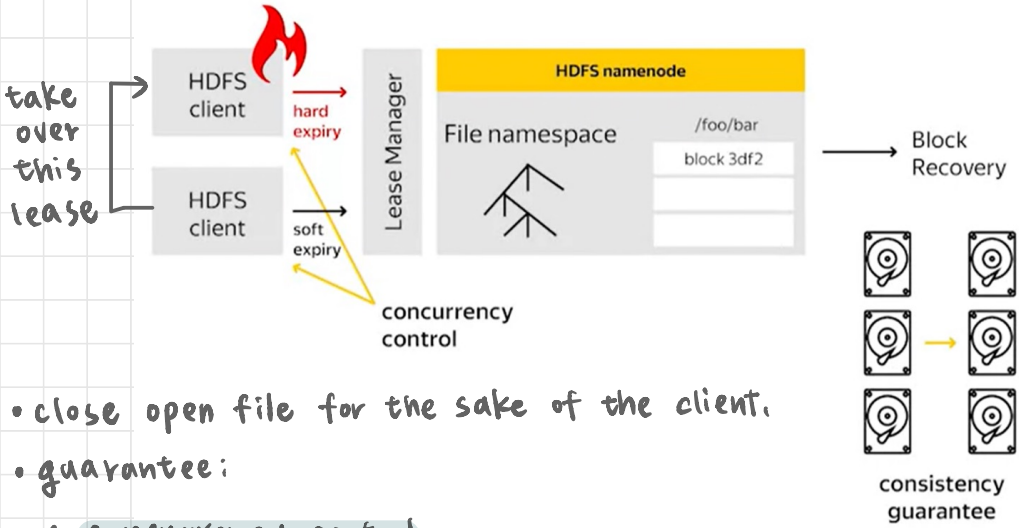
- block recovery could happen only as a part of the lease recovery process.

Replica Recovery

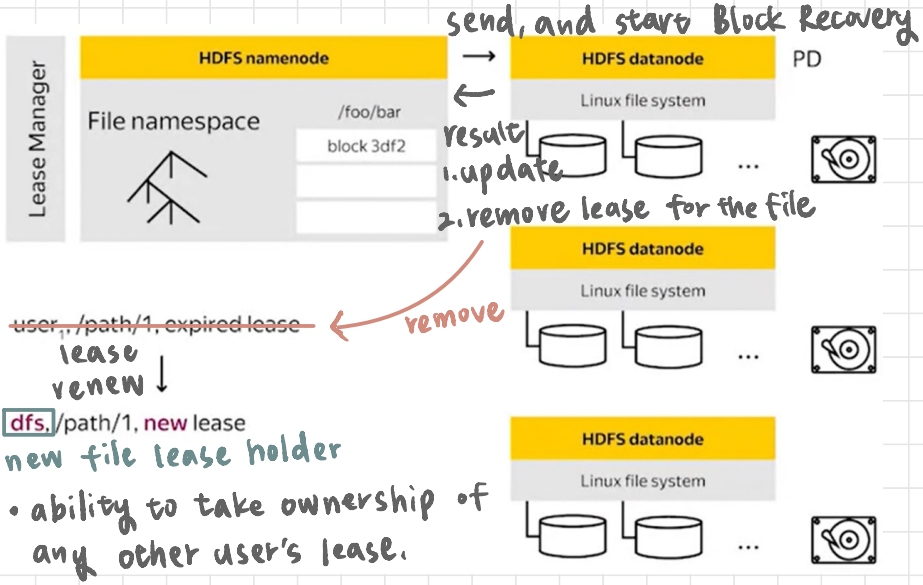
- aborting active clients right into replica.
- aborting previous replica.
- participating in final replica size agreement process.

Lease Recovery





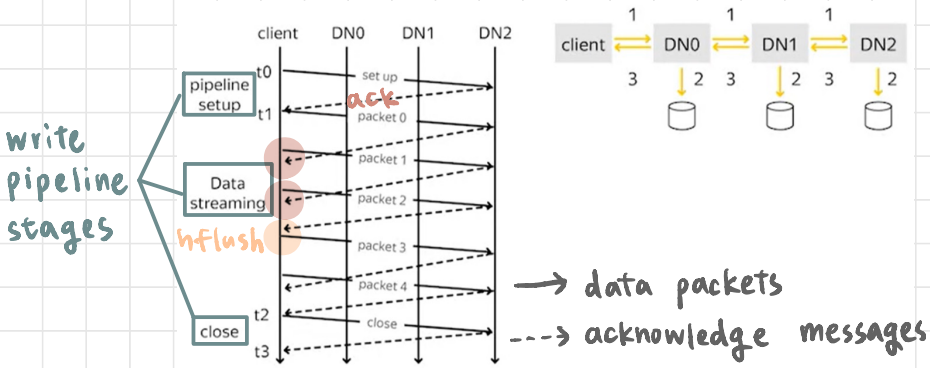
- close open file for the sake of the client.
- guarantee:
  1. concurrency control
     even if client is still alive, it won't able to write data.
  2. consistency guarantee
     all replicas should draw back to a consistence state, same on-disk data, GS



- ability to take ownership of any other user's lease.
- other client request will be rejected.

* 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

1. Failure happens during pipeline setup stage
   - cases: new file / append mode
   - can easily abandon datanode pipeline, request a new one.
2. 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.
3. Failure in Close
   - rebuild pipeline with good datanodes, bumps GS and request to finalized replicas.

* Namenode Architecture

- capacity of datanode and namenode

  suppouse have 10 PB data

  datanode

  suppose use 2TB hard drive, replication factor = 3, we need

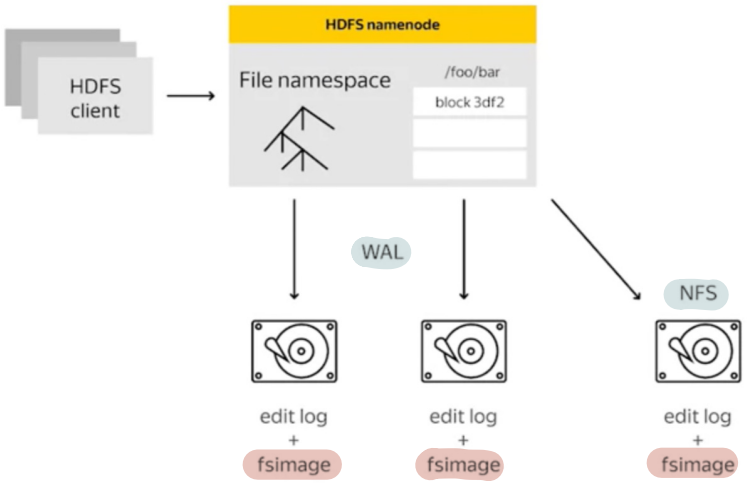
  10PB / 2TB * 3 ≈ 15 K hard drives.

  namenode

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

  10PB / (128 MB * 3) * 150 B ≈ 3.9 GB memory to store metadata.

- Namenode Durability and Speed up Namenode Recovery Process



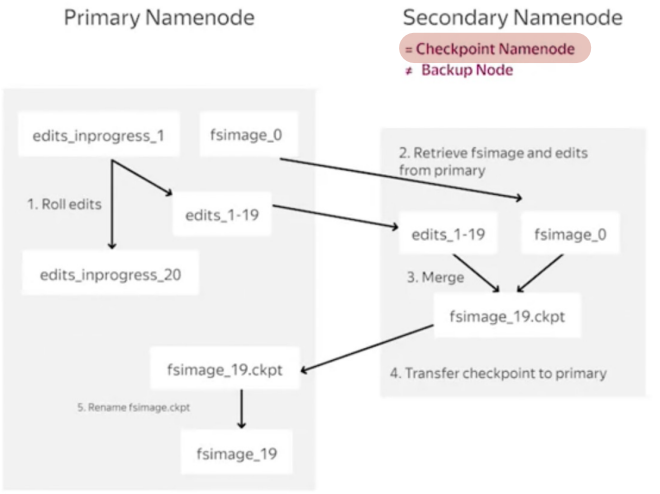
- **WAL** (write-ahead log)

  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

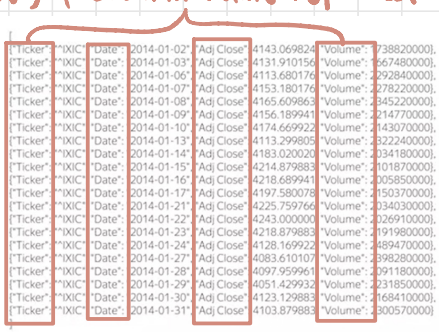
- Drawback: you need to parse it, convert it from textual form into programmatic data structure.

1. CSV & TSV (comma- & tab-seperated values)

- space efficiency: BAD, ① contains a single repetitive value
  ② use 11 bytes, native binary double precision float just 8 bytes.
- Speed: GOOD
- Data types: only strings
- Splittable: Splittable w/o header
- Extensibility: BAD.
  not easy to remove or reorder fields

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
  easily add and remove fields from data items. problematically infer if a particular field is present, and query its types.



3. XML

- space efficiency: BAD (Worse than CSV)
- Speed: ?
- Datatypes: strings, numbers, booleans, maps, lists.
- Splittable: YES
- Extensibility: YES

* 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

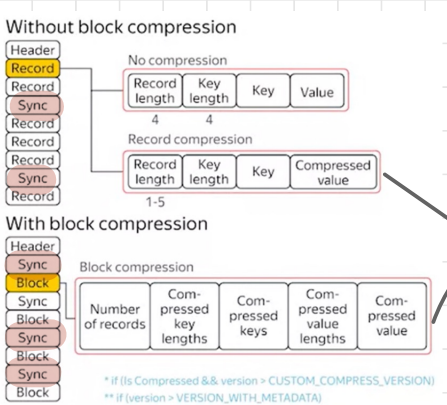
arbitary type.

- Java-specific serialization / deserialization.
- composition of header

Header	
4 bytes - 'SEQ' + VERSION	format version
Text - Key Class Name	class name for key and value types
Text - Value Class Name	
Boolean - Is Compressed	flags syndicating compression
Boolean - Is Block Compressed	
Text - Compress Codec Class Name*	
Metadata**	metadata
16 bytes - Sync Marker	sync marker

- 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)

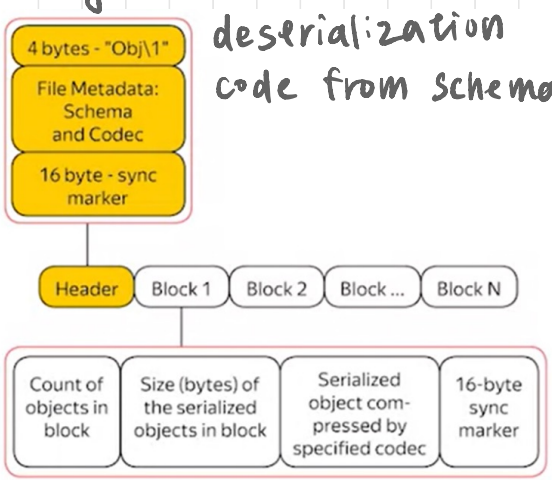


value is compressed with the code specified in the header

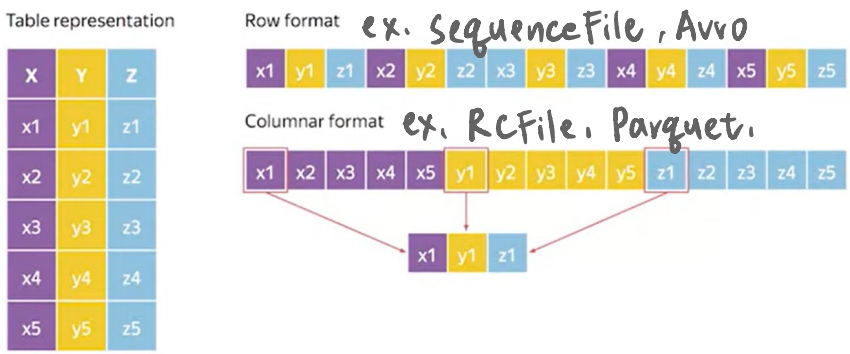
sync marker: similar to the new line character in text file.

- 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.

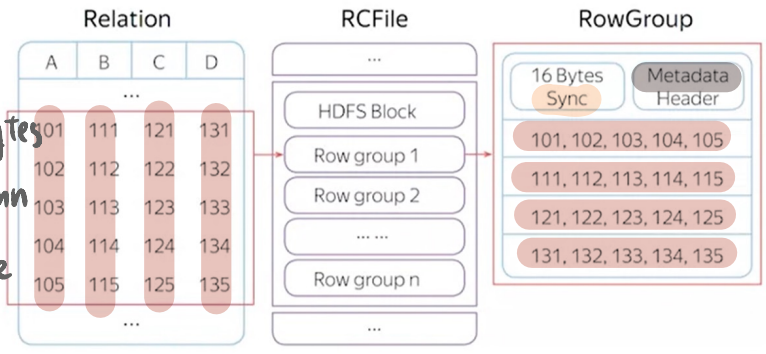


3. RCFile (Record Columnar)

- 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 groups, every row group contains sync marker, metadata, column data
- metadata used by decoder to read consequent column data.

contains:

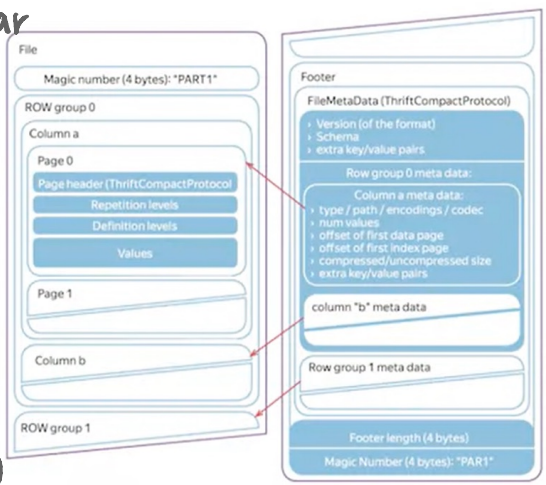
1. # of rows
2. # of columns
3. total # of bytes
4. bytes per column
5. bytes per value



- 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.x. 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 compression 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

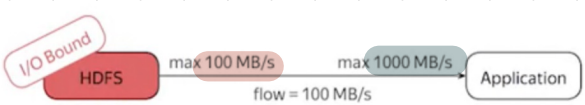
codec	compression speed	decompression speed	ration
Gzip	16~90 MiB/s	250~320 MiB/s	2.77~3.43
Bzip2	12~14 MB/s	38~42 MiB/s	4.02~4.80
LZO	77~150 MiB/s	290~314 MiB/s	2.10~2.48
Snappy	200 MiB/s	475 MiB/s	2.05

- CPU Bound v.s. I/O Bound
  - CPU Bound

compute speed << I/O speed

*cannot befit from compression

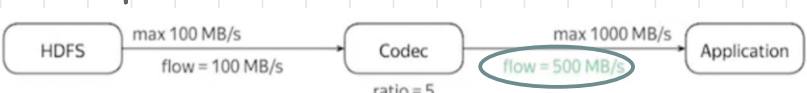
  - I/O Bound



I/O speed << compute speed

*can befit from compression.

e.x. 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.