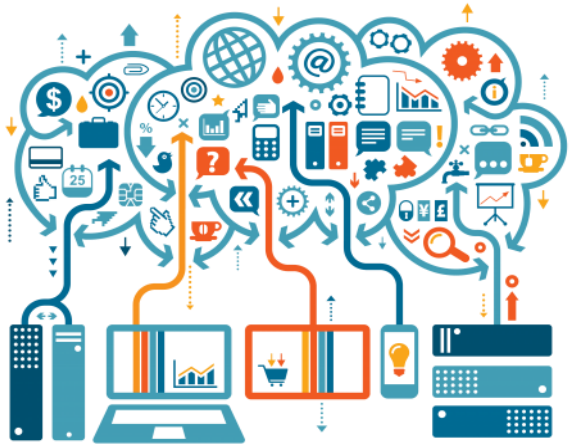
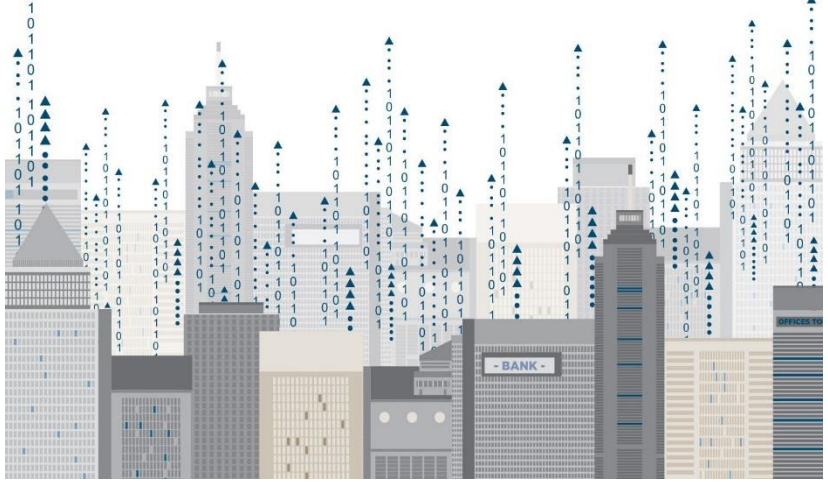




Big Data Analytics

Fall 2025

Lecture 9

HDFS

- WORM: Write-once, read-many-times storage pattern
  - Once the data is recorded, it is permanently fixed. You cannot overwrite or erase it.
  - The stored data can be read, copied, or retrieved any number of times without changing it.
  - Ensures data integrity (tampering, accidental delete)

HDFS

- Commodity hardware – which are not server machines
  - Multi-core processor
  - 16-64 GB RAM
  - SSD/HDD 2 TB – but SSD for NameNode
  - 1 Gbps minimum Internet
  - Linux
- Not good for low-latency (video, Internet, online game, IoT):
  - Because designed for batch processing
- Now Hadoop with Spark/Flink is enough to hands low-latency

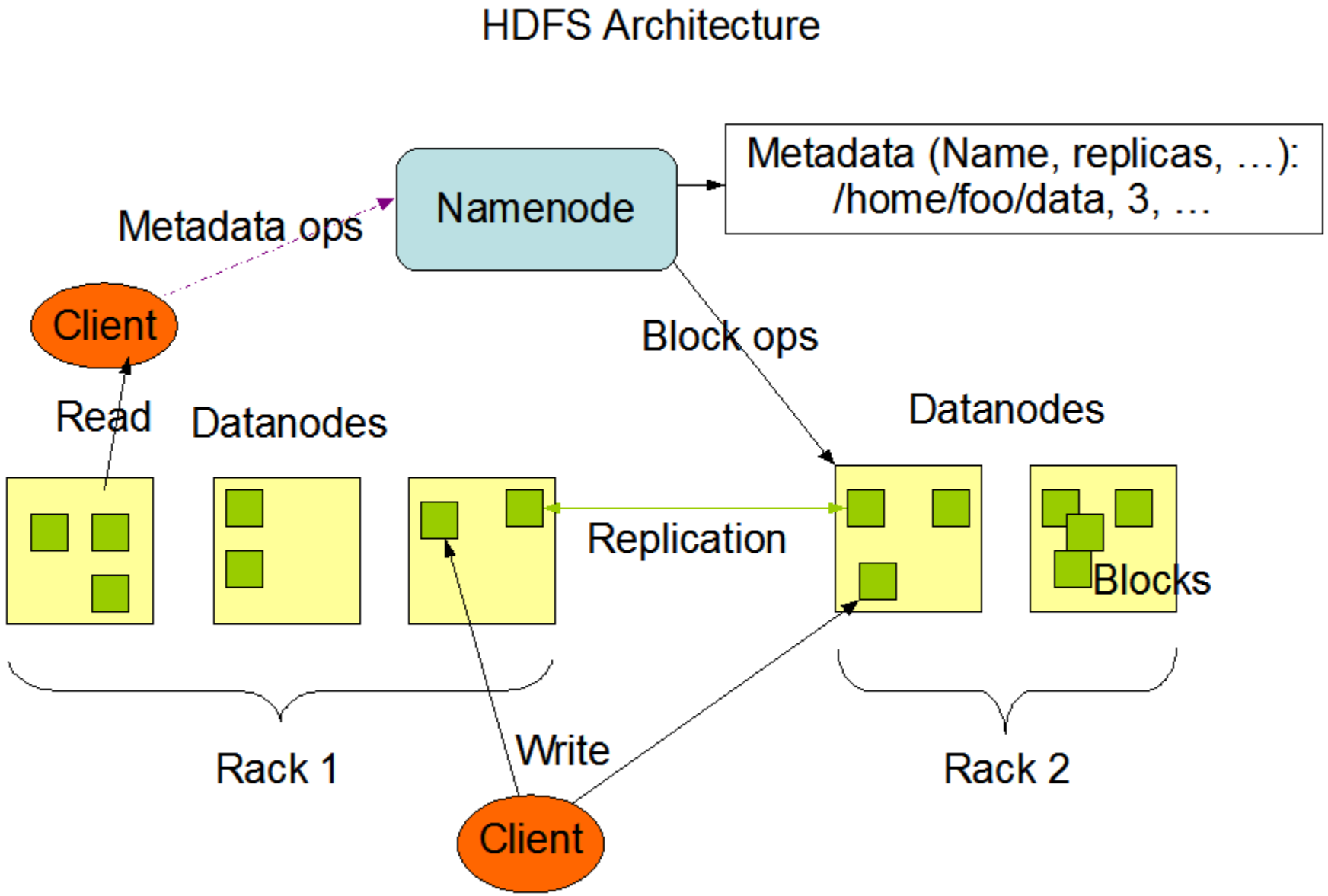
HDFS

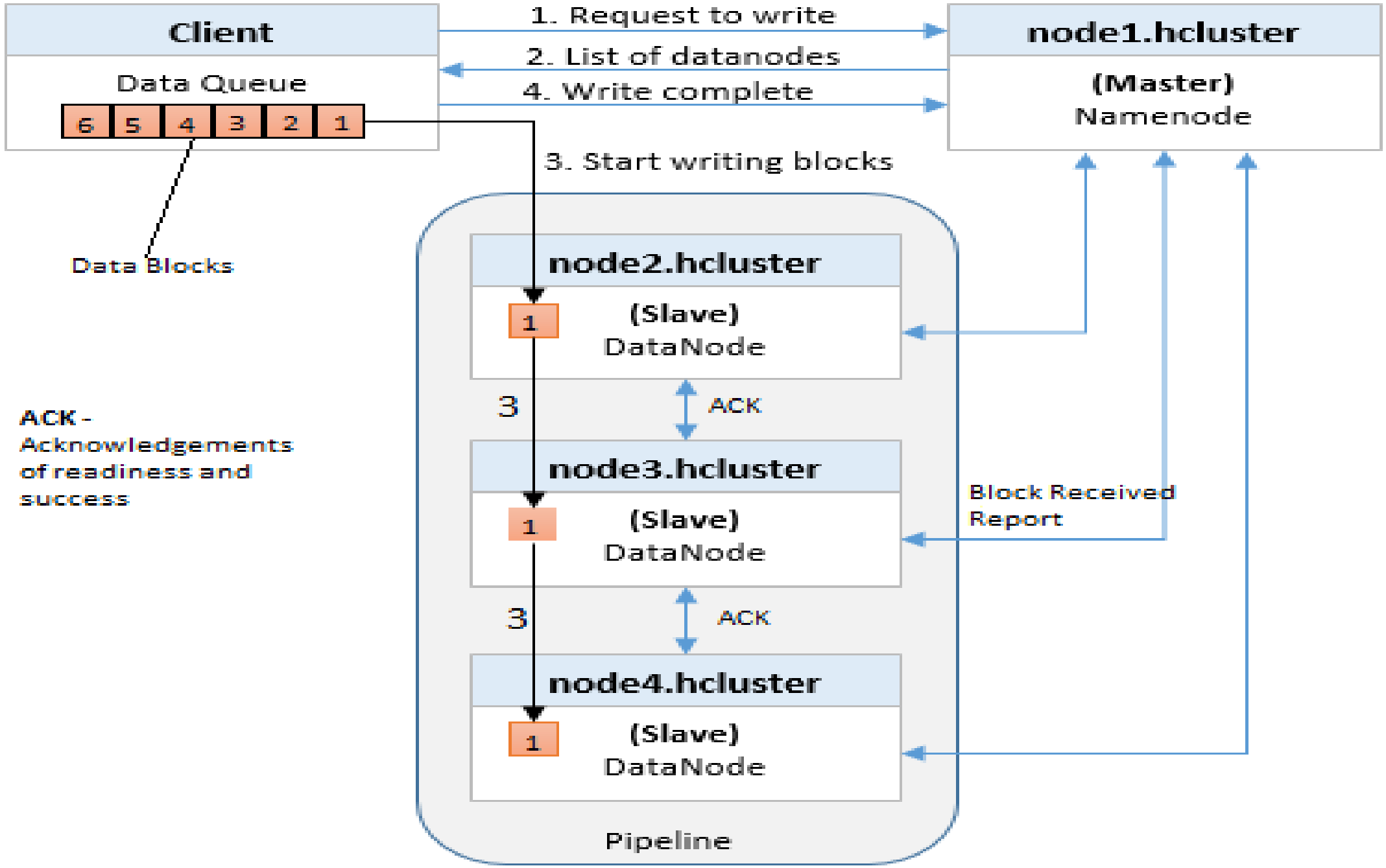
- Not a good-fit for lots of small files:
  - Because the NameNode holds HDFS metadata in memory - the limit to the number of files is governed by the amnt of memory on namenode.
- ROT:
  - Each block metadata takes about 150 bytes on namenode
  - Each file inode is about 150 bytes as well
  - Total per file with 1 block = 300 bytes
- If 1 million files, each taking one block, you would need at least 300 MB of memory (calculate and verify).

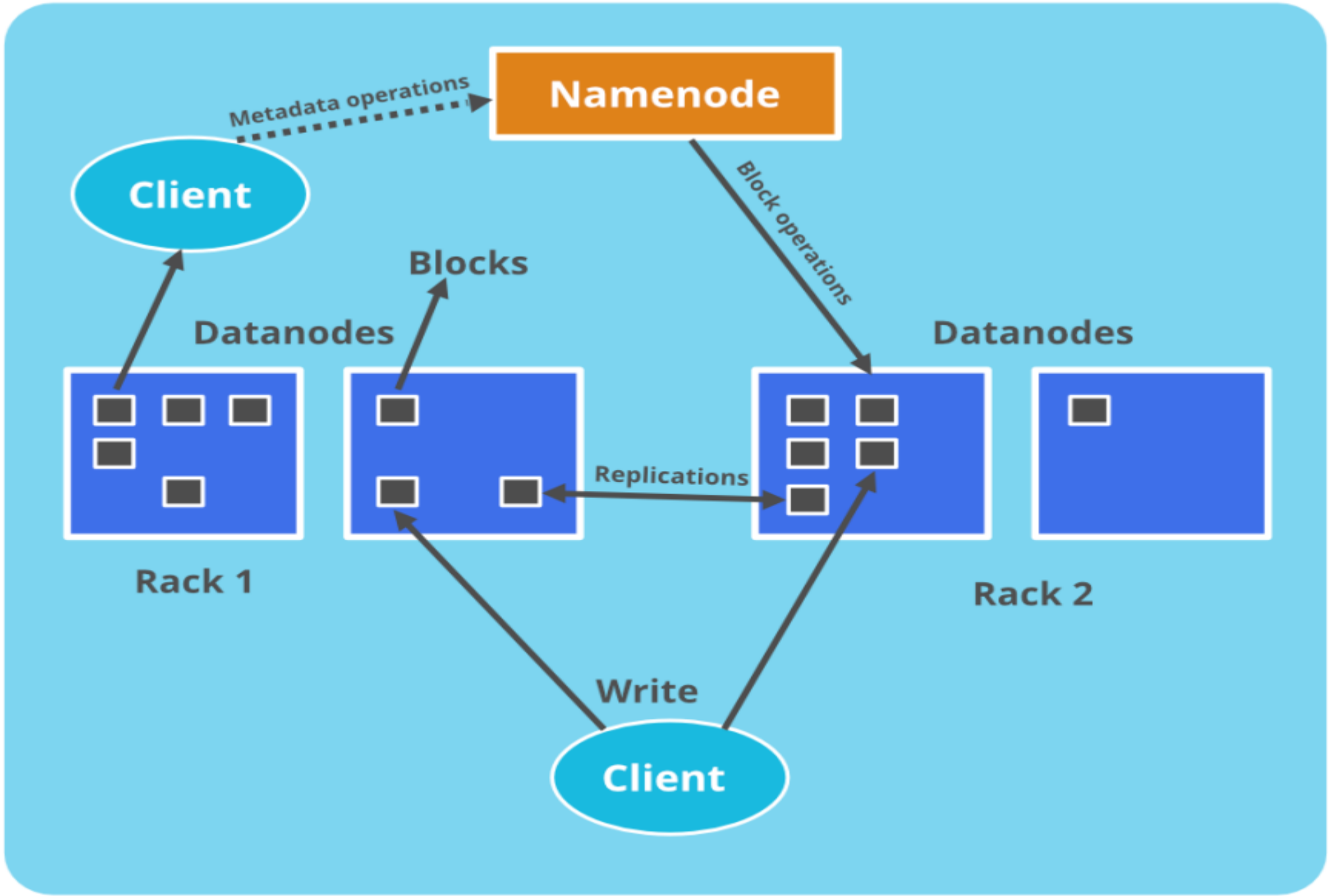
Components

- An HDFS cluster: NameNode (the master) and a number of datanodes (workers).
- The NameNode manages the filesystem (tree) namespace.
- This is metadata which is stored as two files: the namespace image (fsimage) and the edit log (edits)
- FSimage: snapshot (checkpoint) of the entire HDFS namespace
- Edits: transaction log that records every change or modification to the HDFS namespace after the last fsimage checkpoint
- Current state = fsimage + edit log (applied)
- Too much work for the NameNode!

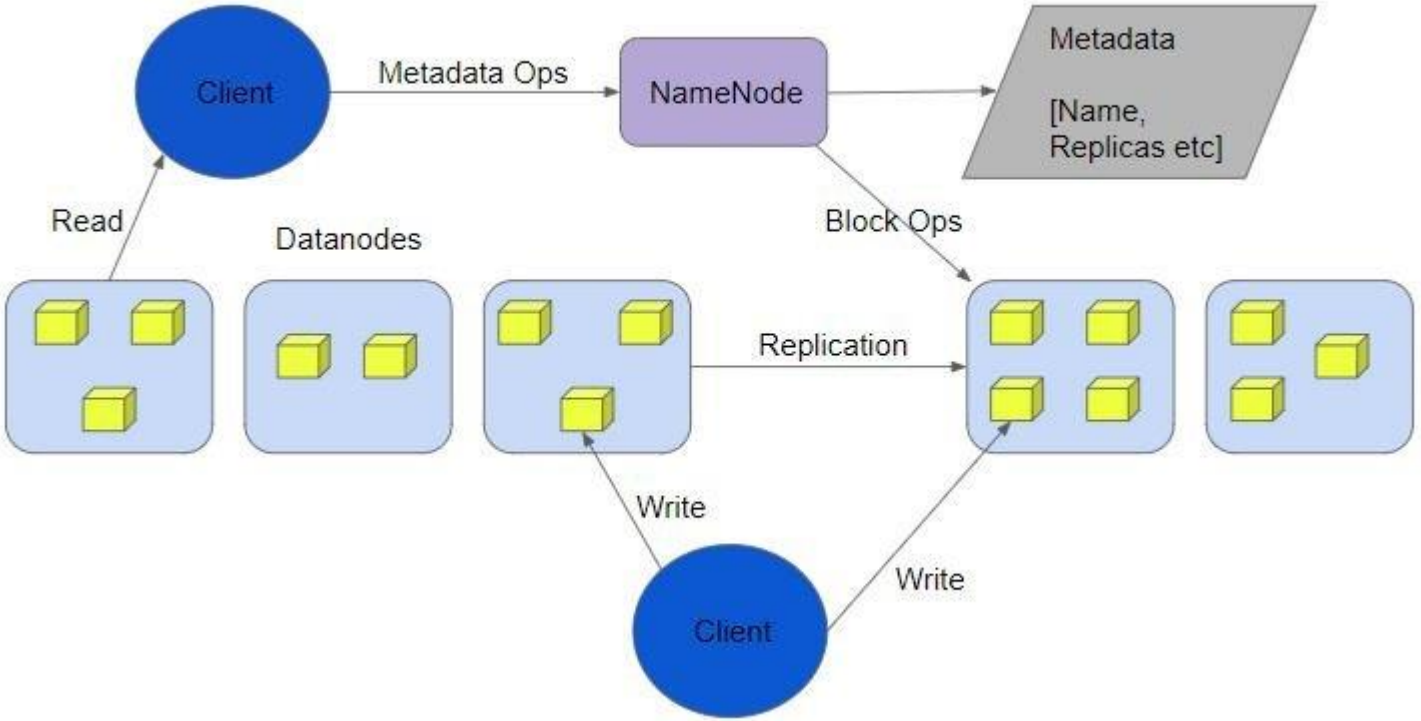
Components


HDFS Architecture
Namenode
Metadata (Name, replicas, …):
/home/foo/data, 3, …
Metadata ops
Client
Block ops
Read
Datanodes
Datanodes
Replication
Blocks
Rack 1
Write
Rack 2
Client

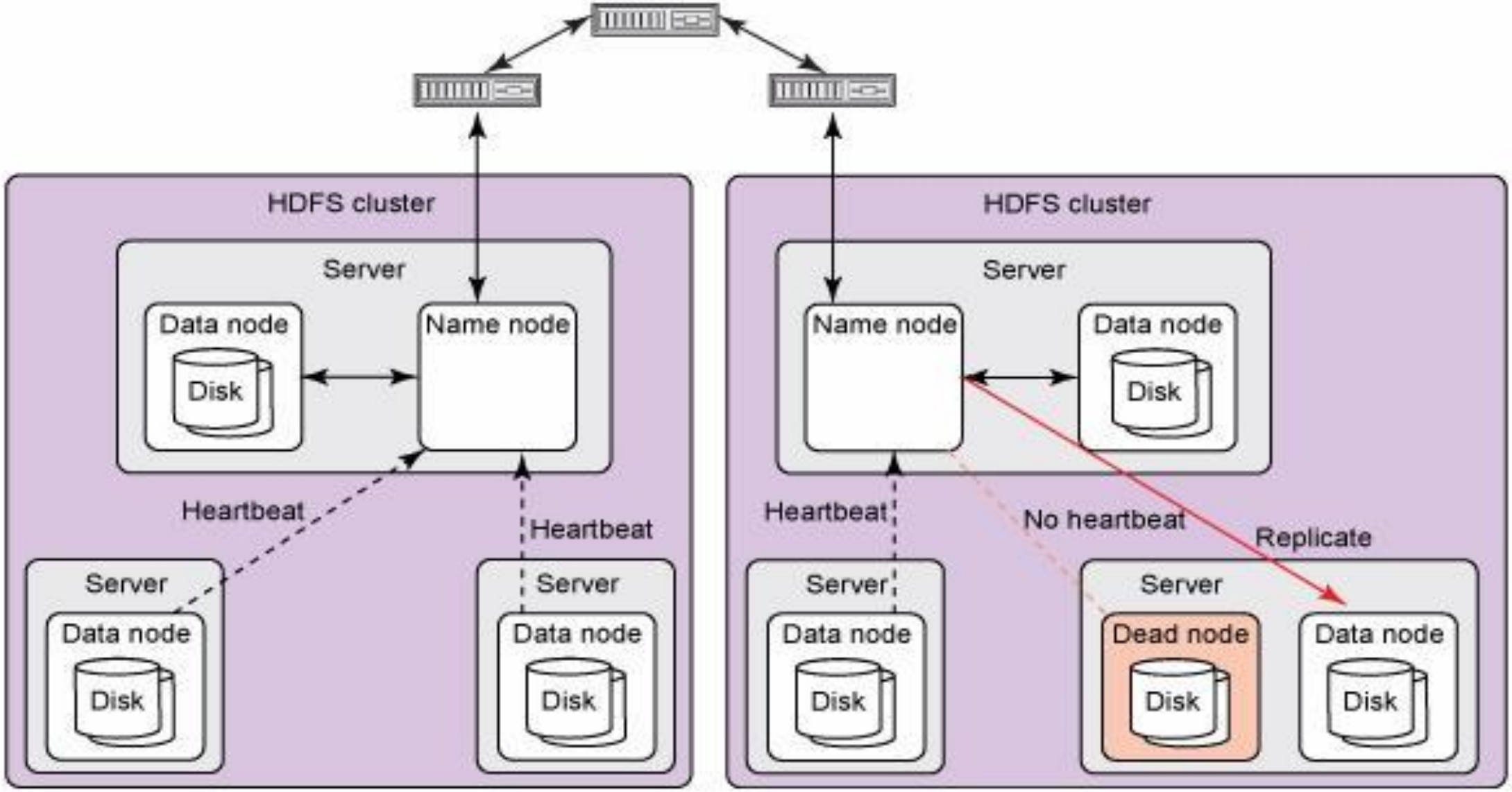

Client
Data Queue
6
5
4
3
2
1
Data Blocks
1. Request to write
2. List of datanodes
4. Write complete
node1.hcluster
(Master)
Namenode
3. Start writing blocks
node2.hcluster
1
(Slave)
DataNode
3
ACK
node3.hcluster
1
(Slave)
DataNode
3
ACK
node4.hcluster
1
(Slave)
DataNode
Pipeline
Block Received
Report
ACK -
Acknowledgements
of readiness and
success

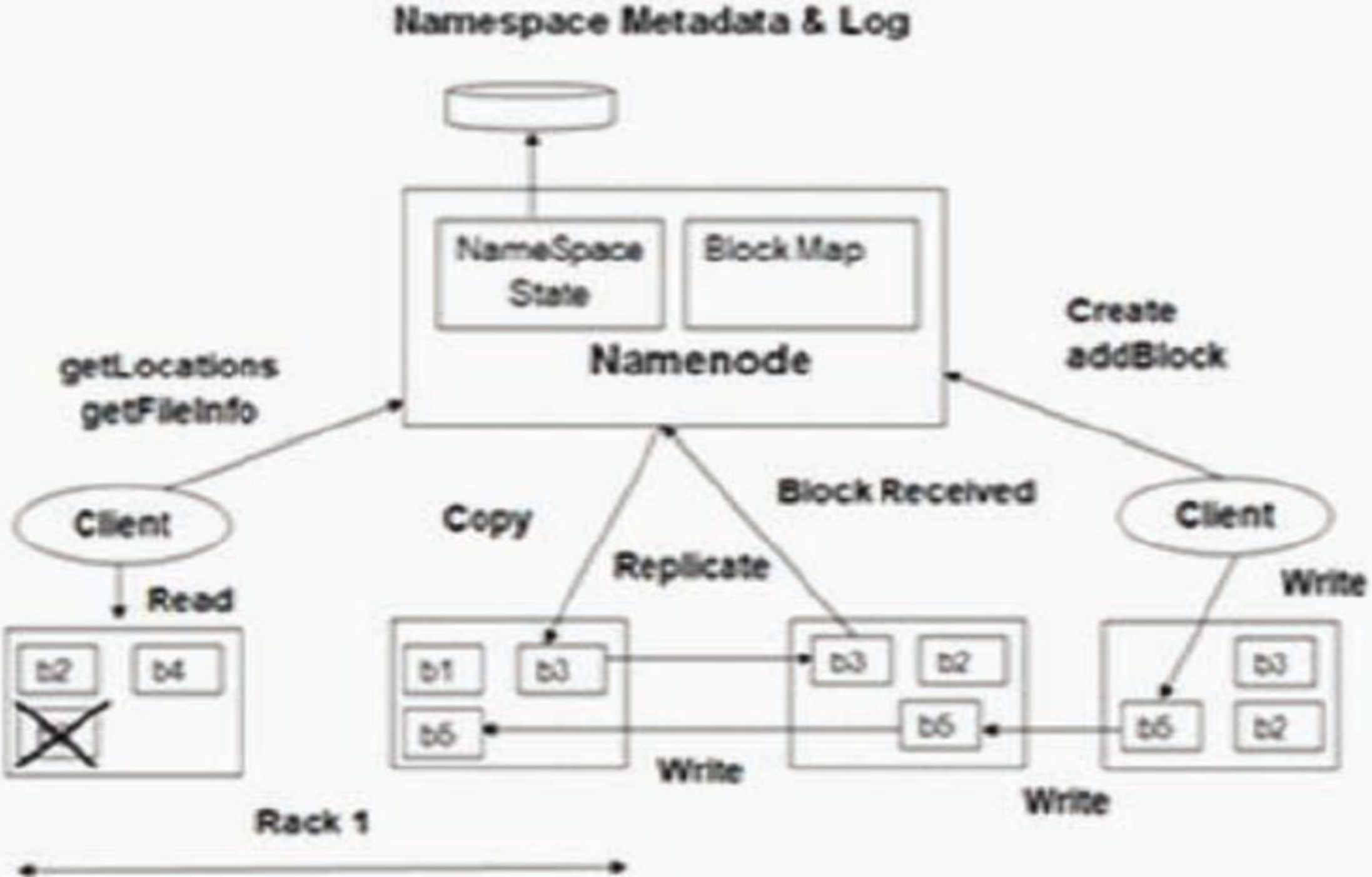

Metadata operations
Namenode
Client
Block operations
Blocks
Datanodes
Datanodes
Replications
Rack 1
Rack 2
Write
Client

Components

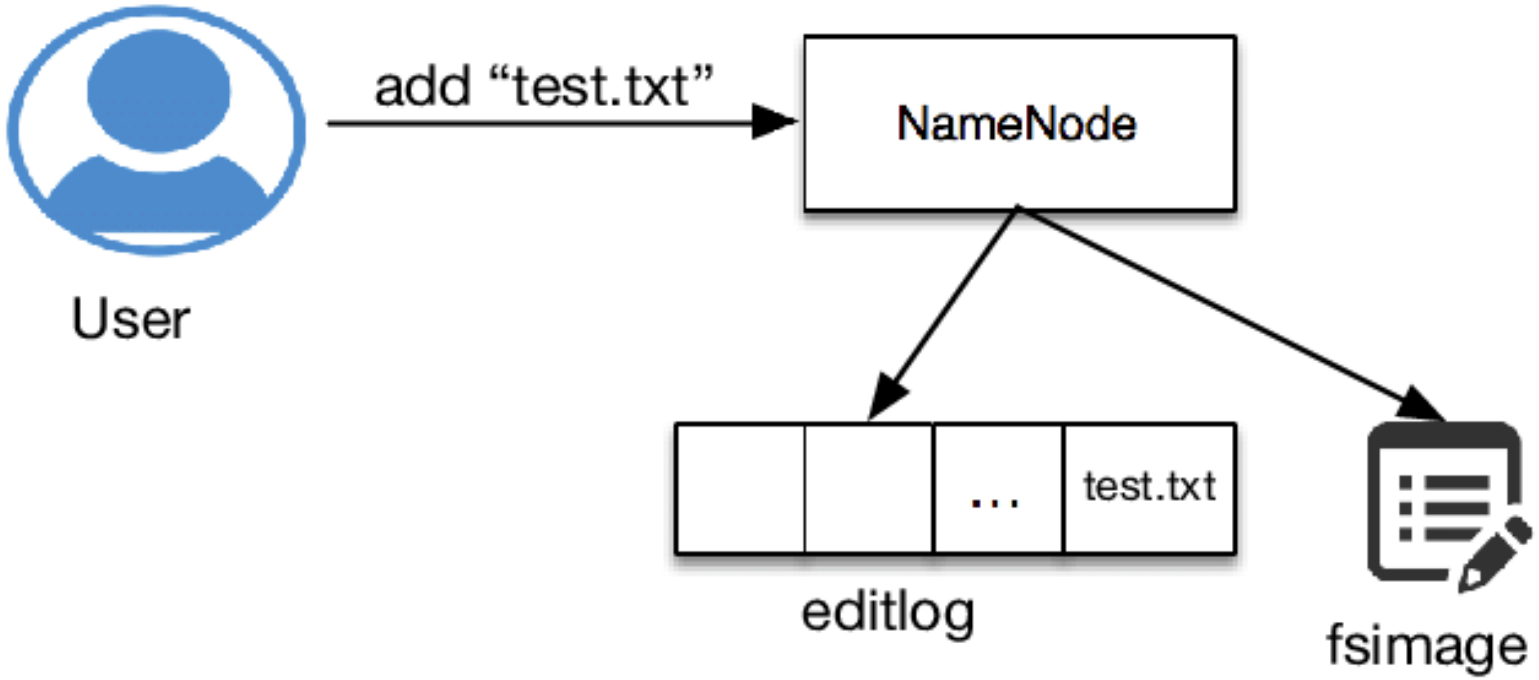

Client
Metadata Ops
NameNode
Metadata
[Name,
Replicas etc]
Read
Datanodes
Block Ops
Replication
Write
Write
Client

Components


HDFS cluster
Server
Data node
Disk
Name node
Heartbeat
Heartbeat
Server
Data node
Disk
Server
Data node
Disk
HDFS cluster
Server
Name node
Data node
Disk
Heartbeat
No heartbeat
Replicate
Server
Data node
Disk
Server
Dead node
Disk
Data node
Disk


Namespace Metadata & Log
NameSpace State
Block Map
Namenode
getLocations
getFileInfo
Create
addBlock
Client
Read
Copy
Replicate
Block Received
Client
Write
b2
b4
b1
b3
b5
b3
b2
b5
b3
b5
b2
Write
Write
Rack 1

Components



Components

- A client accesses the filesystem (RPC calls) by communicating with the name-node and datanodes
- Datanodes are workhorses: store and retrieve blocks when they are told to (by clients or namenode) - report back periodically with lists of blocks that they are storing
- Without namenode, no filesystem - if namenode machine is obliterated, all files on filesystem would be lost
- Important to make the namenode resilient to failure, and Hadoop provides two mechanisms for this

Components

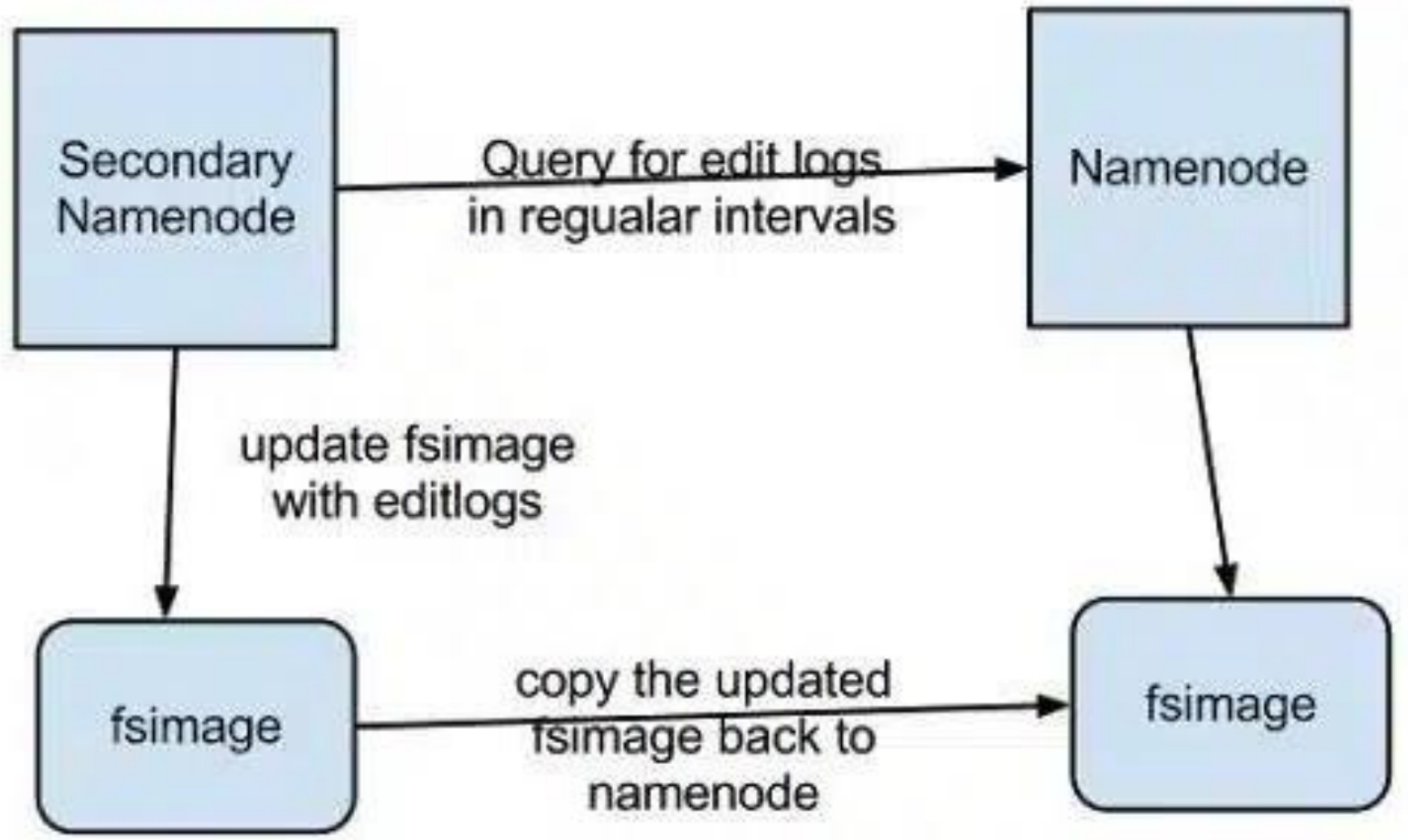
- Current state = fsimage + edit log (applied)
- Use a Secondary NameNode to help with checkpointing work!

Tasks
Download the current fsimage and edits from the NameNode (via HTTP).
Load them into memory and apply all edits to update the namespace.
Create a new merged fsimage.chkpoint file (a fresh checkpoint).
Send back this new fsimage to the NameNode (or store it locally).
The NameNode replaces its old fsimage and resets the edit log.

Secondary NameNode

- Periodically merge the namespace image with the edit log to prevent the edit log from becoming too large.
- Runs on a separate physical machine
- Keeps a copy of the merged namespace image (used in failure)
- State of secondary lags that of the primary – if total failure of the primary, then data loss is almost certain.
- Then: You can use local backup and transfer it to the secondary.

Components


Secondary Namenode
Query for edit logs in regualar intervals
Namenode
update fsimage with editlogs
fsimage
copy the updated fsimage back to namenode
fsimage

HDFS High Availability

- The combo: replicating metadata on multiple filesystems and using secondary to create checkpoints is good - but it does not provide high availability of the filesystem.
- The namenode is still a single point of failure (SPOF).
  - All clients—including MapReduce jobs—would be unable to read, write, or list files, because the namenode is the sole repository of the metadata and the file-to-block mapping.
- The whole Hadoop would effectively be out of service until a new namenode could be brought online!

NameNode and Secondary NameNode was a good combo

But when the NameNode failed:

- HDFS became out of access
- Data Nodes could not be reached
- Copying metadata from Secondary was slow

HDFS High Availability

- We need a new primary namenode with one of the filesystem metadata replicas and to configure datanodes and clients to use this new namenode.
- The new namenode:
  - Loads its namespace image into memory
  - Replays its edit log
  - Receives enough block reports from the datanodes to leave safe mode.
- On large clusters (many files and blocks), the time it takes for a namenode to start from cold can be 30 minutes or more.
- The long recovery time is a problem for routine maintenance - since unexpected failure of the namenode is so rare

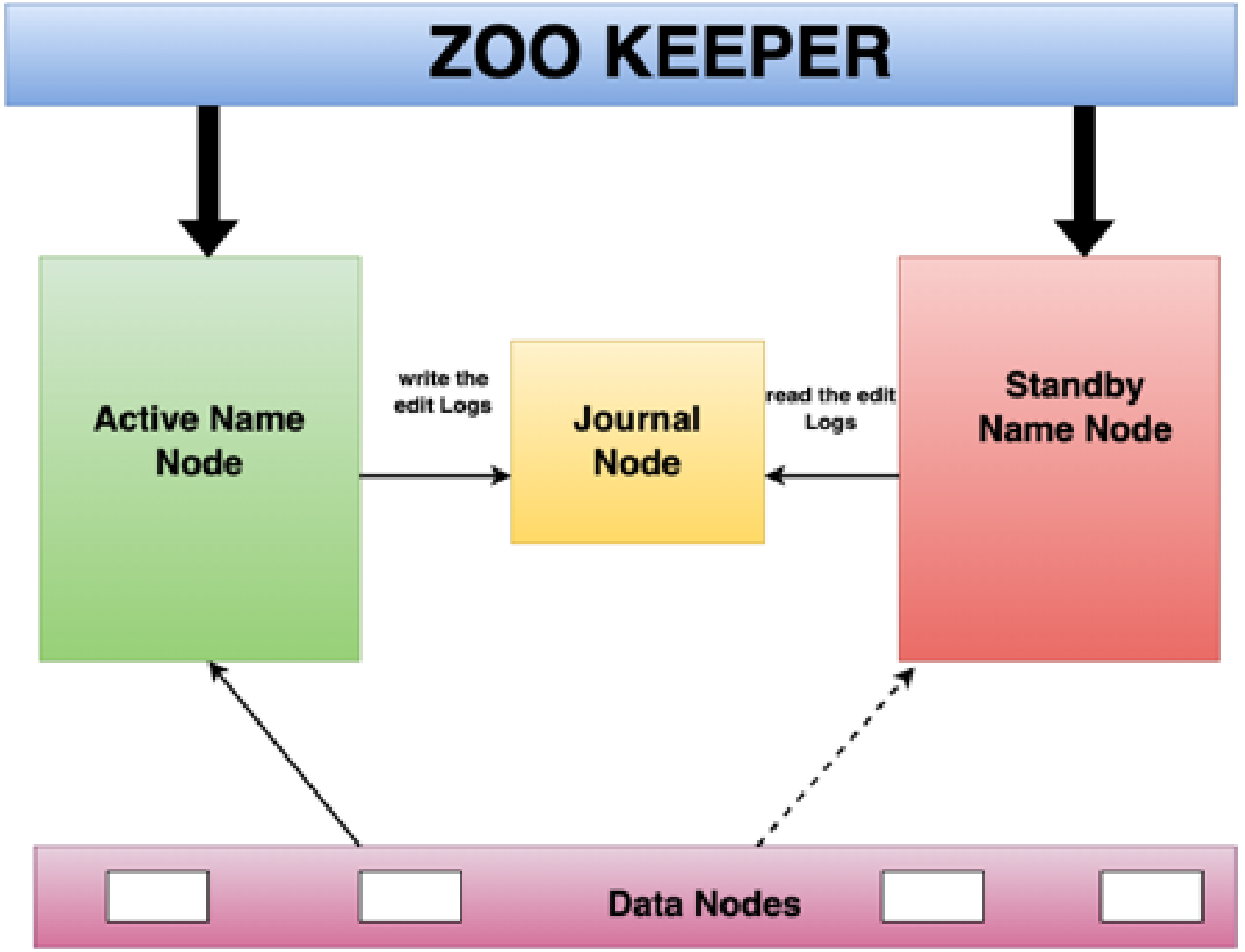
HDFS High Availability

- Hadoop 2: adds support for HDFS high availability (HA).
- A pair of namenodes in an active-standby configuration.
- In the event of the failure of the active namenode, the standby takes over its duties to continue servicing client requests without a significant interruption.

HDFS High Availability

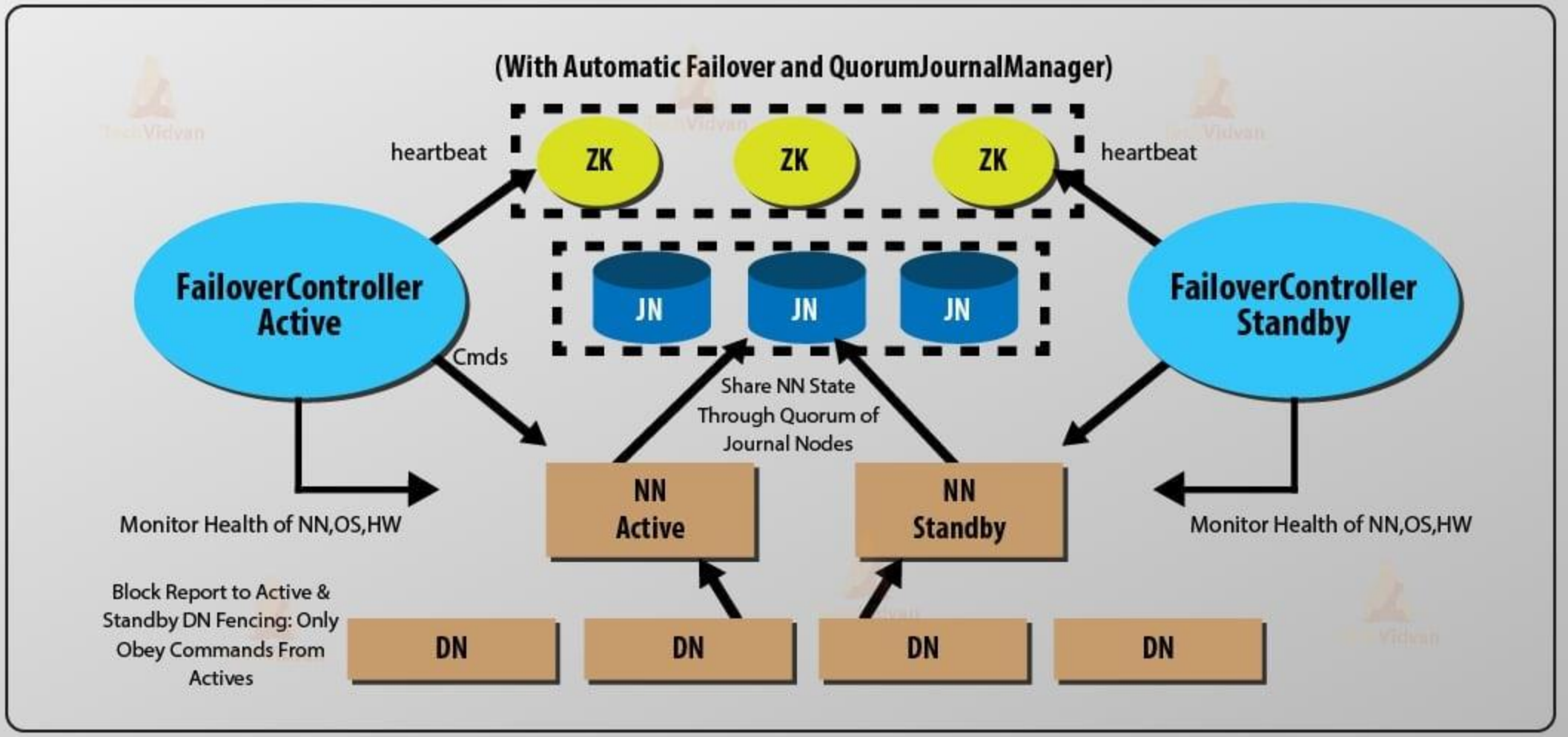
- Namenodes must use highly available shared storage to share the edit log.
- When a standby comes up, it reads up to the end of the shared edit log to synchronize its state with the active namenode
- Then continues to read new entries as they are written by the active namenode.
- Datanodes must send block reports to both namenodes because the block mappings are stored in a namenode's memory
- Clients must be configured to handle namenode failover
- The secondary namenode's role is consumed by the standby node

Components

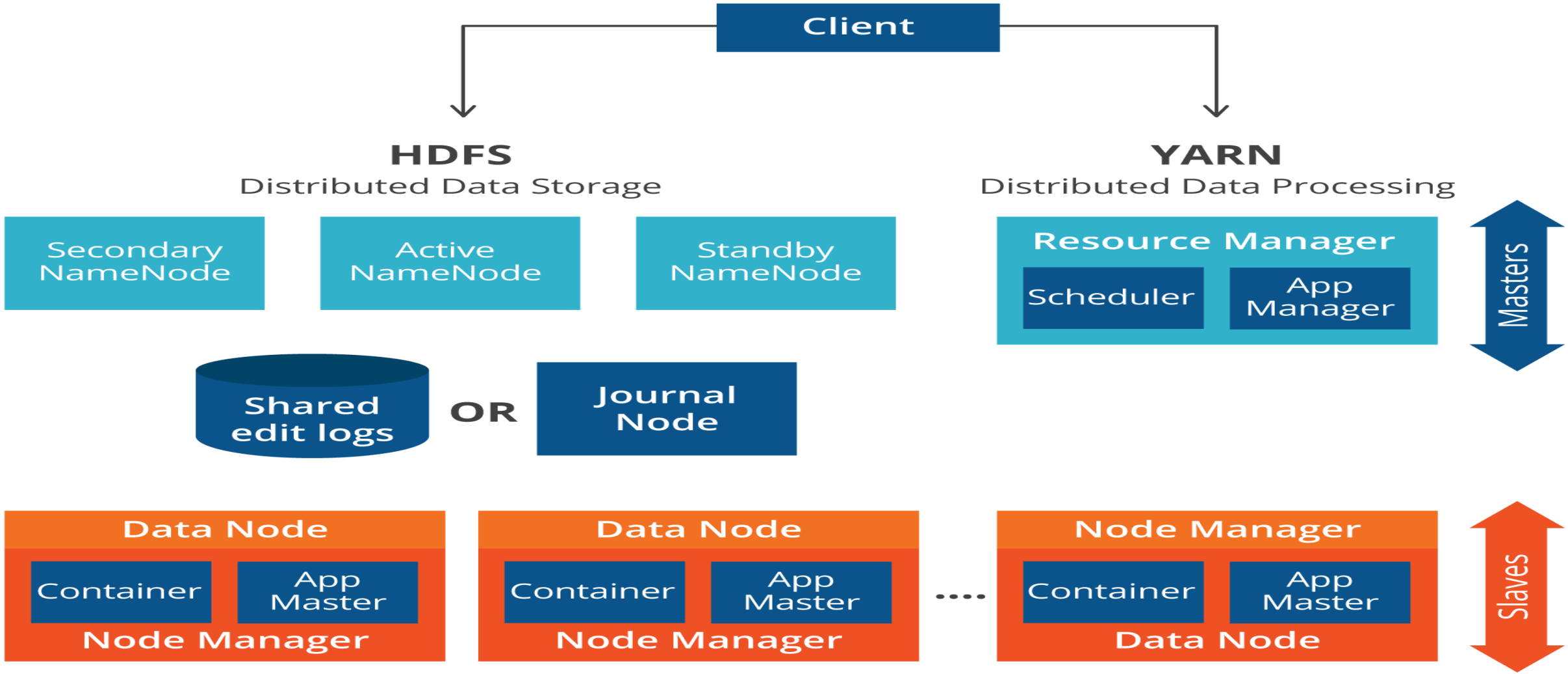

ZOO KEEPER
Active Name Node
write the edit Logs
Journal Node
read the edit Logs
Standby Name Node
Data Nodes

TechVidvan

HDFS NameNode HA Architecture



Apache Hadoop 2.0 and YARN



Component	Description
Active NameNode	Master node that manages the file system namespace and client access. It knows where every block is stored across DataNodes.
Standby NameNode	A hot backup of the Active NameNode that keeps an up-to-date copy of metadata. It can instantly take over if the Active fails.
Secondary NameNode	Checkpointing node (not a backup) - periodically merges Active's edits and fsimage to keep metadata compact (**through slow HTTP requests**)
DataNodes	Worker nodes that store actual data blocks. They regularly send heartbeats and block reports
JournalNodes (for HA)	Used in HA setups to synchronize metadata **in real-time** between Active and Standby. They store edit logs in a quorum (usually 3 or 5 nodes – has to be odd number).

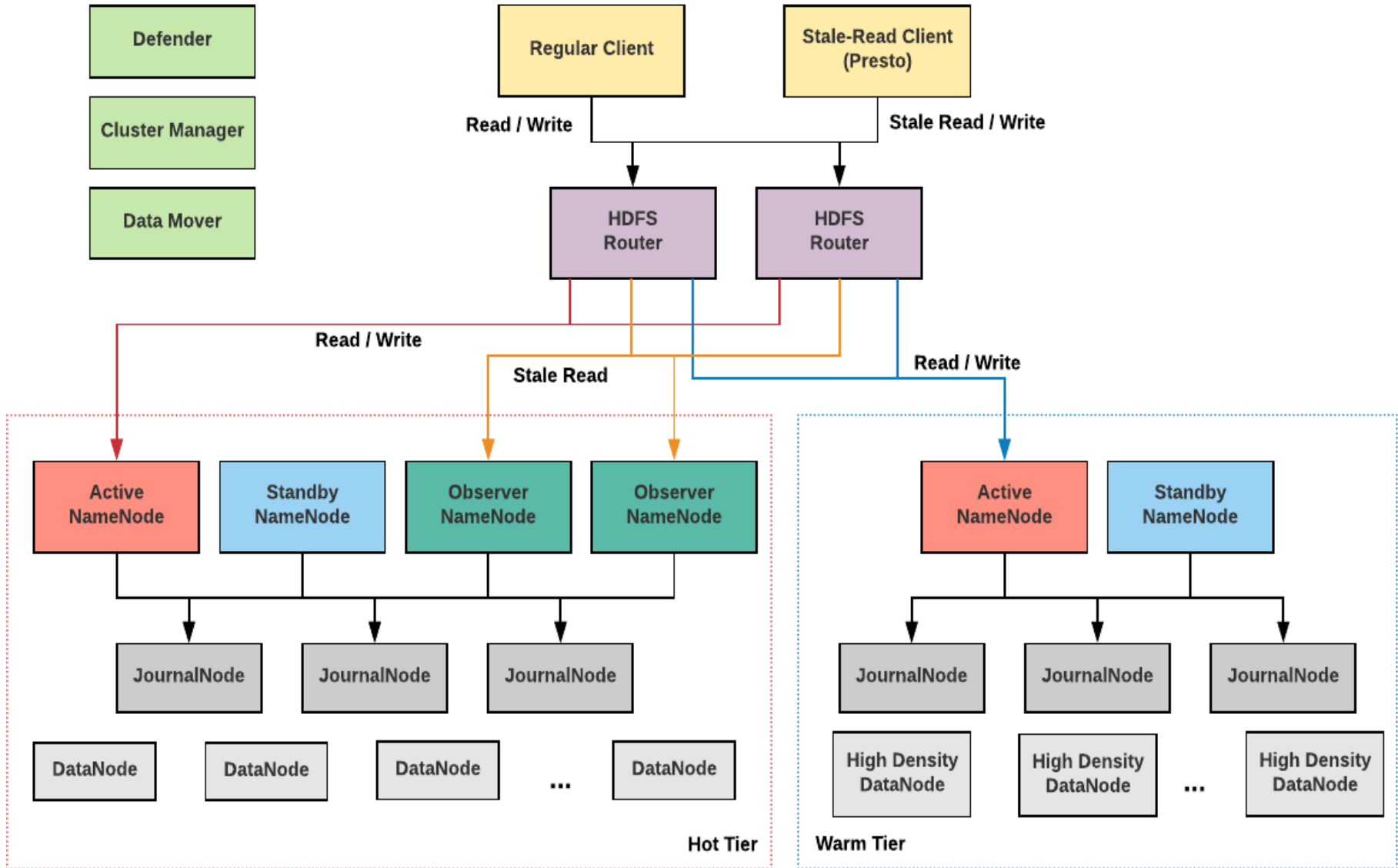
⚙️ Workflow:

1. Cluster starts:
    - Both NameNodes (Active & Standby) load same `fsimage` from shared storage or prior synchronization.
2. **Active NameNode**:
    - Handles **all client requests** (reads & writes).
    - Writes every metadata change (edit) to the **JournalNodes quorum**.
3. **Standby NameNode**:
    - Monitors JournalNodes.
    - Continuously **reads edits** and **applies them** to its own namespace image.
    - Maintains a **real-time, synchronized copy** of Active's metadata.
4. **ZooKeeper Failover Controller (ZKFC)**:
    - Watches both NameNodes' health.
    - If Active fails → ZKFC automatically triggers failover:
        - Demotes failed Active.
        - Promotes Standby to new Active.
5. Clients are automatically redirected to the new Active NameNode.
6. ✅ **High Availability achieved** — no downtime, no manual recovery.
7. ❌ But still only one node (Active) serves **read requests** → potential bottleneck in read-heavy workloads.

All good but:

Only Active was serving both read and write requests
Became a real bottleneck in big clusters

So, now, let’s keep an observer node for read.
It is a read-only NameNode
Slightly lags behind Active!


Defender
Cluster Manager
Data Mover
Regular Client
Stale-Read Client (Presto)
Read / Write
Stale Read / Write
HDFS Router
HDFS Router
Read / Write
Stale Read
Read / Write
Active NameNode
Standby NameNode
Observer NameNode
Observer NameNode
JournalNode
JournalNode
JournalNode
DataNode
DataNode
DataNode
...
DataNode
Hot Tier
Active NameNode
Standby NameNode
JournalNode
JournalNode
JournalNode
High Density DataNode
High Density DataNode
...
High Density DataNode
Warm Tier

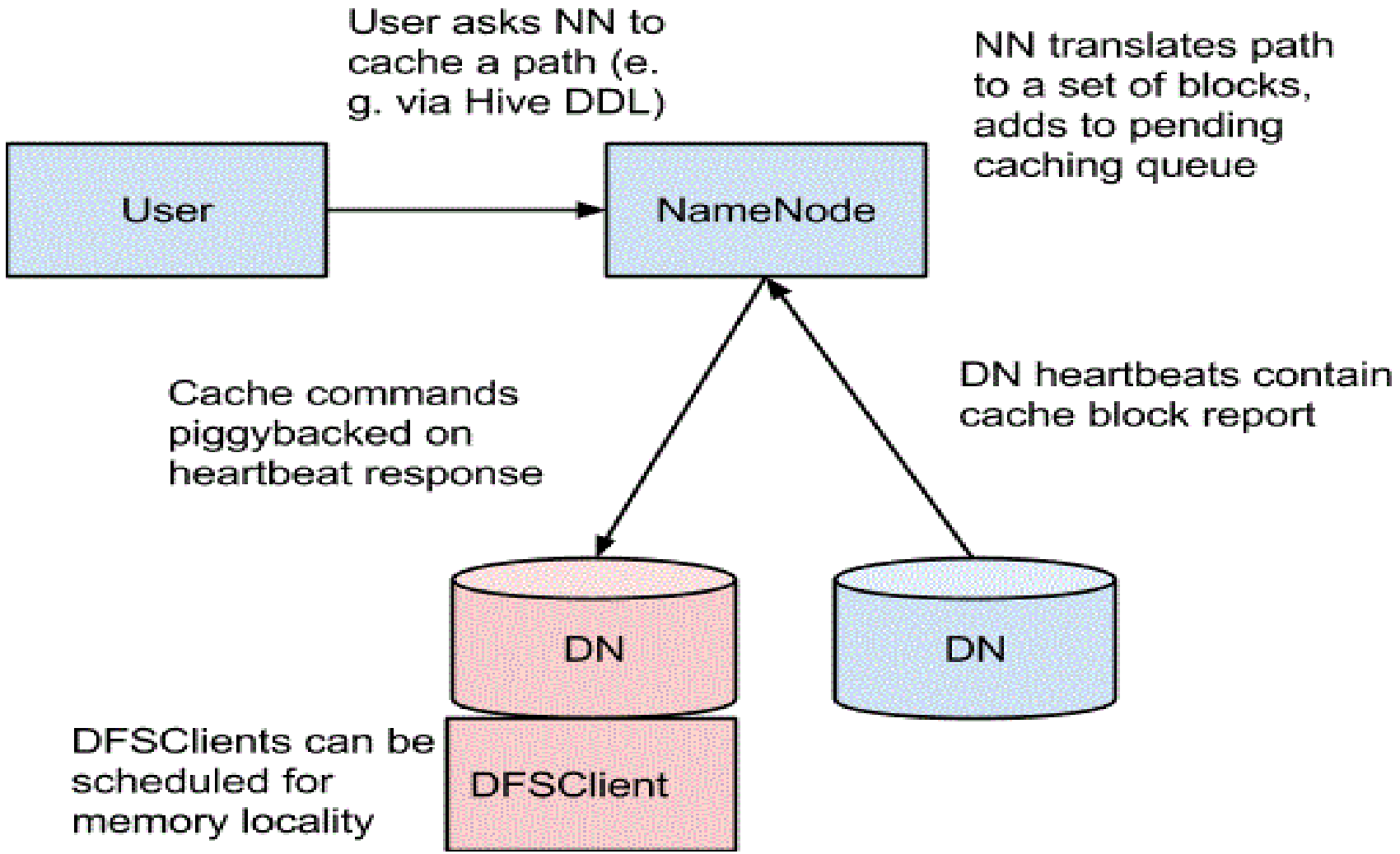
⚙️ Workflow:

1. **Startup phase:**
   - Active, Standby, and Observer(s) all load the same `fsimage`.
   - All connect to **JournalNodes**.
2. **Active NameNode**:
   - Handles **all write and coordination operations**.
   - Writes every metadata change (edit) to JournalNodes.
3. **Standby NameNode**:
   - Continuously reads and applies new edits from JournalNodes.
   - Ready to **take over automatically** if Active fails.
4. **Observer NameNode(s)**:
   - Also read the same edit logs from JournalNodes asynchronously.
   - Keep an *almost up-to-date* copy of namespace.
   - Can serve **read-only client requests** (`ls`, `stat`, `getBlockLocations`).
5. **Client behavior**:
   - Read operations → sent to **Observer** (for better performance).
   - Write operations → sent to **Active**.
   - If Observer is lagging or unavailable, client falls back to Active.
6. **Consistency model**:
   - Slight lag possible (eventual consistency for reads).
   - Clients requiring "read-your-write" consistency can request Active explicitly.
7. **Failover**:
   - ZKFC still monitors Active and Standby only.
   - Observers remain read-only — they don't take over automatically.
8. ✅ **Result:**
   - High Availability (like Hadoop 2.x)
   - Plus **Read Scalability** (thanks to Observers)
   - Perfect for **large-scale, read-heavy** clusters (e.g., Hive, Spark, Impala).

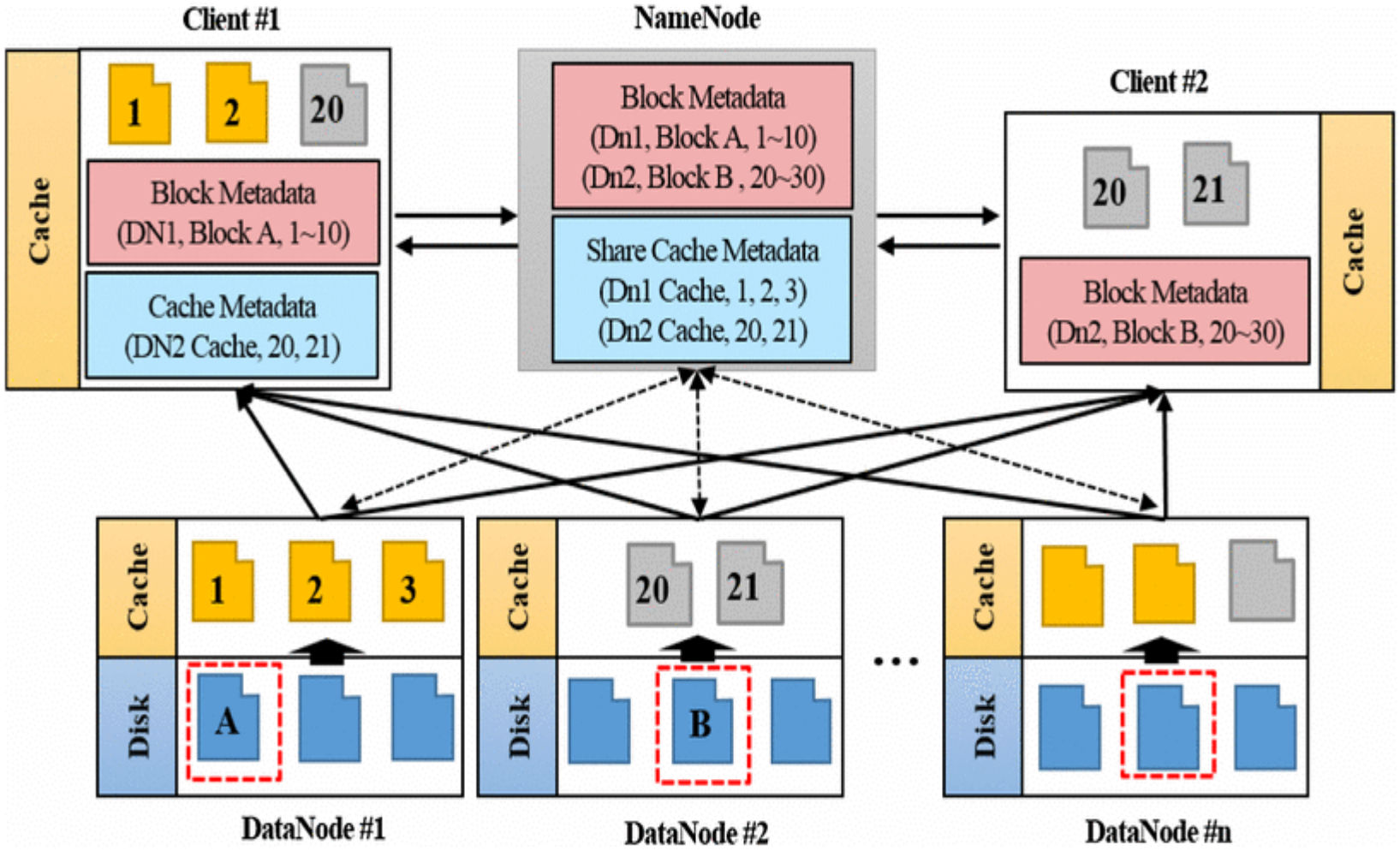
Note:

- You can also back up the files that make up the persistent state of namenode metadata.
- hdfs dfsadmin -fetchImage /backup/fsimage
- hdfs dfsadmin -saveNamespace
- If both Active, Standby and Observers are lost, you can restore the cluster using this offline backup.

Block Caching

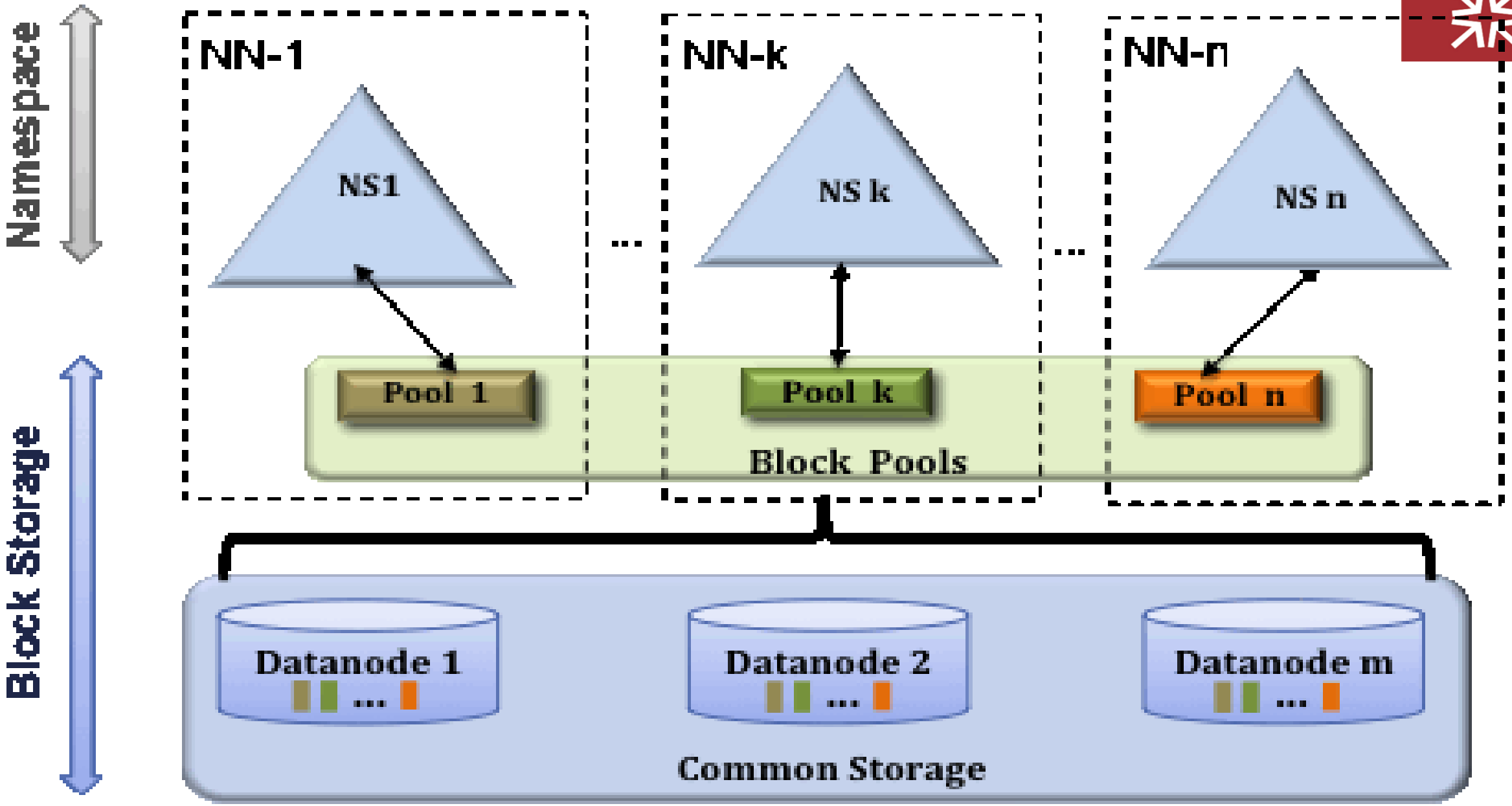


Components


Client #1
NameNode
Client #2
Cache
1
2
20
Block Metadata
(DN1, Block A, 1~10)
Cache Metadata
(DN2 Cache, 20, 21)
Block Metadata
(Dn1, Block A, 1~10)
(Dn2, Block B , 20~30)
Share Cache Metadata
(Dn1 Cache, 1, 2, 3)
(Dn2 Cache, 20, 21)
20
21
Block Metadata
(Dn2, Block B, 20~30)
Cache
Cache
1
2
3
Disk
A
Cache
20
21
Disk
B
Cache
Disk
DataNode #1
DataNode #2
DataNode #n

Block Caching

- A datanode reads blocks from disk
- For frequently accessed files the blocks may be explicitly cached in the datanode's memory, in an off-heap block cache.
- A block is cached in only one datanode's memory, although the number is configurable on a per-file basis.
- Job schedulers (for MapReduce, Spark) take advantage of cached blocks by running tasks on the datanode where a block is cached
- A small lookup table used in a join is a good candidate for caching, for example.


Namespace
Block Storage
NN-1
NN-k
NN-n
NS1
NS k
NS n
Pool 1
Pool k
Pool n
Block Pools
Datanode 1
Datanode 2
Datanode m
Common Storage

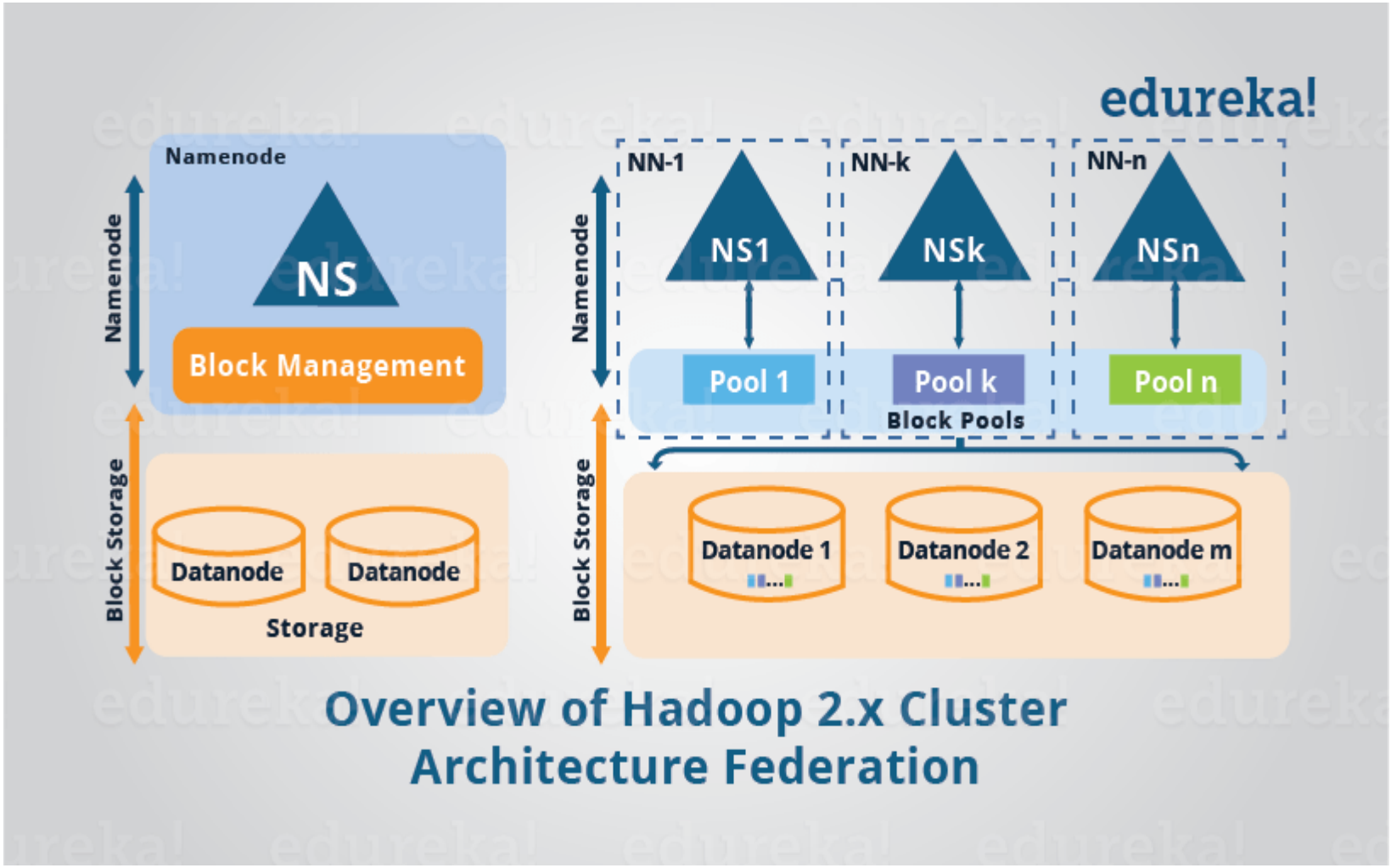
HDFS Federation

- HDFS federation, allows a cluster to scale by adding namenodes, each of which manages a portion of the filesystem namespace.
- For example, one namenode might manage all the files rooted under /user, say, and a second name- node might handle files under /share.
- Hadoop Federation allows **multiple independent NameNodes**, each managing a separate part of the namespace (called a “namespace volume”)
- This gives both **scalability** and **partial redundancy** — if one NameNode fails, others are unaffected

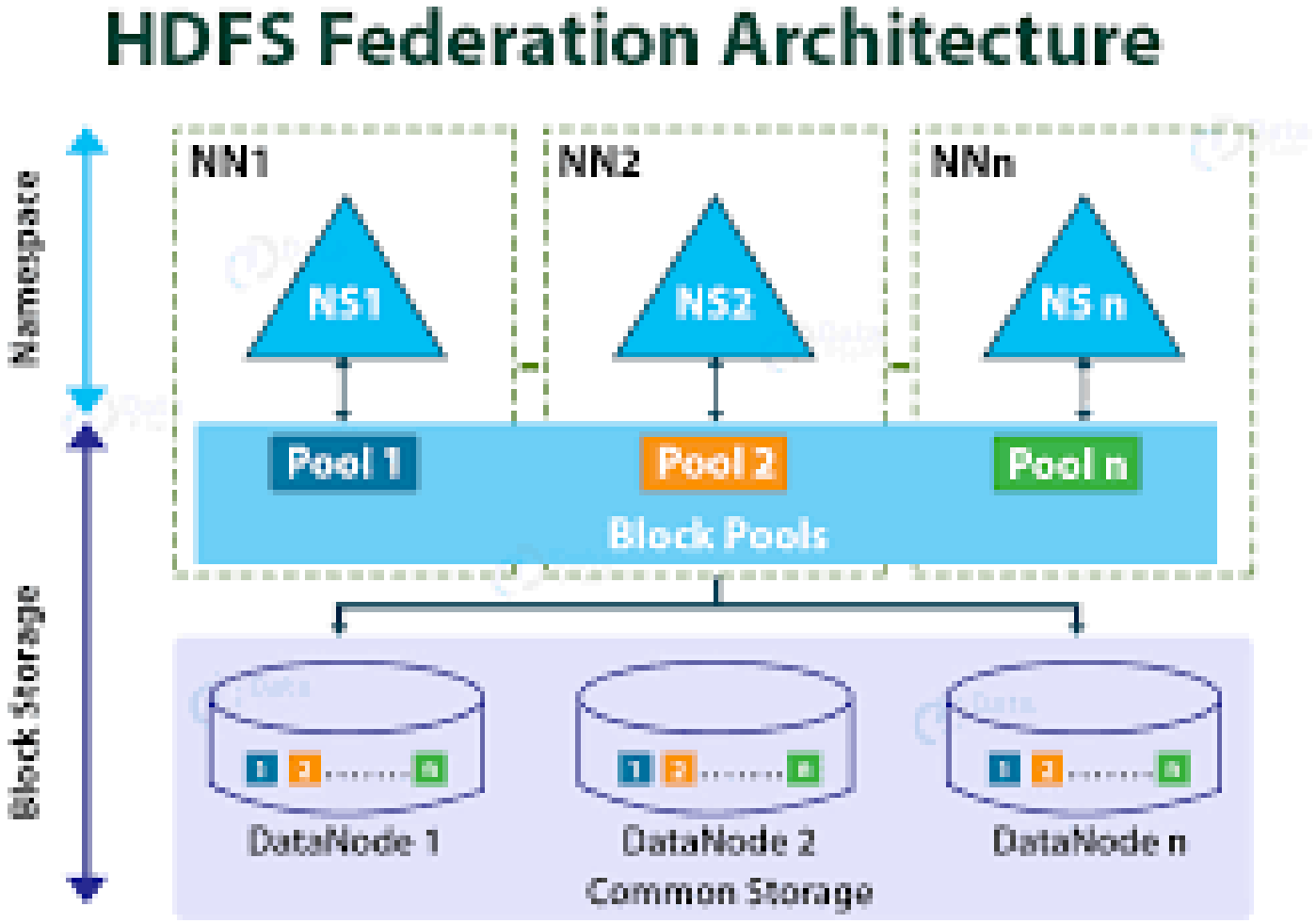
HDFS Federation

- Under federation, each namenode manages a namespace volume: metadata + block pool containing all the blocks for the files in the namespace.
- Namespace volumes are independent:
  - Namenodes do not communicate with one another
  - Failure of one namenode does not affect availability of namespaces managed by other namenodes.
- Block pool storage is not partitioned
  - Datanodes register with each namenode in the cluster and store blocks from multiple block pools

HDFS Federation



HDFS Federation



https://www.uber.com/en-PK/blog/scaling-hdfs/

Uber

We use Hadoop for both batch and streaming analytics across a wide range of use cases, such as fraud detection, machine learning, and ETA calculation.

Uber's Data Infrastructure team massively overhauled scaling HDFS by implementing new adjustments:

- View File System (ViewFs)
- Frequent HDFS version upgrades
- NameNode garbage collection tuning
- Limiting the number of small files that filter through the system
- An HDFS load management service
- A read-only NameNode replica

hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ViewFs.html?uclick_id=939db9f2-5d44-48df-85af-bb76c7f49c6a

APACHE hadoop

The Apache Software Foundation
http://www.apache.org/

Wiki | git | Apache Hadoop | Last Published: 2024-10-09 | Version: 3.4.

ViewFs Guide

Provides a way to manage multiple Hadoop file system namespaces

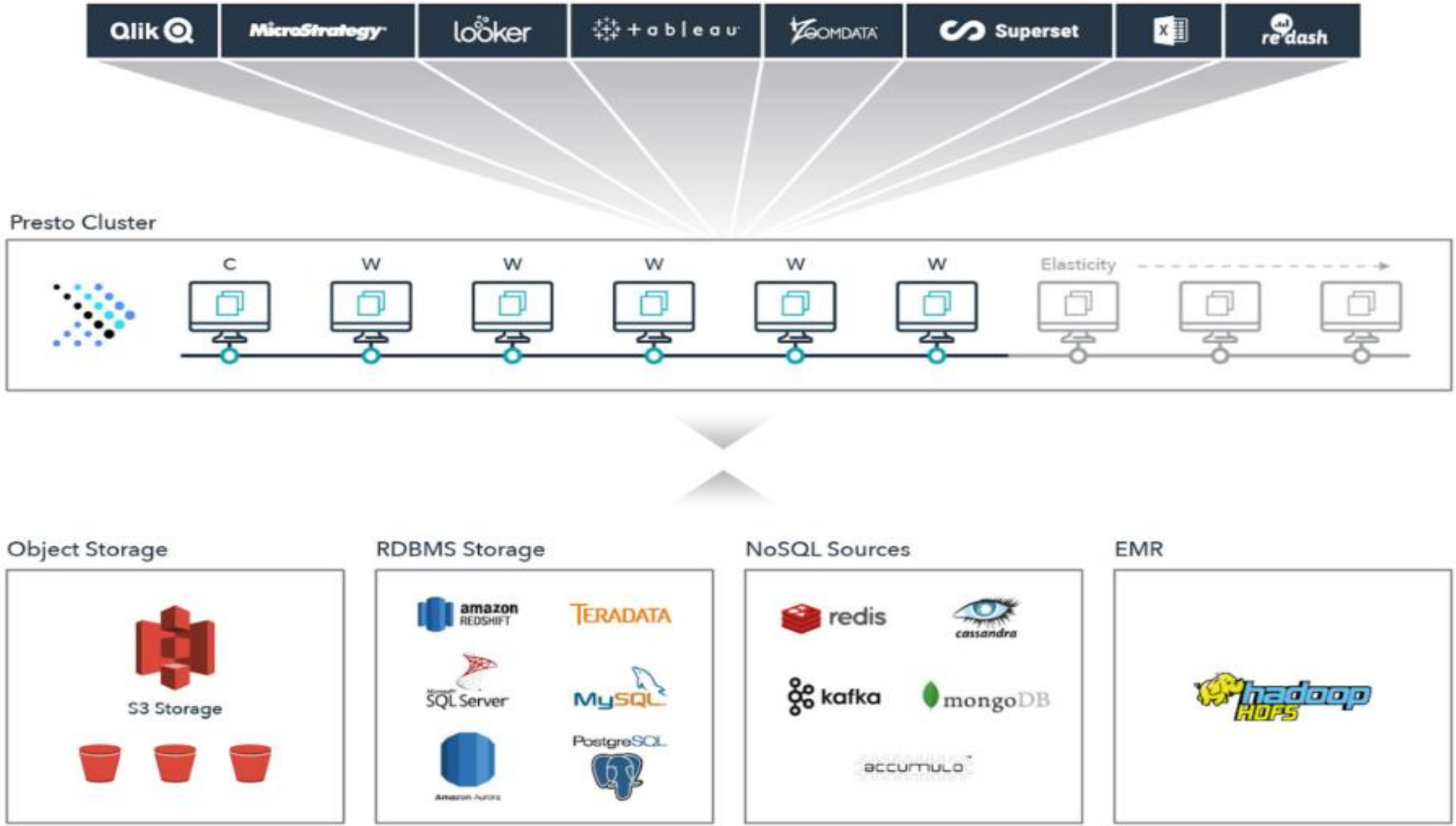
It is particularly useful for clusters having multiple namenodes, and hence multiple namespaces in the HDFS Federation

https://www.uber.com/en-PK/blog/scaling-hdfs/

Uber

HDFS was designed to support thousands of nodes within a single cluster. With enough hardware, scaling to over 100 petabytes of raw storage capacity in one cluster can be easily—and quickly—achieved.

For Uber, however, the rapid growth of our business made it difficult to scale reliably without slowing down data analysis for our thousands of users making millions of Hive or Presto queries each week.

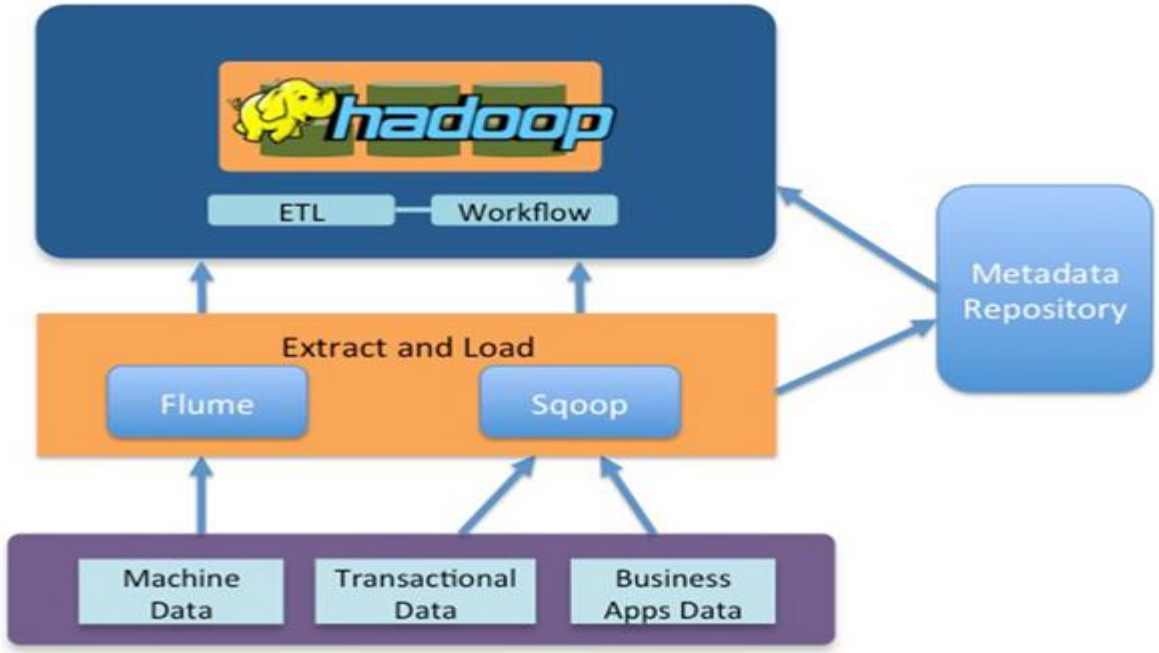

Qlik
MicroStrategy
looker
+ableau
ZOOMDATA
Superset
re dash
Presto Cluster
C
W
W
W
W
W
Elasticity
Object Storage
S3 Storage
RDBMS Storage
amazon REDSHIFT
TERADATA
Microsoft SQL Server
MySQL
Amazon Aurora
PostgreSQL
NoSQL Sources
redis
cassandra
kafka
mongoDB
accumulo
EMR
hadoop HDFS

https://www.uber.com/en-PK/blog/scaling-hdfs/

Uber

Presto accounts for more than half of the access to HDFS, and 90 percent of Presto queries take ~100 seconds to process - If our HDFS infrastructure is overloaded, Presto queries in the queue pile up (completion delays) - Also data should be available immediately.

We engineered ETL to occur in the same clusters where users run queries to reduce replication latency - but results in the generation of small files to accommodate frequent writes, which further clogs the queue.

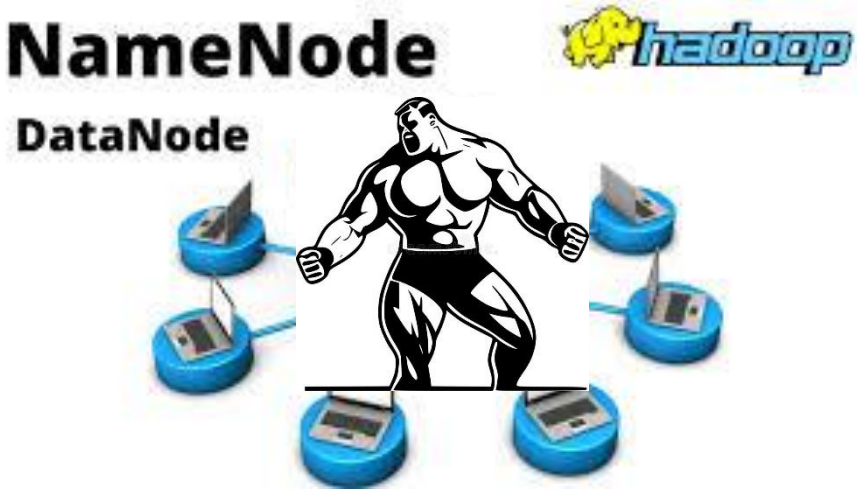


https://www.uber.com/en-PK/blog/scaling-hdfs/

Uber

Another one! Multiple teams require a large percentage of stored data, making it impossible to split clusters by use case or organization without duplications - decrease efficiency and increase costs!

The root of these slowdowns:
performance and throughput of the NameNode - the directory tree of all files in the system that tracks where data files are kept.



All metadata stored in NameNode - client requests to an HDFS cluster must first pass through it.

Uber

https://www.uber.com/en-PK/blog/scaling-hdfs/

We started experiencing high NameNode remote procedure call (RPC) queue time - NameNode queue time could exceed 500 milliseconds per request - every single HDFS request waited for at least half a second in the queue (normal = 10 ms)

https://www.uber.com/en-PK/blog/scaling-hdfs/

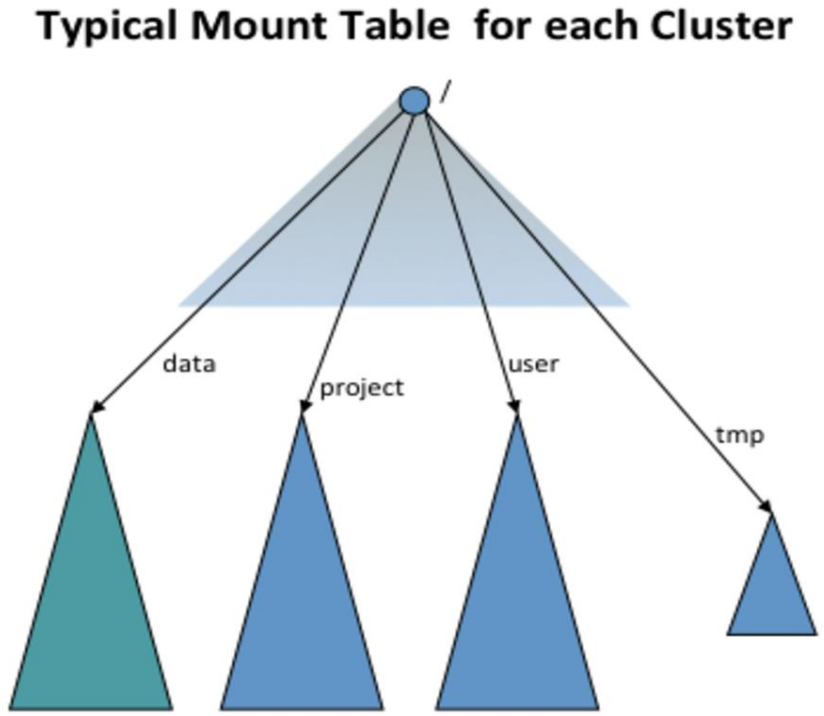
Uber

Inspired by a similar effort at Twitter, we utilized View File System (ViewFs) to split our HDFS into multiple physical namespaces and used ViewFs mount points to present a single virtual namespace to users,

Separated our HBase from HDFS cluster of the YARN and Presto operations.

Greatly reduced the load on the main cluster

Reduced the HBase cluster restart from hours to minutes.



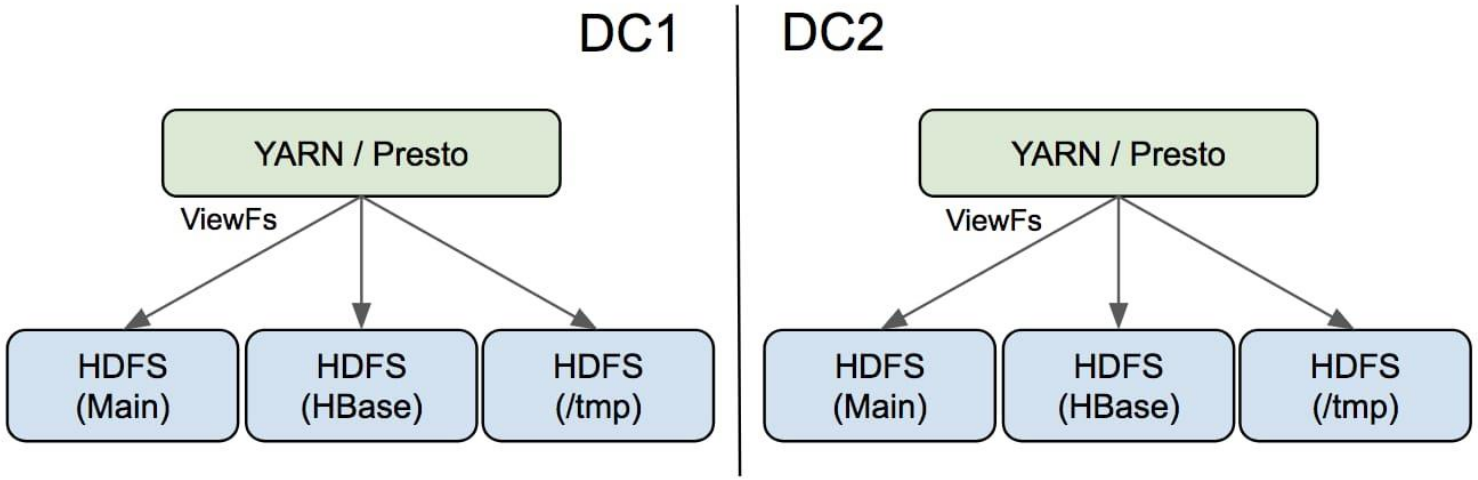
https://www.uber.com/en-PK/blog/scaling-hdfs/

Uber

Implemented separate HDFS clusters behind ViewFS instead of HDFS Federation –

Now, HDFS upgrades gradually rolled out to minimize the risk of large-scale outages; complete isolation also helps improve system reliability

Downside: separate HDFS clusters = slightly higher operational costs.



https://www.uber.com/en-PK/blog/scaling-hdfs/

Uber

A second solution for our scaling challenges was to upgrade our HDFS to keep up with the latest release versions.

Upgrading HDFS can be risky because it can cause downtime, performance degradation, or data loss.

To counter these possible issues, we spent several months validating 2.8.2 before deploying it in production.

https://www.uber.com/en-PK/blog/scaling-hdfs/

Uber

Run different versions of YARN and HDFS on the same servers at the same time - critical to scaling.

Since both YARN and HDFS are part of Hadoop, they are normally upgraded together.

However, major YARN upgrades take much longer to fully validate and roll out

Because some production applications running on YARN might need to be updated due to YARN API changes or JAR dependency conflicts between YARN and these applications.

https://www.uber.com/en-PK/blog/scaling-hdfs/

Garbage collection (GC) tuning has also played an important role in our optimization and given us much needed breathing room as we continue to scale out our storage infrastructure.

We prevent long GC pauses by forcing Concurrent Mark Sweep collectors (CMS) to do more aggressive old generation collections

For reference, our customized GC-related Java Virtual Machine (JVM) arguments for NameNode with 160GB heap size are:

- XX:+UnlockDiagnosticVMOptions
- XX:ParGCCardsPerStrideChunk=32768 -XX:+UseParNewGC
- XX:+UseConcMarkSweepGC -XX:+CMSConcurrentMTEnabled
- XX:CMSInitiatingOccupancyFraction=40
- XX:+UseCMSInitiatingOccupancyOnly
- XX:+CMSParallelRemarkEnabled -XX:+UseCondCardMark
- XX:+DisableExplicitGC

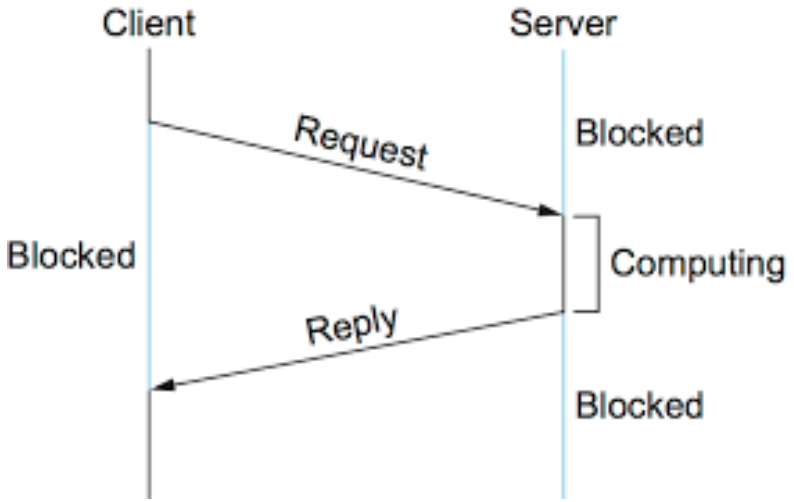
https://www.uber.com/en-PK/blog/scaling-hdfs/

Uber

NameNode loads all file metadata in memory, storing small files increases memory pressure on the NameNode - increase of read RPC calls for accessing same amount of data when clients are reading files

Built new ingestion pipelines based on our Hoodie library that generates much bigger files than those created by our original data pipelines.

‘stitcher’ - merges together small files into larger ones, mostly greater than 1GB in size.



https://www.uber.com/en-PK/blog/scaling-hdfs/

Uber

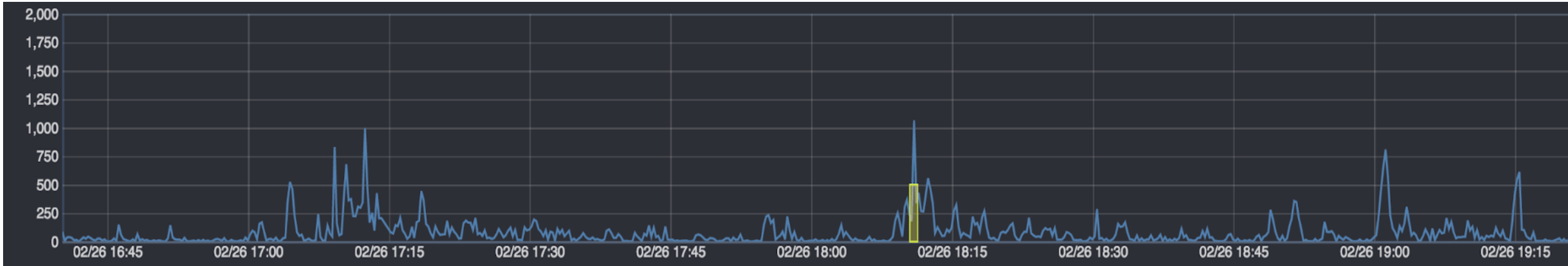
Second, we set a strict namespace quota on Hive databases and application directories - created a self-service tool for users to manage quotas within their organizations - allocated at the ratio of 256MB per file - encourage users to optimize their output file size.

Another one! Detecting which applications are causing unusually large loads, and from there, taking quick action to fix them.

We built an in-house HDFS load management service, referred to as Spotlight, for this purpose.

https://www.uber.com/en-PK/blog/scaling-hdfs/

In Spotlight, an audit log is streamed from active NameNode and processed in real time through a backend based on Flink and Kafka - output is shown on dashboard - automatically disable accounts or kill tasks causing slowdown.



Switch View (Itemized / Aggregated)

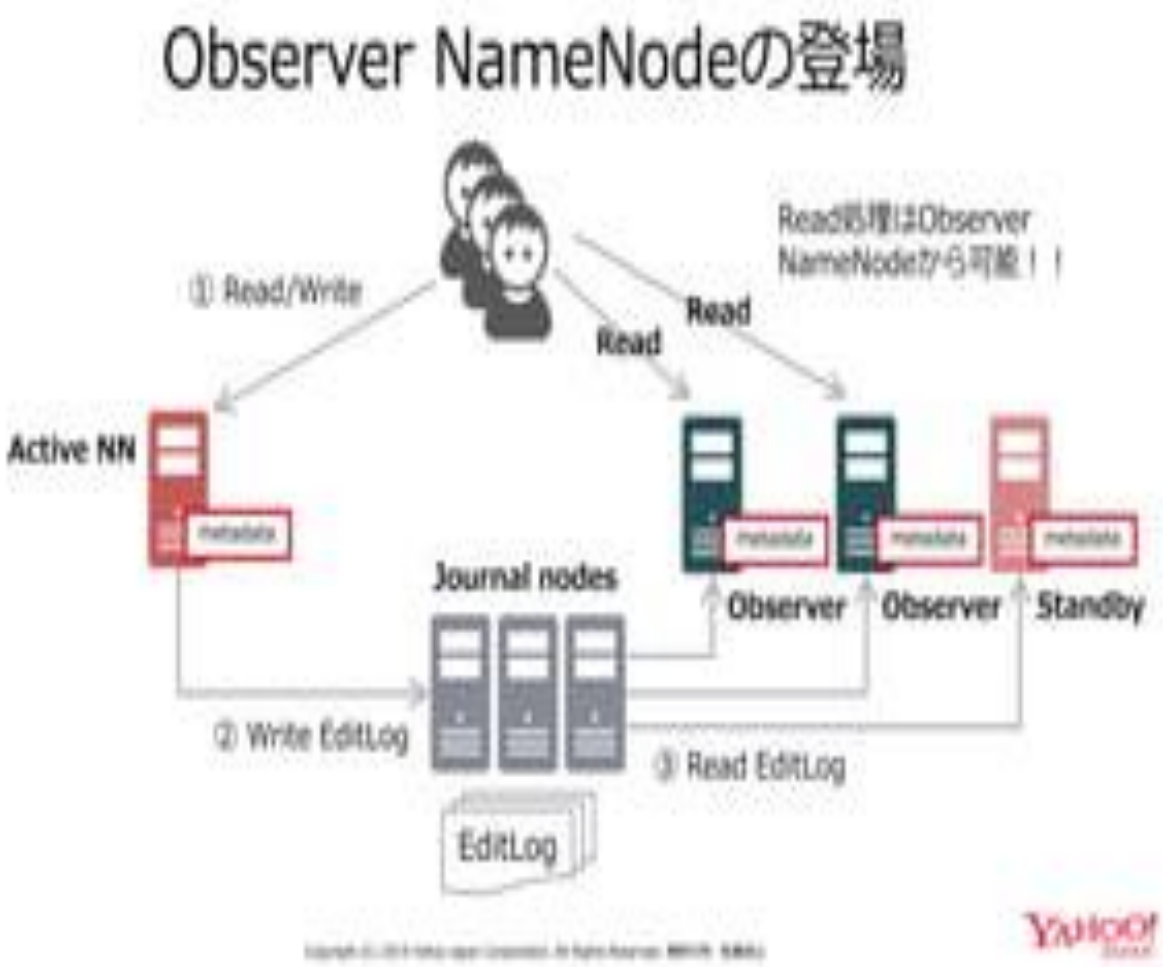
time	rpcquetime	cmd	count	ugi	directory
		contentSummary	332	hive	/user/hive/.sparkStaging
02-26 18:10:42 – 02-26 18:11:13		create	24973	hive	/tmp/hive_stagingdir/hive_hive_2018-02-27_02-02-28_397_5474197884312459232-59776
		create	22438	hive	/tmp/hive_stagingdir/hive_hive_2018-02-27_02-01-30_604_5744798450550671345-59763
		create	19347	hive	/tmp/hive_stagingdir/hive_hive_2018-02-27_02-08-11_715_2470669021781026765-59098
		create	8172	hive	/tmp/hive_stagingdir/hive_hive_2018-02-27_02-01-52_950_8449284600356918220-58896
		create	3845	hive	/tmp/hive_stagingdir/hive_hive_2018-02-27_02-01-13_545_7766103451547089863-22982
		create	1764	hive	/tmp/hive_stagingdir/hive_hive_2018-02-27_02-05-02_486_3190944843817077058-6249
		create	1103	hive	/tmp/hive_stagingdir/hive_hive_2018-02-27_02-07-55_055_25282529972543290-22501

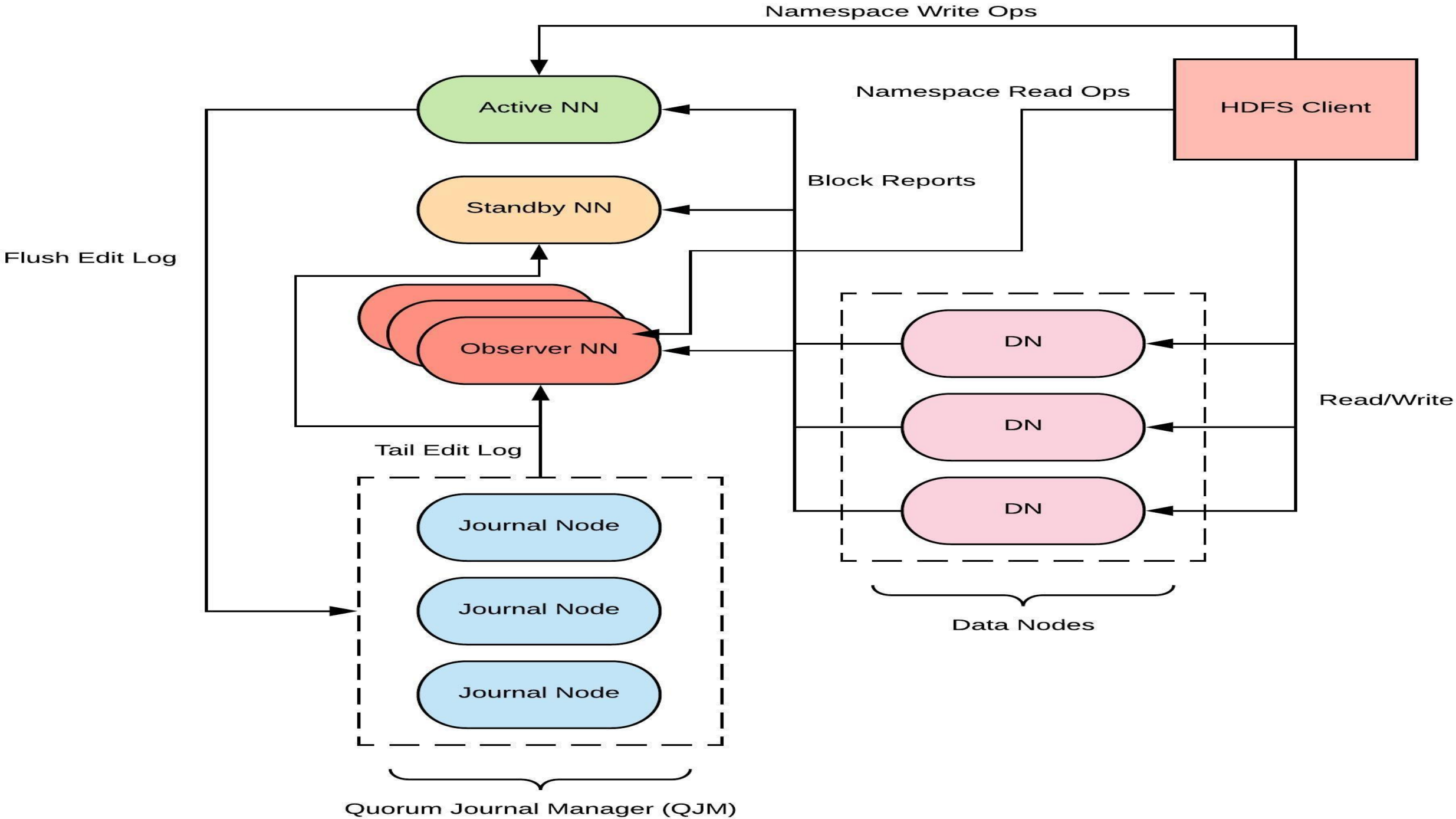
https://www.uber.com/en-PK/blog/scaling-hdfs/

Uber

Development of Observer NameNode - read-only NameNode replica - reduce the load on the active NameNode cluster.

Because more than half of HDFS RPC volume comes from read-only Presto queries, Observer NameNodes help us scale the overall NameNode throughput by close to 100 percent in the first release.




Namespace Write Ops
Active NN
Namespace Read Ops
HDFS Client
Block Reports
Standby NN
Flush Edit Log
Observer NN
DN
DN
DN
Read/Write
Tail Edit Log
Journal Node
Journal Node
Journal Node
Data Nodes
Quorum Journal Manager (QJM)

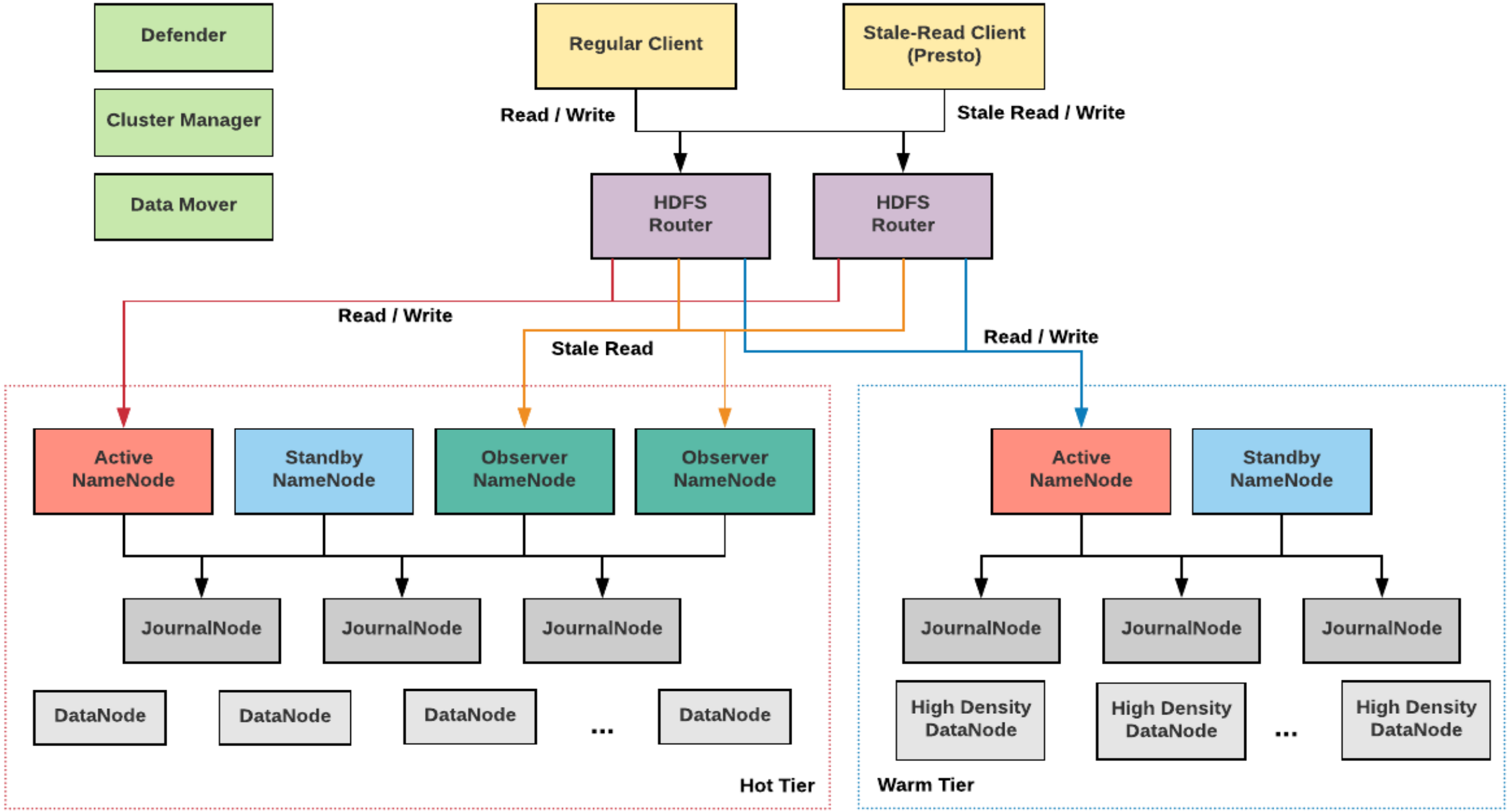
https://www.uber.com/en-PK/blog/scaling-hdfs/

Uber

Layer your solutions: Scalability improvements like Observer NameNode or splitting HDFS into more clusters takes significant effort - Short-term measures like GC tuning and merging smaller files into larger ones via our stitcher gave us a lot of breathing room to develop long-term solutions.

Bigger file sizes are better: Since small files are a threat to HDFS, it is better to tackle them earlier rather than later.

Participate in the community: Hadoop has been around for more than 10 years and its community is more active than ever before, leading to scalability and functionality improvements introduced in nearly every release.

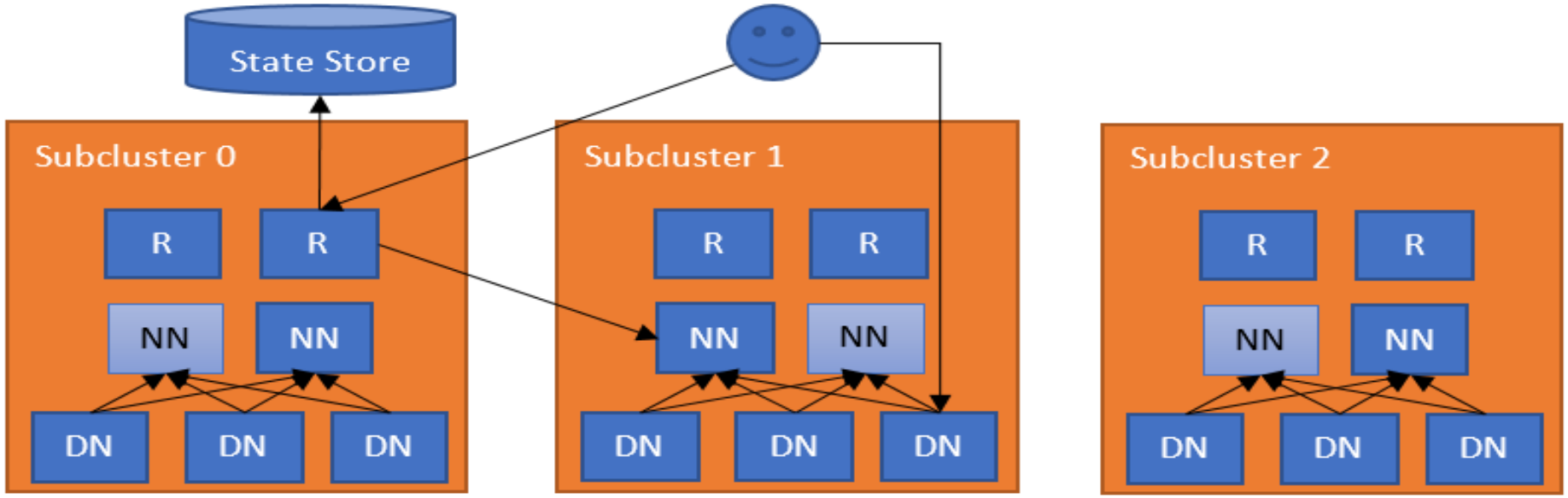

Defender
Cluster Manager
Data Mover
Regular Client
Stale-Read Client (Presto)
Read / Write
Stale Read / Write
HDFS Router
HDFS Router
Read / Write
Stale Read
Read / Write
Active NameNode
Standby NameNode
Observer NameNode
Observer NameNode
JournalNode
JournalNode
JournalNode
DataNode
DataNode
DataNode
...
DataNode
Hot Tier
Active NameNode
Standby NameNode
JournalNode
JournalNode
JournalNode
High Density DataNode
High Density DataNode
...
High Density DataNode
Warm Tier

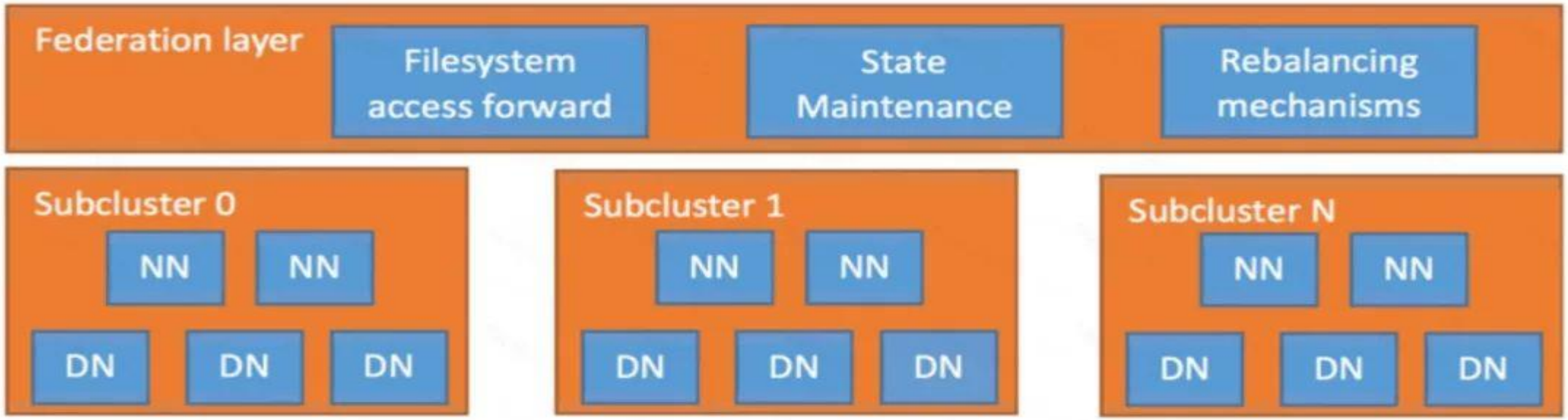
ViewFs - scale out HDFS when subclusters become overloaded – But then client configuration changes are required every time one adds or replaces a new mount point - affects production workflows.

Microsoft’s Router-based Federation - HDFS 2.9 release - natural extension to a ViewFs-based partitioned federation.

Adds a layer of software capable of centralizing HDFS namespaces –

By offering the same interface (a combination of RPC and WebHDFS), gives users transparent access to any subcluster - subclusters manage their data independently.


State Store
Subcluster 0
R
R
NN
NN
DN
DN
DN
Subcluster 1
R
R
NN
NN
DN
DN
DN
Subcluster 2
R
R
NN
NN
DN
DN
DN


Federation layer
Filesystem access forward
State Maintenance
Rebalancing mechanisms
Subcluster 0
NN
NN
DN
DN
DN
Subcluster 1
NN
NN
DN
DN
DN
Subcluster N
NN
NN
DN
DN
DN

https://www.uber.com/en-PK/blog/scaling-hdfs/

Uber

Importance of reducing storage costs - users access recent data (“hot” data) more frequently than old data (“warm” data).

Moving old data to a separate, less resource-intensive tier will significantly reduce our storage costs.

HDFS Erasure Coding, Router-based Federation, high density (more than 250TB) hardware, and a data mover service (to handle moving data between “hot” tier clusters and “warm” tier clusters) are key components of our forthcoming tiered storage design.