Infrastructure for Data Analytics and Machine Learning

Part 1 - Distributed Filesystems

Paweł Wiejacha
RTB House

Plan for this lecture

- Filesystem, Distributed Filesystem, Object Storage
  - what they are?
  - and why we need them?
- What can we expect from Distributed Storage?
- How do Distributed Storage work?

File system

File system - a method and data structure used to control how data is stored and retrieved.

- Digital file systems are named and modeled after paper-based filing systems (e.g. file, directory, folder).
- Separating data into pieces (**files**), naming them, and organizing files into **directories** simplifies data storage and retrival
- Without such separation there would be just unorganized chunks of data written on a storage medium

The file system is responsible for:

- organizing files and directories
- mapping file parts to physical storage blocks
- keeping file and directory metadata (e.g. creation date, owner)

Distributed file system

Distributed file system - a file system where data is stored across multiple networked computers which communicate and coordinate their actions by passing messages.

Just like ordinary file system has to map files to disk sectors, distributed file system has to:

- map files to disk sectors stored on different nodes
- but also handle adding and removing machines to a system, handle network partitioning, etc.

Why we need a distributed storage?

Hardware limits

- We know what Distributed File Systems are, but **do we really need them**?
    - and if so, when?
- Computers become parallel and very powerful, there are technological advancements in the storage field
    - maybe we can buy an expensive computer, big and fast disks and use a simple non-distributed filesystem?
- To answer these questions, we need to examine information about **hardware constraints** that limit systems we can create today

Memory hierarchy

Storage medium	**Max. capacity**	**Reading / Writing speed**	**Seek time / random. IOPS**	**Price per TB**
Magnetic tape	20 TB	read: 400 MB/s write: 400 MB/s	10-100 s	drive + $13
SAS/SATA HDD	20 TB	read: 200 MB/s write: 269 MB/s	4.15 ms (~250 IOPS)	$22
NVMe SSD	30 TB	read: 7 000 MB/s write: 3 600 MB/s	0.001 ms (1e-6 s) read: 930 000 IOPS write: 190 000 IOPS	$125
DRAM	4 TB	190 000 MB/s	68-100ns (6.8e-8 s)	$7 500
L3 Cache (SRAM)	256 **MB**	~553 000 MB/s	~12ns (1.2e-8 s)	$7 000 000

Hardware limits and bottlenecks

- Number of processors in a single server
  - 2 CPUs (2x 48 CPU cores)
- Expansion bus limits
  - 128-160 PCIe gen4 lanes
    - ~2GB/s per PCIe gen4 lane
- Network adapters
  - 200 GbE (~25 GB/s)
  - latency: 1.01 microseconds
  - price ~$1600 per two port NIC

Storage failure rates

Annualized failure rate (AFR) - estimated probability that a component will fail during a full year of use.

Mean Time Between Failures (MTBF) - $\frac{1}{\lambda}$ where λ is failure rate.

$$AFR = 1 - e^{-8766/MTBF}$$

Both are population statistic estimated on a sample of given hardware components

Storage medium	*Real* AFR
Magnetic tape [1]	3.44%
SAS/SATA HDD	0.80% - 1.10%
NVMe SSD	0.78% - 1.22%
DRAM (Reg. ECC)	0.07 - 0.11%

[1] assuming unrealistic 100% duty cycle

Failures in real world

- **disk level**: assuming 1.22% AFR, given a system with 1450 disks, we can expect a disk failure every 3 weeks
- **node level**: power supply unit failures, motherboard failures
- **multi-node level**: network switch failures, a rack issues
- **data-center level**: maybe not earthquakes, but DNS/BGP problems
- human errors

With failures so common, they **cannot degrade** storage system performance - they need to be transparent to users.

Hardware setup for non-distributed storage

Let's try to select hardware for a imaginary **non-distributed** storage:

- **10 PB of stored data**
- batch task that reads only a part of that data (**1 PB**)

10 PB of data is not that much, e.g.

- 1.5e9 smartphone photos (~4 photos of every product available on Amazon)
- or 1080p movies uploaded to YouTube over 125-250 days
- or bid-logs uploaded to RTB House during 12 hours

Hardware setup for non-distributed storage

Let's try to select hardware for a imaginary non-distributed storage (10 PB of stored data, 1 PB batch task)

- reading 1 PB using a single HDD: `1 PB / 200 MB/s = 62 days` - too long

Hardware setup for non-distributed storage

Let's try to select hardware for a imaginary non-distributed storage (10 PB of stored data, 1 PB batch task)

- reading 1 PB using a single HDD: `1 PB / 200 MB/s = 62 days` - too long
- another reason to use more disks is storage space:
  - 10 PB / 30 TB disks = 341 disks
  - we want to handle simultaneous failure of 2 disks: `341 + 25% = 426 disks`
  - we want to have at least 20% of free space: `426 + 25% = 532 disks`
  - our system/company is expecting to grow: `532 + 15% = 600+ disks`

Hardware setup for non-distributed storage

Let's try to select hardware for a imaginary non-distributed storage (10 PB of stored data, 1 PB batch task)

- reading 1 PB using a single HDD: `1 PB / 200 MB/s = 62 days` - too long
- another reason to use more disks is storage space:
  - 10 PB / 30 TB disks = 341 disks
  - we want to handle simultaneous failure of 2 disks: `341 + 25% = 426 disks`
  - we want to have at least 20% of free space: `426 + 25% = 532 disks`
  - our system/company is expecting to grow: `532 + 15% = 600+ disks`
- reading 1 PB using 600 HDDs: `1 PB / 600 / 200 MB/s = 2.5 hours` - acceptable

Hardware setup for non-distributed storage

Let's try to select hardware for a imaginary non-distributed storage (10 PB of stored data, 1 PB batch task)

- 600 HDDs is too many for a single machine
  - slots for disks - max 72 disks per pachine (usually 12-48)
  - 600 HDDs: 420 kg, 12 meters high stack or 90 meters long chain

Hardware setup for non-distributed storage

Let's try to select hardware for a imaginary non-distributed storage (10 PB of stored data, 1 PB batch task)

- 600 HDDs is too many for a single machine
  - slots for disks - max 72 disks per pachine (usually 12-48)
  - 600 HDDs: 420 kg, 12 meters high stack or 90 meters long chain
- `600 disks * 200 MiB/s = 117 GiB/s` - slightly too much for a single machine:
  - 59 PCIe lanes just to read data from disks
  - copy to memory (12 cores needed - for a simple `memcpy()`)
  - decrypt, compute checksums (46 cores needed for decryption)
  - read from memory (another 12 cores)
  - compute or send over network
    - sending over network requires 64 PCIe lanes and 5 network adapters
    - Linux kernel would have a problem to handle that many network packets

Hardware setup for non-distributed storage

Let's try to select hardware for a imaginary non-distributed storage (10 PB of stored data, 1 PB batch task)

Reliability and economy:

- in real world, it's likely you are scaling up an existing system (e.g. hardware bought 2 years ago)
  - you have lots of 15TB disks instead of 30 TB ones
  - and some PCIe gen3 motherboards instead of new PCIe gen4
- having spare PSU, motherboard, CPUs, RAM modules is a good idea
- but it's not possible to achieve high availability this way

Advances in modern hardware allow to create truly high performant storage systems.

But in many cases we still need distributed storage

Distributed storage interfaces for batch processing

Block storage

Block-level storage - storage where interface emulates behaviour of a traditional block device (e.g. HDD).

Data is organized as **blocks** which are identified by an arbitrary identifier which can be used to store and retrieve given block. There are no files, directories, no structure - just blocks.

Examples:

- non-cloud: Ceph RADOS Block Device
- cloud: Amazon EBS, Google Cloud Persistent Disks

Verdict: **Too low level to for batch processing** - we need higher level of abstraction.

POSIX file system interface

Portable Operating System Interface (POSIX) - a family of standards specified in the 1988 for maintaining compatibility between (UNIX-like) operating systems.

There is a section of the standard that defines the semantics of the POSIX-compatible file system, e.g.:

- allows hierarchical file names and resolution (e.g. `/dir/dir/file`)
- strong consistency, atomic writes, atomic renames, deleting open files
- implement certain operations, like:
  - random access reads/writes (`pread()`, `pwrite()`)
  - access control (`chown`, `chmod`, etc.)
  - symlinks, hardlinks
  - `ftruncate()`, `fsync()`, `fcntl()`, `mmap()`, `fadvise()`, `fallocate()`

POSIX file system interface

Examples[1]:

- local filesystems: XFS, ext4, Btrfs, ZFS, ...
- distributed filesystems: CephFS, GlusterFS, MooseFS/LizardFS, Lustre, JuiceFS

Verdict:

- **too complex** - implementing a POSIX-compliant distributed filesystem is a challenge
- not all features nor consistency guarantees are needed for batch processing

[1] - with various level of POSIX compliance

Object storage

Object storage, blob storage - storage architecture that manages data as objects that

- have globally unique identifier
- contain variable length data
- and optionally a metadata attached to them

Usually, the structure is flat (limited number of buckets, objects inside buckets) - there are no nested file directories.

Basically, a KVS (Key-Value Store) with big "Values" and metadata.

Object storage

Limited set of operations, usually:

- `get_object(name)`, `put_object(name, data)`, `delete_object(name)`, `set_metadata(name, metadata)`
- `list_objects(prefix)` - since there are no directories, we need to use prefix searches like:
    - "list all objects with names starting with `/folder/sub-folder/`" to simulate file directories

Limited functionality:

- sometimes only eventual consistency
- immutable objects
    - no positioned writes, no `mmap()`
    - in some cases: append-only writes
- sequential reads or limited random reads

Object storage

Examples:

- Cloud: Amazon S3, Google Cloud Storage, Azure Blob Storage, Cloudflare R3
- Open Source: Ceph Object Gateway, MinIO, OpenStack Swift, HDFS [1]

Verdict: **just right for batch processing**

- have all operations needed for batch processing
- simplified semantic does not limit storage performance

[1] - not a POSIX filesystem, not a object storage

Databases?

Why can't we use a database for **batch processing**?

We can. Sometimes.

We should use the best means to achieve our goal. In many cases using a specialized database for batch processing is a good choice, e.g.

- columnar databases, when we only use small subset of columns
- time series database, when processing time series
- document databases with decent indexes, if that's our use case

Sometimes it's worth considering using both database and object storage during single batch processing task.

Database vs Object Storage - blobs and records

Object storage is kind of KVS but with "big" values, e.g.

1. **binary data**: images, videos, or recorder sounds
2. files that contain multiple "**records**" in formats like CSV, JSON, Avro, Parquet

Let's skip the obvious case of batch processing binary data, and focus on the second one.

Sometimes it's better to keep this data records as they are:

- do not restructure records
- do not index them
- just roughly partition them (e.g. by type or creation date) while writing
- and process them sequentially when needed

Database vs Object Storage - denormalization

Example use case

Imagine you own a big e-commerce store that records every action user performs on your website. And that you want to answer **ad-hoc** questions like:

- What was the price of product ABC 200 days ago?
- What how many unexpired promo tokens user XYZ had 183 days ago?
- And what **exactly** was detailed user XYZ browsing history just **before** request 2342-234237483-56452-234232 was received?

Database vs Object Storage - denormalization (2)

You could create complex and normalized database schema, with hundreds of tables and answer those questions performing really **huge joins** that need huge DB indices.

But sometimes this is not very realistic approach. And not only because of performance reasons:

- the world, your company, your platform and you **will evolve**
  - maintaining database schema and existing queries would be a nightmare
- sometimes you cannot build an index, because you cannot predict the future, e.g.
  - what queries analytic team will need to perform next quarter?
  - what algorithms Machine Learning team will create?
- **sometimes it better to store unstructured data** (e.g. raw requests in a JSON format)
  - it might be easier than reaching consensus about taxonomy, schema, shared indices, priorities

```
SELECT
    event.time,
    event.type,
    product.name,
    product_prices.price,
    user_segment, ...
FROM
    events JOIN
    products JOIN
    product_prices JOIN
    i18n_product_details JOIN
    users JOIN
    order_requests
    ...
```

Database vs Object Storage - denormalization (3)

Sometimes to avoid complex joins it's worth to consider keeping:

- **raw events**
- (redundant) big **complex snapshots** (e.g. snapshot of all records related to user XYZ)
    - those can be stored
        - periodically (e.g. every day at midnight)
        - or even with every request
- optionally, redundant **periodical deltas** containing consolidated information that changed since last snapshot

In our example, to materialize user profile at requested point of time

- instead of reading all records about user XYZ stored in hundreds of files created during last 2 years
- we could use the latest snapshot of user XYZ history + few deltas + few raw events

Storing redundant materialized snapshots will consume **huge amounts of storage space**, but it's worth considering as it might speed up **simplify** batch processing.

Database vs Object Storage - data locality

- Sometimes the easiest or fastest way is to processes unstructured data sequentially
  - This processing might require a lot of computing power
  - and performing computations **close to data** might be desired property
    - property that might not be possible to achieve when standard database is used

Distributed storage - features and requirements

Distributed storage - basic requirements

What are our basic requirements for a distributed storage?

- **storing data** and providing reasonable interface
- having **scalable storage capacity**
- having **scalable** data access **throughput**
    - i.e. increasing number nodes/disks increases total bandwidth/IOPS
- being **resilient** to hardware failures
    - both temporary (e.g. HA) and persistent (broken disks)

What else can we expect from a distributed storage used for batch processing?

Encryption at rest

If someone breaks into a data center and steals our servers/disks he should not be able to read our data

- usually we can encrypt drives and store keys in memory, and store the master key in some secure location (TPM, TEE)

Encryption at rest this might be a legal requirement

Metadata

- sometimes you don't want to read files to get some basic information
  - `gdpr_cleanup_id`, `schema_version`
- and you cannot store it in the object name
  - sometimes you want to attach or change metadata long after file was created
- **having those metadata indexed is a nice feature**
  - easier than managing the metadata and consistency in a separate database

Object versioning

- the idea is that when you overwrite an object, storage keeps previous and the current version of it
  - and when you delete it keeps old version and a deletion maker
- this is not something you cannot achieve with sensible naming policy
  - but sometimes it simplifies things and is a nice feature
  - especially if paired with lifecycle policy

Constant-time file concatenation

- Needed in scenarios when multiple writers create multiple files (e.g. in M/R job) that logically are a single file
  - and when processing concatenated file is easier than processing hundreds of files)
- Also appending header/footer without the need of rewriting source file can be useful

Snapshots

- This is rather file system not object storage feature
- With snapshots, we can make a **instantaneous read-only point-in-time snapshot** of filesystem subtree
    - and later modify the source subtree without changing the snapshot
- We can take multiple snapshots of a given directory
- This is quite simple operation in a filesystem when you cannot modify files

HDFS supports filesystem subtree snapshots.

Compression

- Compression not only **increases storage capacity**
- It also reduces disk utilization
    - and network bandwidth (if we decompress on client)
- If we use compression algorithms that focus on compression/decompression speeds (lz4, zstd, snappy) we can also **increase download and uploads speeds**

Manual compression - compressing data by ourselves when uploading and decompress when downloading

Transparent compression - with compression implemented by the file system we reduce amount of code/logic to write and maintain

Atomic rename or Compare-and-Set

With **Compare-and-Set** operation or at least **atomic rename**, we can implement few useful algorithms and create complex systems on top of a object storage (i.e. without using additional database, distributed locks, etc.)

```
# returns current object data and object sequence number (monotonic version number)
def read_object(key) -> (value, object_seq_no): ...

# atomically writes object *only* if current object's sequence number is expected_seq_no
def compare_and_set(key, new_value, expected_seq_no) -> Success | Failure: ...
```

Unfortunately, not many object storages implement this.

- HDFS has atomic rename
- GCS has no rename operation, no CAS
- S3 implemented strong-consistency for read-after-write (for a single PUT/DELETE) two years ago

Lifecycle and retention policies

Sometimes storage system can store store object in different ways:

- with more or less resiliency (e.g. handling only one not three disk failures)
- or on a cheaper medium (e.g. tapes) where object retrieval is delayed

It's very convenient if storage system allows to set **retention policies**, e.g.:

- delete objects older than X days
- move objects older than X days to a cold storage

Distributed storage

building blocks and common design patterns

Checksums

Computing checksums (error detecting codes) is the **absolute minimum** distributed storage should do.

- the simplest way is parity check (split data into n-bit words and xor them)
- CRC (cyclic redundancy check)
- using cryptographic hash function that uses dedicated CPU instructions (e.g. SHA instruction set)

```
import zlib
zlib.crc32(b"some-data-abcdefgh") # 2142146772
zlib.crc32(b"some-ERRORabcdefgh") # 1649843363
```

Resiliency - replication

Replication is the simplest way to achieve storage that is resilient to disk or node failures.

Basically, to handle $n - 1$ disk failures, instead of writing each object (file or block) once, we need to **write it n times** on different disks (and different nodes if we want to handle node failures).

Pros:

- the simplest way to achieve resiliency
- can speed up reading process by reading from different disks
    - if the same file is read simultaneously by multiple readers

Cons:

- you need to buy n times more disks

Resiliency - erasure coding

[optimal] erasure code is a forward error correction:

- that transforms a message of n symbols into a message with $m = n + r$ symbols (n original and r redundant)
- so that the original message can be recovered from **any n-symbol subset** of the m symbol message
- the "symbol" can be any number of bits/bytes (processor word, 512 byte sector, 64 KB block)
- we are assuming "erasure" (disappearing) of symbols instead of "error" (corruption) of symbols
  - that's why checksums are so important

Resiliency - erasure coding (2)

- the simplest erasure code is xor/parity code ($r = 1$)

```
 123  ^  561  ^  913  # == 475 (parity)
(475) ^  561  ^  913  # == 123
 123  ^ (475) ^  913  # == 561
 123  ^  561  ^ (475) # == 913
```

- for general case ($r > 1$), one of methods is polynomial oversampling
  - you need at least $n + 1$ points to determine n-degree polynomial coefficients [over some finite field]
  - so you can:
    - use original message symbols as polynomial $p(i)$ coefficients
    - then generate $\{ p(i) : i \in 1..m \}$ samples and use them as a erasure-coded message

```
# input message: [3, 5]
p(x) = 3*x + 5

# output 2+2 message: [p(1), p(2), p(3), p(4)]
[8, 11, 14, 17]
```

Distributed storage - building blocks

Resiliency - example

Input: 2000 KiB file, block size = 1000 KiB

Replication (3x):

- split file into 2 blocks: b_1 and b_2, then for each block
  - write the same block 3 times using 3 different disks

```
                                    block_1      block_2
[0123456789abcdefghijkl..] -> [0123456789abcdef] [ghijkl..]
                              [0123456789abcdef] [ghijkl..]
                              [0123456789abcdef] [ghijkl..]
```

8+2 erasure coding:

- split file into 2 blocks: b_1 and b_2, then for each block
  - split each block into 8 chunks (125 KiB each): $c_1, \ldots, c_8$
  - compute parity chunks:
    - $c_9 = parity_\alpha(c_1, \ldots, c_8)$,
    - $c_{10} = parity_\beta(c_1, \ldots, c_8)$
  - write each chunk on a different disk

```
                                    block_1      block_2
[0123456789abcdefghijkl..] -> [0123456789abcdef] [ghijkl..]
                              [01]               [gh]
                              [23]               [ij]
                              [45]               [kl]
                              [67]               ..
                              [89]               ..
                              [ab]               ..
                              [cd]               ..
                              [ef]               ..
         redundant chunks:    [UV]               [PQ]
         redundant chunks:    [XY]               [RS]
```

Resiliency - erasure coding (3)

- the more chunks the **smaller storage overhead** (e.g. 1000+2 vs 3*1000)
    - but need multiple disk accesses (e.g. 1000 seeks) to read a single block
    - and you need system with at least 1002 **different** disks
- computation redundant chunks requires more computing power than a simple copy done by replication
- you need more disks (e.g. 10 vs 3 in 8+2 EC vs 3x replication case) to get the same resiliency level
- EC is a much more **complex solution**, e.g.
    - in case of failure of a disk containing original chunk, you need to reconstruct the block on the fly

Sharding

Sharding - partitioning data horizontally (e.g. by rows not columns) so that each data shard is stored or handled by a different instance, process or disk.

Examples:

- storing all blocks with `hash(block_uuid) mod num_machines == i` on machine i
- storing all metadata with `hash(object_name) mod num_metadata_servers == i` on server i

This way we can distribute load and responsibility evenly and scale our system.

```
# file_names
['file0', 'file1', 'file2', 'file3', 'file4', 'file5', 'file6', 'file7', 'file8', 'file9']

>>> [hash(file_name) % 3 for file_name in file_names]
[0, 1, 2, 1, 0, 2, 0, 2, 0, 0]

# server0: 'file0', 'file4', 'file6', 'file8', 'file9'
# server1: 'file1', 'file3'
# server3: 'file2', 'file5', 'file7'
```

Tiering and caching

Tiering - arranging something in tiers (layers)

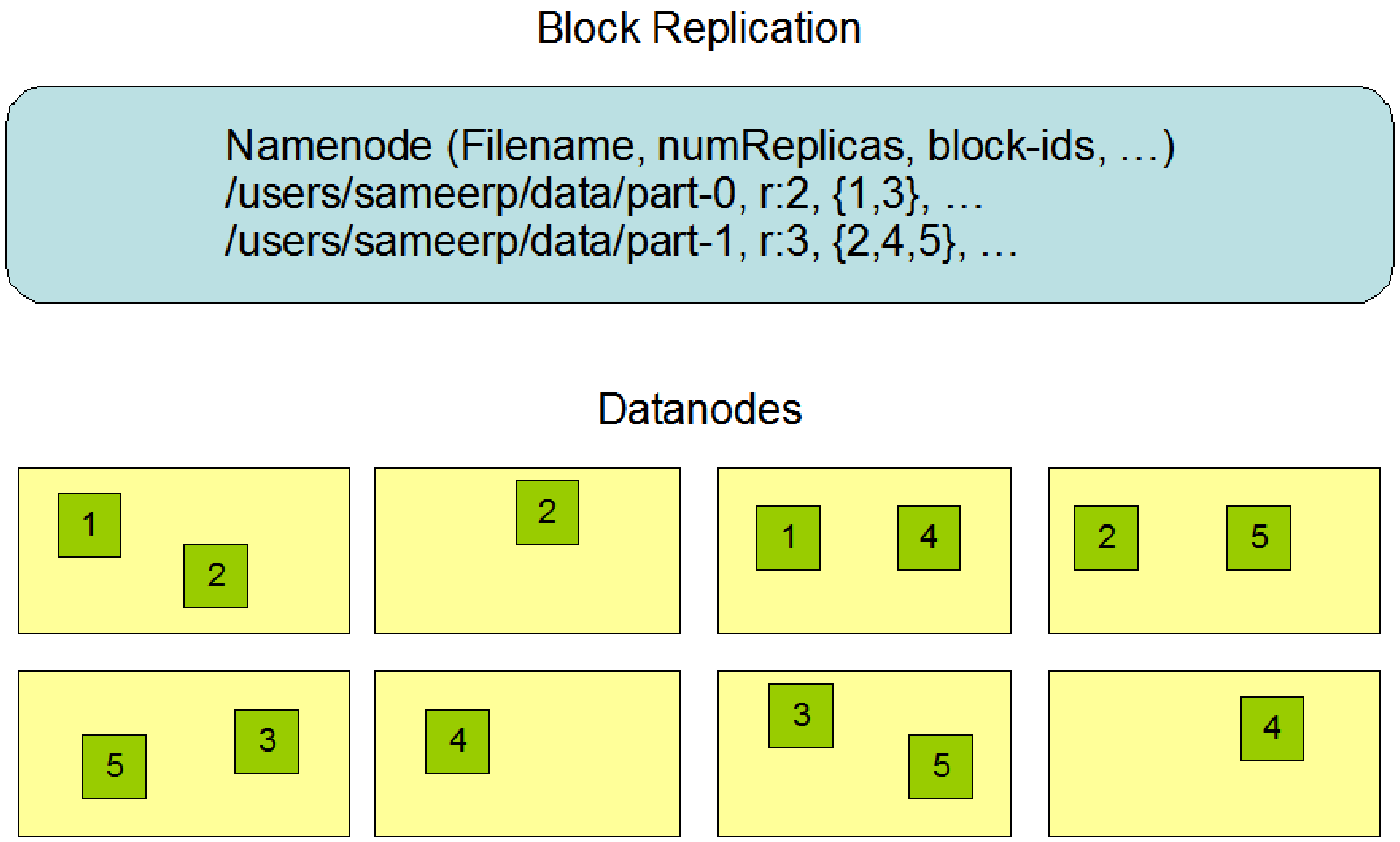
For example:

- Keeping frequently used data (filesystem hierarchy, indexes, file metadata) in RAM to reduce latency
- Keeping hot data (recently read files, blocks prefetched by the read-ahead mechanism) in RAM or SSD
- Keeping cold and ordinary data on slower medium (e.g. HDD)

Data organization

- storing data on multiple machines (*data nodes*)
- splitting files into **blocks** with limited size (e.g. up to 128 MB)
  - simplifies things like reconstruction, balancing, out of space handling
- placement awareness (same disk, same node, same rack)
- replication or erasure coding
- data transfers should be done directly to/from a data node
- the simplest way to store blocks is to use a normal filesystem (XFS, EXT4)
  - examples: HDFS, MinIO, OpenStack Swift,
  - this way we can use `fsck.ext4`, `rsync`, `ls`, `cp`, software RAID, etc.
- if we need to squeeze more performance we can create storage layer by ourselves
  - example: BlueStore in CephFS

Data organization

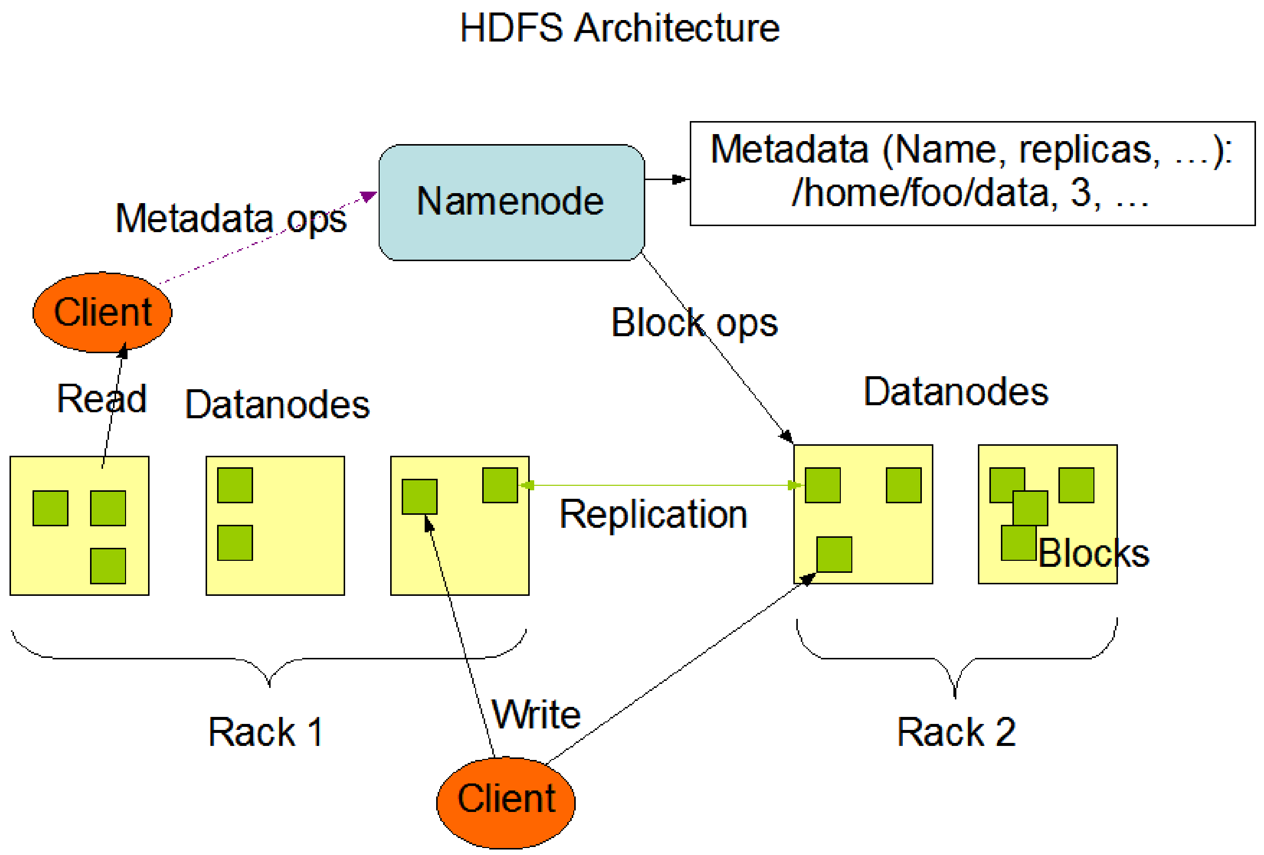

Block Replication
Namenode (Filename, numReplicas, block-ids, …)
/users/sameerp/data/part-0, r:2, {1,3}, …
/users/sameerp/data/part-1, r:3, {2,4,5}, …
Datanodes
1
2
2
1
4
2
5
5
3
4
3
5
4

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

Metadata organization (1)

- We stored data blocks in a resilient way on data nodes.
- During read request, client is going to connect directly to data nodes to fetch blocks
- But which data node client should connect?
- We need some place to keep **metadata**
    - (e.g. file size, list of file's blocks and their locations, index of stored files)
- We can use deterministic algorithms and placement policies, sharding, broadcast messages, etc. to reduce size of metadata
- But we need to keep some metadata
- And if we want to provide reasonable consistency guarantees we need to coordinate (e.g. serialize) all metadata operations

Metadata organization (2)



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

Metadata organization (3)

- the simplest way is to keep all metadata on a single server
  - powerful enough to handle all metadata operations and keep hot data in RAM
  - forcing every metadata request to go through this server to guarantee consistency
- next level is to use active/standby configuration, e.g.
  - writing metadata/log on a secure storage before confirming
  - waiting for replica confirmation storage before confirming to client
- next level: multiple metadata servers that use distributed consensus algorithms (Raft, Paxos)
- don't forget about sharding
- how to store metadata?
  - filesystem
  - some basic local KVS/DB (e.g. LevelDB)
  - custom database

Rebalancing

Storage system should handle:

- **extension** - adding more disk and nodes to extend storage and throughput
- **retiring (decomissioning)** - marking old or partially broken nodes for removal or repair

Those actions can result in unbalanced storage cluster state

- e.g. 200 disks are 95% full and 50 new disks are empty

Unbalanced system should automatically start a **rebalancing process** that moves data blocks to new places.

Ideally this process should not degrade user reads/writes.

Reconstruction

In case of (multiple) node or disk failures number of original/redundant chunks falls

- we either lost some data (if we lost too many chunks)
- there are less chunks than required to handle next failure

Storage system can reconstruct blocks on-the-fly during read operations, but it should perform **reconstruction process** to rebuild missing chunks permanently.

- especially for files that are not going to be read in the near future

Implementing it correctly is kind of tricky:

- we should not degrade user experience
- but we also cannot delay/throttle reconstruction because another failure can happen when we are not ready for it
- we also should handle temporary "failures" (e.g. failed switch, machine reboot) gracefully

Scrubbing

Scrubbing - process of reading stored data in the background in order to detect errors.

- it can detect hardware errors (broken sectors, broken drives) and sometimes software errors (incorrect programs)
- read data can be compared to:
  - other replicas
  - reconstructed data from EC
  - stored checksum
- assuming some redundant information is present, data can be reconstructed

It's better to reconstruct data when it's still possible than have a storage system with hidden possibly unrecoverable corrupted data.

Summary

We have discused following topics:

- What is a Distributed Storage
- Why batch processing may require Distributed Storage
- What interface for Distributed Storage is best suited for batch processing
- What can we expect from a Distributed Storage
- How (in general) Distributed Stores work