Introduction to Big Data

HADOOP DISTRIBUTED FILESYSTEM

Le Ngoc Thanh – Nguyen Ngoc Thao

{lnthanh, nnthao}@fit.hcmus.edu.vn

Outline

- Hadoop Distributed Filesystem (HDFS)
  - NameNode and DataNode
  - HDFS Federation
  - HDFS High Availability
  - Data processing on HDFS
- Read data from HDFS
- Write data to HDFS

Cluster terminologies: Node

- A node is simply a physical computer.

Node 1

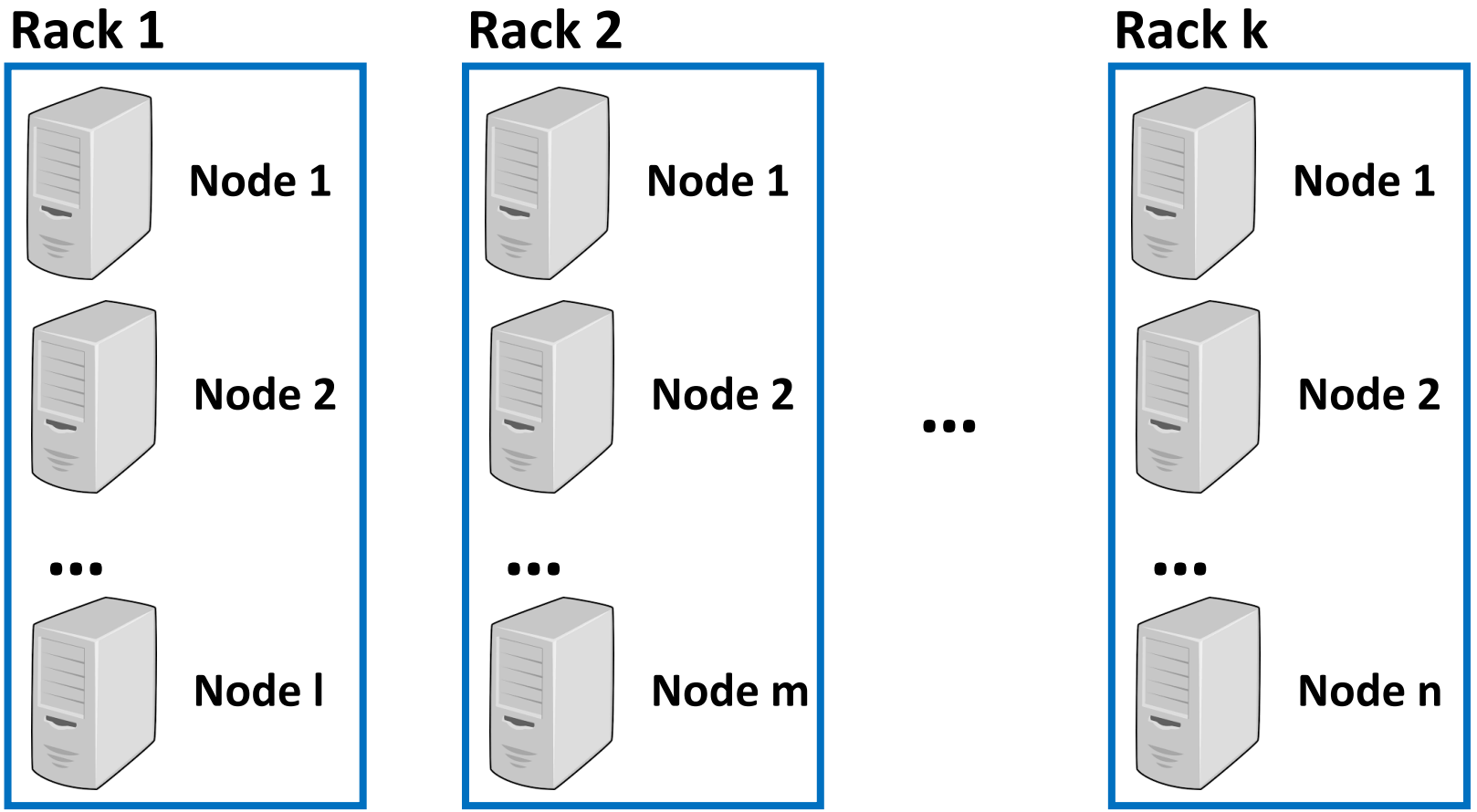
Node 2

...

Node l

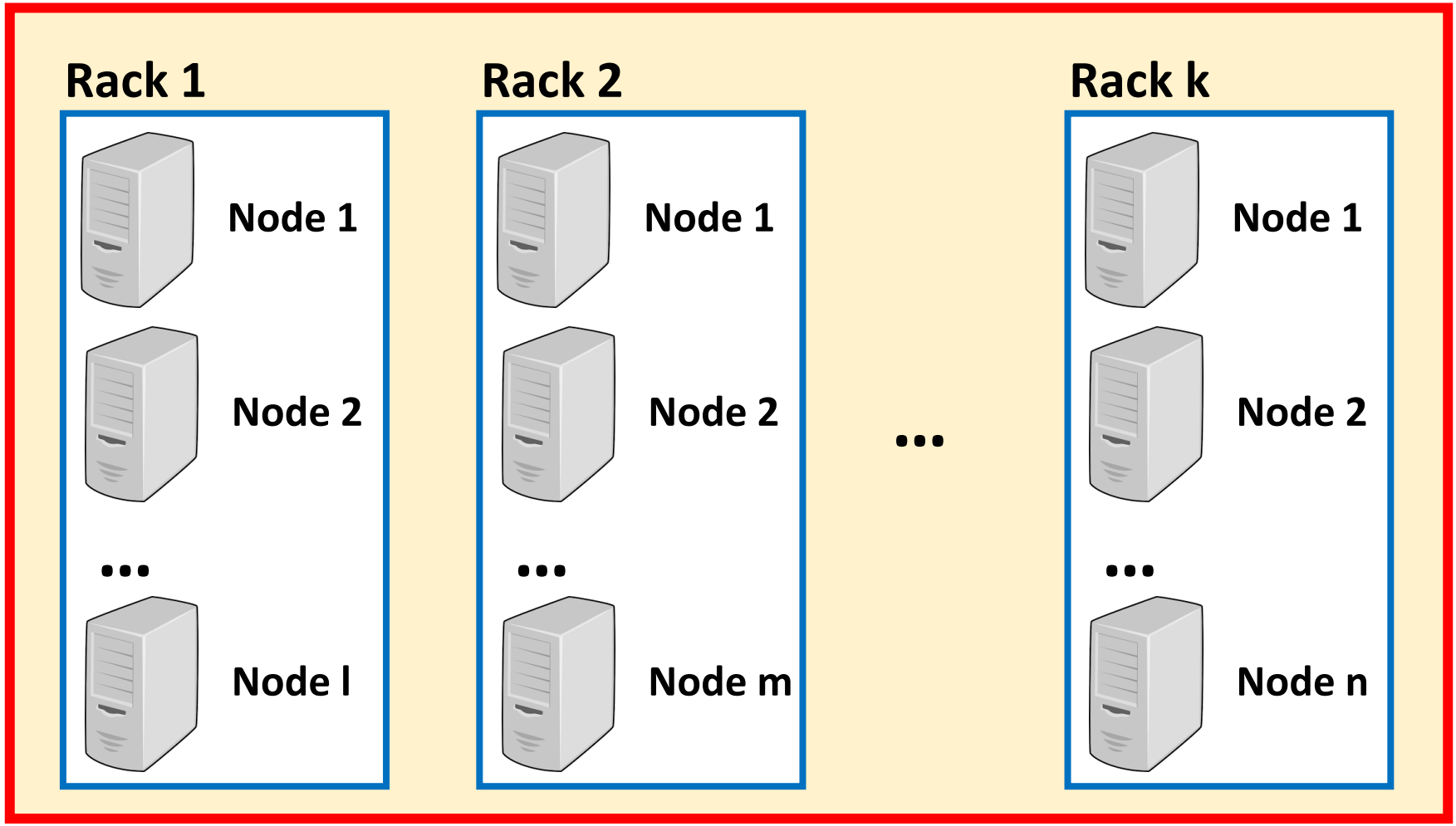
Cluster terminologies: Rack

- A rack consists of **30** or **40** nodes physically stored close together and all connected to the same network switch.

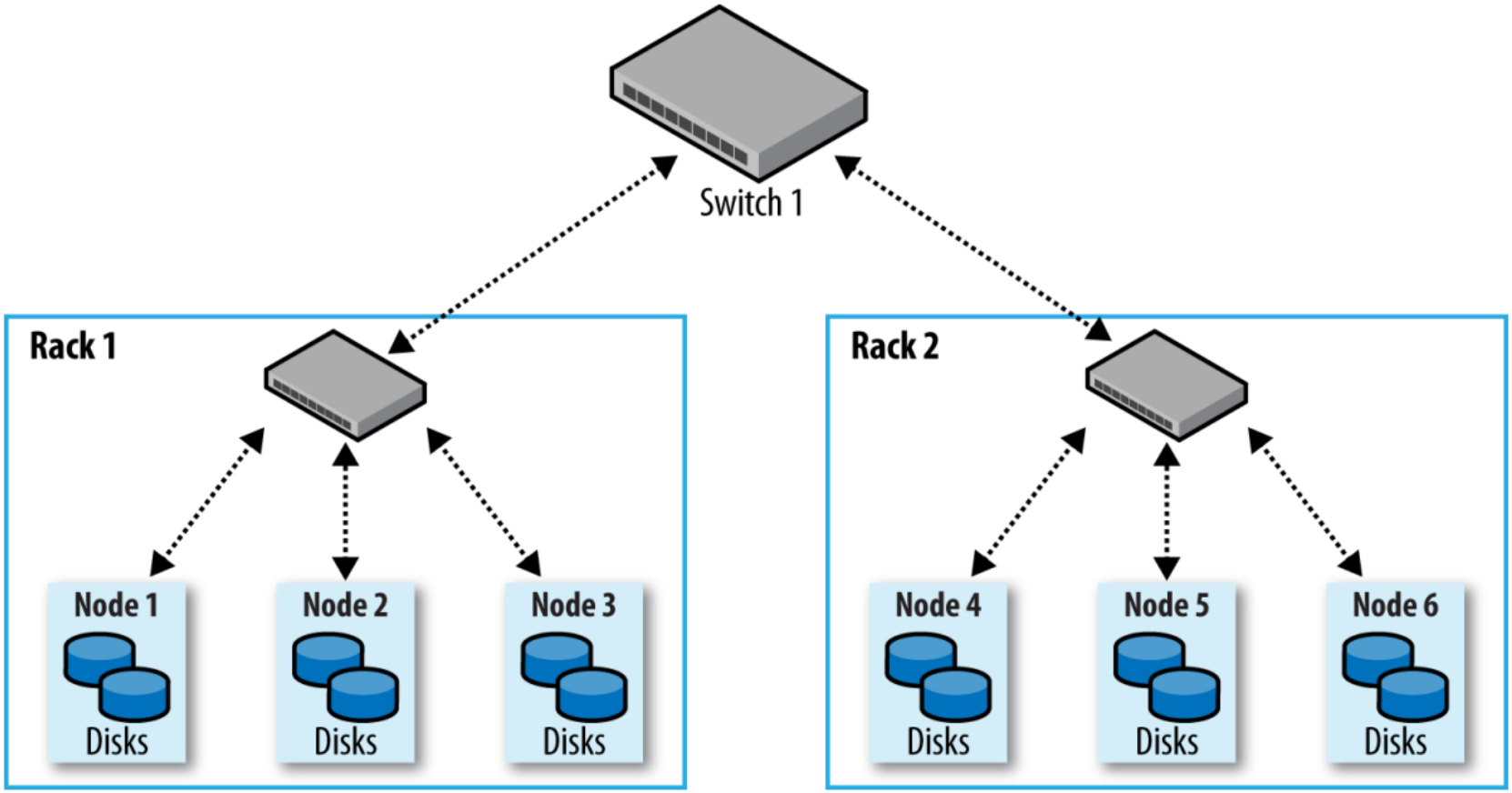


Cluster terminologies: Cluster

- A Hadoop cluster is a collection of racks.



Terminology review: Cluster



A typical two-level network architecture for a Hadoop cluster.

Apache Hadoop: Architecture

- The core of Hadoop includes HDFS, MapReduce, and YARN.

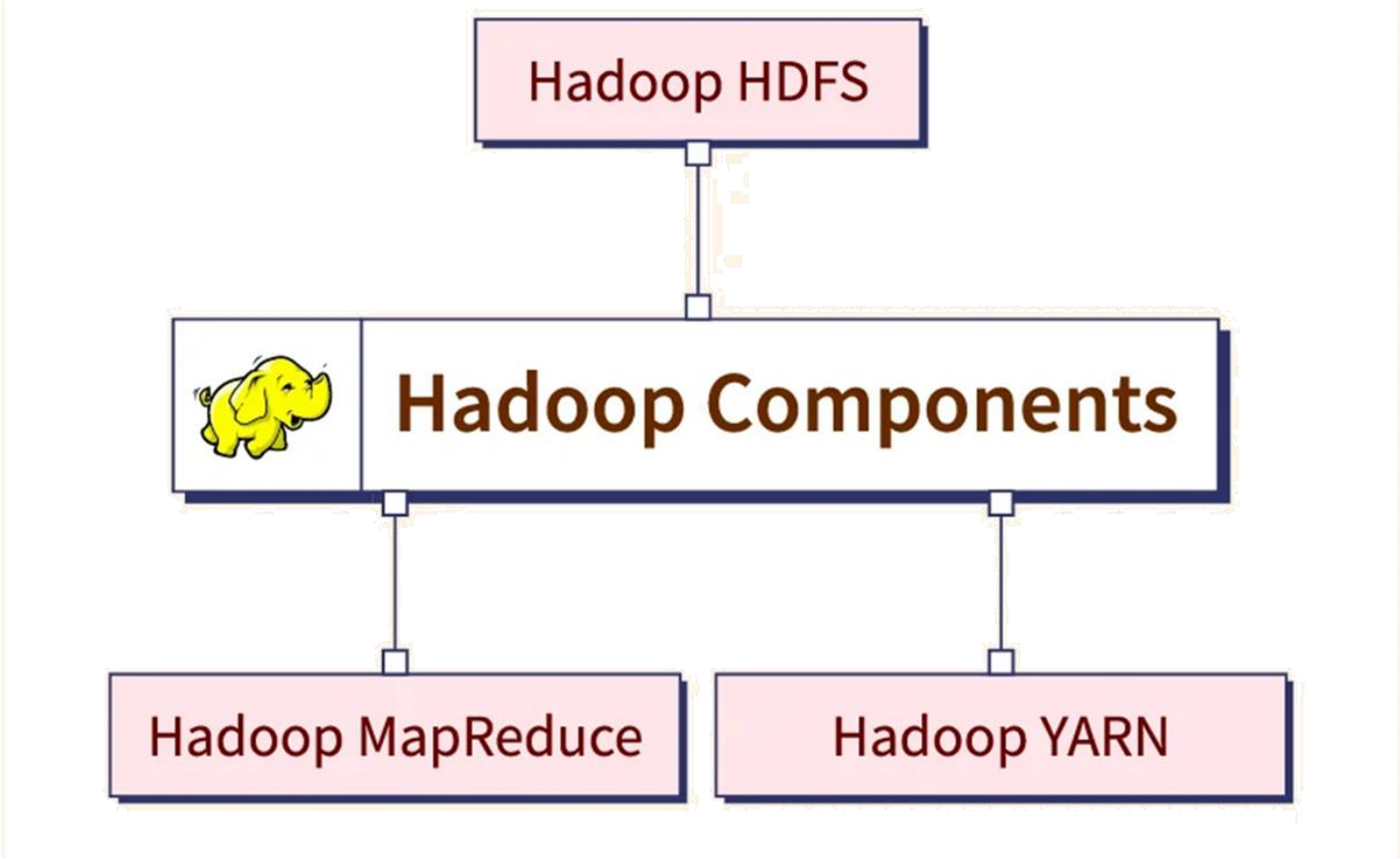


Image credit: Scaler Topics

Distributed Filesystem

- Mainly **Hadoop Distributed Filesystem** (HDFS), while other FSs are also supported, like IBM Spectrum Scale, Amazon S3, etc.
- Designed to store large files as multiple blocks on multiple nodes
- Data is automatically re-replicated on need.

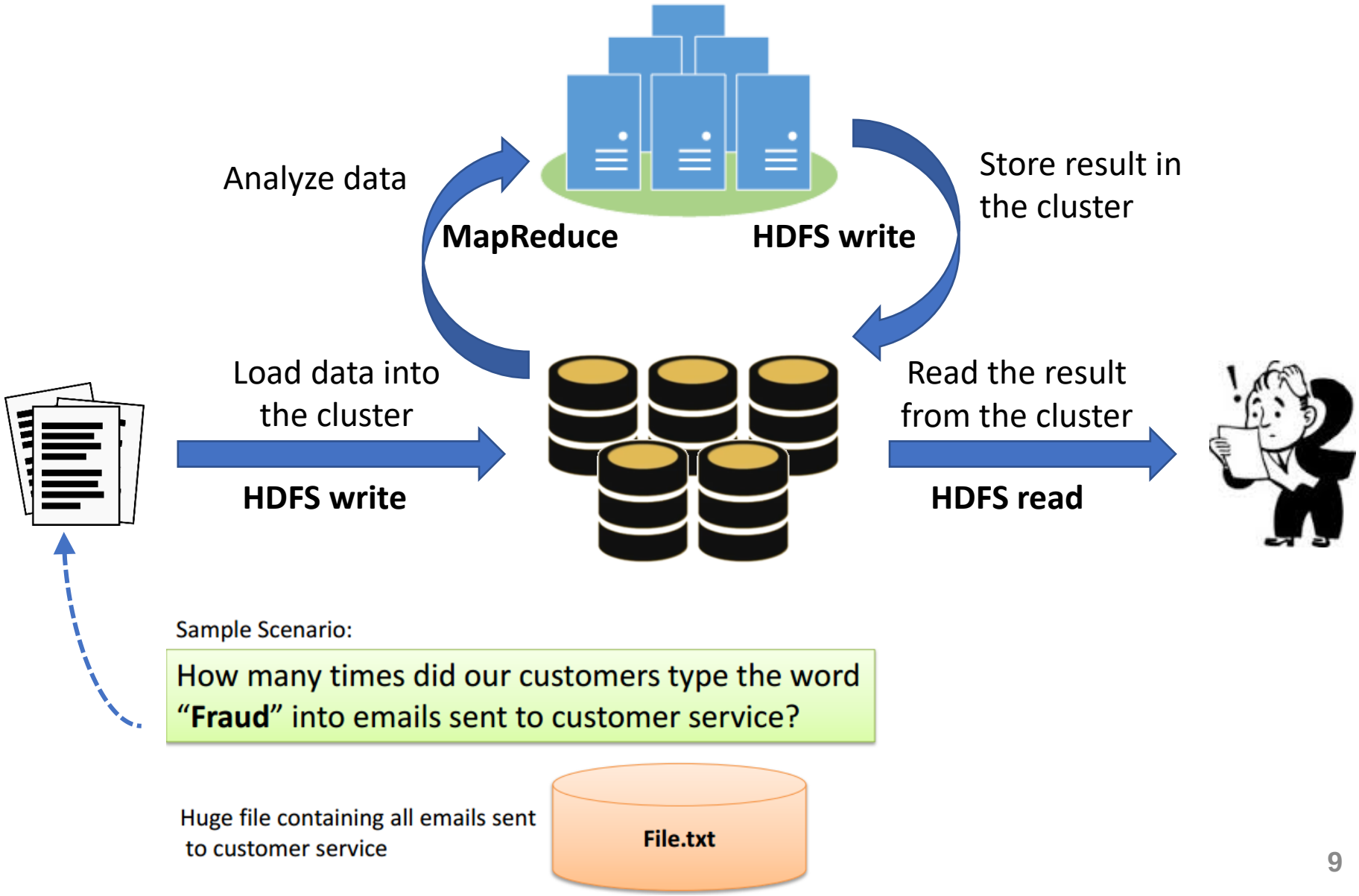
MapReduce framework

- Perform calculations on the data stored in the distributed filesystem
- Make schedules to run jobs in Hadoop cluster and monitor them

YARN

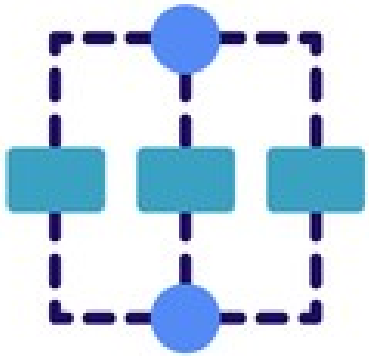
- Designed to manage and allocate resources in a Hadoop cluster.

A typical workflow in Hadoop cluster

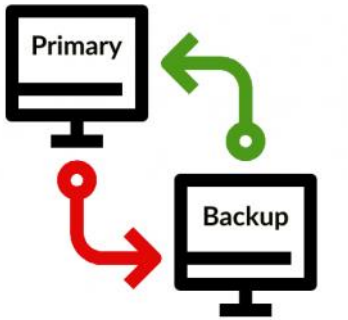


Hadoop: Advantages

Distributed storage



Parallel processing

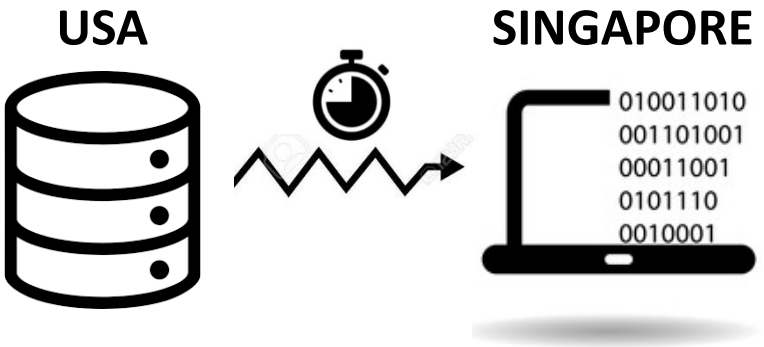


Automatic failover management

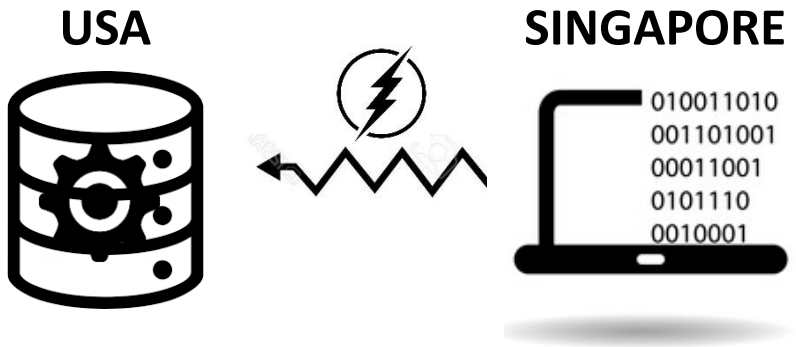
Cost effective

Hadoop: Advantages

- Hadoop has a better scheme for data processing by using data locality optimization.



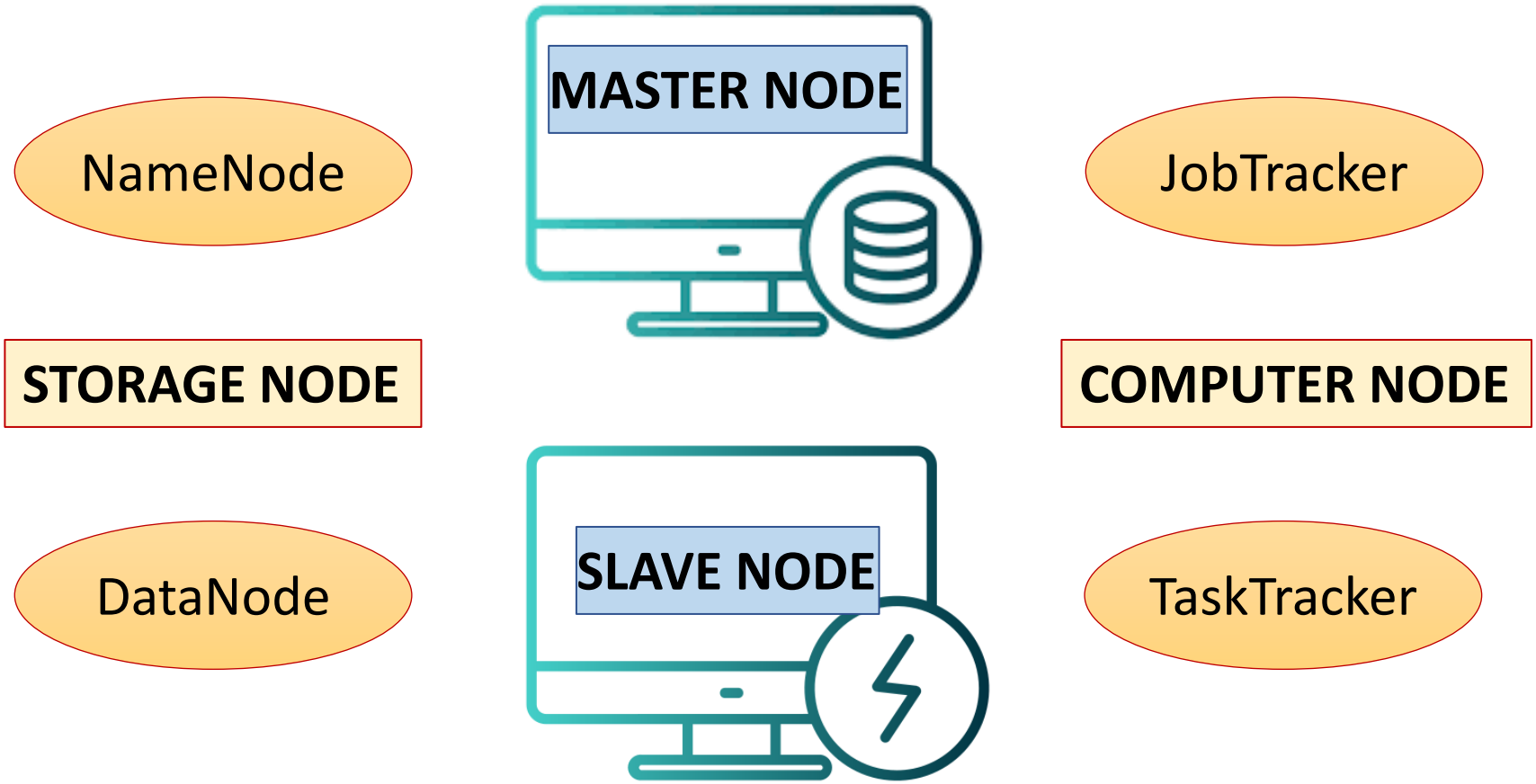
Traditional scheme: copy the data (in PBs) from USA to Singapore
→ not good in terms of performance and time



Better scheme: copy the source code (in MBs) to USA from Singapore

Hadoop Pre 2.2 architecture

- Working nodes in a Hadoop cluster can be categorized into



Hadoop Pre 2.2 architecture

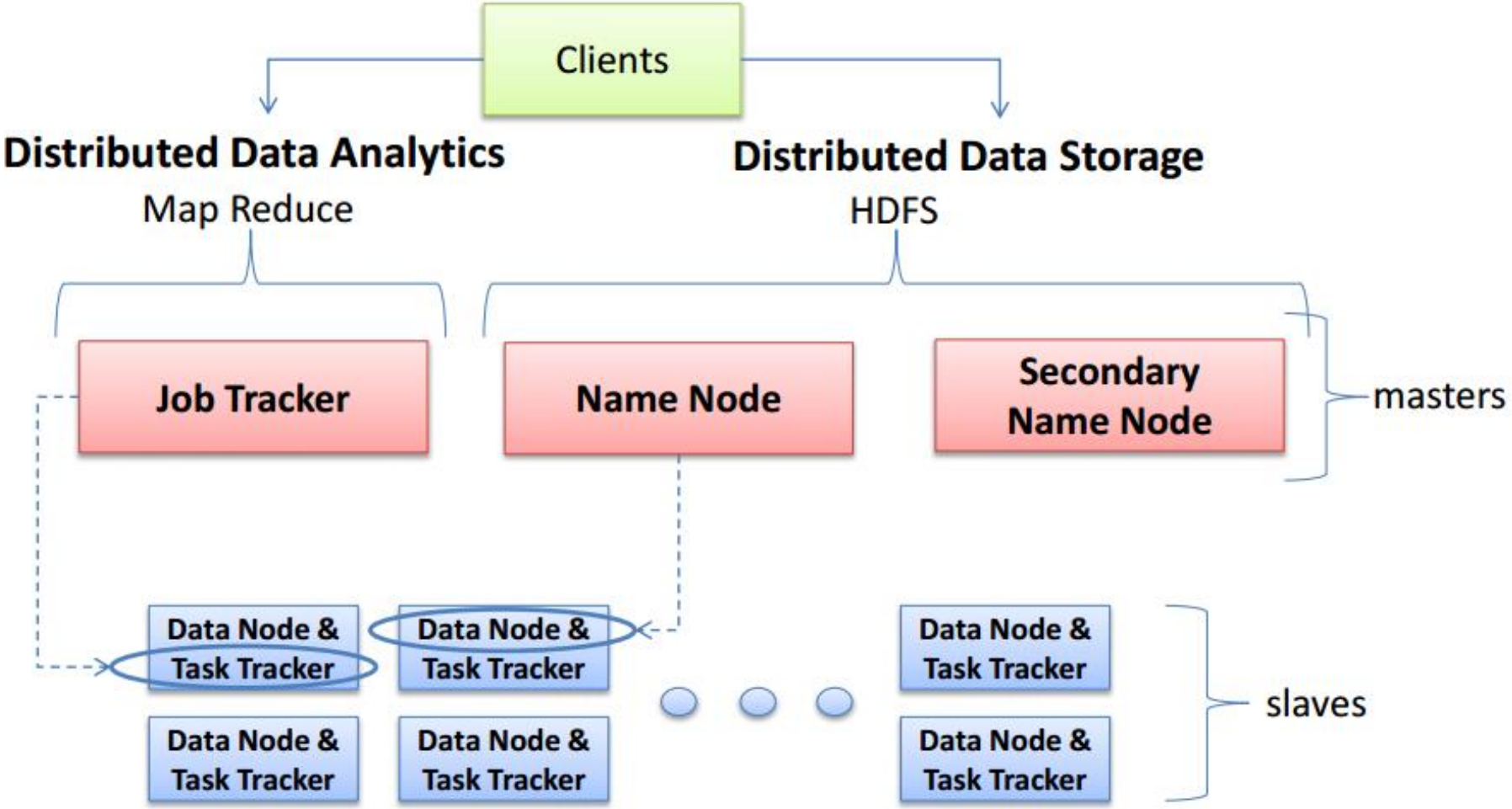


Image credit: Brad Hedlund

Hadoop 2.2 architecture

- HDFS Federation: Horizontal scalability for NameNode
- NameNode High Availability: NameNode is no longer a SPOF (Single Point of Failure)
- YARN: A new framework for cluster resource management
  - Support data processing of size TBs or PBs using Non-MapReduce applications (e.g., MPI, Giraph)
  - Split up JobTracker into two separate daemons: a global Resource Manager and per-application ApplicationMaster
- Additional features
  - Capacity Scheduler (Enable Multi-tenancy support in Hadoop), Data Snapshot, Support for Windows, NFS access, etc.

Hadoop 3.0 architecture

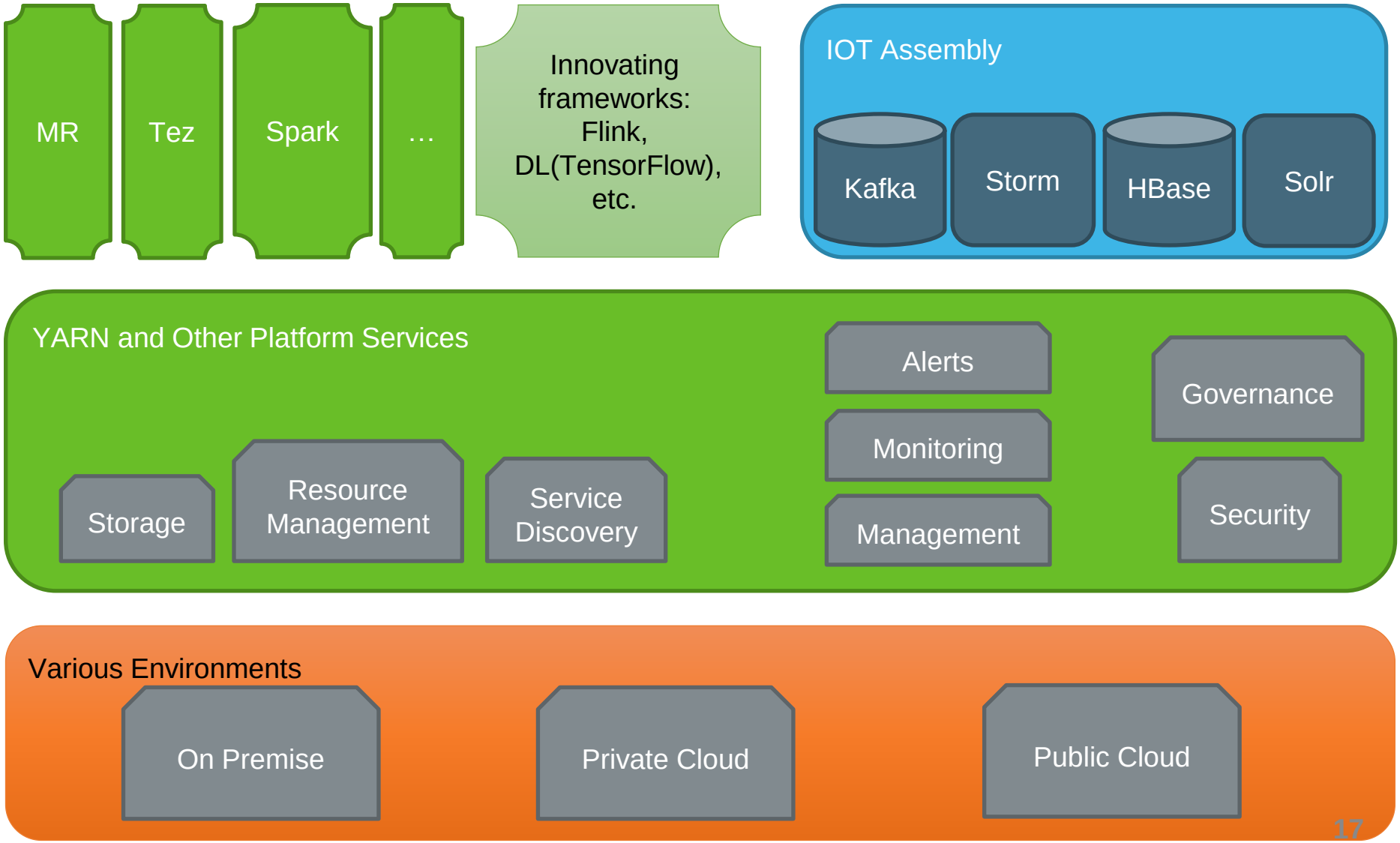
There are significant enhancements over the previous major release line (hadoop-2.x)

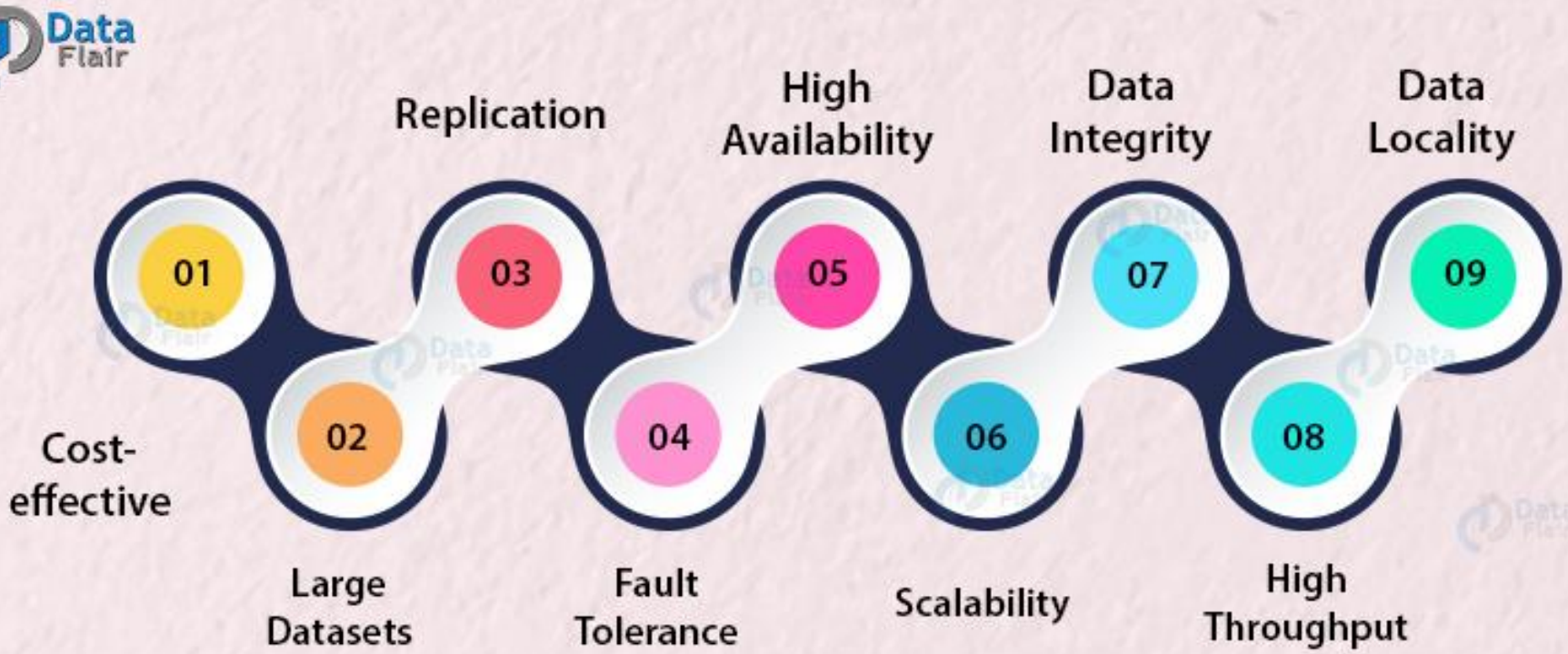
- Minimum Required Java Version is 8
- Support for Erasure Encoding in HDFS
- YARN Timeline Service v.2
- Shell Script Rewrite
- Shaded Client Jars
- Support for Opportunistic Containers
- MapReduce Task-Level Native Optimization
- Support for More than 2 NameNodes
- Default Ports of Multiple Services changed
- Support for Filesystem Connector
- Intra-DataNode Balancer
- Reworked Daemon and Task Heap Management

Hadoop 3.0 architecture

- Support for erasure encoding in HDFS
  - Store data with significant space savings compared to replication
- YARN Timeline Service v.2
  - Improve scalability and reliability of Timeline Service, and enhance usability by introducing flows and aggregation
- Support for multiple Standby NameNodes
  - Allow for higher abilities of fault tolerance and availability for HDFS
- New default ports for several services
  - Default ports for NameNode, Secondary NameNode, DataNode, etc. have been moved out of the Linux ephemeral range (32768-61000).
- Intra-DataNode Balancer
  - Address the intra-node skew when disks are added or replaced

Hadoop 3.0 architecture

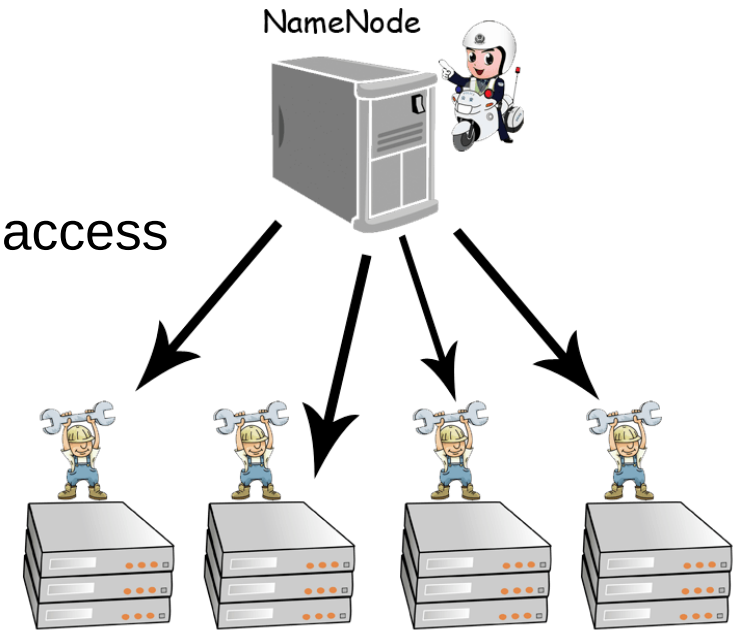




Hadoop Distributed Filesystem

Hadoop distributed filesystem

- A filesystem for storing very large files with streaming data access patterns, running on clusters of commodity hardware
- Run on top of the existing filesystem
  - Tolerate high component failure rate through replication of the data
  - Not POSIX compliant
- Work best with very large files
  - Designed for streaming or sequential access rather than random access

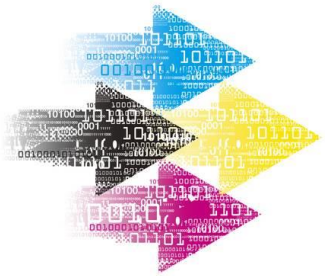


HDFS are designed for

Very large files

Files of hundreds of megabytes, gigabytes, or terabytes in size.

There are modern Hadoop clusters that store petabytes of data.



Streaming data access

Most efficient data processing pattern: write-once and read-many

The time to read the whole dataset is more important than the latency in reading the first record.

Commodity hardware

Expensive and/or highly reliable hardware is not obligatory.

It overcomes failure without a noticeable interruption to the user.

Yet, HDFS are NOT designed for

Low-latency data access

Not for applications that require access in the tens of milliseconds

Optimized for delivering a high throughput of data → this may be at the expense of latency

Lots of small files

The number of files in a filesystem is limited by the amount of memory on the NameNode.

Multiple writers, arbitrary file modifications

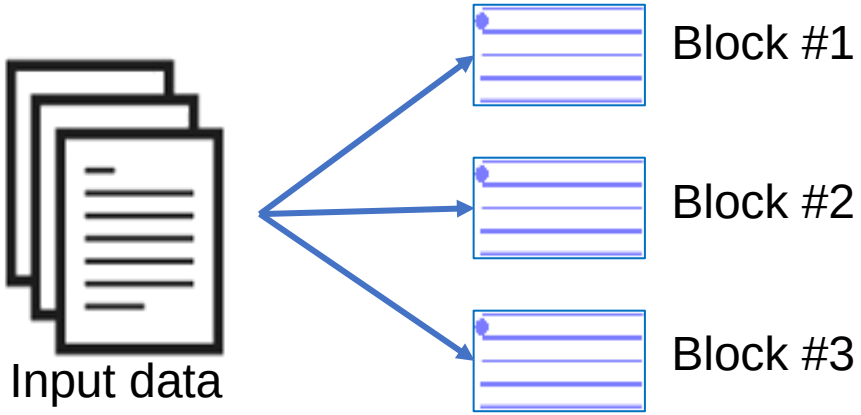
Writes are always made at the end of file in append-only fashion.

There is no support for multiple writers or for modifications at arbitrary offsets in the file.

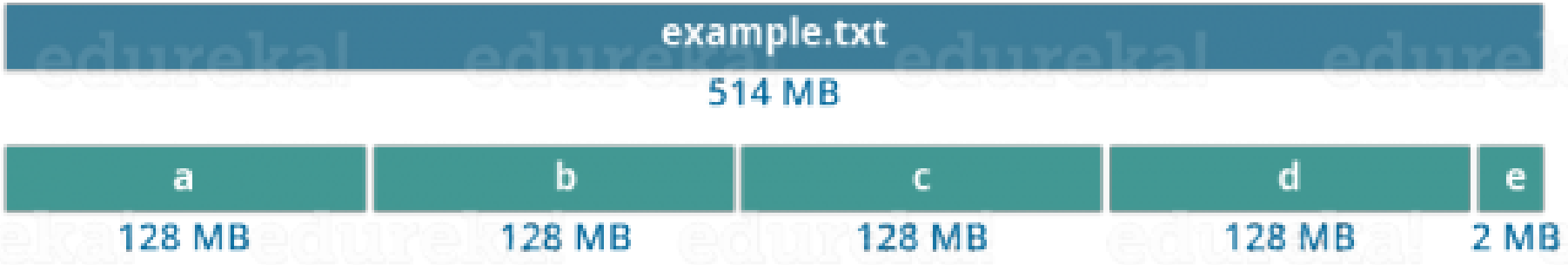
Hadoop Distributed File System HDFS	Google File System GFS
Cross Platform	Linux
Developed in Java environment	Developed in C, C++ environment
At first developed by Yahoo and now it is an open-source framework	Developed by Google
NameNode and DataNode	Master Node and Chunk server
The default block size is 128MB	The default block size is 64MB
Heartbeat: DataNode → NameNode	Heartbeat: Chunk server → Master Node
Commodities hardware were used	Commodities hardware were used
WORM – Write Once and Read Many times	Multiple writer, multiple reader model
Deleted files are renamed into particular folder and then it will be removed via garbage	Deleted files are not reclaimed immediately. They are renamed in hidden namespace and will be deleted after 3 days if not in use
No Network stack issue	Network stack Issue
Journal, editlog	Operational log
Only append is possible	Random file write possible

HDFS file blocks

- Hadoop uses blocks to store a file or parts of a file.



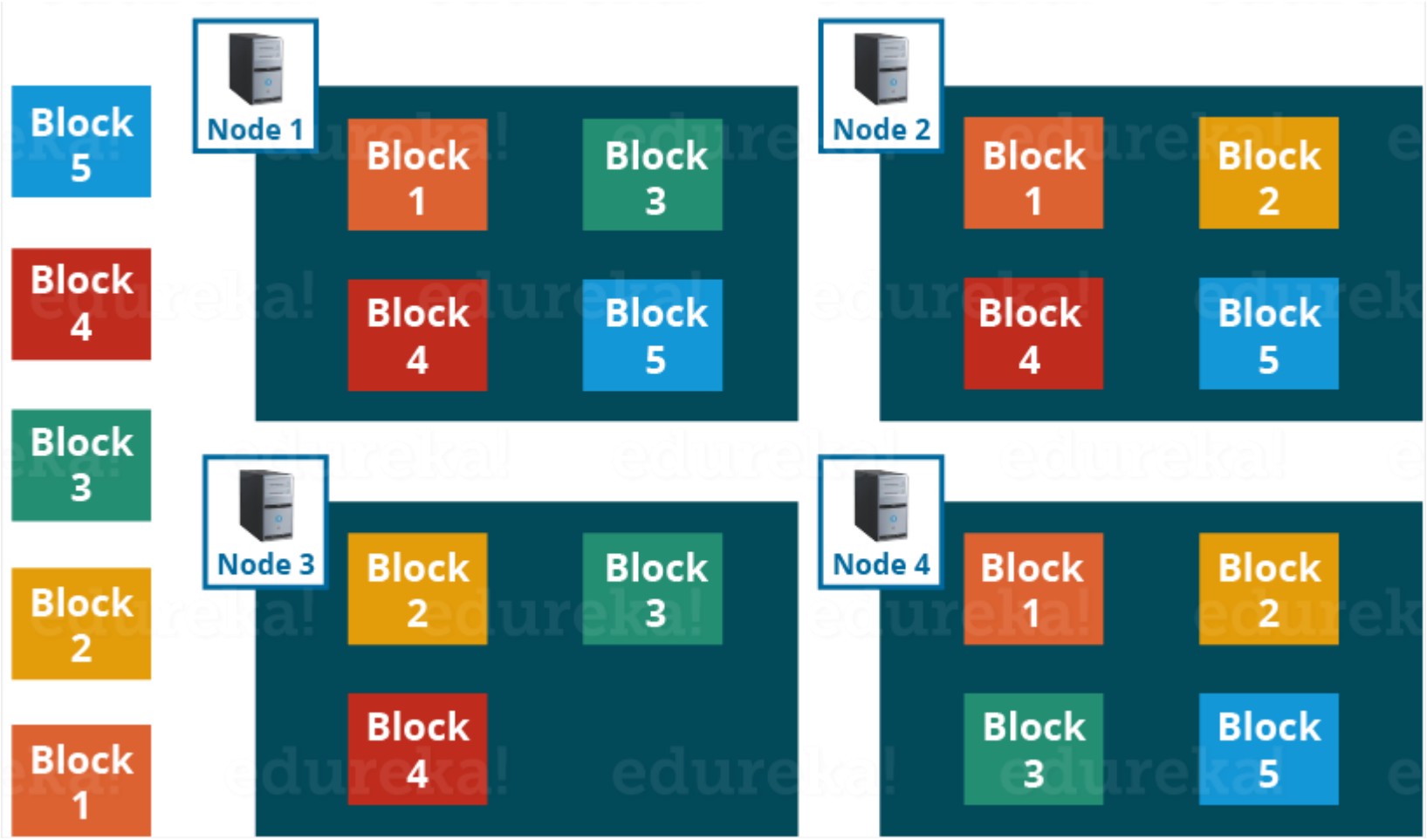
- A HDFS block is a file on the underlying filesystem.
  - Block size: commonly 128 MBs (Hadoop 2-3, IBM BigInsights)

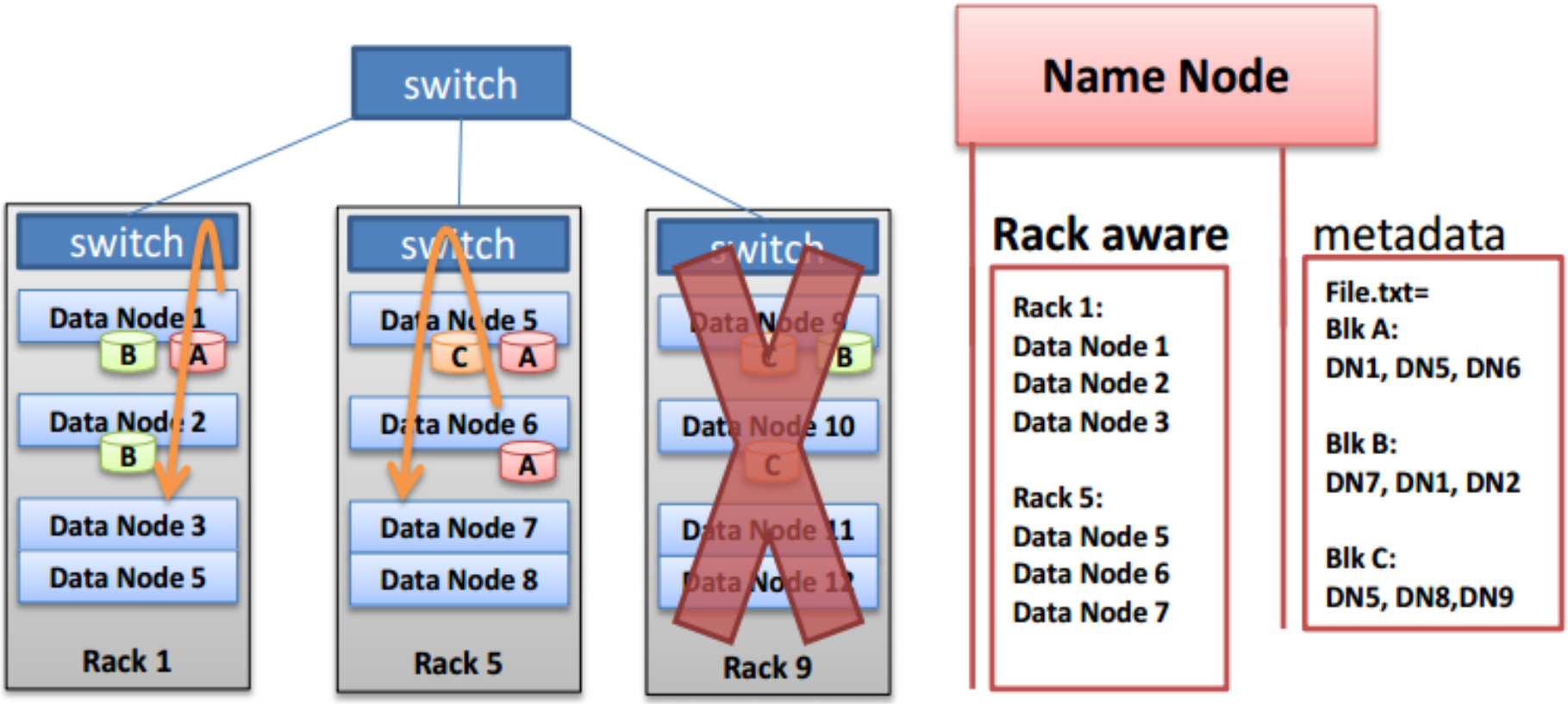


HDFS file blocks

- Blocks fit well with replication, allowing HDFS to be fault tolerant and available on commodity hardware.
- They can be spread over multiple nodes.
  - A file can be larger than any single disk in the cluster.
- Fixed in size $\rightarrow$ calculate how many can fit on a disk easily
- HDFS blocks also do not waste space.
  - If a file is not an even multiple of the block size, the block containing the remainder does not occupy the space of an entire block.

Block replication: An example





Rack Awareness

- **Two copies in one rack, one copy in another rack**

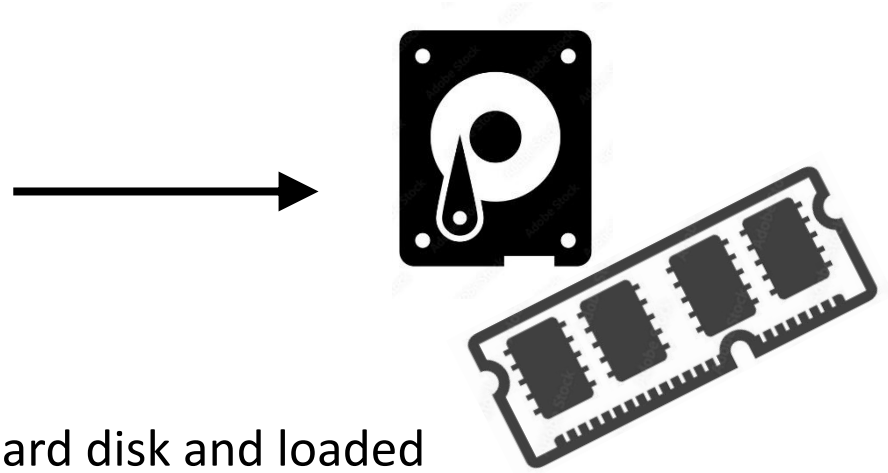
Image credit: Brad Hedlund

NameNode

NameNode: Functionalities

- Maintain and execute the filesystem namespace
  - Manage the filesystem tree and metadata (location, permissions, file size, etc.) for all the files and directories in the tree.
  - The information is stored persistently on the local disk in the **namespace image** (**fsimage**) and **transactional log** (**editLog**) files.

File name
File permissions
File ownership
File location

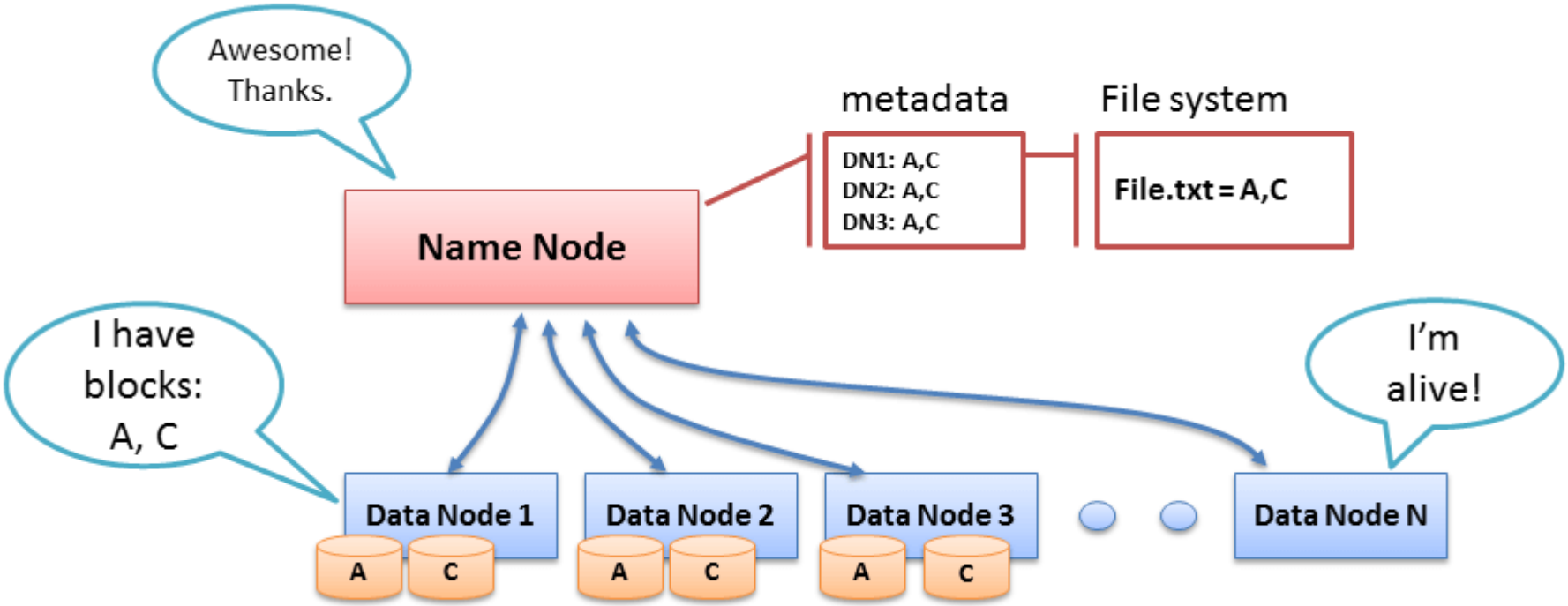


The metadata is stored in hard disk and loaded into main memory when starting Hadoop.

NameNode: Functionalities

- Govern the mechanism of storing data
  - Map a file to a set of blocks and direct a block to the DataNodes
  - Keep a record of how the files in HDFS are divided into blocks and in which nodes these blocks are stored
  - Take care of the **replication factor** of blocks
- Make sure that the DataNodes are working properly
  - Regularly receive heartbeats and block reports from all DataNodes
  - Allocate new DataNodes for replicas, in case of a DataNode failure, balances disk usage and manages the network traffic
- Direct DataNodes to execute the low-level I/O operations
- By and large the NameNode manages cluster configuration.

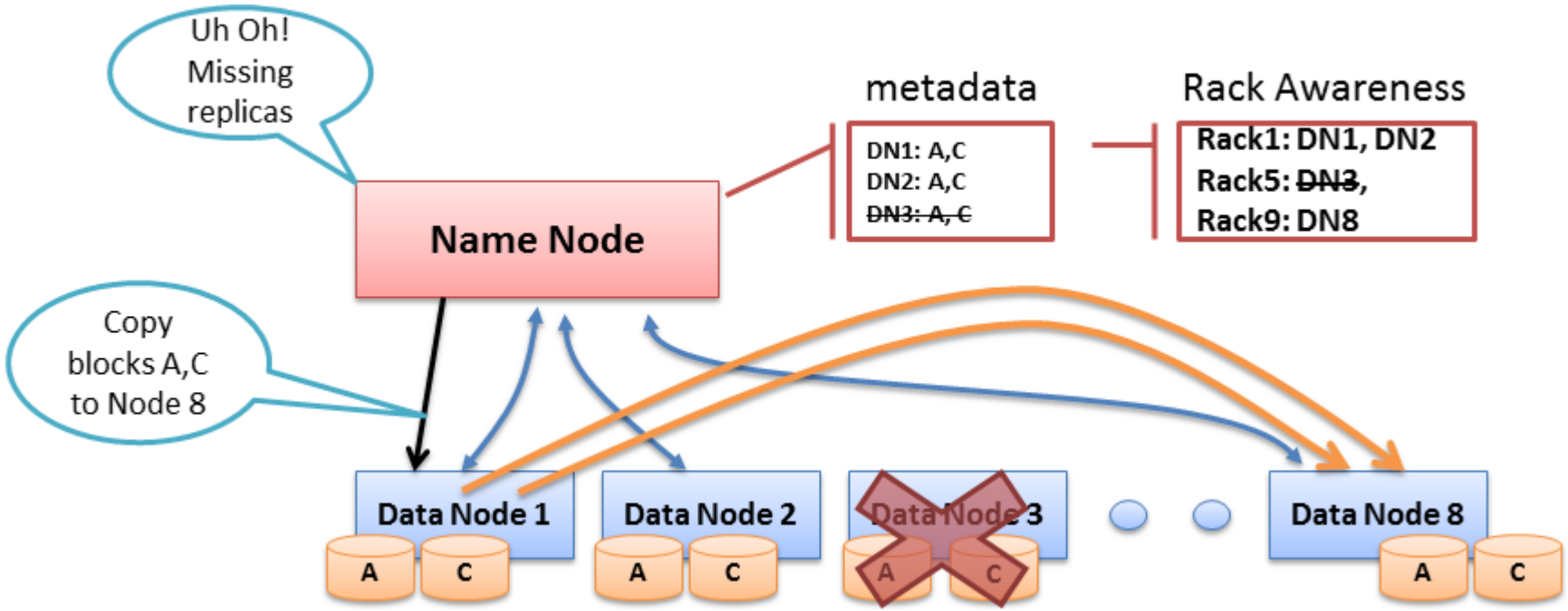
NameNode in use



- A DataNode sends a heartbeat every 3 seconds via a TCP handshake.
- For every 10th heartbeat, there is a Block report, allowing NameNode to build its metadata and insure **(3) copies** for each block in the system.
- **If NameNode is down, HDFS is down.**

Image credit: Brad Hedlund

Re-replicate missing replicas



- Missing heartbeats signify lost nodes.
- NameNode consults metadata to find affected data.
- It selects a DataNode for replication based on the Rack Awareness policy.

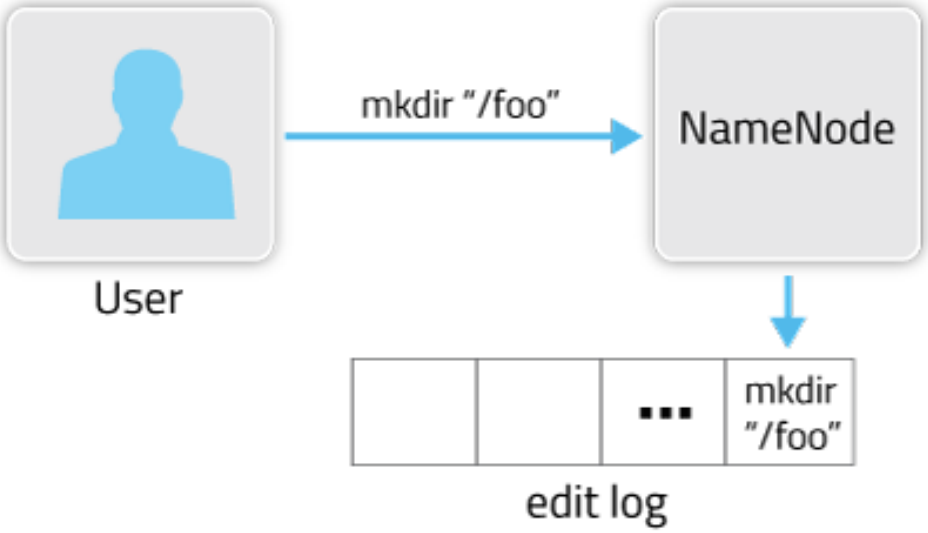
Image credit: Brad Hedlund

HDFS fsimage file

- A persistent checkpoint of HDFS metadata, on starting the NameNode
- Binary file, containing information about files and directories
  - E.g., full path, replication factor, modification time, access time, permissions and ownership, block size, number of locks, file size
- We can learn from the file how fast HDFS grows and draw appropriate capacity planning.
  - Combine fsimage with “external” datasets for a comprehensive report
  - Number of daily/monthly active users, total size of logs generated by users, and number of queries / day run by data analysts.

HDFS edit/audit file

- A transactional log that tracks a series of modifications done to the filesystem after starting the NameNode
  - Any modifications in the filesystem namespace or in its properties.
  - All filesystem access requests sent to the NameNode
- It is easy to parse and aggregate.

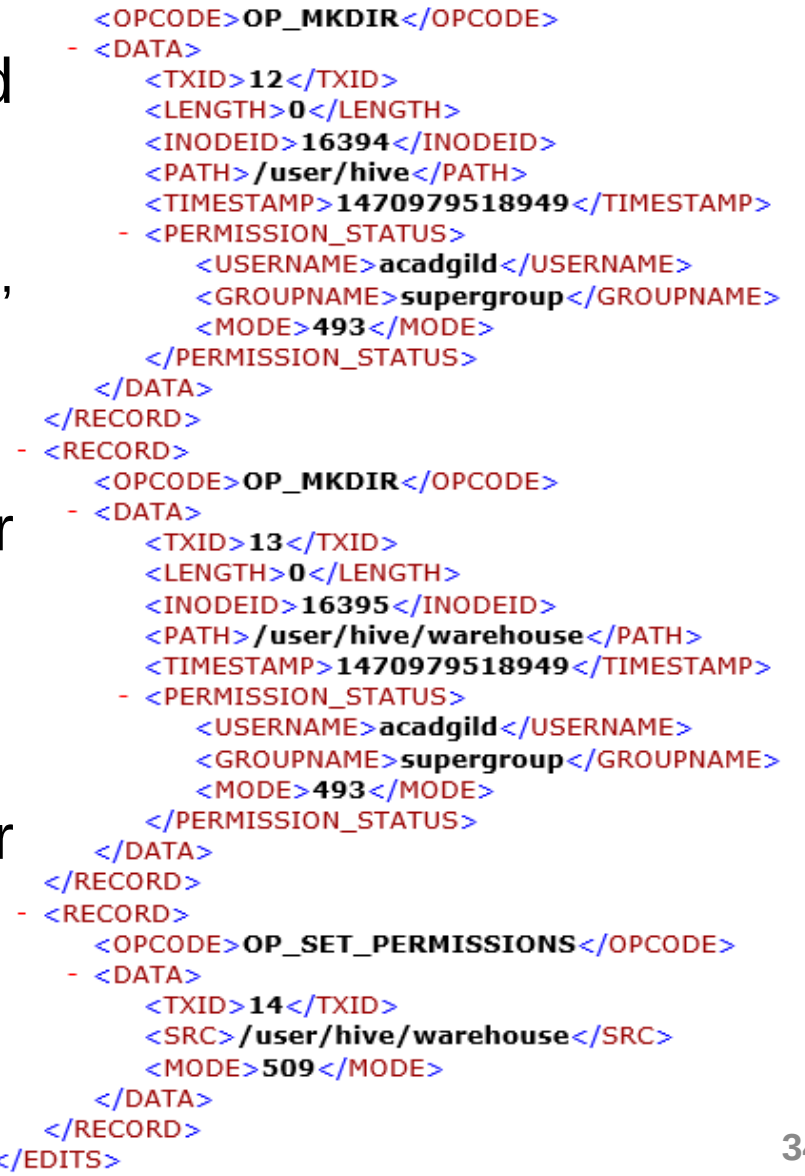


HDFS edit/audit file

- Some files/folders are accessed more often than others.
  - E.g., fresh logs, core datasets, dictionary files, etc.
- Increase its replication factor while it is “hot”

  → faster processing
- Decrease its replication factor when it becomes “cold”

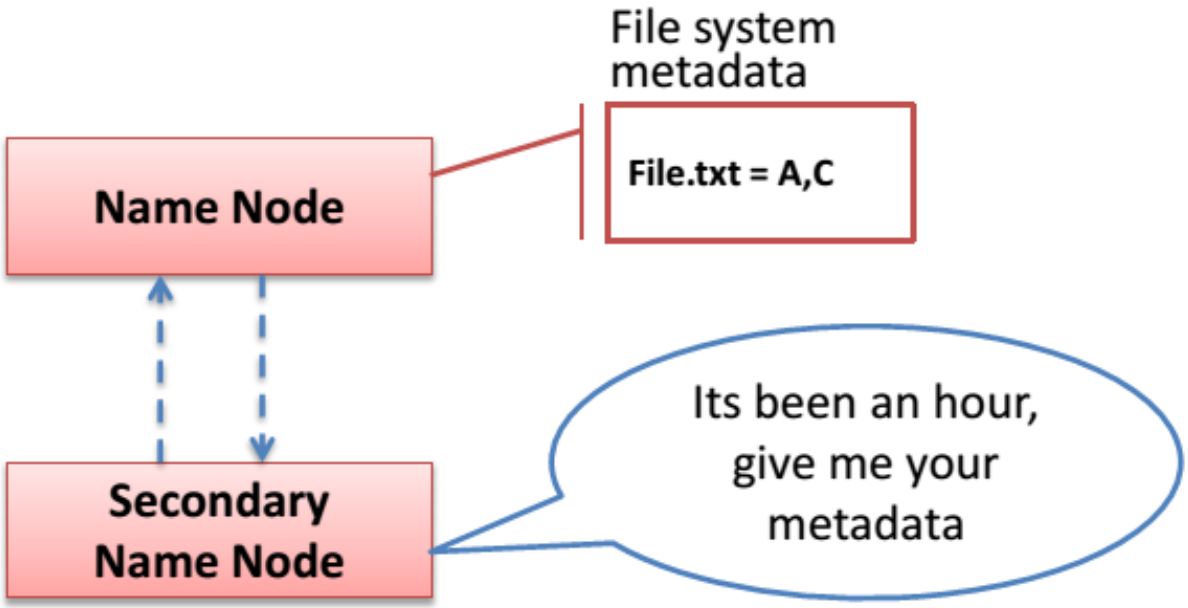
  → disk space saving



Secondary NameNode

- NameNode failure makes the Hadoop cluster inaccessible.
  - All the files on the filesystem would be lost, though the blocks are still on the DataNodes.
- It is important to make the NameNode **resilient to failure**.
- Conventional solution: backup the files on local disks or a remote NFS mount.
- More elastic solution: run a secondary NameNode.

Secondary NameNode

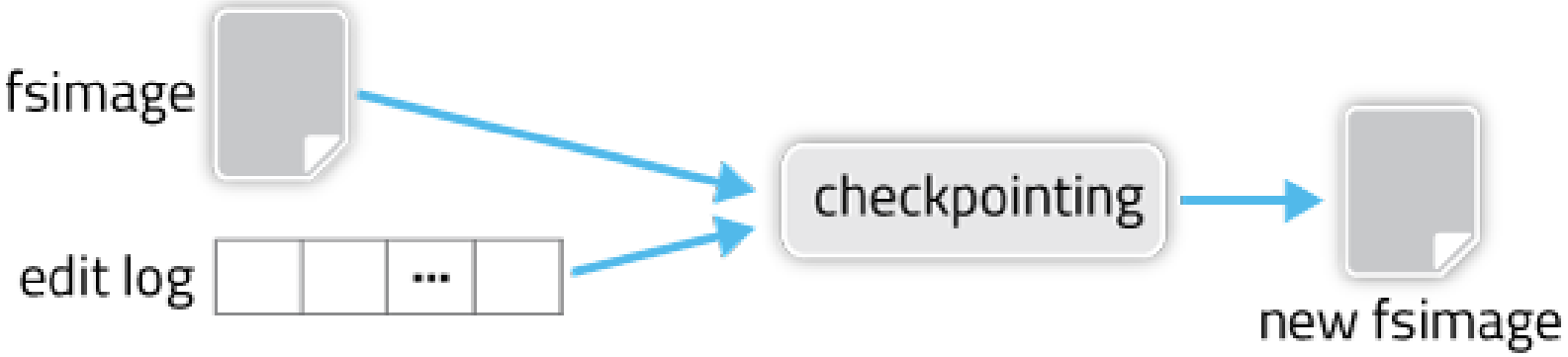


- Not a hot standby for the NameNode
- It serves as a housekeeper to back up the NameNode metadata, which can be used to recover a failed NameNode.
- It connects to the NameNode every hour (by default).

Image credit: Brad Hedlund

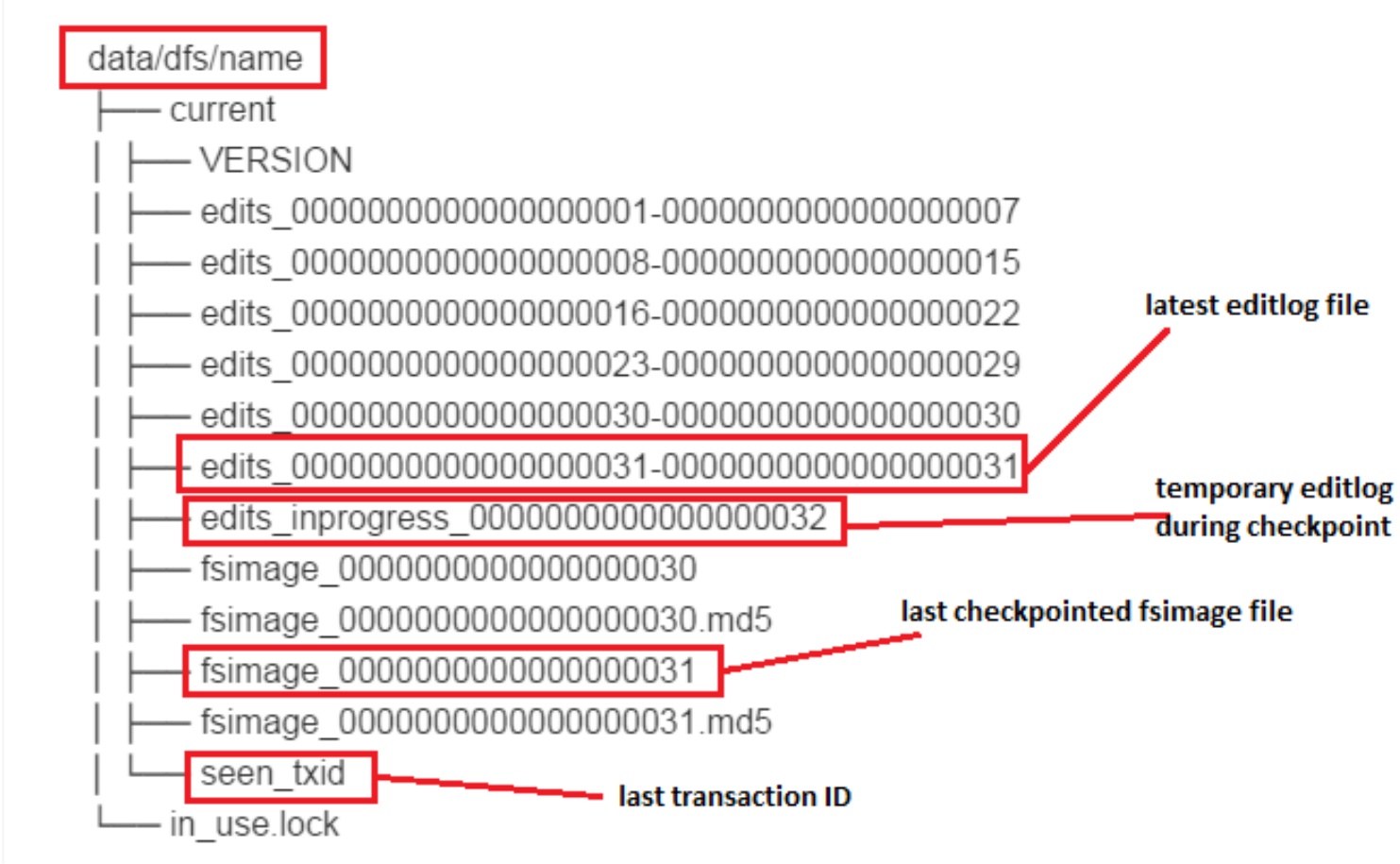
Checkpointing

- Checkpointing is a process that compacts a fsimage and an editLog into a new fsimage.



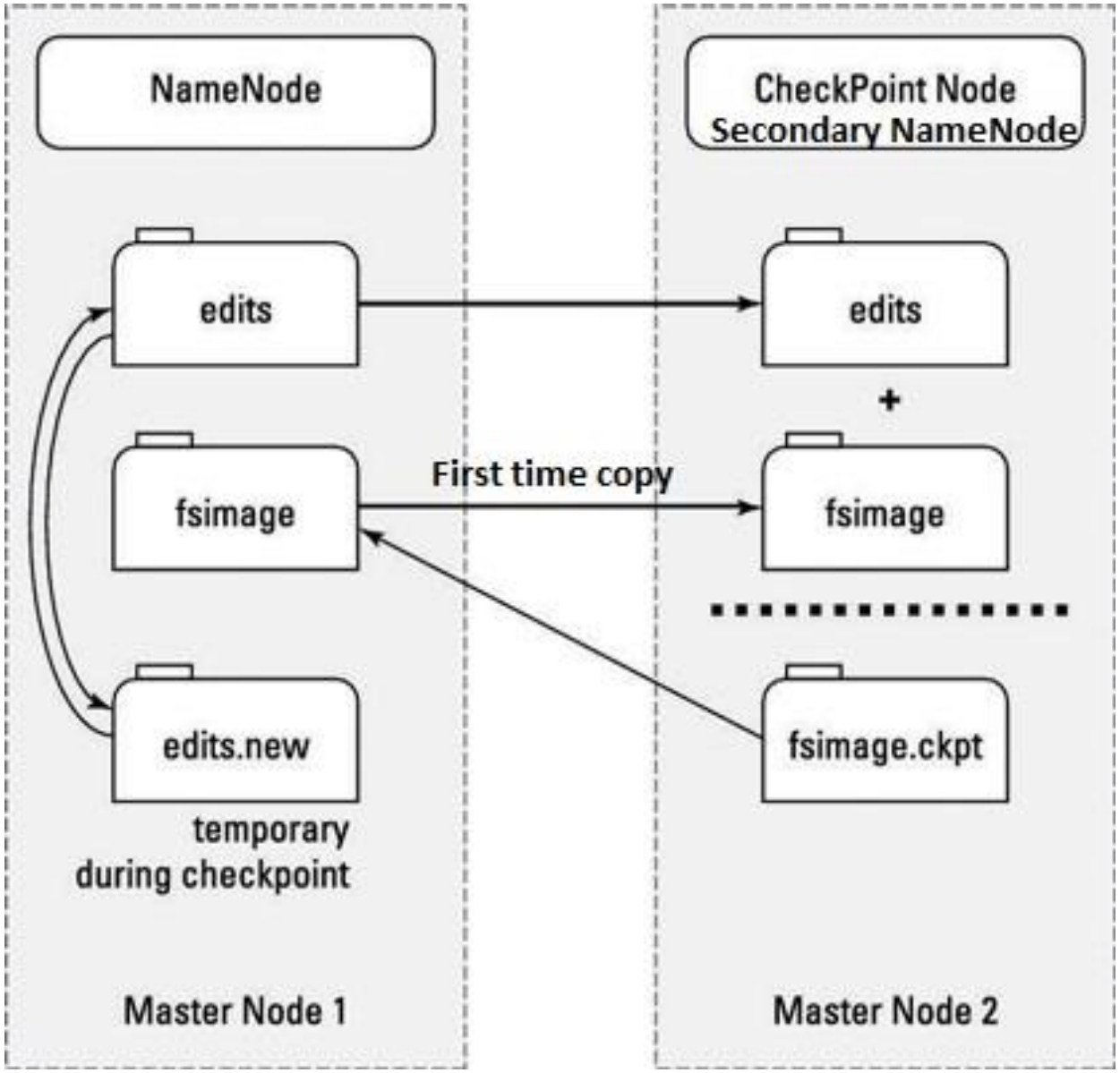
- The NameNode can load the final in-memory state directly from the fsimage → startup time reduced
  - This excludes the need of replaying a potentially unbounded edit log.

Checkpointing: An example



An example of HDFS metadata directory taken from a NameNode.

A typical workflow of checkpointing



DataNode

DataNode: Functionalities

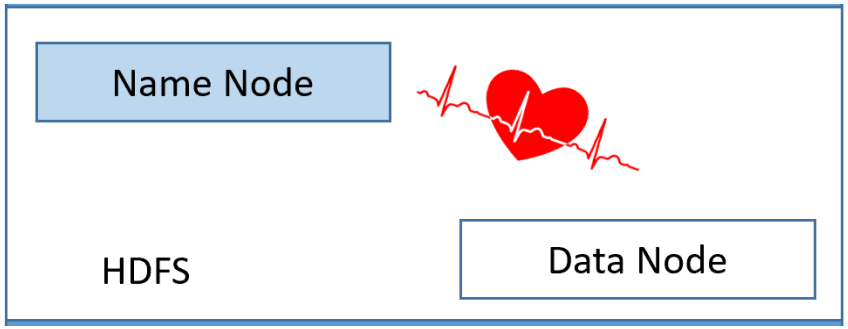
- Workhorses of the filesystem, registered with NameNode
- Store and retrieve blocks when they are told to (by clients or the NameNode)
- Report back to the NameNode periodically with lists of blocks that they are storing

DataNode: Functionalities

- Perform low-level read and write requests from clients
  - Enable pipelining of data, forward data to other specified DataNodes
- Manipulate blocks following the NameNode's command
  - Create blocks, deleting blocks and replicating them
  - Store each HDFS data block in separate files in its local filesystem
- Send heartbeats to the NameNode every 3 seconds
  - Total storage capacity, fraction of storage in use, and the number of data transfers currently in progress $\rightarrow$ the overall health of HDFS
- Periodically send a report on all the blocks present to the NameNode
  - Block ID, generation stamp and the length for each block replica
  - When getting started: scan the local filesystem to create a list of blocks corresponding to these local files and send a Block report to the NameNode

Heartbeat messages

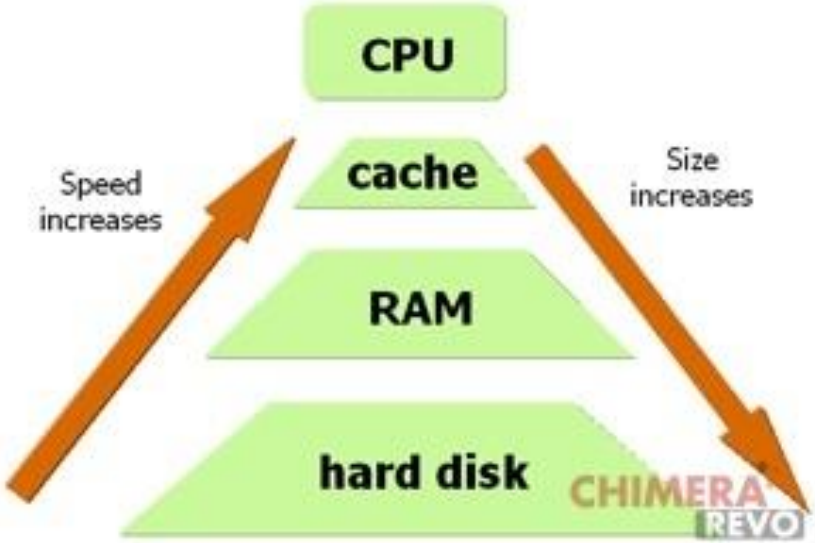
- These messages are periodically sent from each DataNode to the NameNode, ensuring the connectivity between nodes.
  - The default interval is three seconds.



- NameNode marks those with missing heartbeats as dead DataNodes.
  - Data stored on a dead node is no longer available, which is effectively removed from the system.
  - Further requests will not be sent to dead nodes.
- Some blocks may no longer have enough copies, which is specified by the replication factor.
  - The NameNode initiates additional replication to meet the requirement.

Block caching

- Blocks of frequently accessed files may be explicitly cached in the DataNode's memory, in an off-heap block cache.
  - By default, a block is cached in the memory of only one DataNode.
- The user instructs the NameNode which files to cache (and how long) by adding a cache directive to a cache pool.

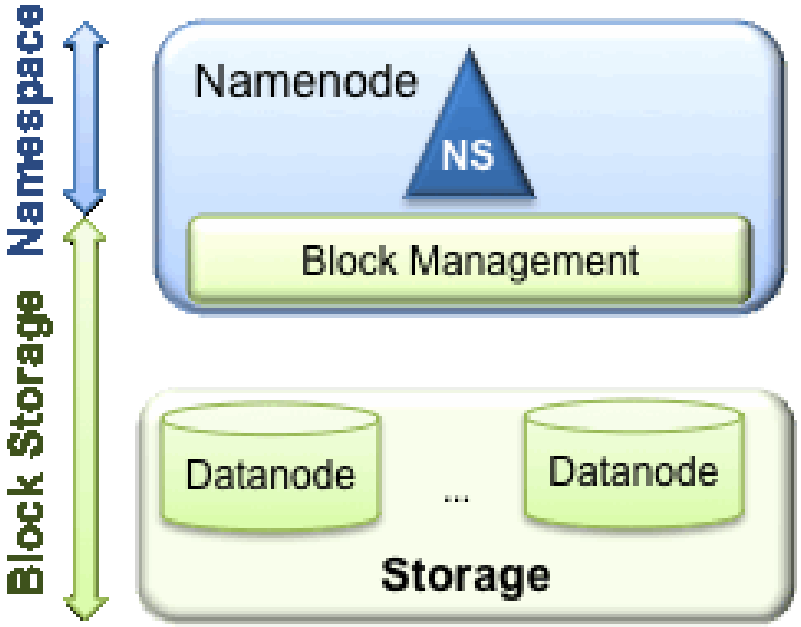


HDFS Federation

HDFS in Hadoop 1.0

Namespace

- Consist of directories, files and blocks
- Support all the namespace related operations, e.g., create, delete, and modify files and directories

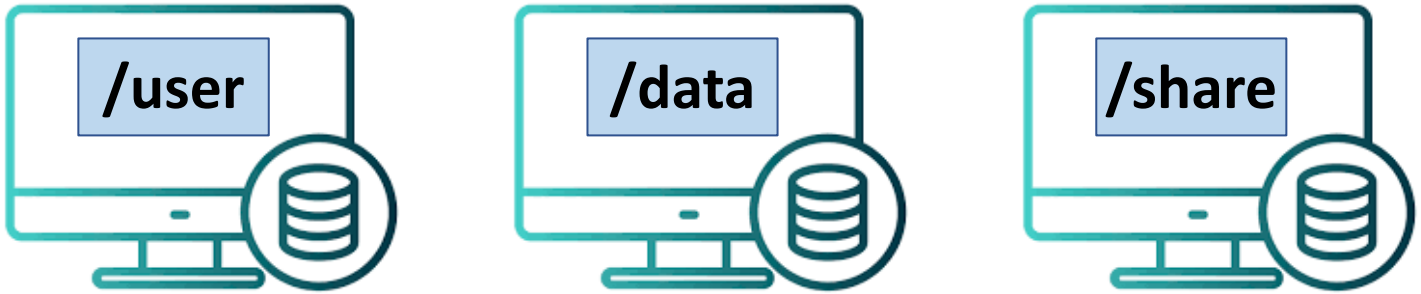


Block Storage

- **Block Management** (done in NameNode)
  - Provide DataNode cluster membership
  - Manage blocks: locations, reports, and related operations
  - Govern the replication of blocks
- **Storage** (provided by DataNode)
  - Store blocks on the local filesystem and allow read/write access

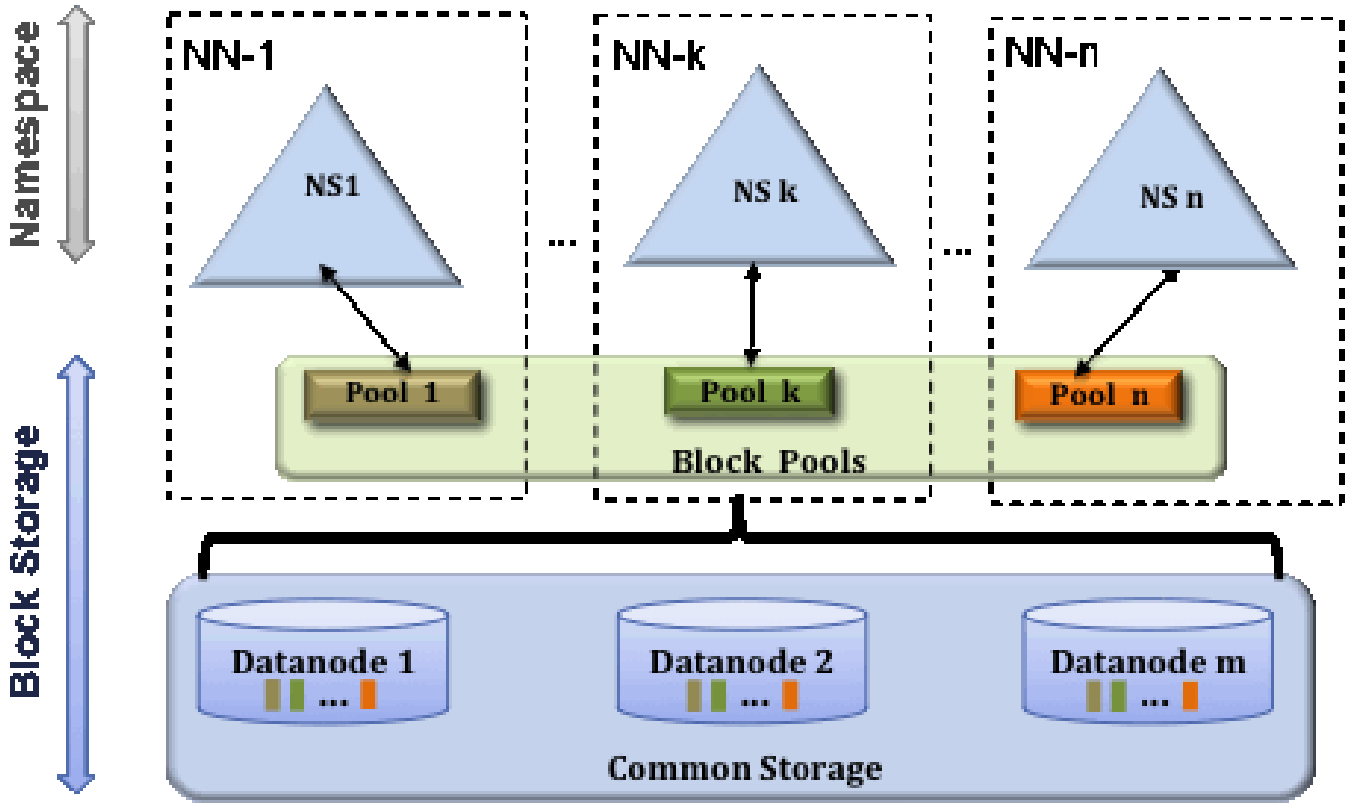
HDFS Federation in Hadoop 2.0

- HDFS Federation allows for adding multiple NameNodes, and hence multiple namespaces, to the HDFS filesystem

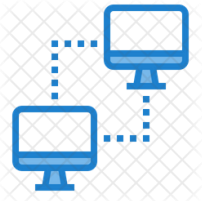


- Namespace volumes are independent of each other, the failure of one NameNode does not affect the others.

HDFS Federation in Hadoop 2.0



- Namespace Volume = Namespace + Block Pool
- Block Storage as generic storage service
  - The set of blocks for a Namespace Volume is called a **Block Pool**.
  - DataNodes store blocks for all the Namespace Volumes – **no partitioning**.

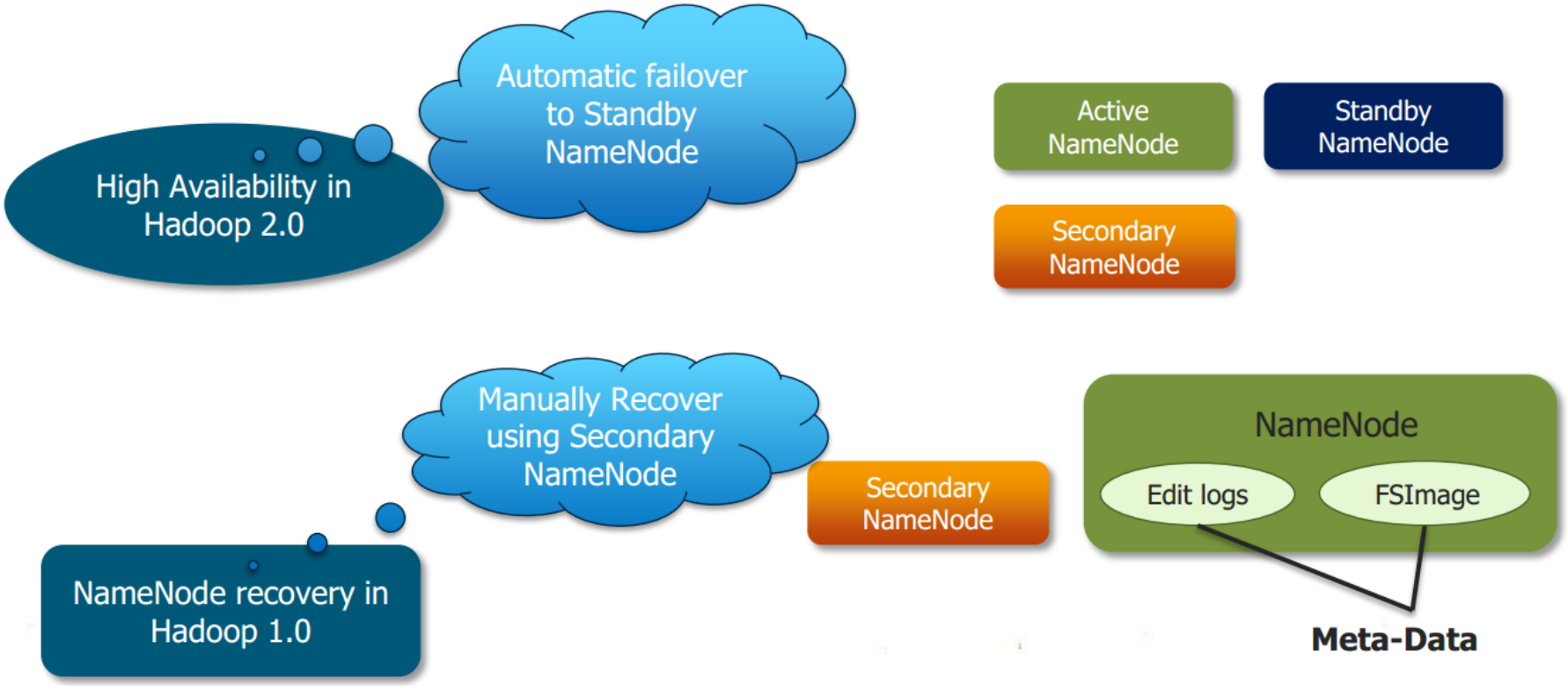


HDFS High Availability

NameNode failure and recovery

- NameNode in HDFS 1.0 is a Single Point of Failure (SPOF).
  - Secondary NameNode only keeps the checkpoints, yet it cannot take over the role of a primary NameNode.
- Solution: start a new primary NameNode and configure DataNode and clients to use this new NameNode
  - Note that the NameNode is not able to serve requests until it has
    - Loaded its namespace image into memory.
    - Replayed its edit log, and
    - Received enough block reports from DataNodes to leave safe mode.
  - If clusters is large, it takes about 30 minutes or more.

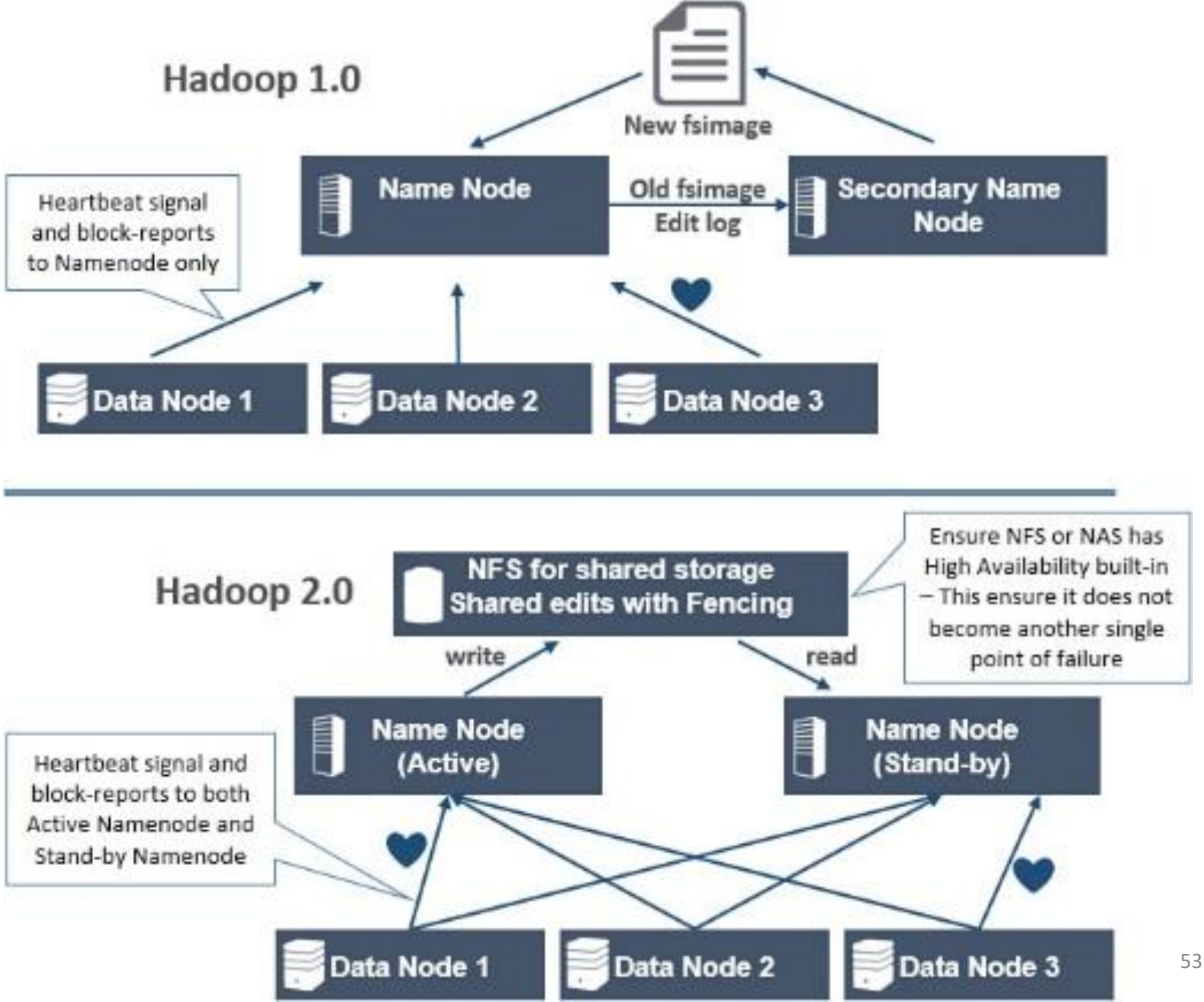
HDFS High Availability (HA)



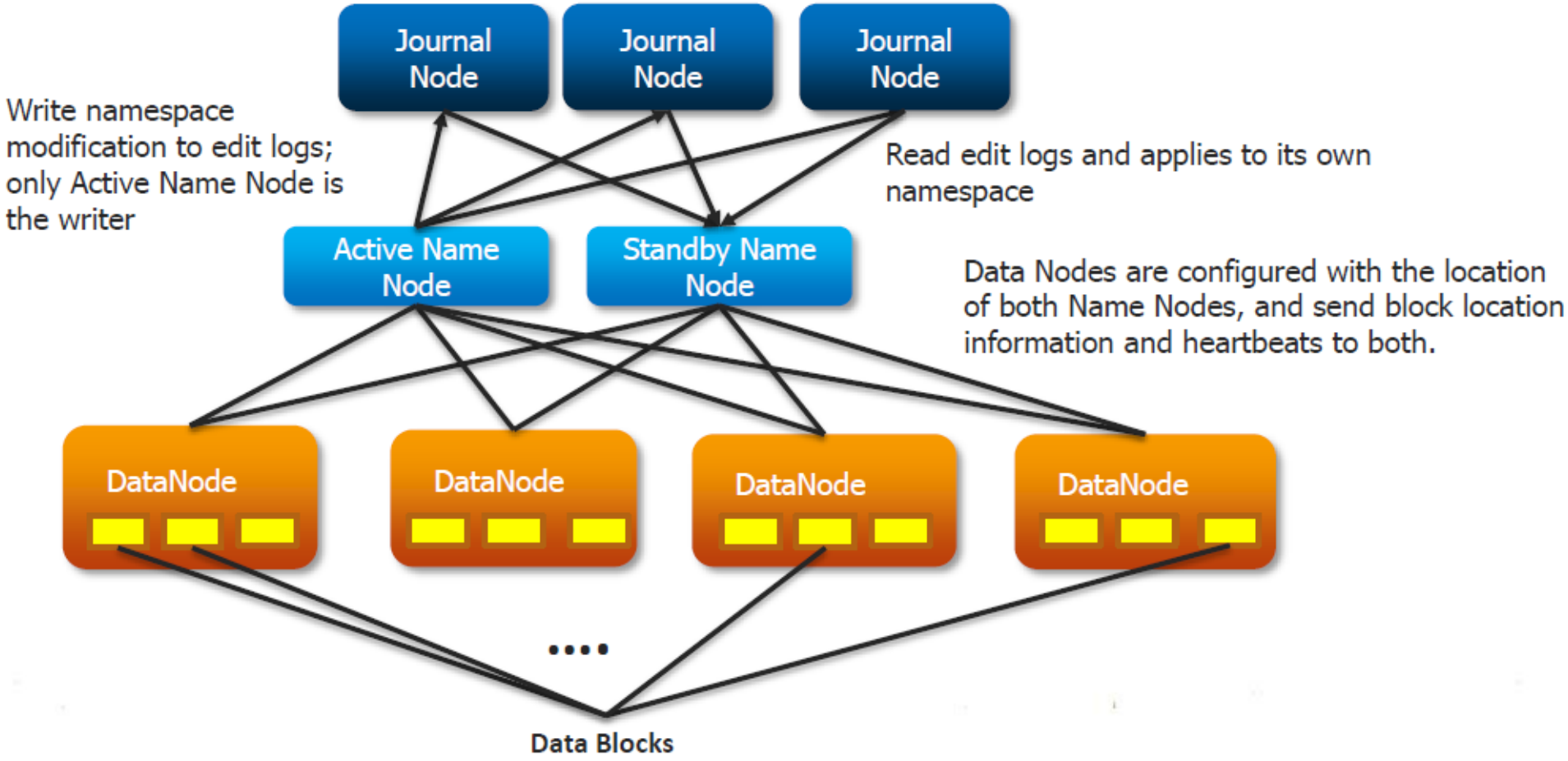
- A pair of NameNodes in an active-standby mode
- In failure, the standby takes over its duties to continue servicing client requests without a significant interruption.

HDFS High Availability (HA)

- HDFS HA uses shared storage to share edit logs
  - Shared storage techniques: NFS filer, Quorum journal manager (QJM), etc.
- A standby NameNode frequently synchronizes its states with the active NameNode via reading the shared edit log.
  - It read new log entries as they are written by the active NameNode.
- DataNodes must send block requests to both NameNodes.
- Clients must be configured to handle NameNode failover.


Hadoop 1.0
New fsimage
Name Node
Old fsimage
Edit log
Secondary Name Node
Heartbeat signal and block-reports to Namenode only
Data Node 1
Data Node 2
Data Node 3
Hadoop 2.0
NFS for shared storage
Shared edits with Fencing
Ensure NFS or NAS has High Availability built-in – This ensure it does not become another single point of failure
write
read
Name Node (Active)
Name Node (Stand-by)
Heartbeat signal and block-reports to both Active Namenode and Stand-by Namenode
Data Node 1
Data Node 2
Data Node 3

Shared storage with Journal Nodes

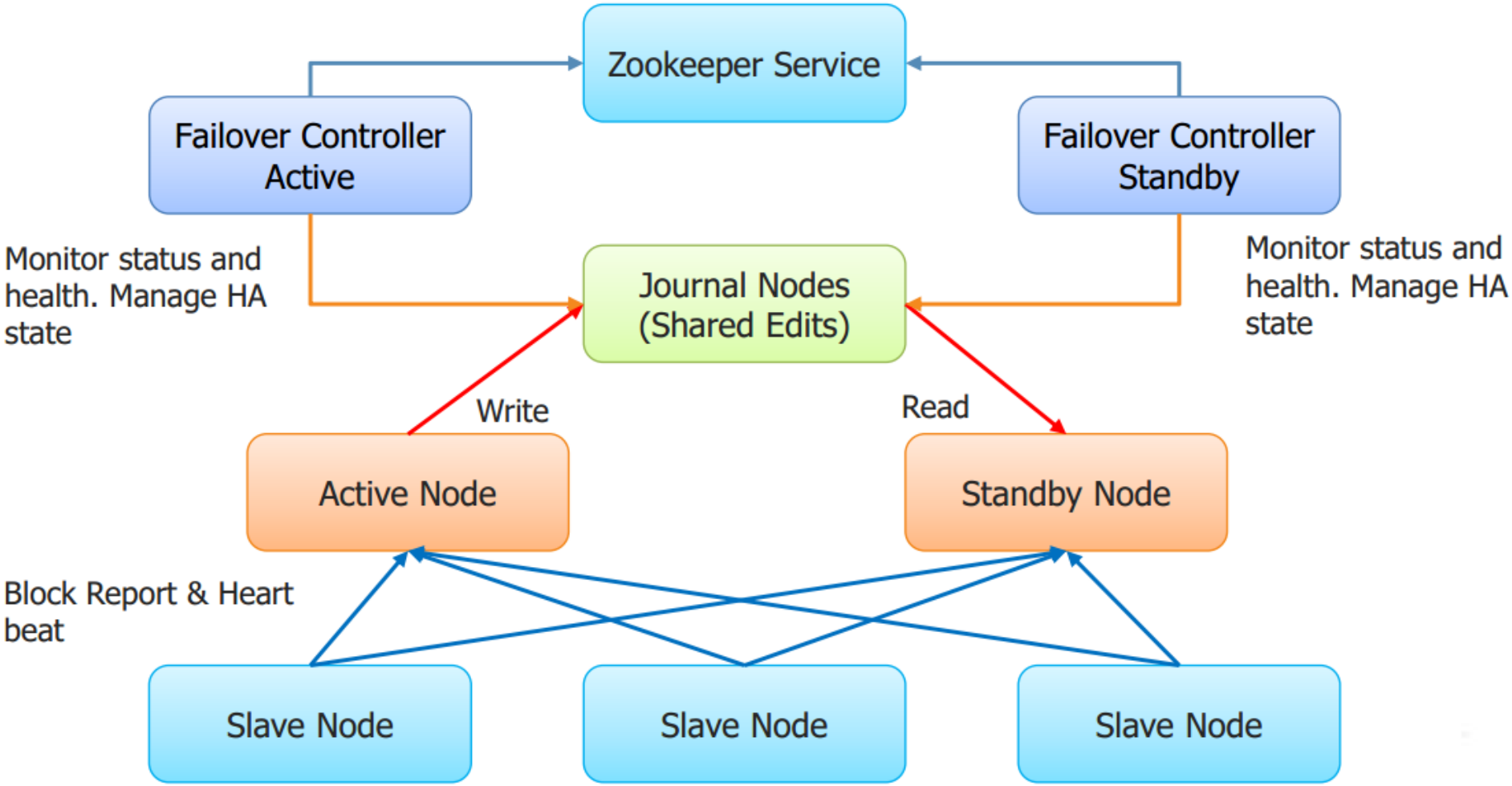


Failover controller

- An entity integrated in the system that governs the transition from the active NameNode to the standby one.
  - ZooKeeper Failover Controller (ZKFC), etc.
- It arranges an orderly transition for both NameNodes to switch roles.

- **Graceful failover:** A kind of failover that may be initiated manually by an administrator, e.g., routine maintenance.

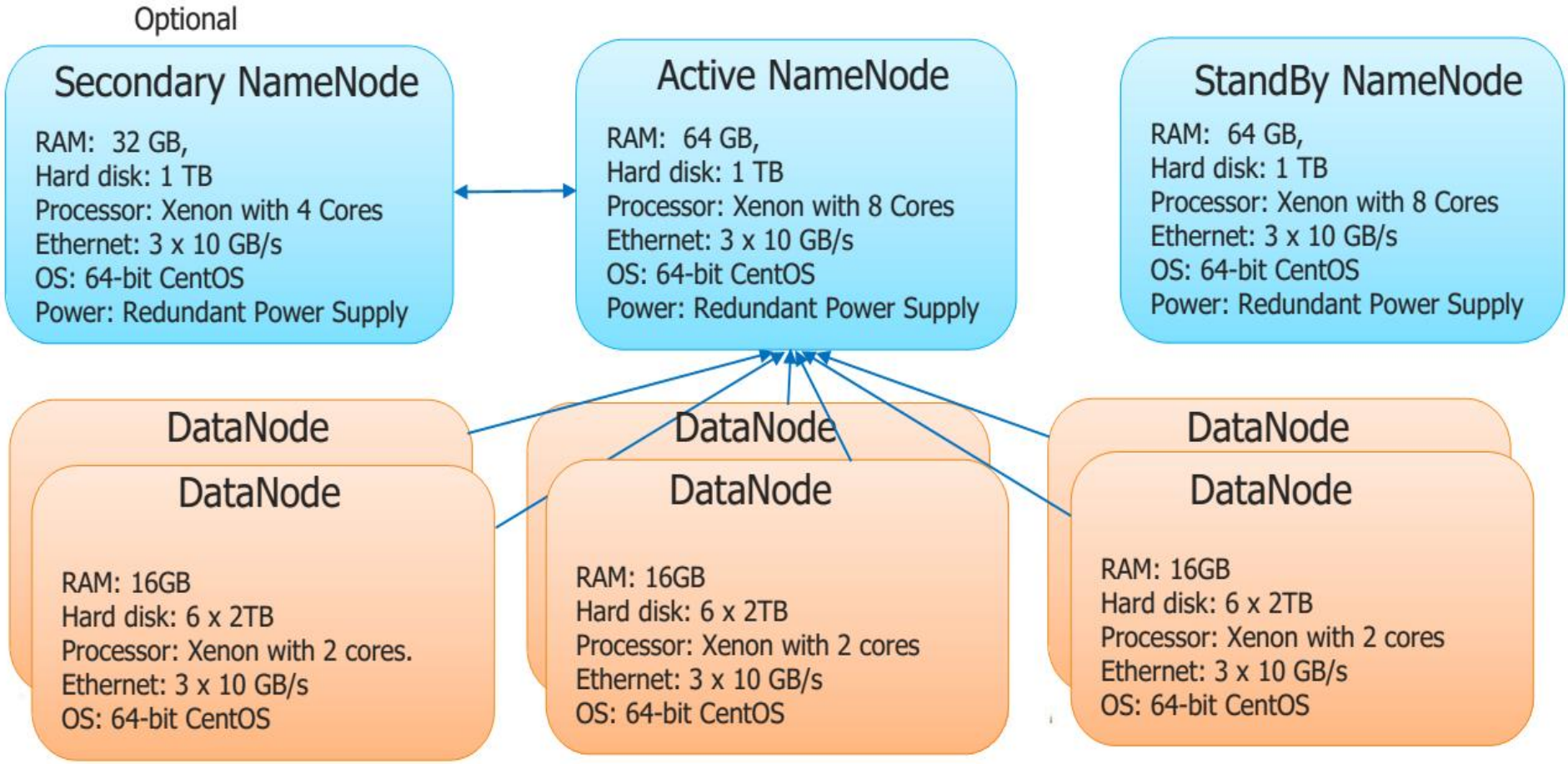
Failover controller



Fencing

- It is not sure that the failed NameNode has stopped running.
  - A slow network / network partition may trigger a failover transition.
  - The (previously) active NameNode is still running and thinks that it is still the main NameNode.
- Fencing ensures that the previously active NameNode is prevented from doing any damage and causing corruption.
  - It can be implemented by using ZooKeeper or something similar.

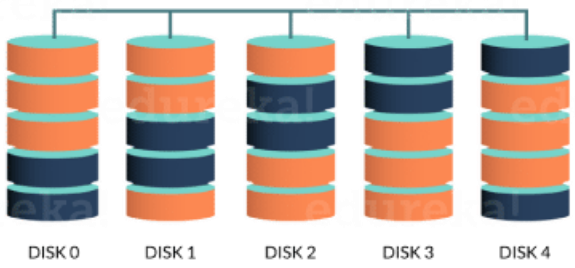
An example of HDFS configurations

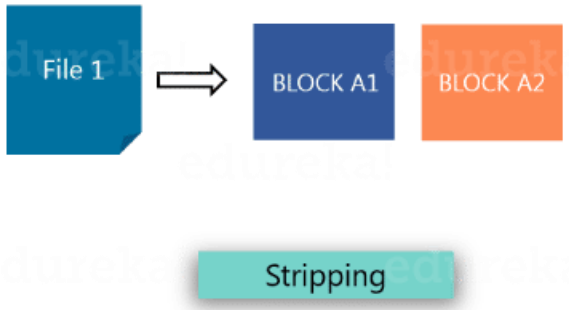


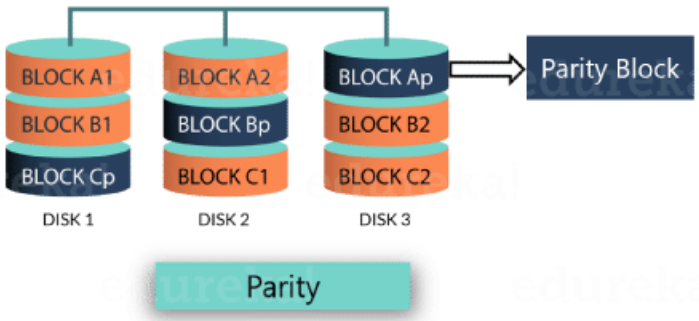
HDFS in Hadoop 3.0

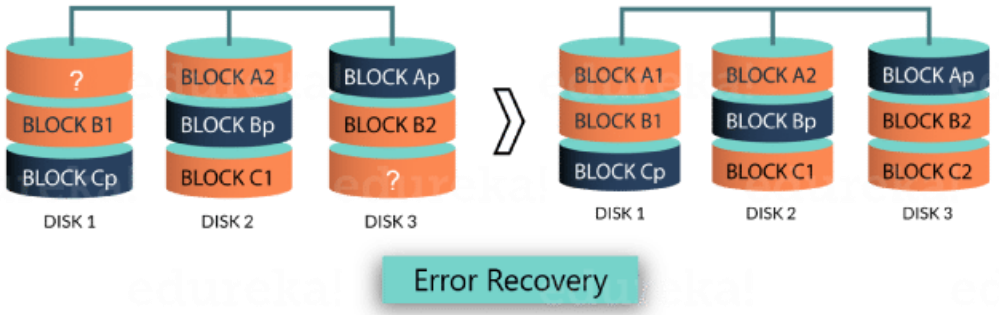
Support for Erasure Encoding

- Erasure Coding is mostly used in RAID.



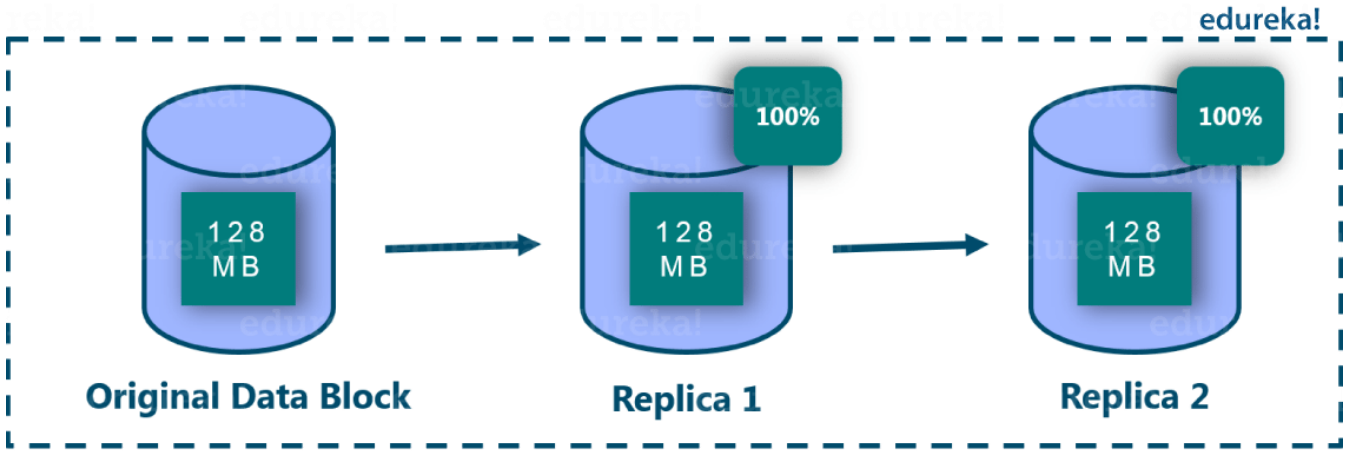


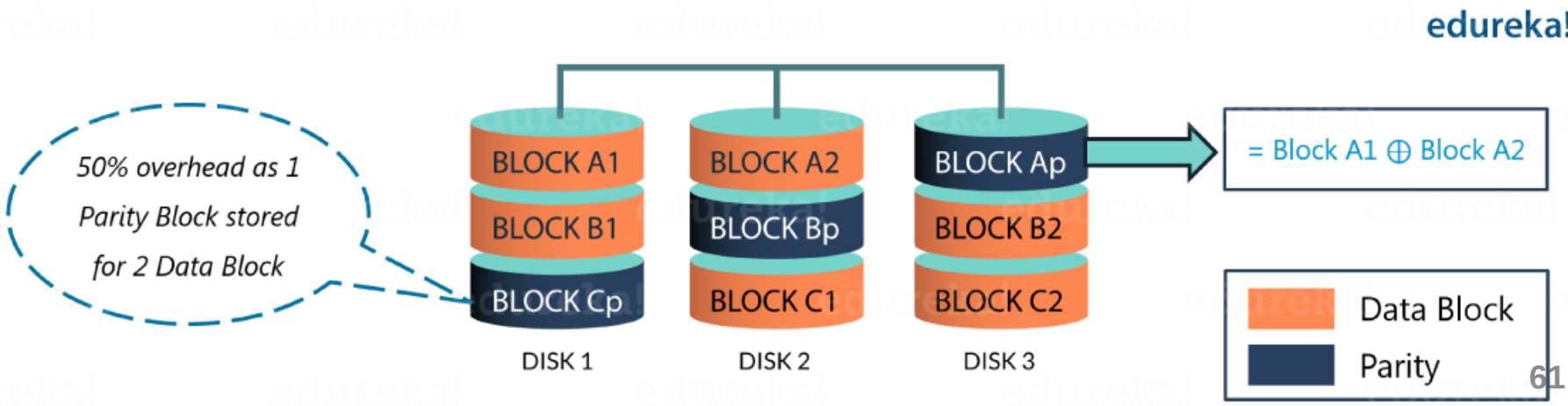




Support for Erasure Encoding

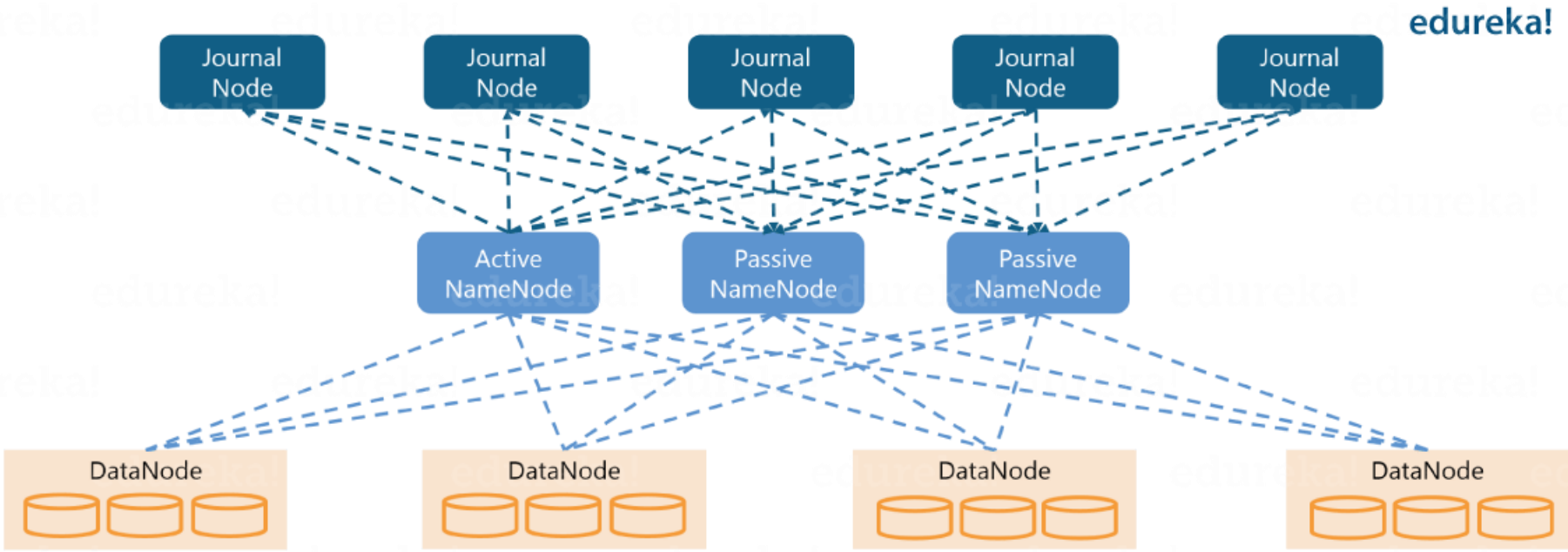
- Store the data and provide fault tolerance with less space overhead as compared to HDFS replication





Support more Standby NameNodes

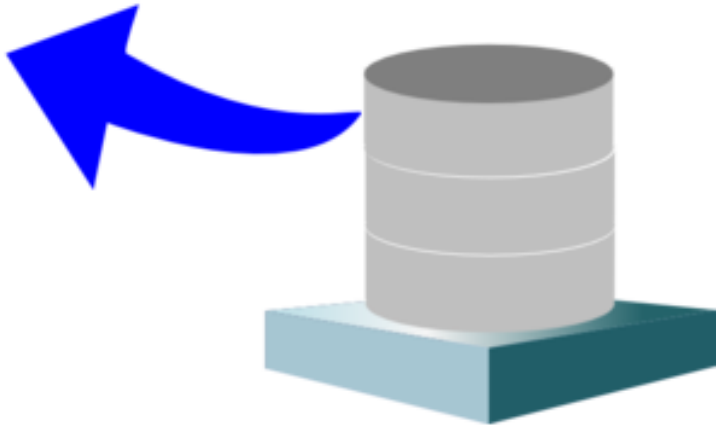
- Business critical deployments require higher degrees of fault-tolerance.



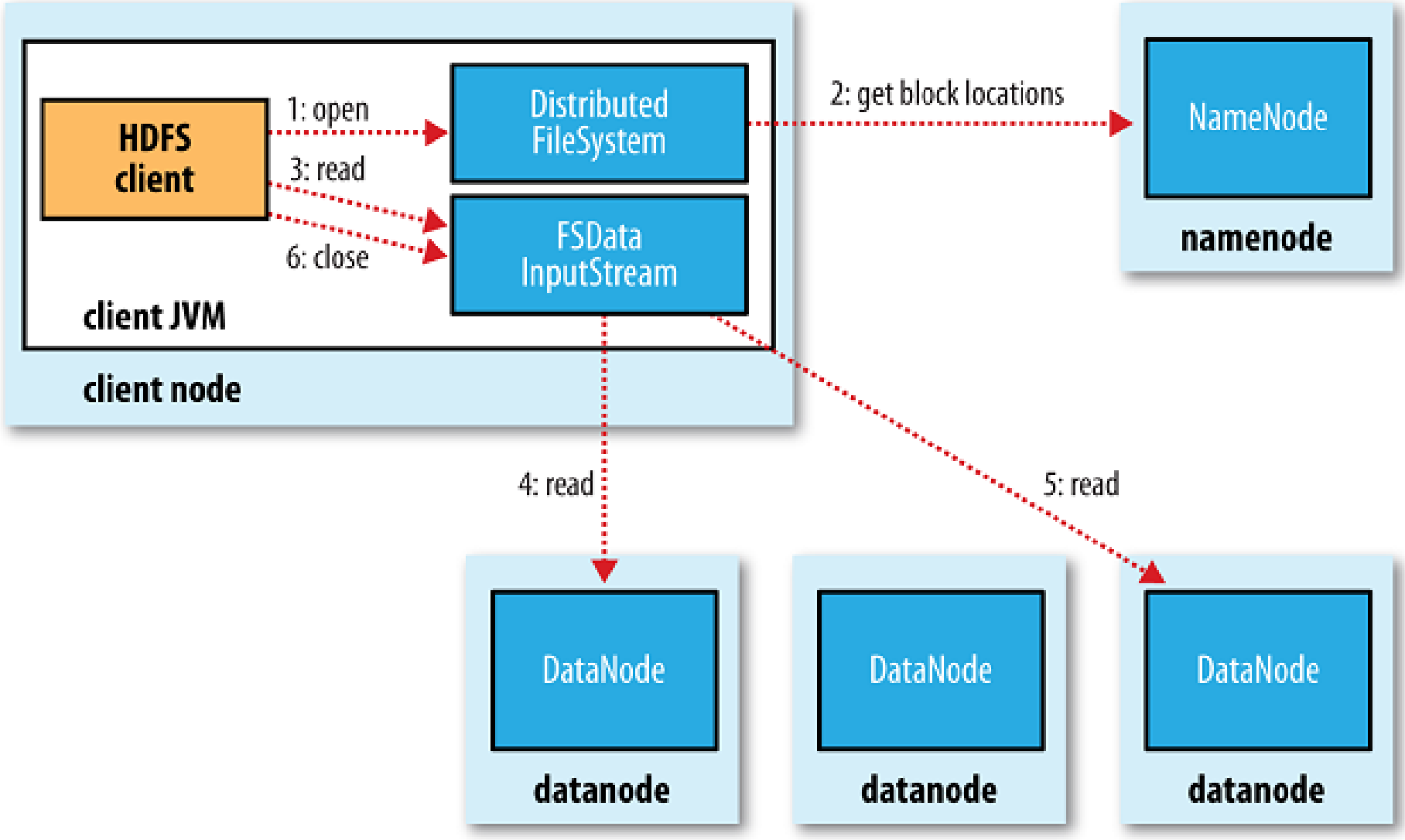
Other improvements

- Support for Filesystem Connector
  - Hadoop now supports integration with Microsoft Azure Data Lake and Aliyun Object Storage System.
- Intra-DataNode Balancer
  - A single DataNode manages multiple disk, and thus there will be a similar problem to Unbalanced cluster
  - This can be handled via the hdfs diskbalancer CLI.

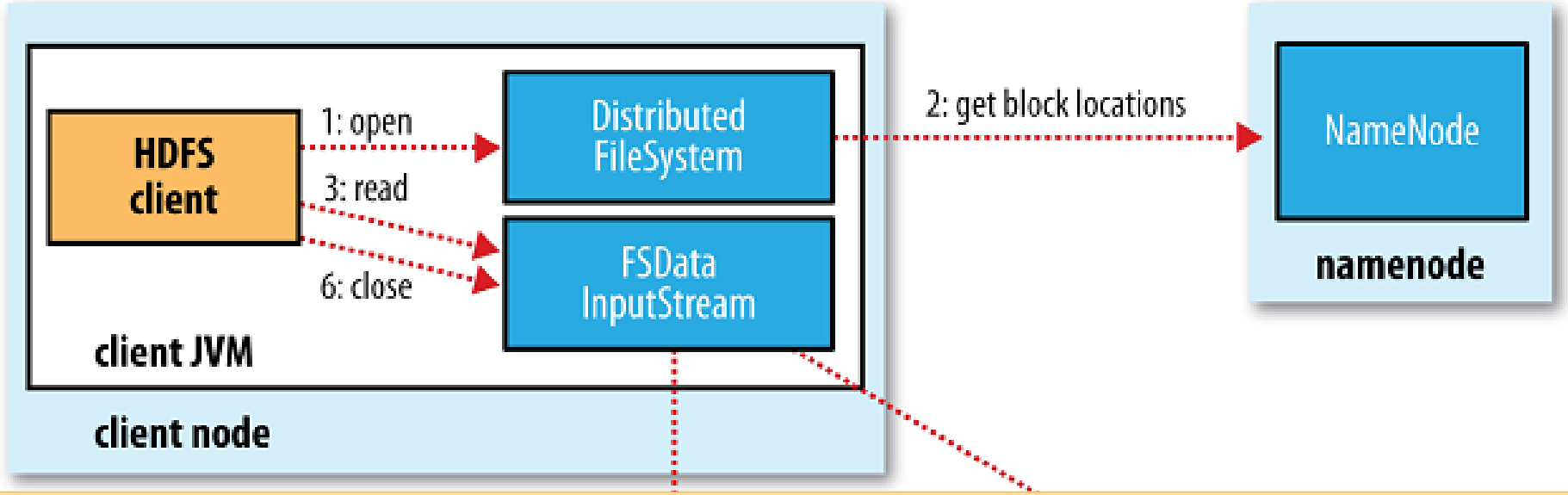
Read data from HDFS



An anatomy of a file read

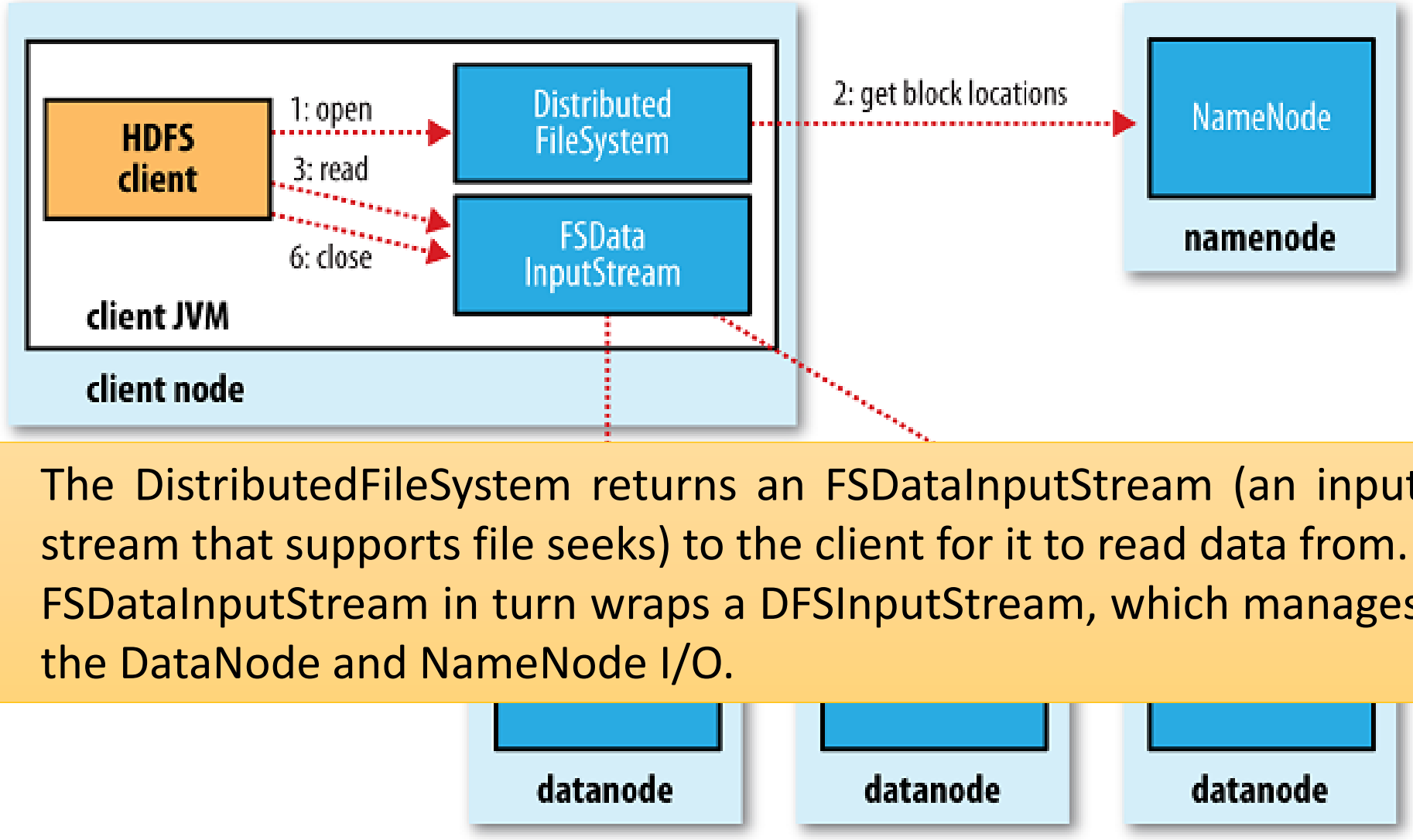


An anatomy of a file read



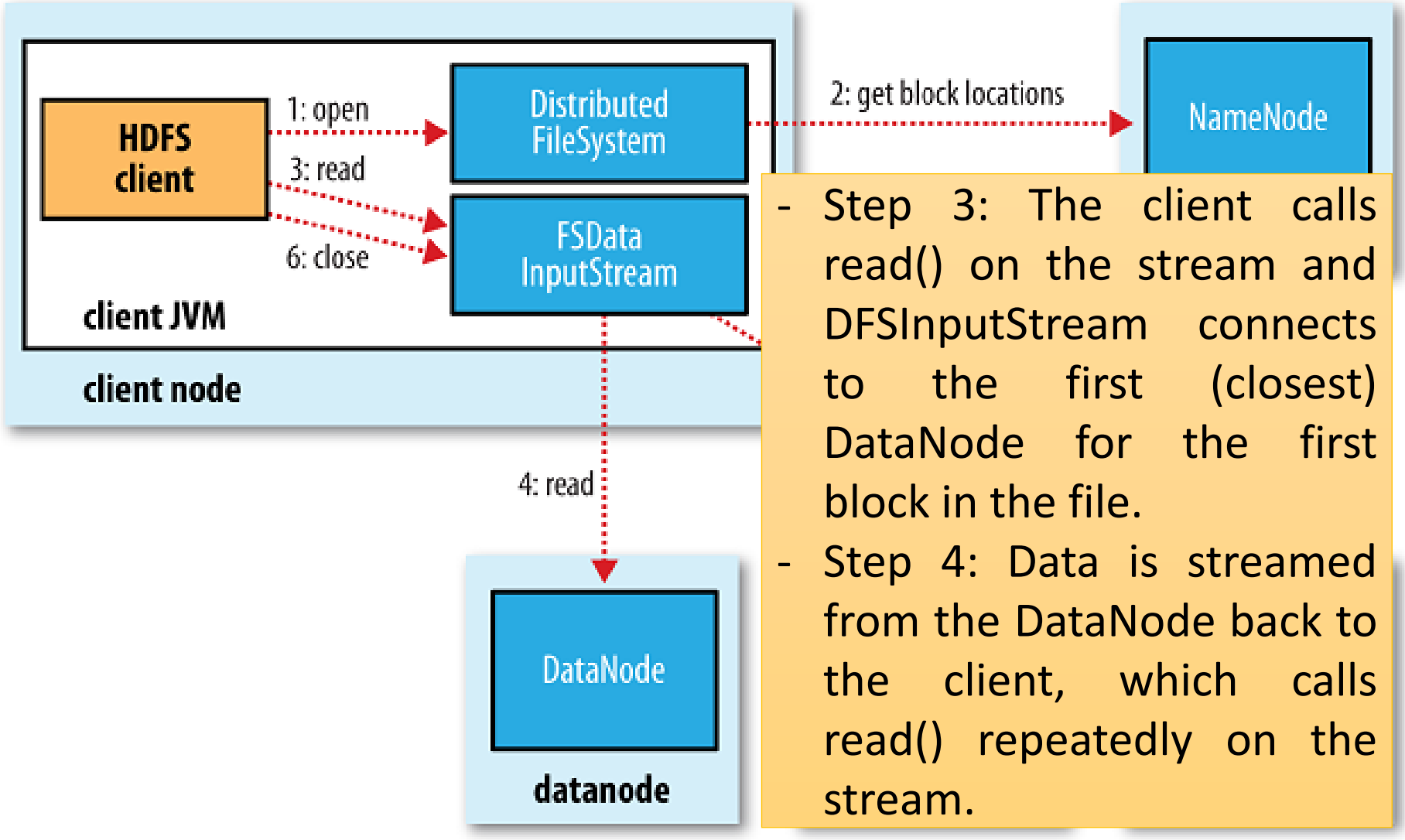
- Step 1: The client calls open() on the FileSystem object.
- Step 2: DistributedFileSystem calls the NameNode to determine the locations of the first few blocks in the file.
  - *For each block, the NameNode returns the addresses of the DataNodes that have a copy of that block, which are sorted according to the proximity between the node and the client.*
  - *If the client is itself a DataNode and it hosts a copy of the block, the client will read from the local storage.*

An anatomy of a file read



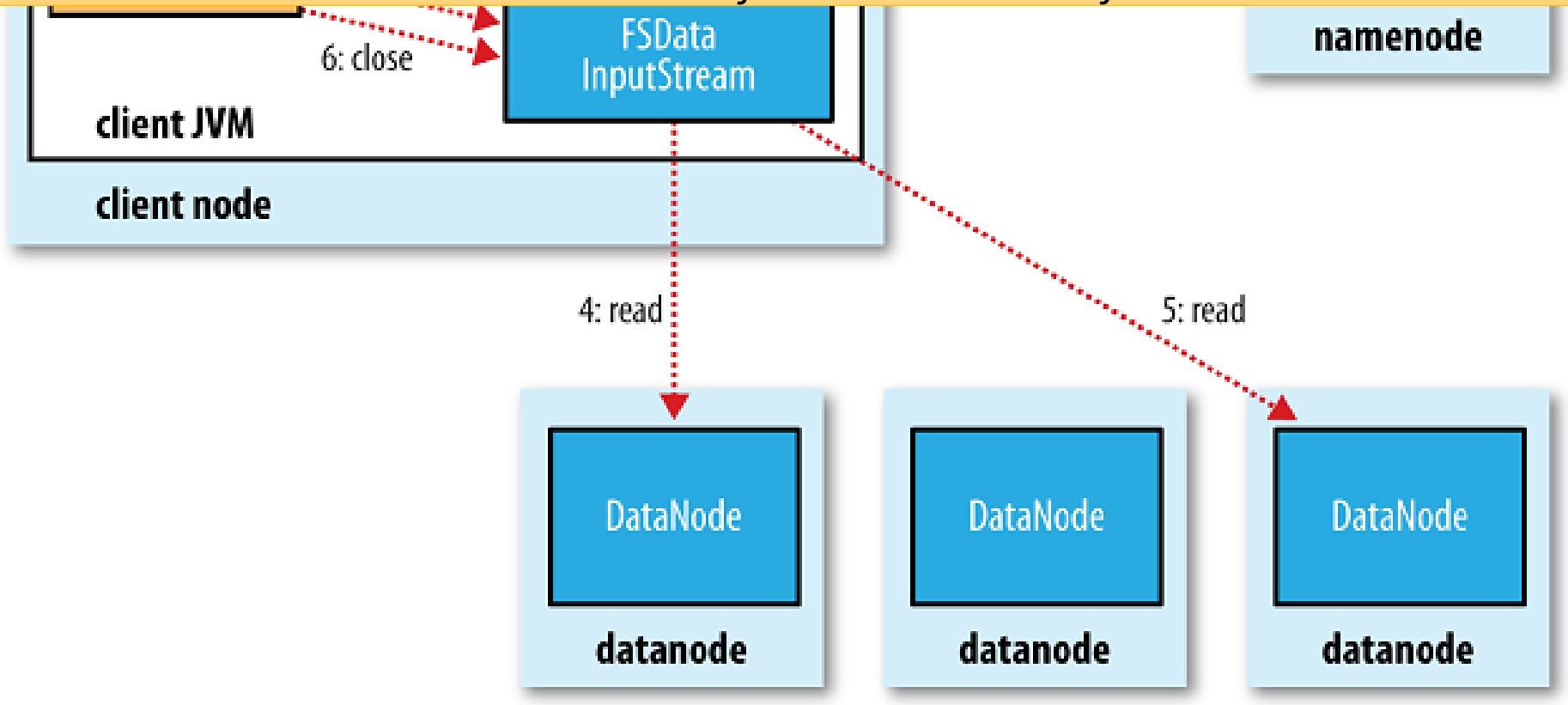
- The DistributedFileSystem returns an FSDataInputStream (an input stream that supports file seeks) to the client for it to read data from.
- FSDataInputStream in turn wraps a DFSInputStream, which manages the DataNode and NameNode I/O.

An anatomy of a file read

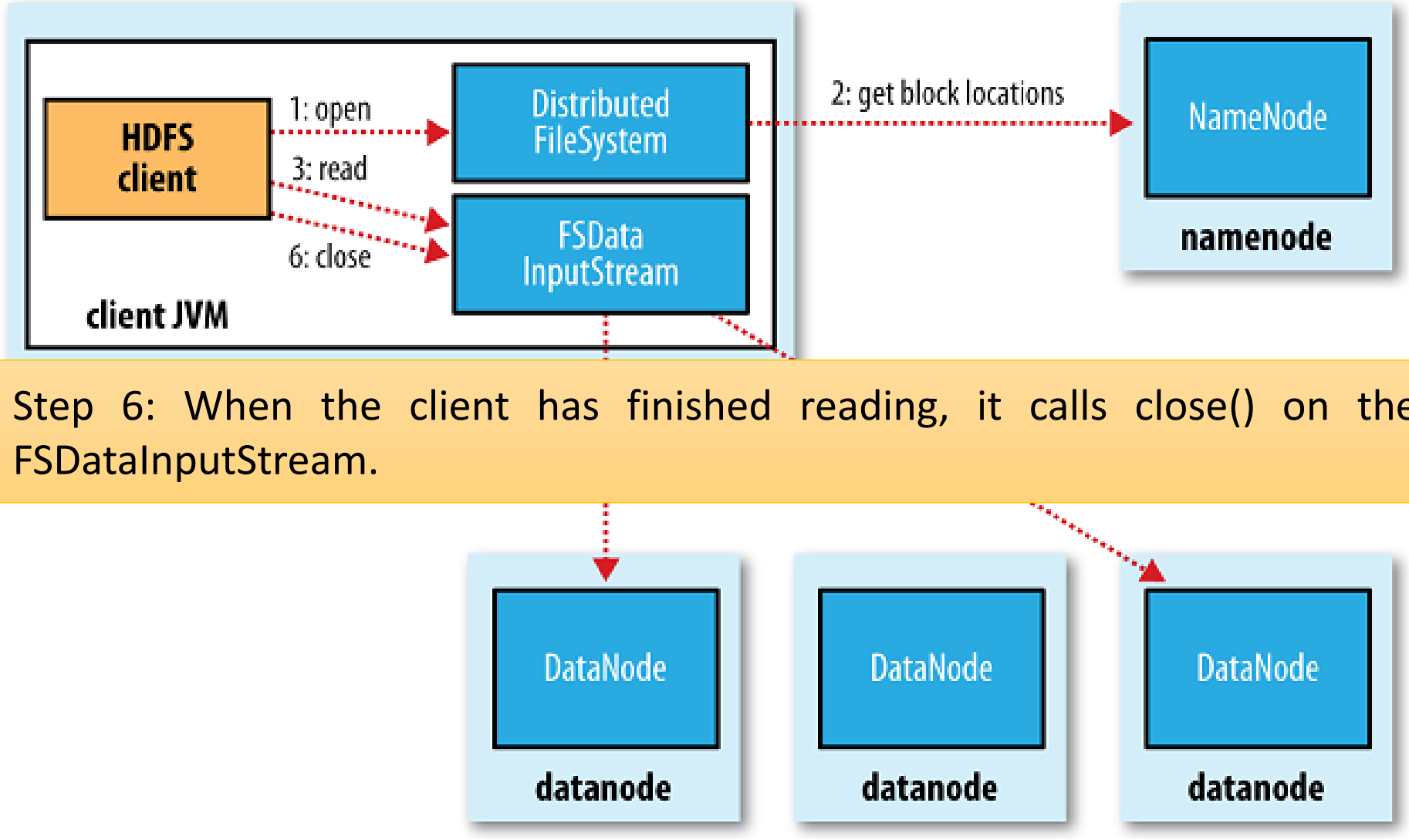


- Step 3: The client calls read() on the stream and DFSInputStream connects to the first (closest) DataNode for the first block in the file.
- Step 4: Data is streamed from the DataNode back to the client, which calls read() repeatedly on the stream.

- Step 5: When the end of the block is reached, DFSInputStream will close the connection to the DataNode, then find the best DataNode for the next block.
  - *This happens transparently to the client, which from its point of view is just reading a continuous stream.*
  - *Blocks are read in order. The DFSInputStream opens new connections to DataNodes as the client reads through the stream. It also calls the NameNode to retrieve the DataNode locations for the next batch of blocks as needed.*



An anatomy of a file read

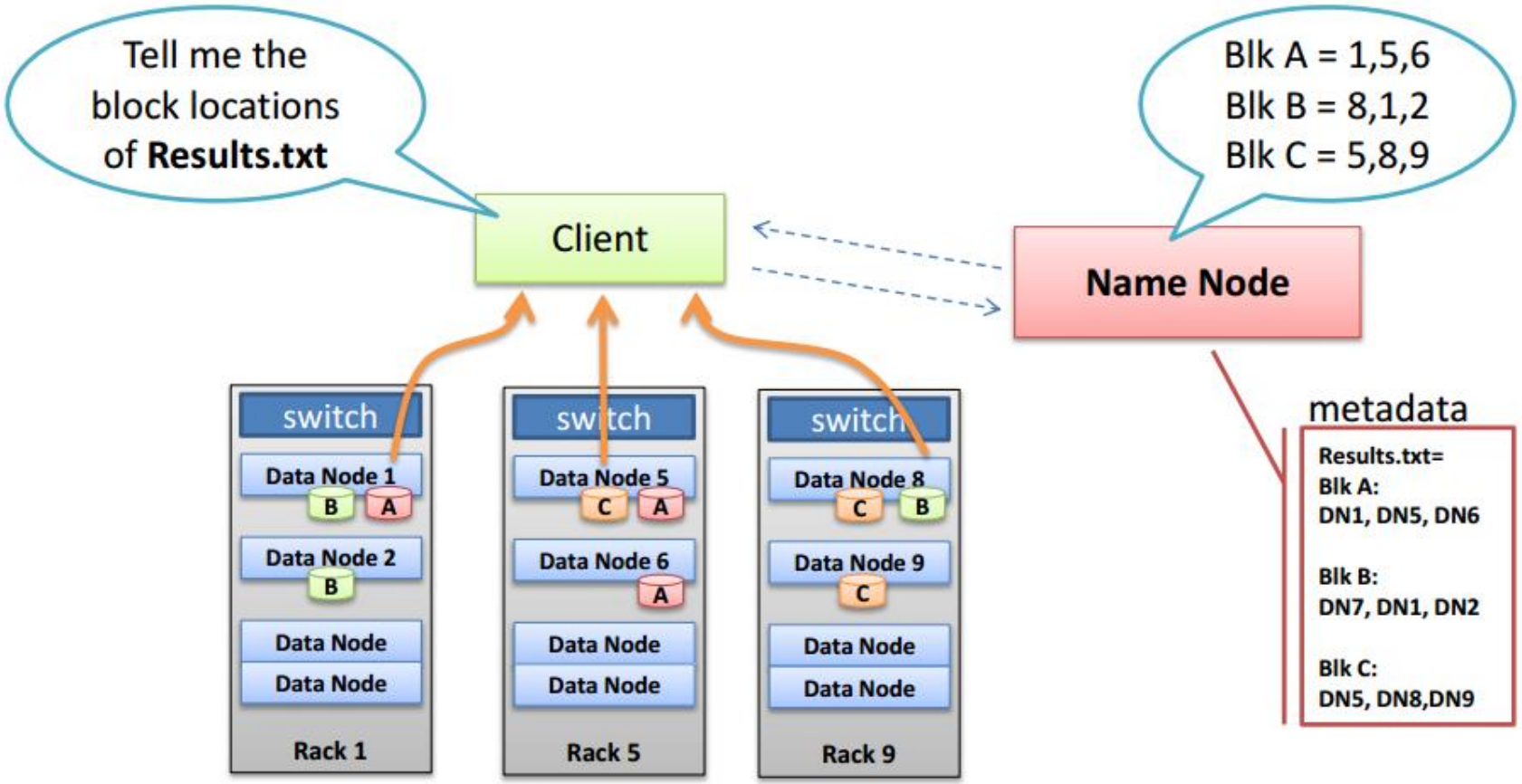


- Step 6: When the client has finished reading, it calls close() on the FSDataInputStream.

An anatomy of a file read

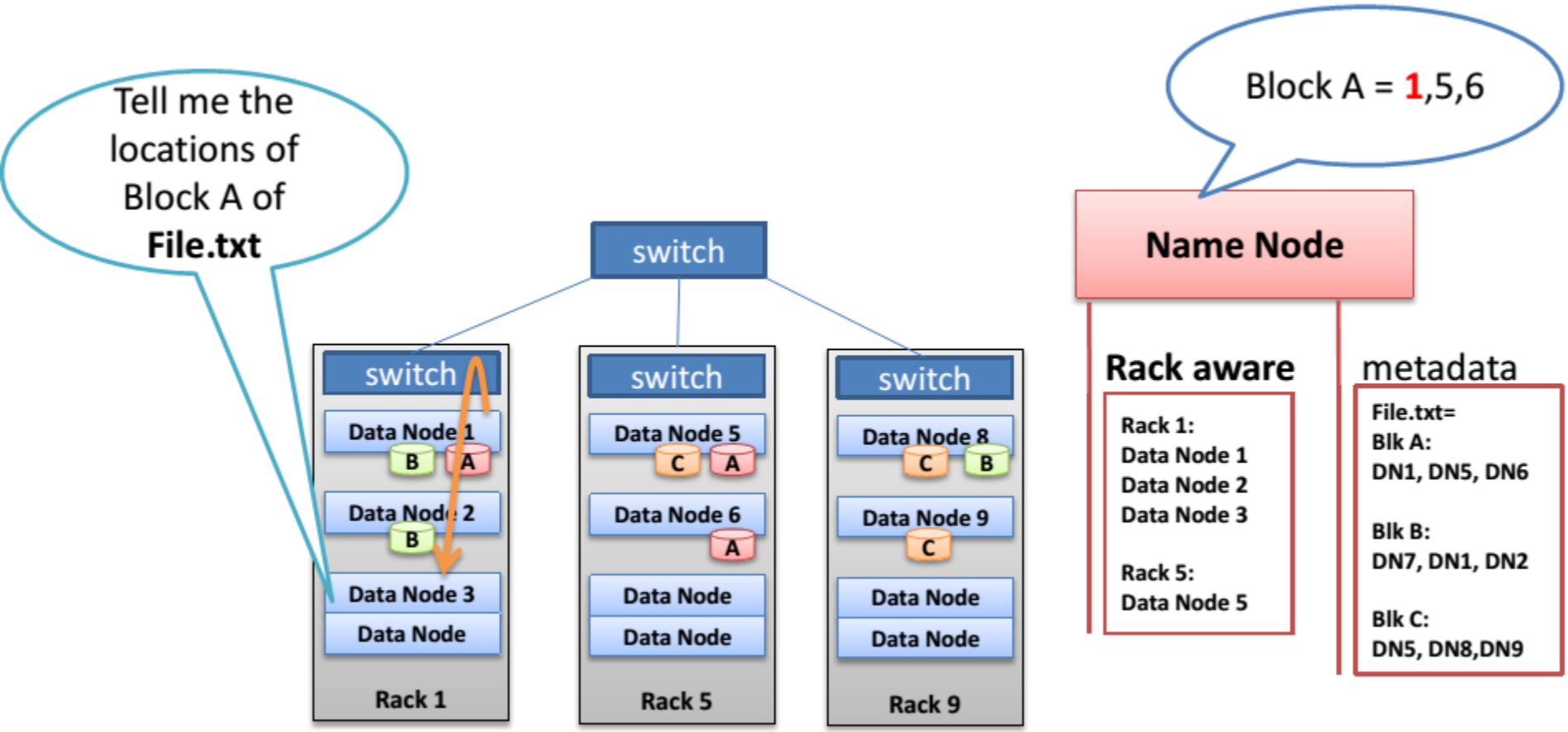
- DFSInputStream will try the next closest DataNode for a block if it fails to communicate with current node.
  - The failed DataNode is marked so that the DFSInputStream does not needlessly retry them for later blocks.
- It also verifies checksums for the data transferred.
  - If a corrupted block is found, the DFSInputStream reads a replica of the block from another DataNode.
  - It also reports the corrupted block to the NameNode.

Read data from HDFS: An example



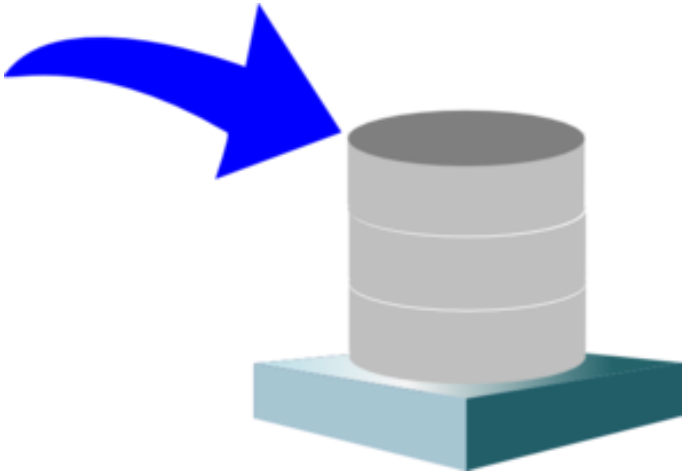
- The client receives from the NameNode a list of DataNodes for each block, and then picks the first Data Node for each block.
- Client reads blocks sequentially.

Read data from HDFS: An example

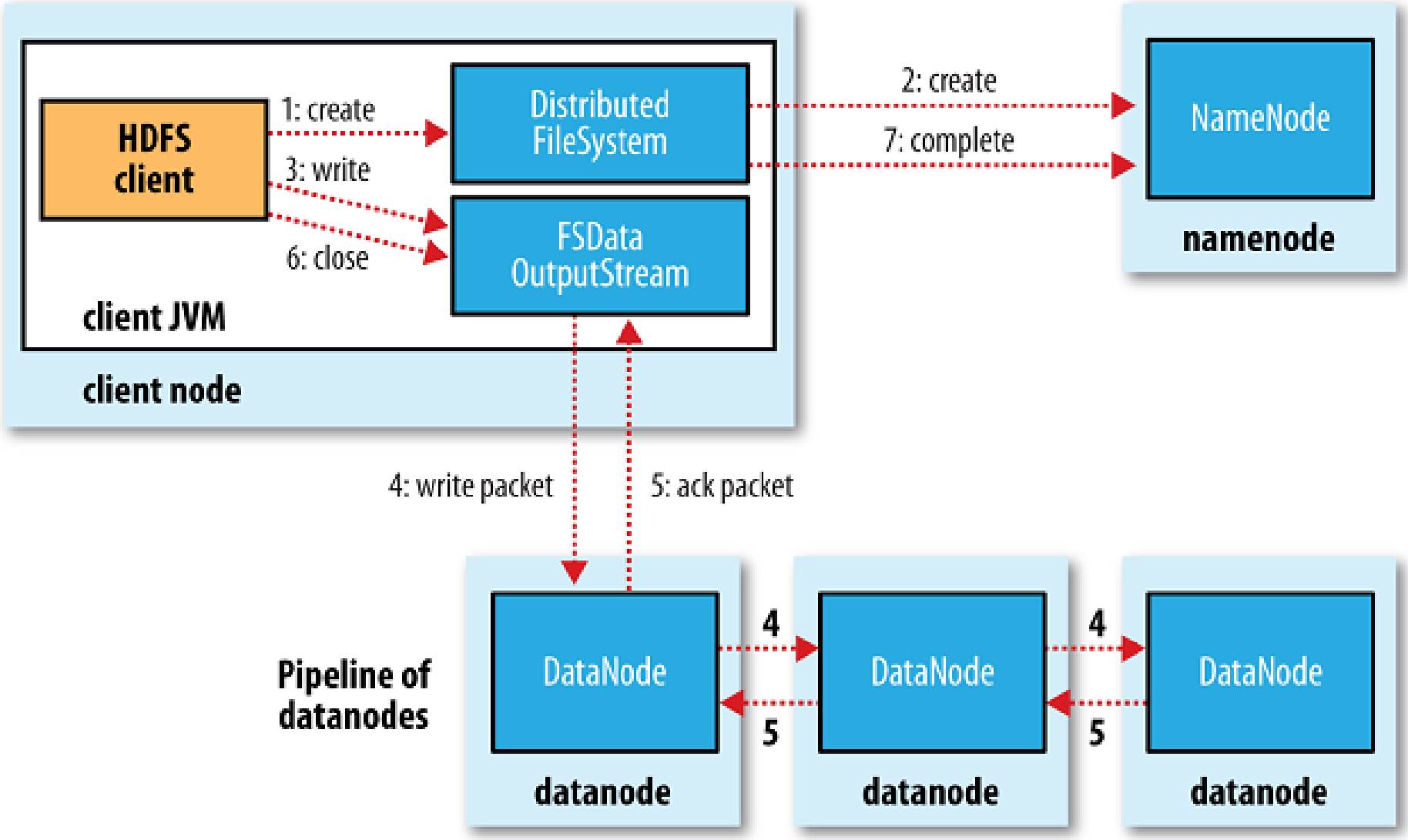


- Name Node provides rack-local nodes first.
- Leverage in-rack bandwidth, single hop.

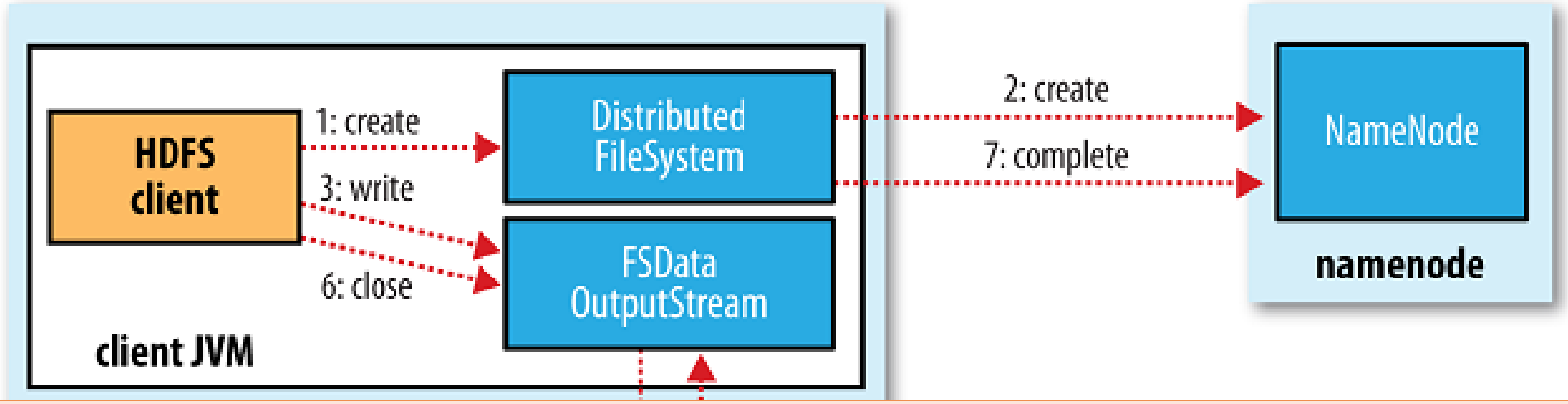
Write data to HDFS



An anatomy of a file write

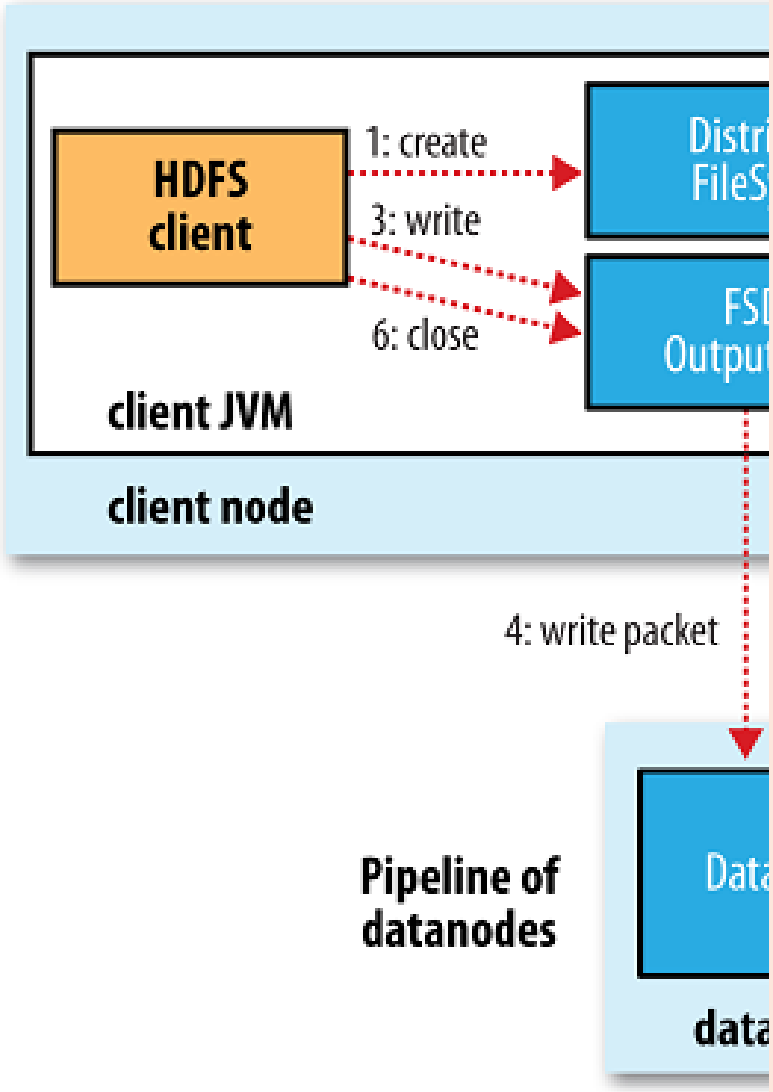


An anatomy of a file write



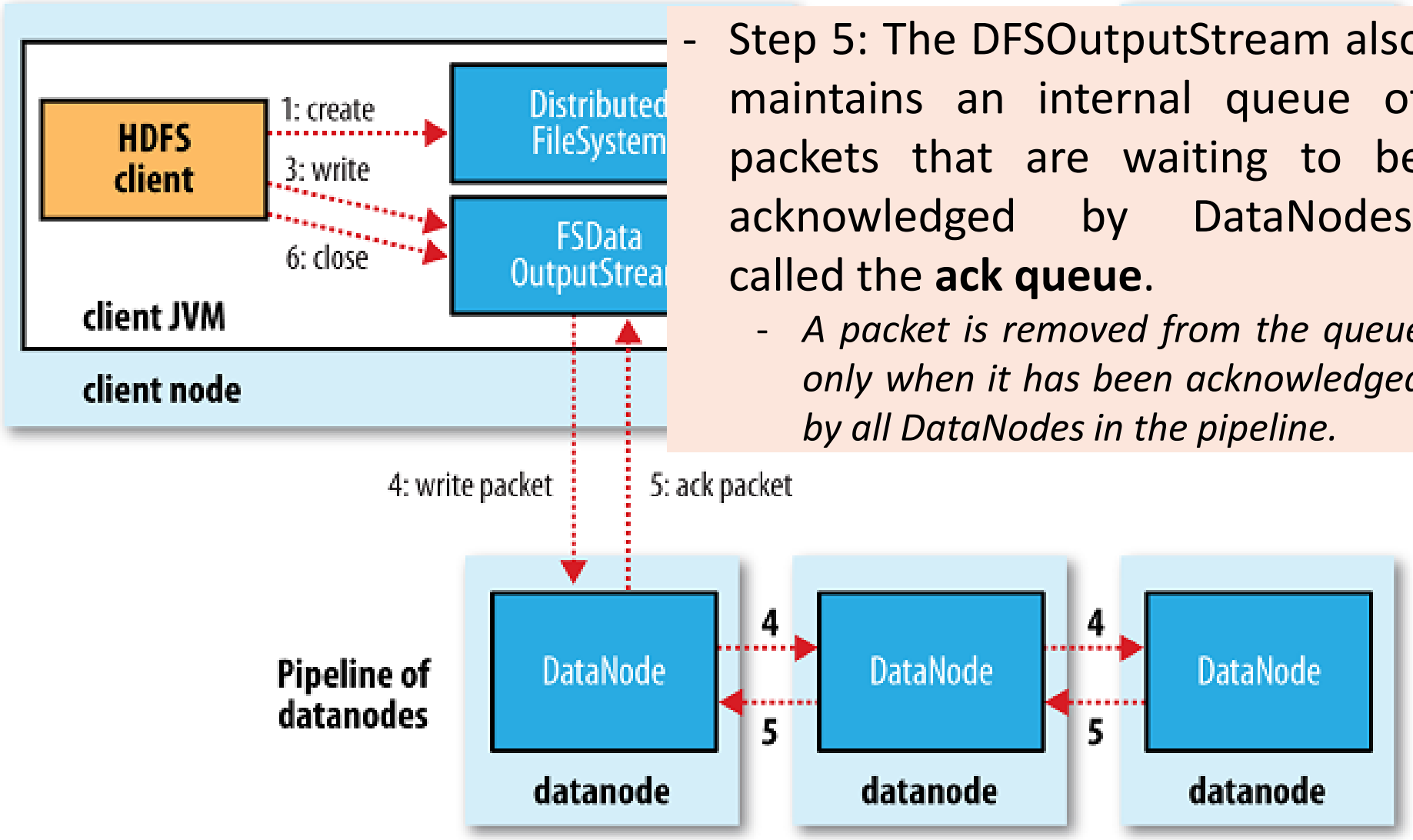
- Step 1: The client calls create() on DistributedFileSystem.
- Step 2: DistributedFileSystem calls the NameNode to create a new file in the filesystem namespace, with no blocks associated with it.
  - *The NameNode performs various checks to make sure the file does not already exist and the client has the right permissions to create the file.*
  - *If checks passed, the NameNode makes a record of the new file; otherwise, file creation fails and the client is thrown an IOException.*
  - *The DistributedFileSystem returns an FSDataOutputStream for the client to start writing data to.*

An anatomy of a file write



- Step 3: As the client writes data, the DFSOutputStream splits it into packets, which it writes to an internal queue called the **data queue**.
  - *The queue is consumed by the DataStreamer, which is responsible for asking NameNode to allocate new blocks by picking a list of suitable DataNodes to store the replicas (usually 3 nodes).*
  - *The list of DataNodes forms a pipeline*.
- Step 4: The DataStreamer sequentially streams the packets through nodes.
  - *The first DataNode in the pipeline stores the packet and forwards it to the second DataNode. Similarly, the second DataNode stores the packet and forwards it to the third (and last) DataNode.*

An anatomy of a file write



- Step 5: The DFSOutputStream also maintains an internal queue of packets that are waiting to be acknowledged by DataNodes, called the **ack queue**.
  - *A packet is removed from the queue only when it has been acknowledged by all DataNodes in the pipeline.*

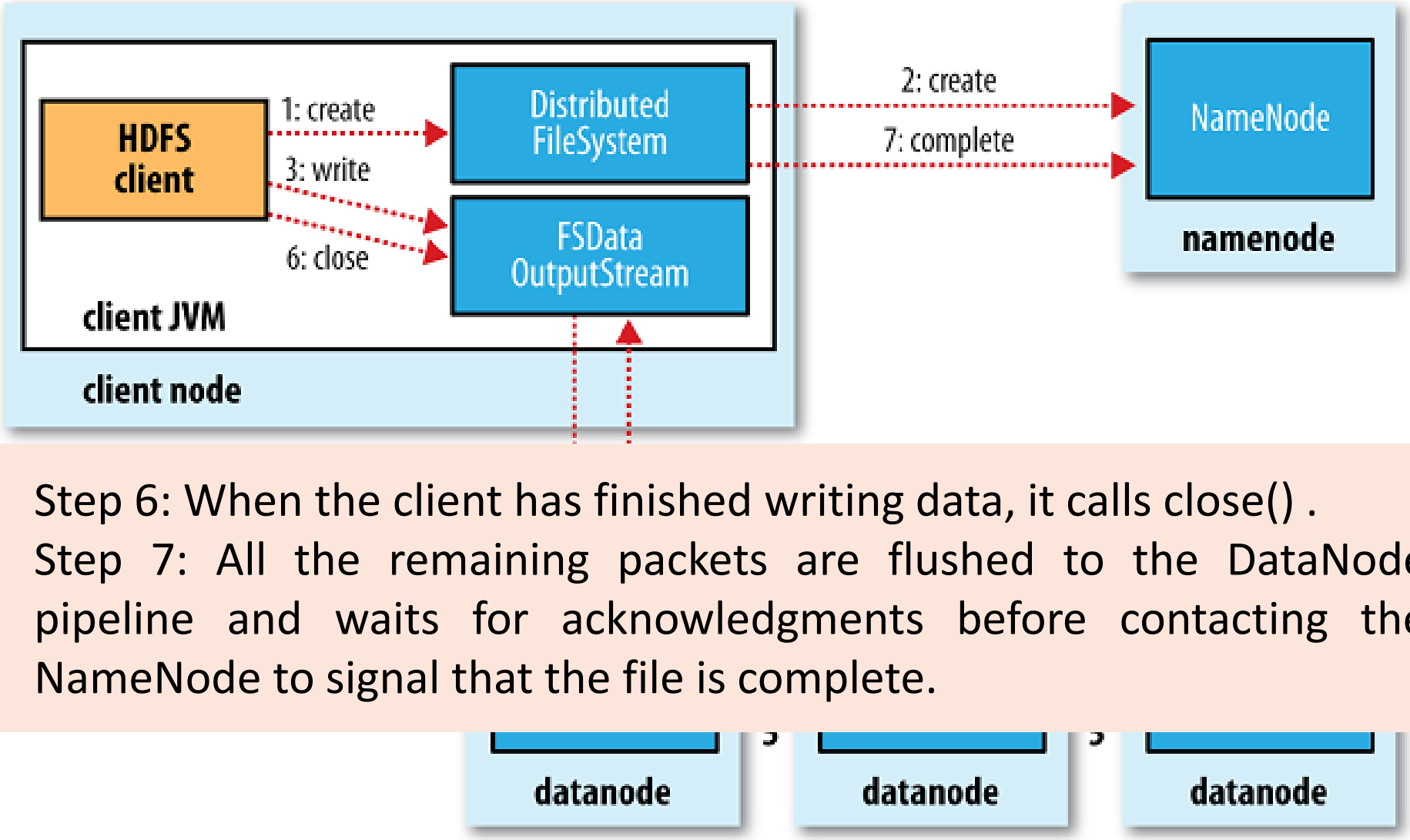
An anatomy of a file write: Failure

1. The pipeline is closed.
2. Any packets in the ack queue are added to the front of the data queue.
   - DataNodes that are downstream from the failed node will not miss any packets.
3. The current block on the good DataNodes is given a new identity, which is communicated to the NameNode.
   - The partial block on the failed DataNode will be deleted if the node recovers later.
4. A new pipeline is constructed from the two good DataNodes.
   - The failed DataNode is removed from the pipeline.
5. The remainder of the block's data is written to the new pipeline.
6. The NameNode arranges for a further replica on another node.

Subsequent blocks are then treated as normal.

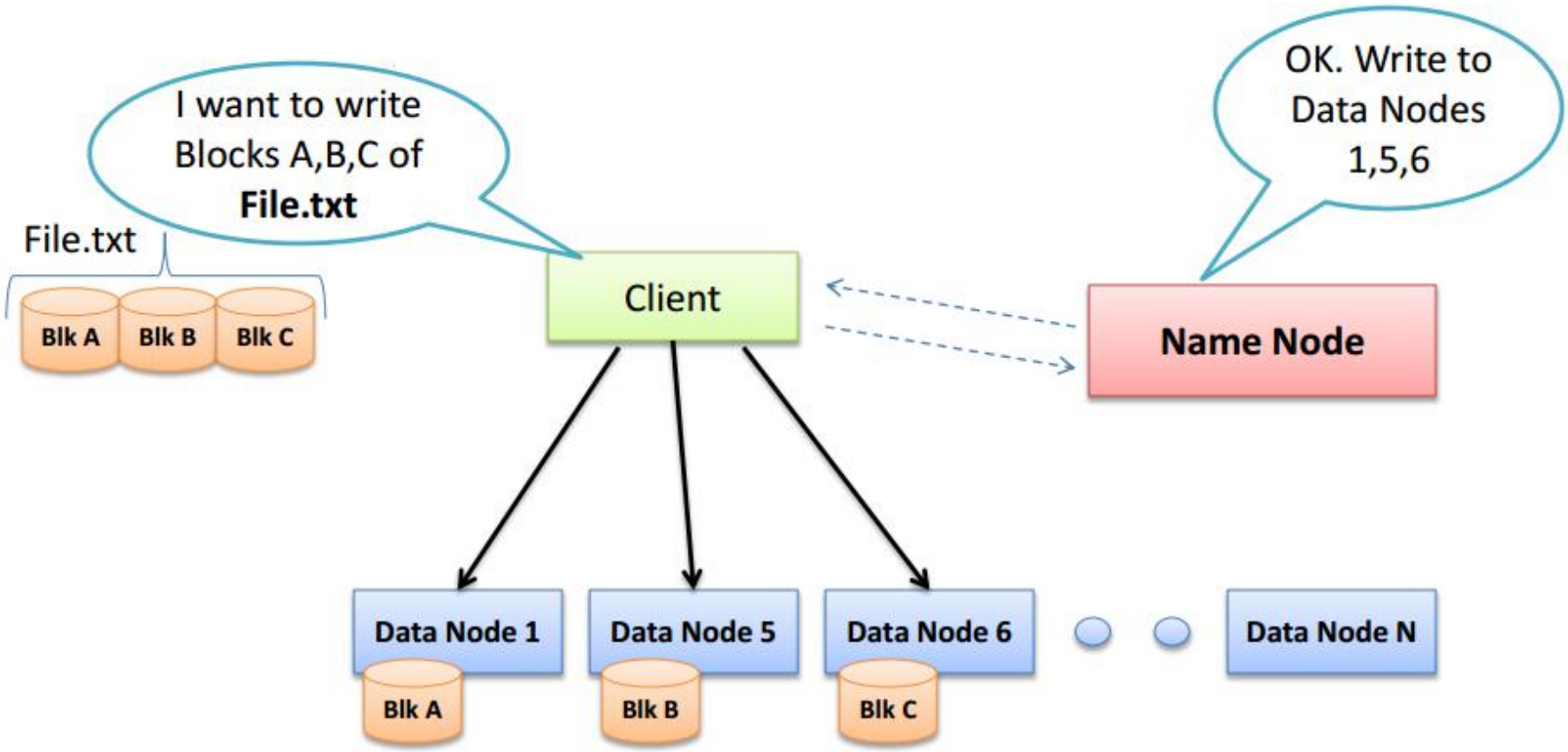
Those actions are transparent to the client writing the data.

An anatomy of a file write



- Step 6: When the client has finished writing data, it calls close() .
- Step 7: All the remaining packets are flushed to the DataNode pipeline and waits for acknowledgments before contacting the NameNode to signal that the file is complete.

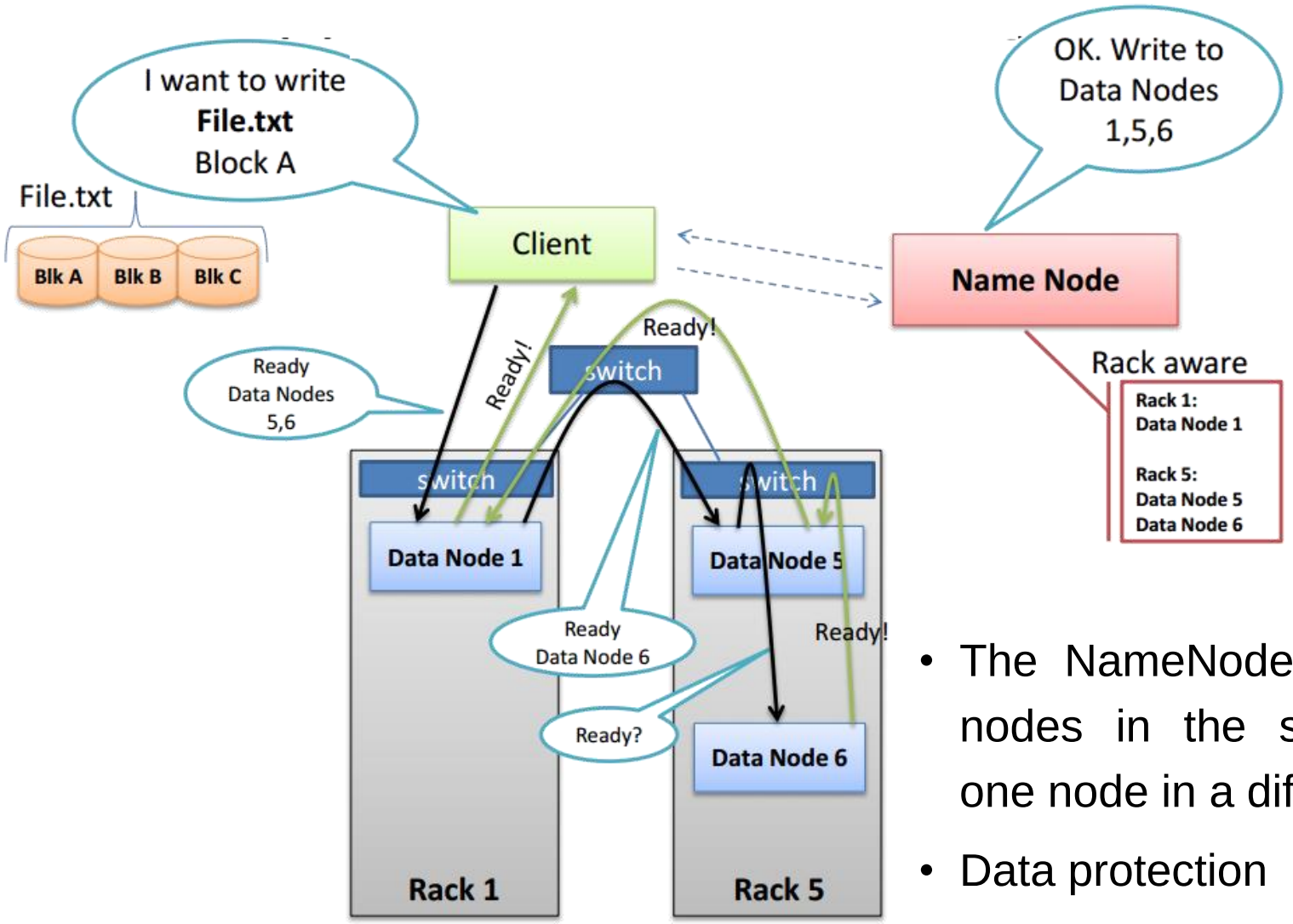
Write data to HDFS: An example



- The client consults the NameNode to write a block directly to a DataNode.
- Each block will be replicated in other DataNodes.
- Cycle repeats for next block

Image credit: Brad Hedlund

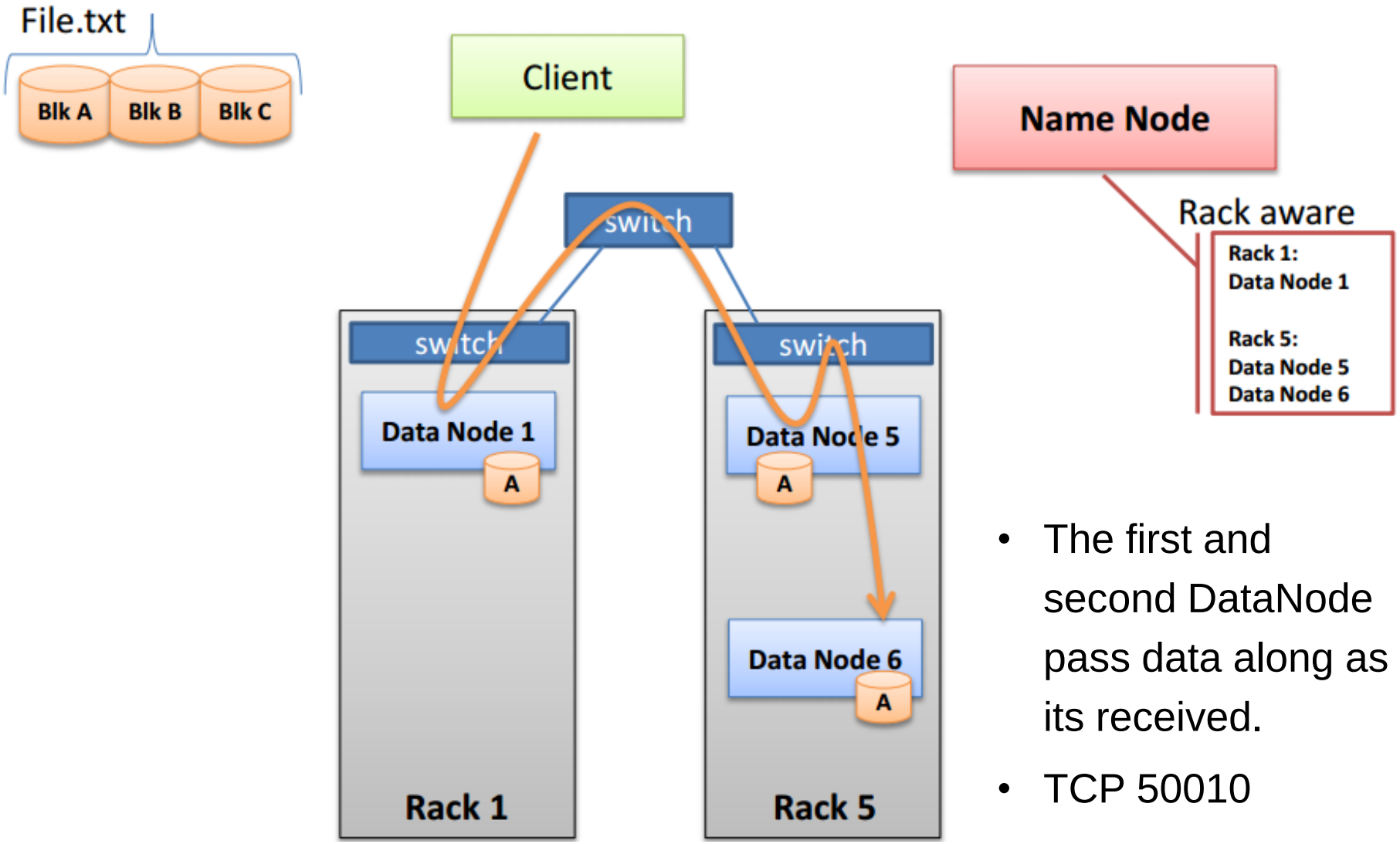
Write data to HDFS: An example



- The NameNode picks two nodes in the same rack, one node in a different rack.
- Data protection
- Locality for MR jobs

Image credit: Brad Hedlund

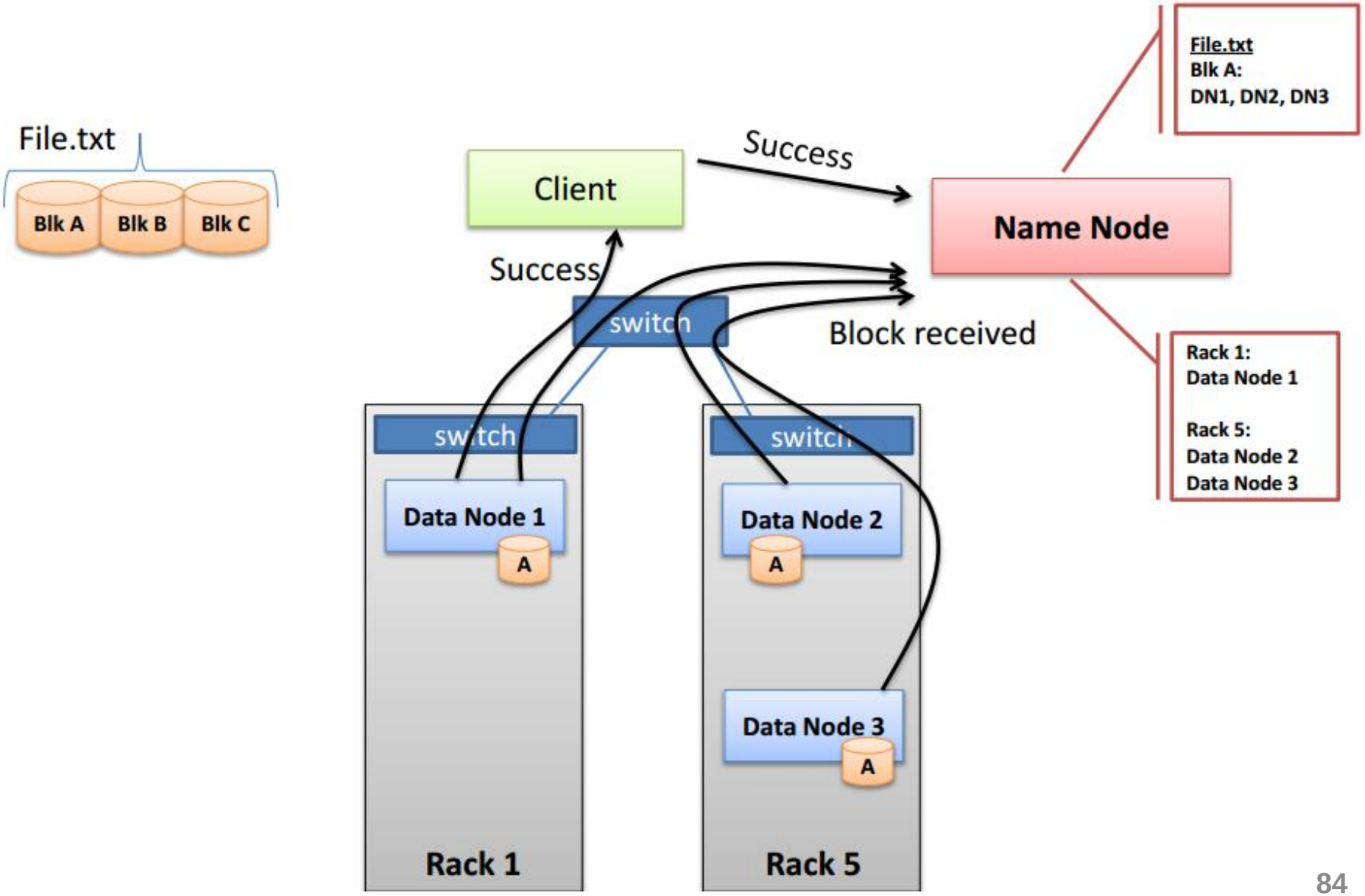
Write data to HDFS: An example



- The first and second DataNode pass data along as its received.
- TCP 50010

Image credit: Brad Hedlund

Write data to HDFS: An example



Write data to HDFS: An example

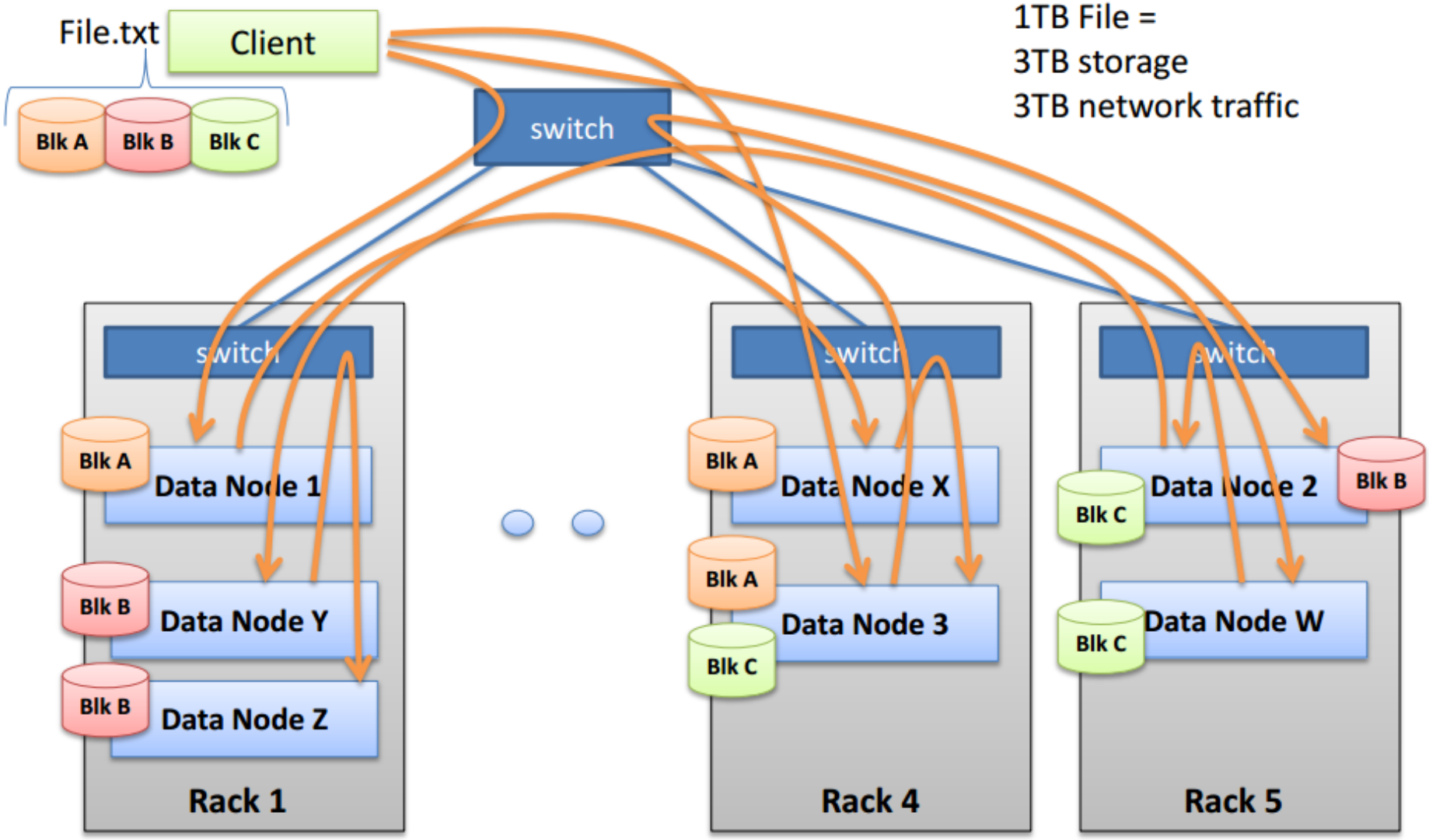


Image credit: Brad Hedlund

Unbalanced cluster

- Hadoop prefers local processing if possible
- New servers underutilized for MapReduce, HDFS*
- Might see more network bandwidth, slower job times**

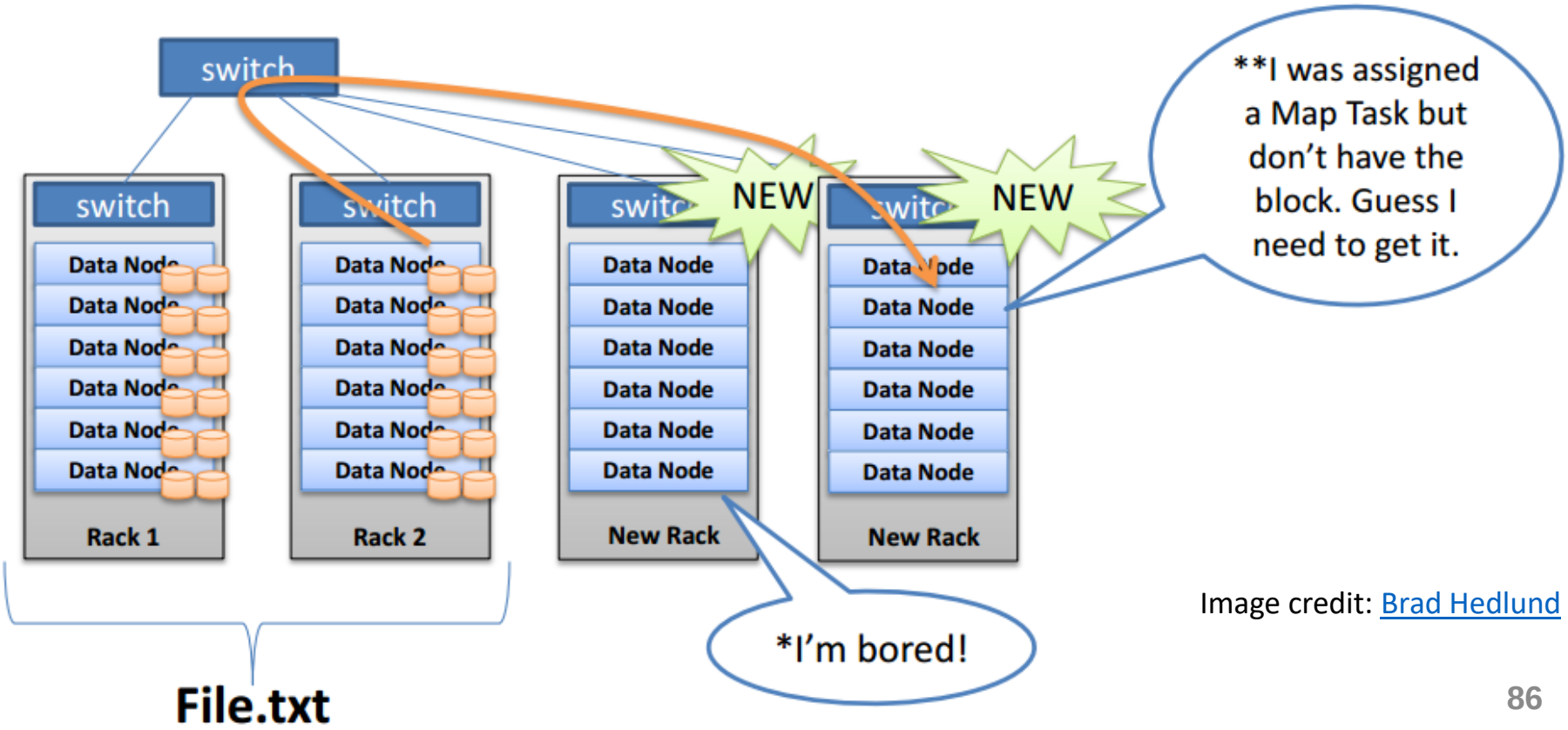


Image credit: Brad Hedlund

Cluster balancing

- **Balancer** utility (if used) runs in the background and does not interfere with MapReduce or HDFS.
- Default speed limit 1 MB/s.

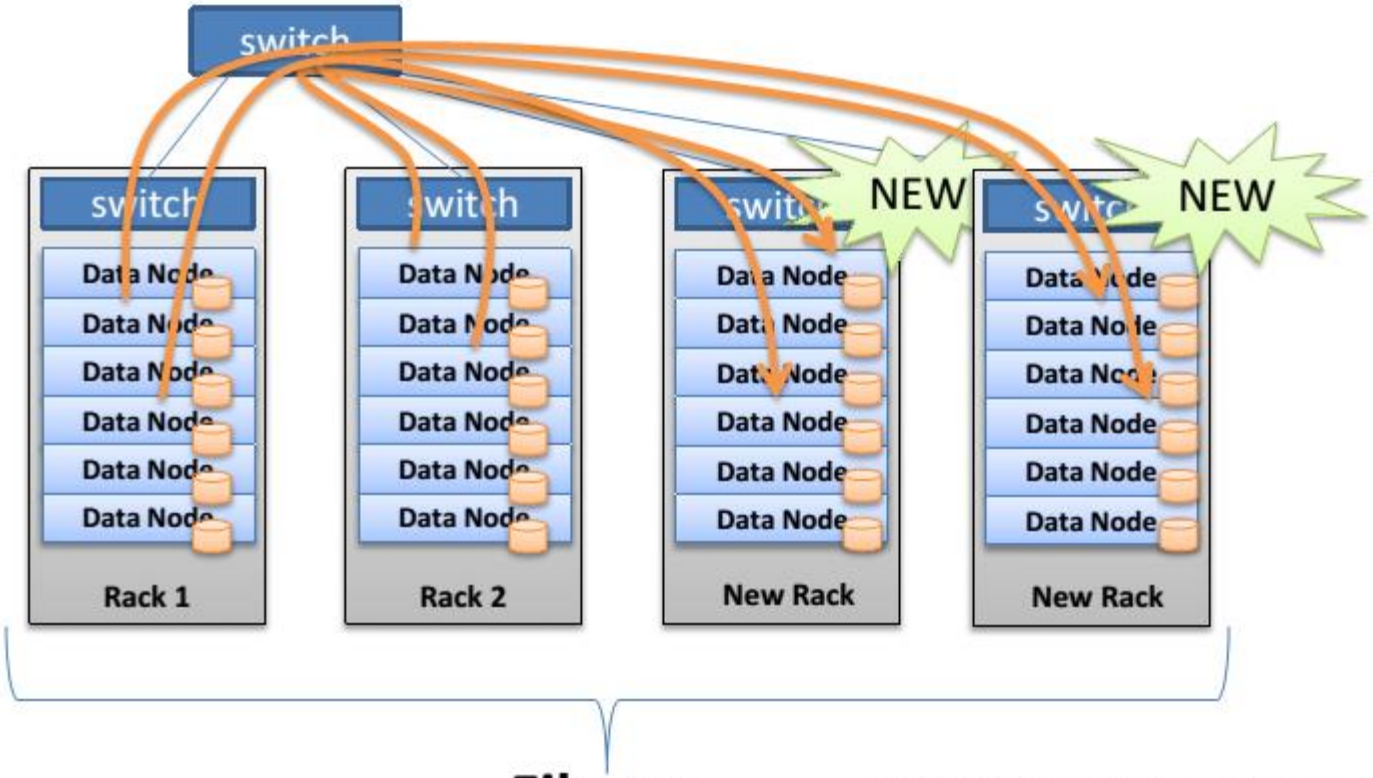


Image credit: Brad Hedlund

```
brad@cloudera-1:~$hadoop balancer
```


...the end.