Lecture slides of the course Introduction to Big Data

HADOOP FUNDAMENTALS

(Part II)

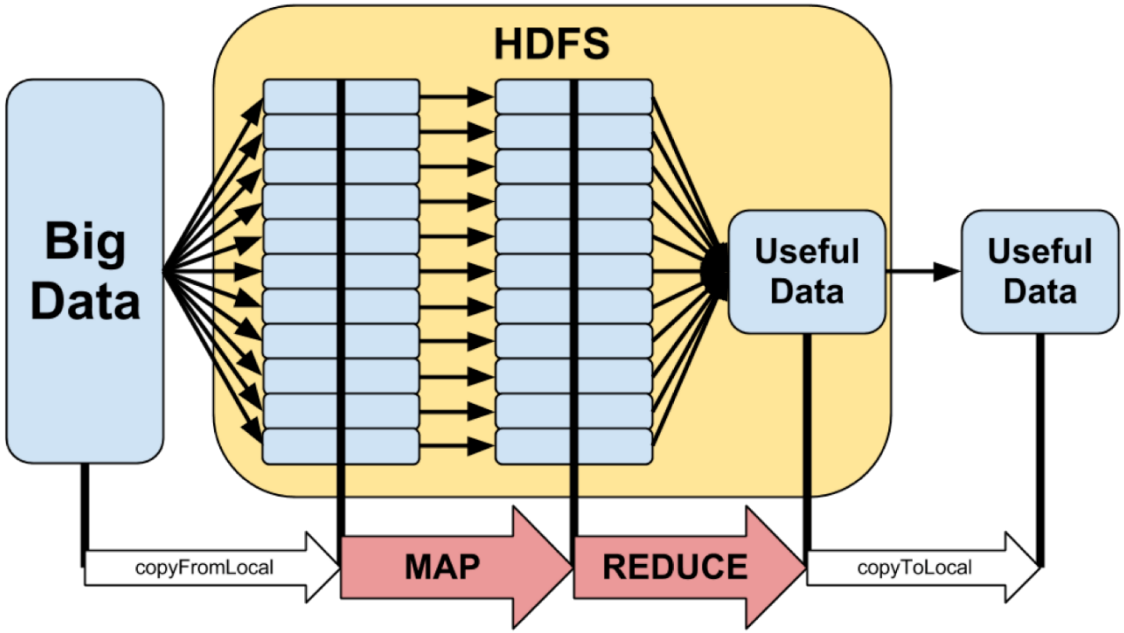
Lecturer: Dr. Nguyen Ngoc Thao – MSc. Le Ngoc Thanh

Department of Computer Science, FIT, HCMUS

Ho Chi Minh City, September 2018

Outline

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

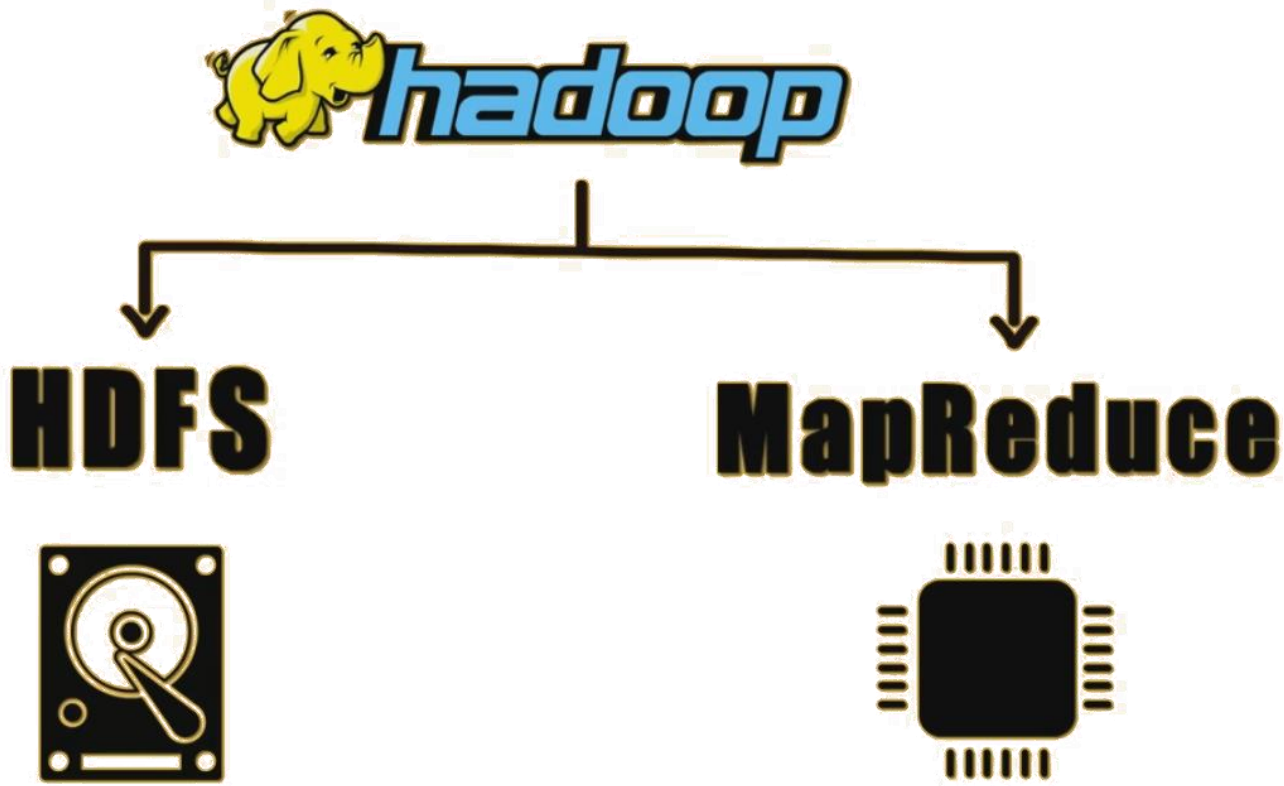


A BRIEF REVIEW OF HADOOP

An essential ecosystem for Big Data

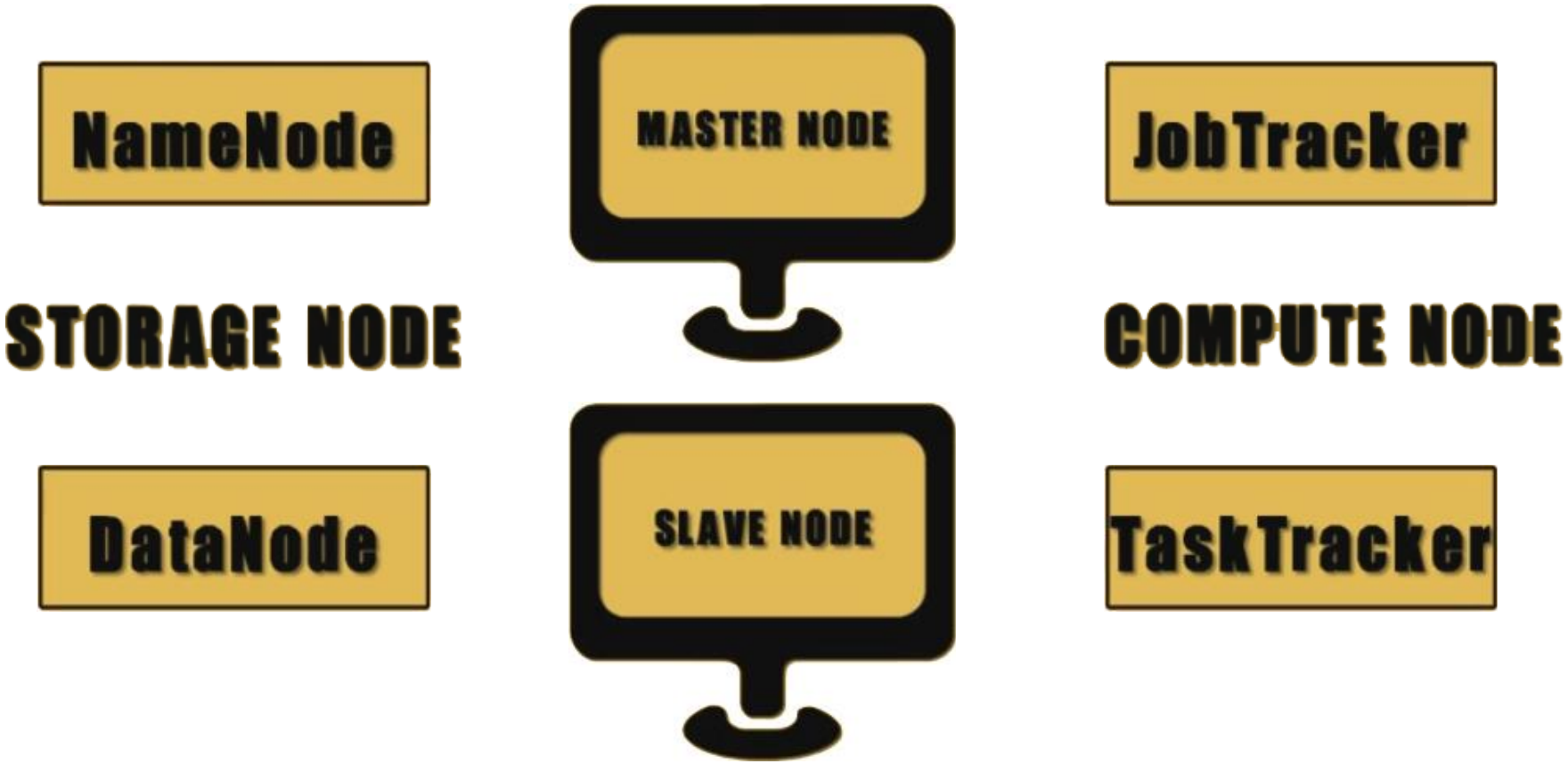
Hadoop architecture

- The core of Hadoop includes HDFS and MapReduce.

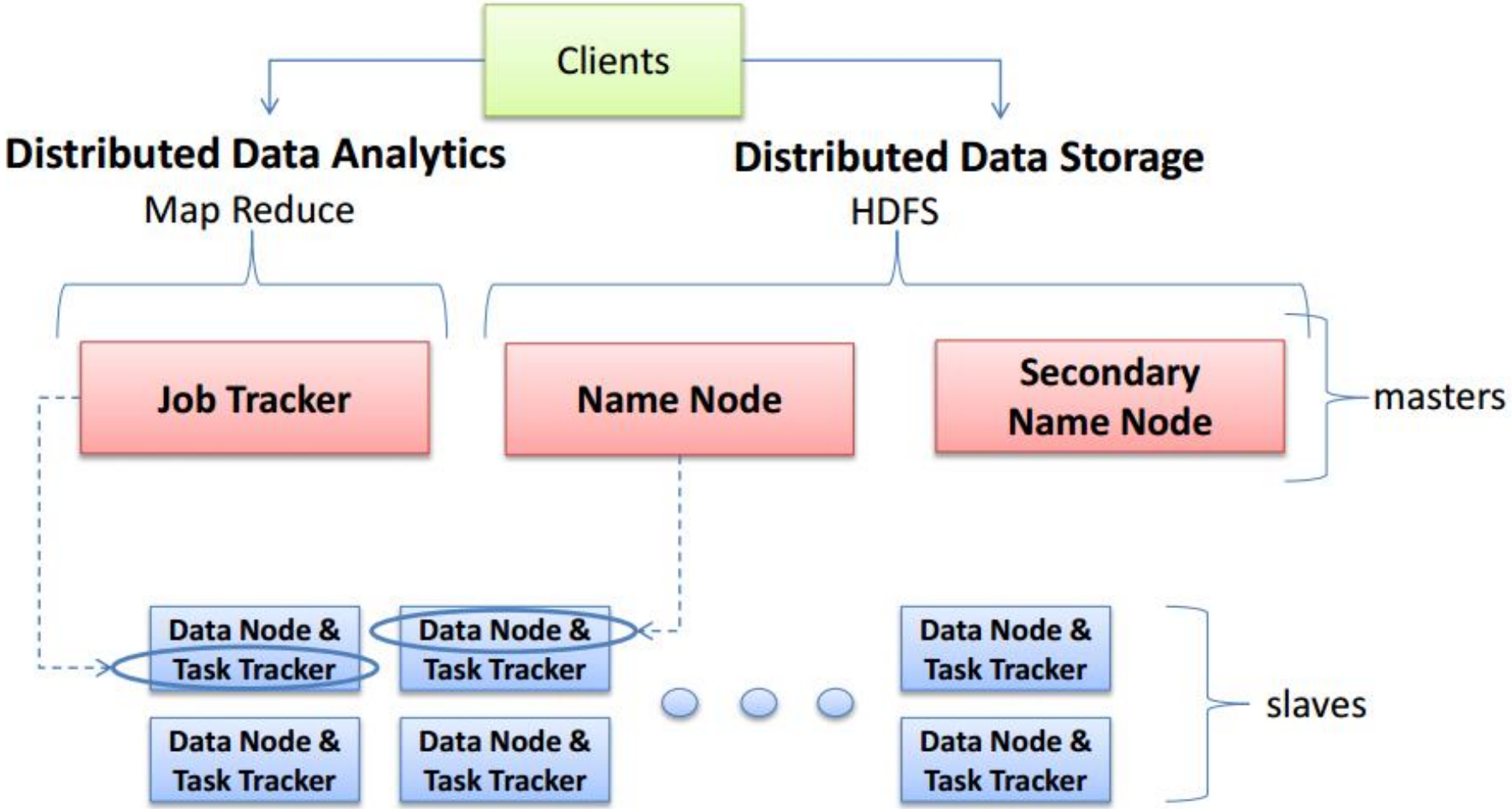


Hadoop architecture

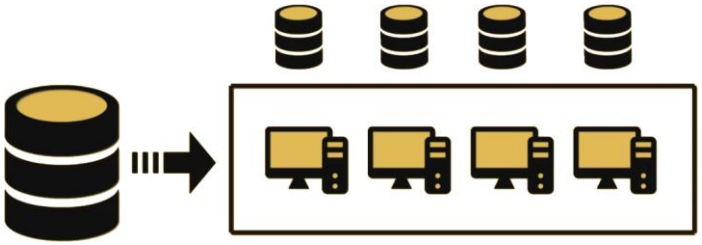
- Working nodes in a Hadoop cluster can be categorized into



Hadoop architecture

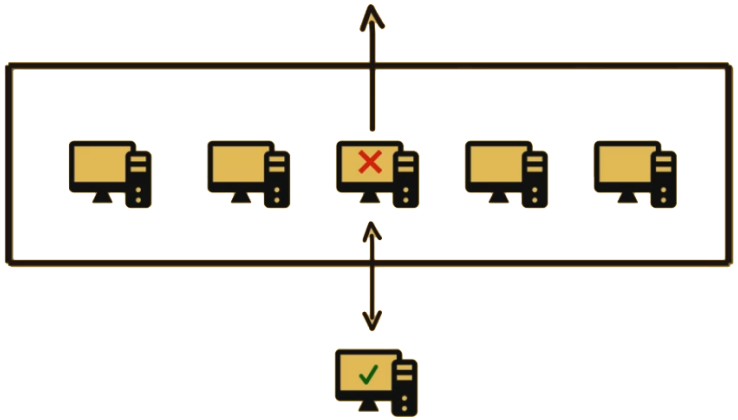


Pros of using Hadoop

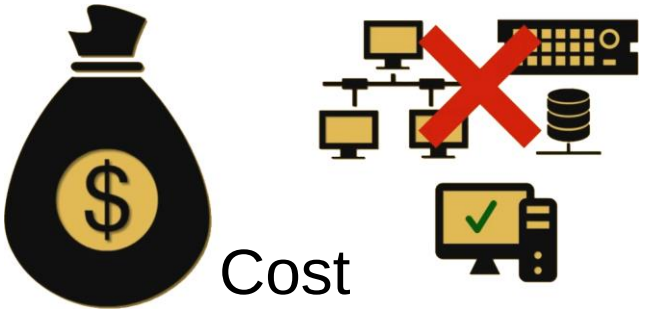


Distributed storage

Parallel processing



Automatic failover management

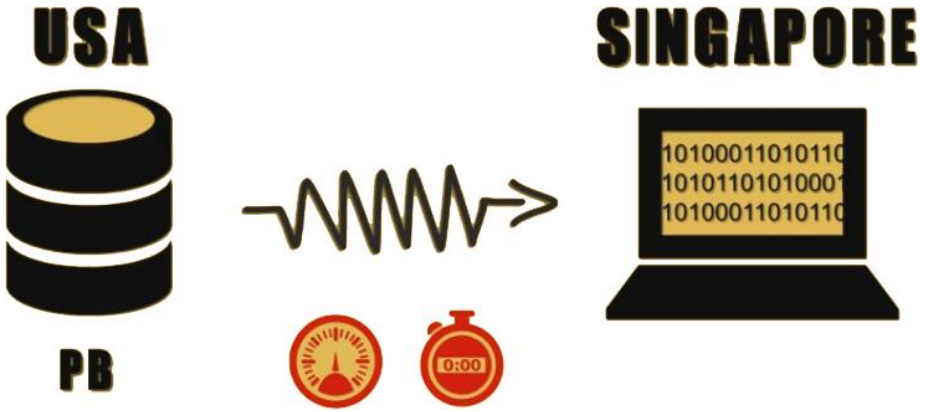


Cost effective

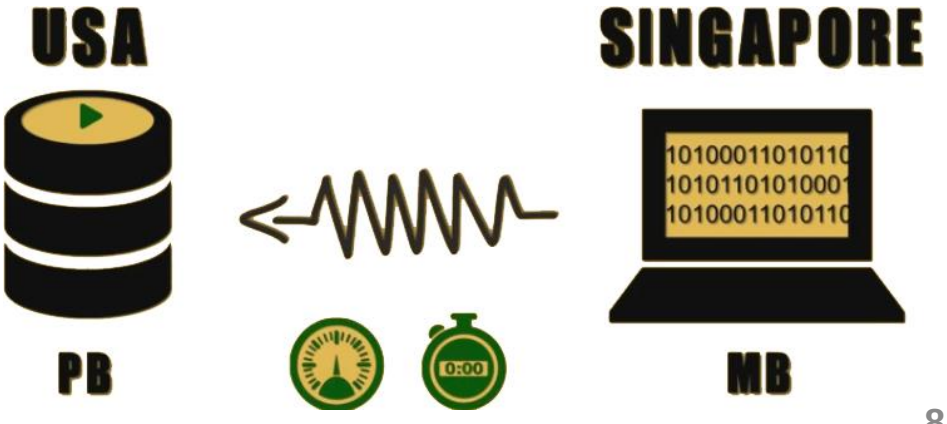
Pros of using Hadoop

Data locality optimization

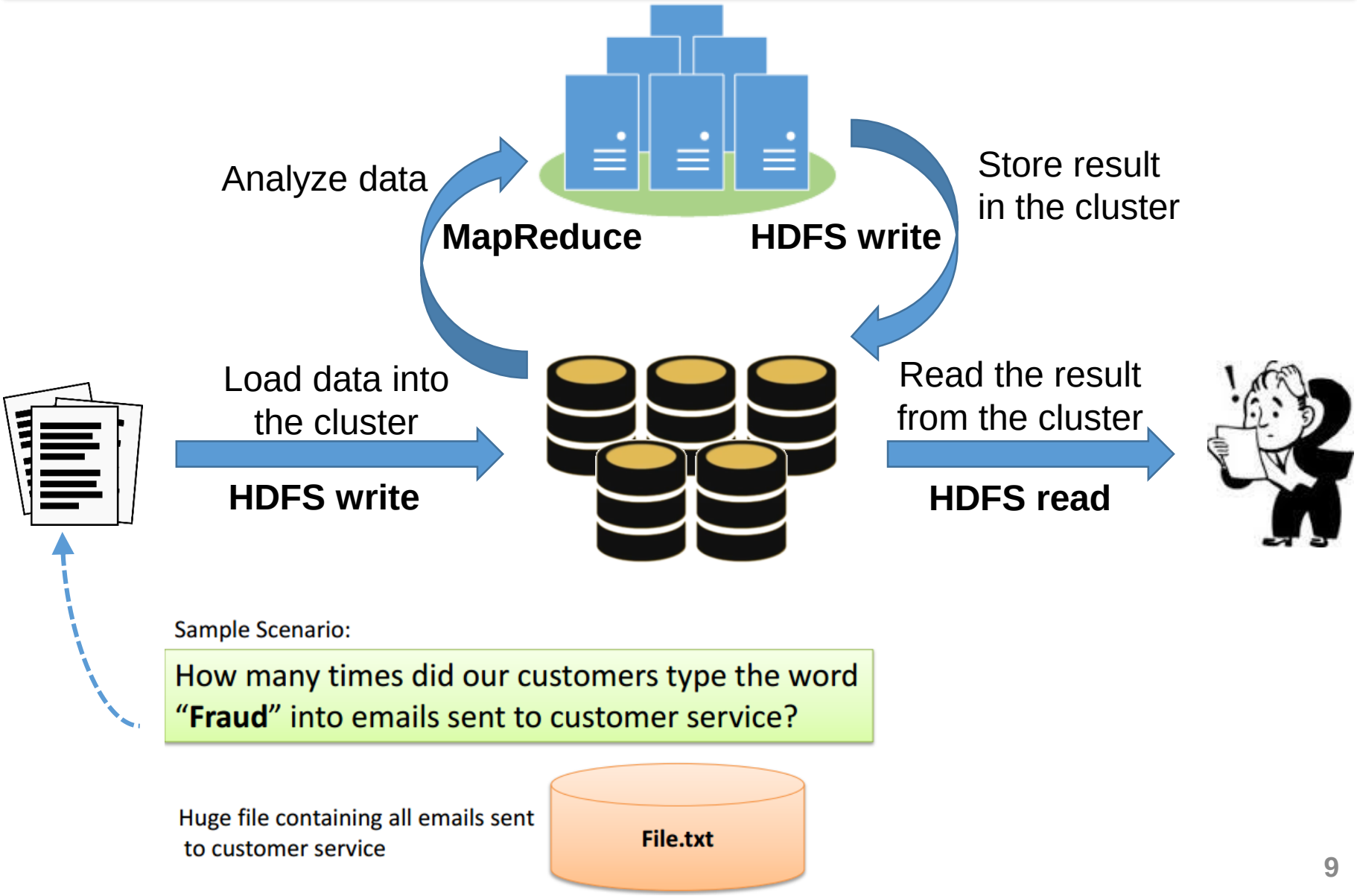
Traditional approach: copy the data from USA to Singapore → not good in term of performance and time.



Better approach: copy the source code to USA from Singapore.



A typical workflow in HDFS

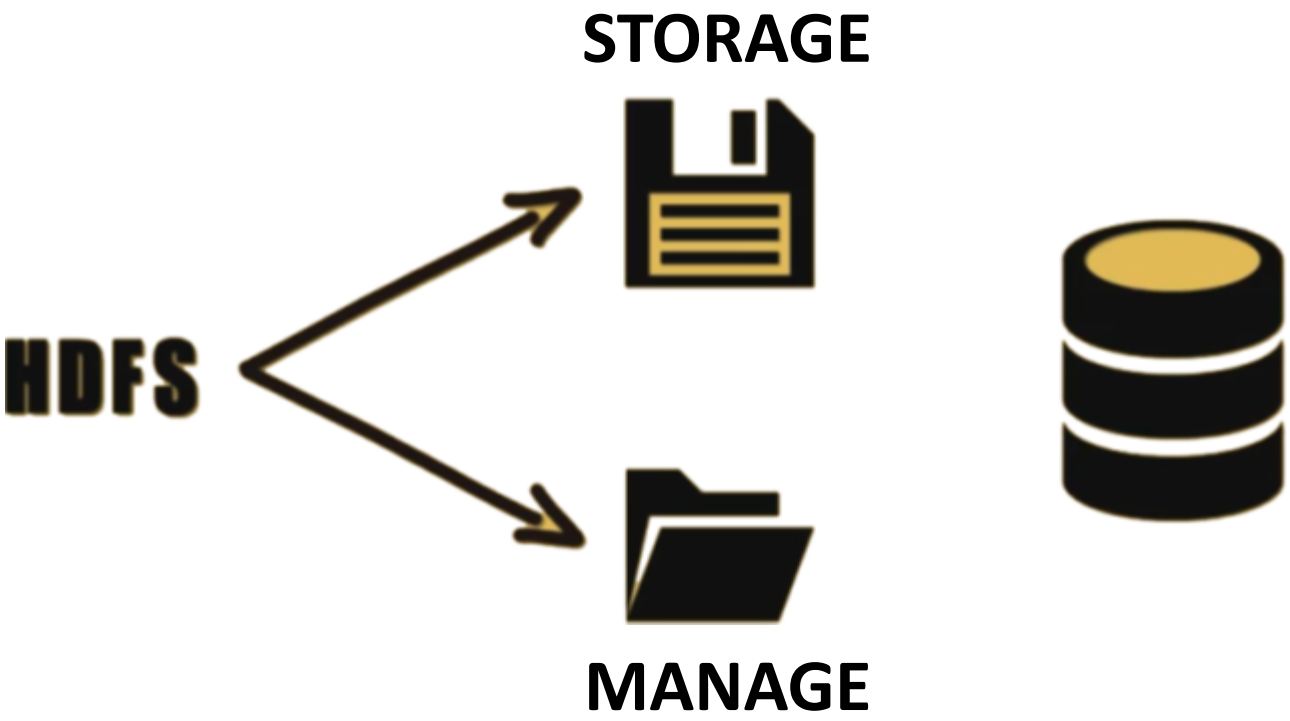


HDFS

A distributed filesystem designed to run on commodity hardware

What is HDFS?

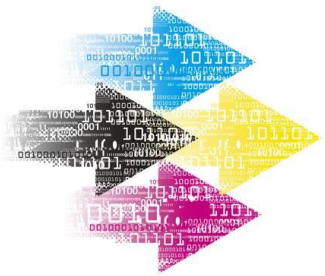
- **Hadoop Distributed Filesystem** (HDFS) is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.



HDFS are designed for

Very large files

Files that are hundreds of megabytes, gigabytes, or terabytes in size. There are Hadoop clusters running today that store petabytes of data.



Streaming data access

Most efficient data processing pattern: write-once, 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. HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure.

But, HDFS are NOT designed for

Low-latency data access

Not for applications that require access in the tens of milliseconds HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency.

Lots of small files

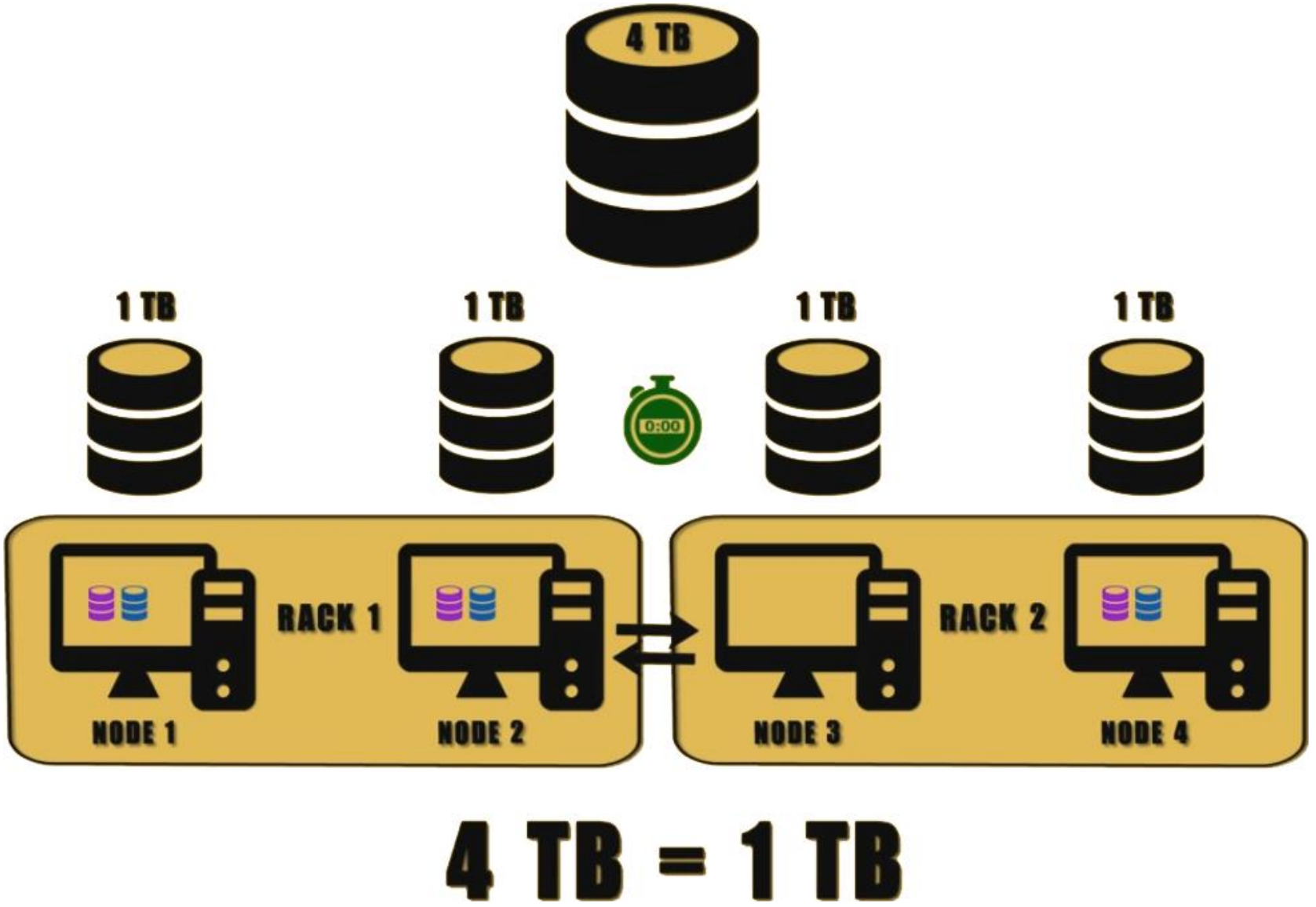
The limit to the number of files in a filesystem is governed by the amount of memory on the namenode.

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.

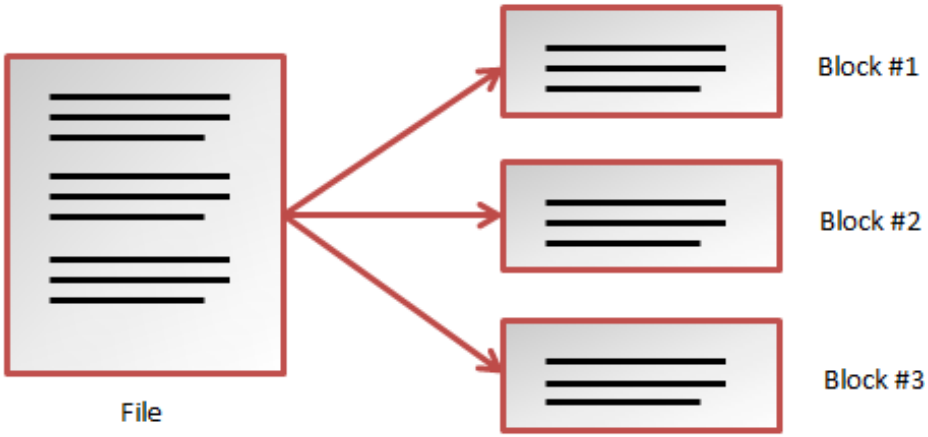
Hadoop Distributed File System HDFS	Google File System GFS
Cross Platform	Linux
Developed in Java environment	Developed in C, C++ environment
At first developed by Yahoo and now it is an open source framework	Developed by Google
NameNode and DataNode	Master Node and Chunk server
The default block size 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 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	Oprational log
Only append is possible	Random file write possible

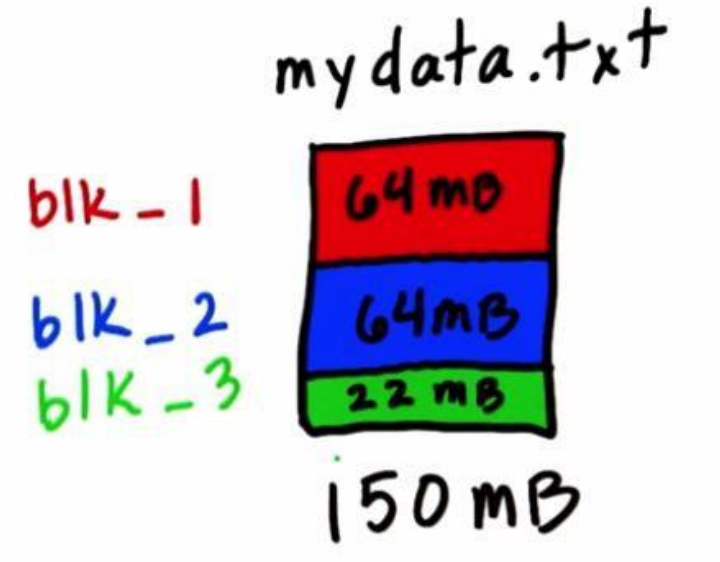
Storing data with HDFS



Blocks in HDFS

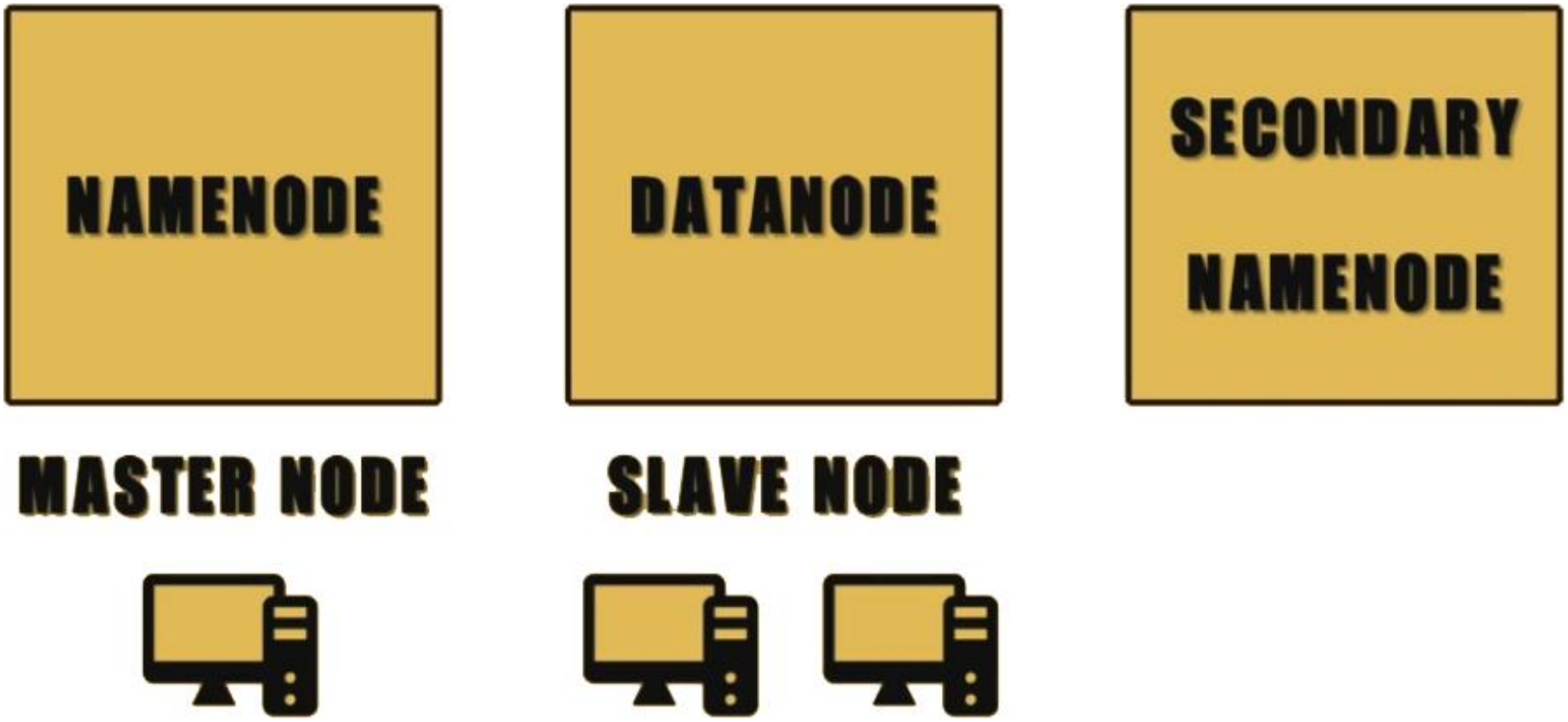
- **Like** in a filesystem for a single disk, files in HDFS are broken into block-sized chunks and stored as independent units.





- **Unlike** a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage.

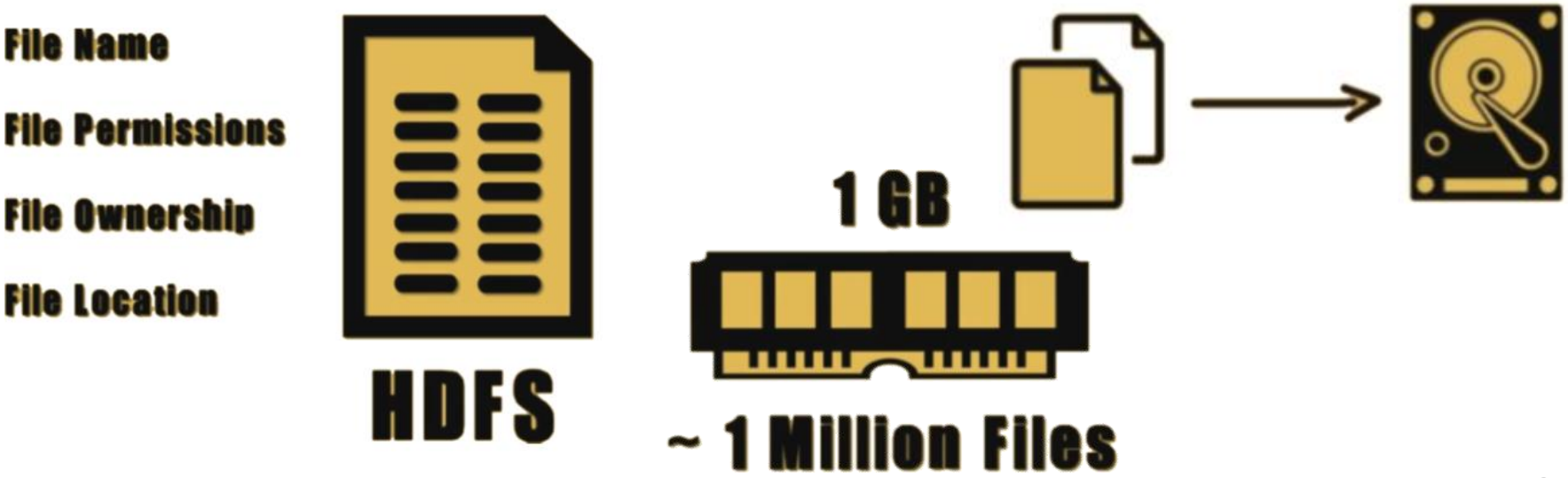
HDFS Daemons



NameNode

NameNode

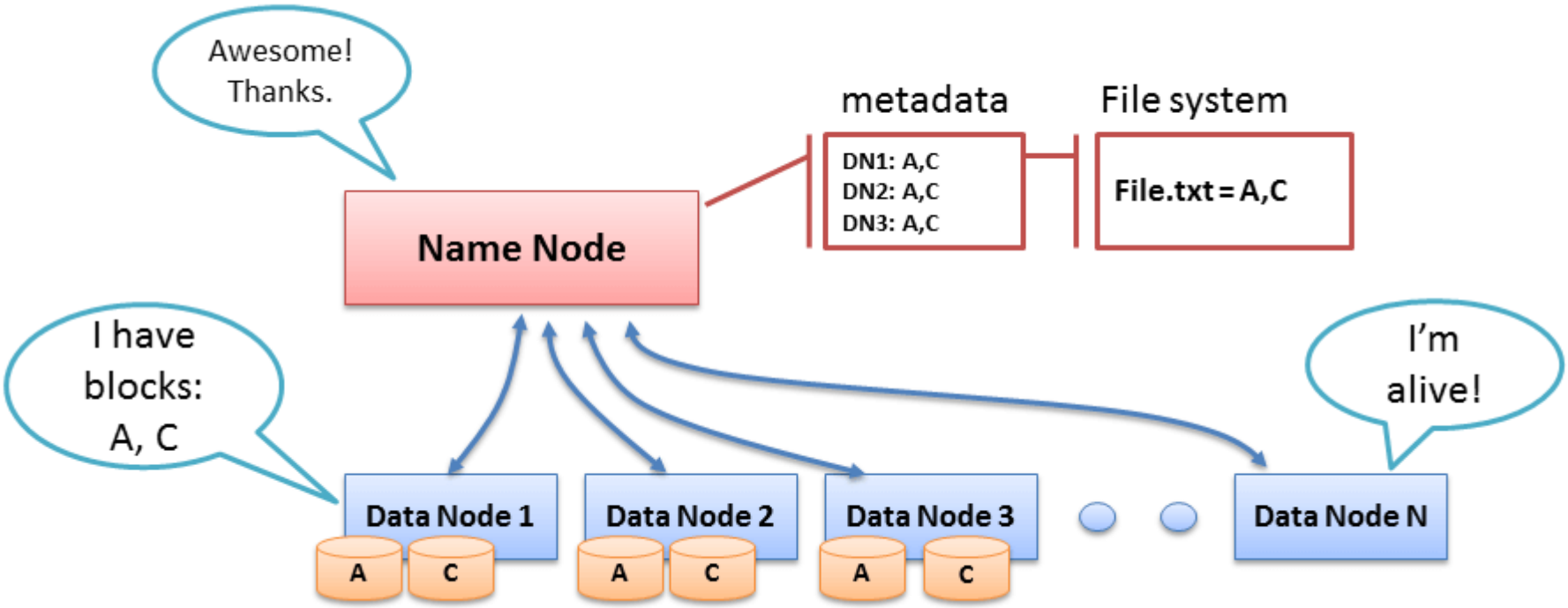
- Manage the filesystem namespace by maintaining the **filesystem tree** and the **metadata** for all the files and directories in the tree
- This information is stored persistently on the local disk in the **namespace image** (**fsimage**) and **edit log** (**editLog**) files.



NameNode

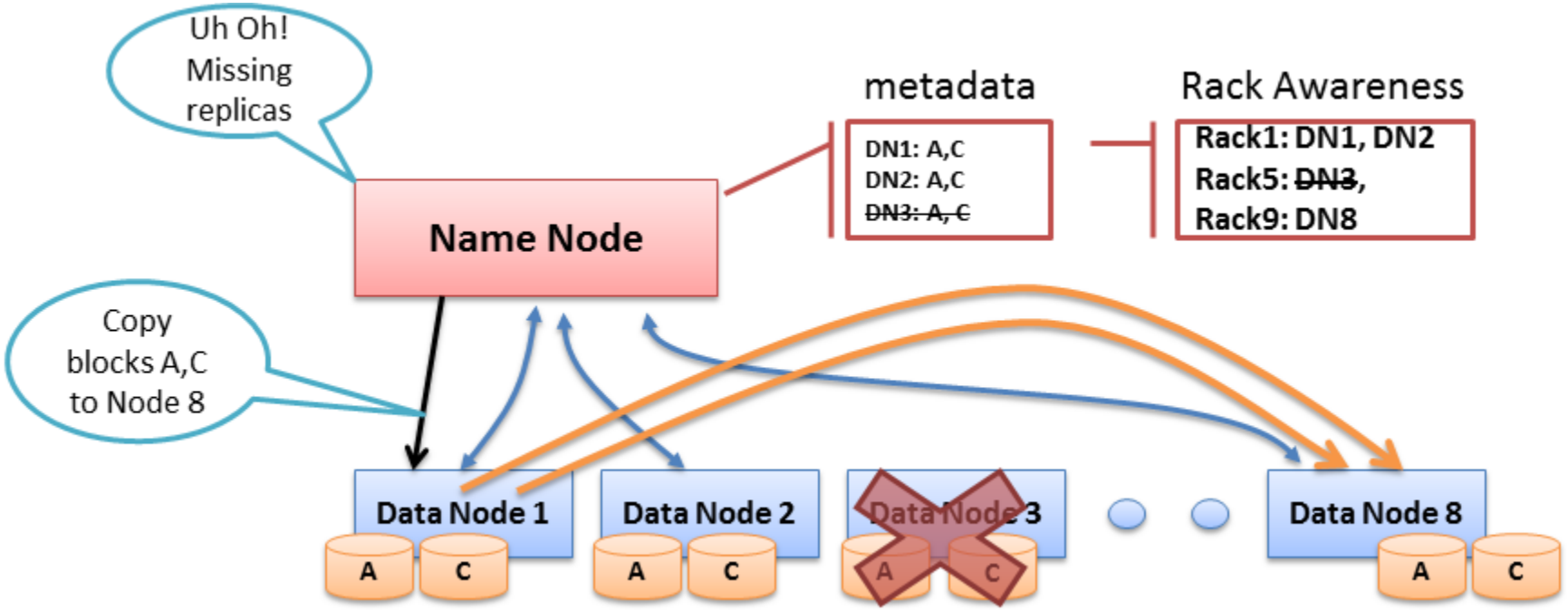
- The NameNode knows the DataNodes on which all the blocks for a given file are located.
- However, it does **not store block locations persistently**.
- This information is reconstructed from DataNodes when the system starts.

NameNode in use



- DataNodes send **heartbeats** every 3 seconds via a TCP handshake.
- Every 10th heartbeat is a Block report, allowing the NameNode builds its metadata and insure (3) copies of the block exist in the system.
- **If NameNode is down, HDFS is down.**

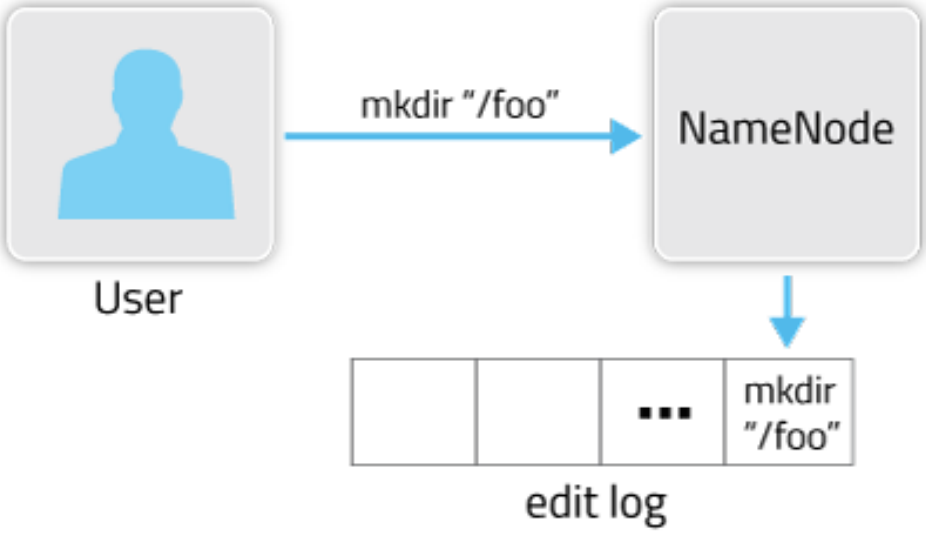
Re-replicating missing replicas



- Missing heartbeats signify lost nodes
- NameNode consults metadata, finds affected data.
- NameNode consults Rack Awareness script.
- NameNode tells a DataNode to re-replicate.

Functions of a NameNode

- **Maintain and execute the filesystem namespace**
  - Any modifications in the filesystem namespace or in its properties is tracked by the NameNode with the help of a transactional log, i.e. **EditLog**.
  - E.g., when a directory is created in HDFS, the NameNode will immediately record this in the EditLog.



Functions of a NameNode

- **Govern the mechanism of storing data**
  - Map a file to a set of blocks and maps a block to the DataNodes where it is located
  - Keep a record of how the files in HDFS are divided into blocks, in which nodes these blocks are stored.
  - Take care of the **replication factor** of blocks
    - A change in the replication factor of any block will be recorded in the editLog by the NameNode.

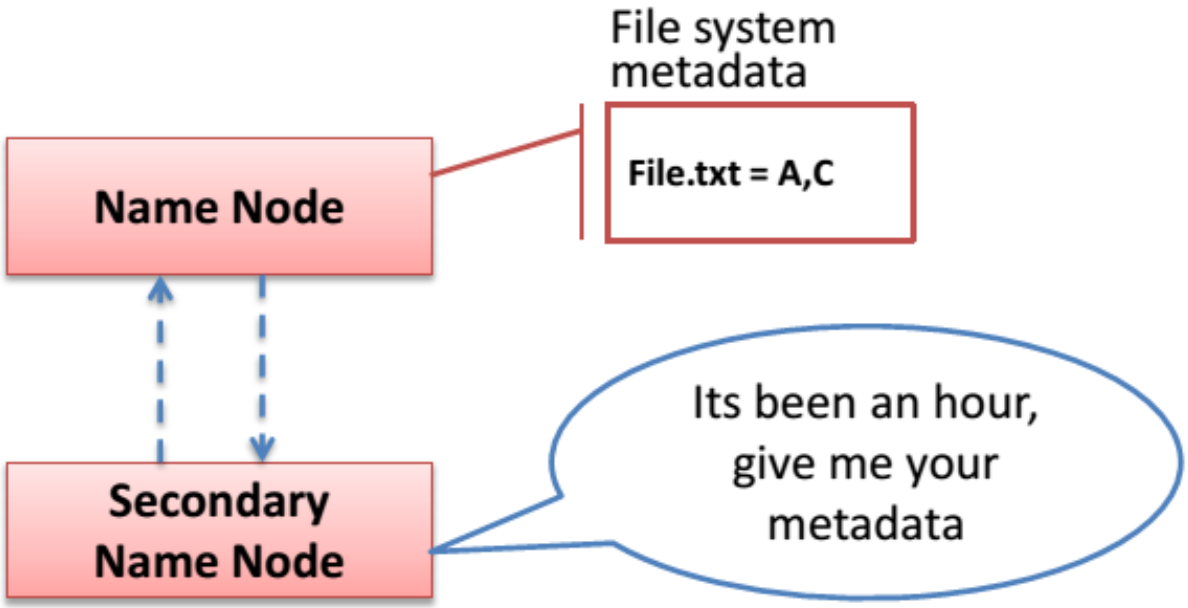
Functions of a NameNode

- **Record the metadata of all the files stored in the cluster**
  - E.g., the location, the size of the files, permissions, hierarchy, etc.
- **Make sure that the DataNodes are working properly**
  - Regularly receive heartbeats and Block reports from all DataNodes.
  - In case of a DataNode failure, it chooses new DataNodes for new replicas, balances disk usage and also manages the communication traffic to the DataNodes.
- Direct DataNodes to execute the low-level I/O operations
- By and large the NameNode manages cluster configuration.

Secondary NameNode

- **The filesystem cannot be used without the NameNode.**
  - All the files on the filesystem would be lost since there would be no way of reconstructing them from the blocks 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
- Connect to NameNode every hour (by default).
- Housekeeping, backup of NameNode metadata
- Saved metadata can rebuild a failed NameNode

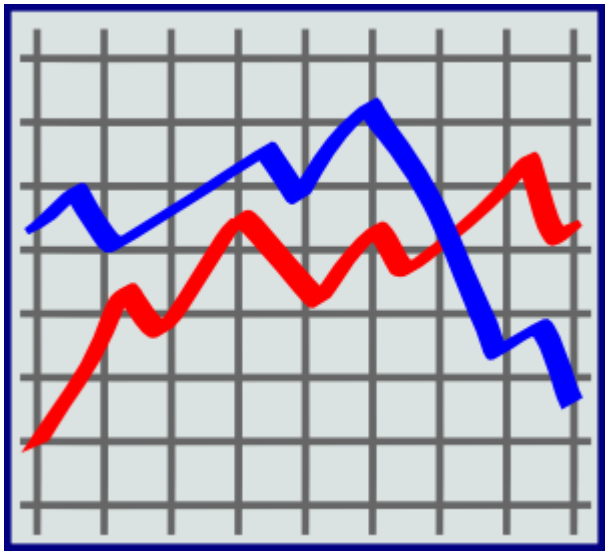
HDFS fsimage file

- A persistent checkpoint of **HDFS metadata**
- Binary file, containing information about files and directories

	FILE	DIRECTORY
Full path	YES	YES
Replication factor	YES	
Modification time	YES	YES
Access time	YES	
Permissions and Ownership	YES	YES
Block size	YES	
Number of blocks	YES	
File size	YES	
Namespace and Diskspace Quota		YES

Advanced HDFS capacity planning

- Analyze the fsimage to learn how fast HDFS grows
- Combine it with “external” datasets
  - number of daily/monthly active users.
  - total size of logs generated by users.
  - number of queries / day run by data analysts.



HDFS edit/audit file

- Log all filesystem access requests sent to the NameNode
- Easy to parse and aggregate

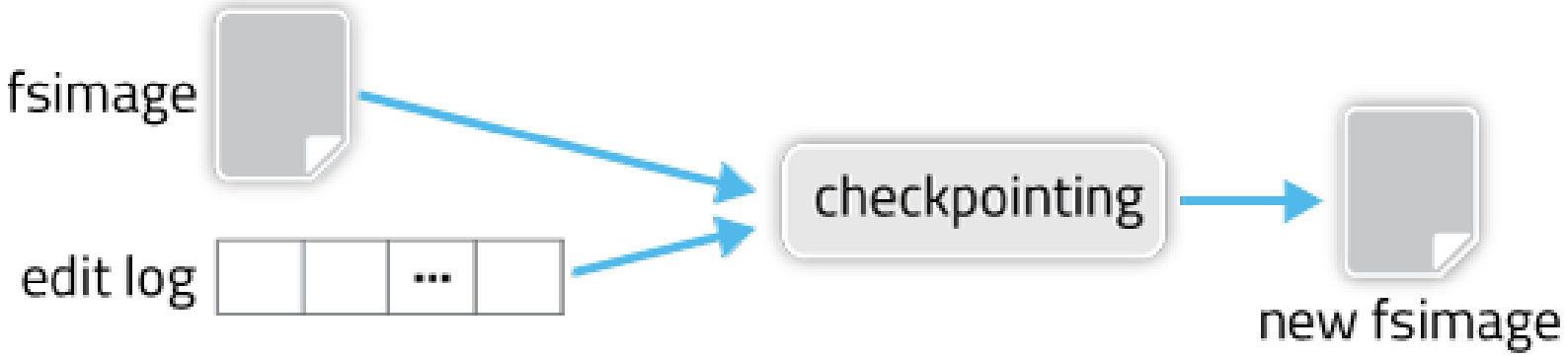
```
    <OPCODE>OP_MKDIR</OPCODE>
  - <DATA>
      <TXID>12</TXID>
      <LENGTH>0</LENGTH>
      <INODEID>16394</INODEID>
      <PATH>/user/hive</PATH>
      <TIMESTAMP>1470979518949</TIMESTAMP>
    - <PERMISSION_STATUS>
        <USERNAME>acadgild</USERNAME>
        <GROUPNAME>supergroup</GROUPNAME>
        <MODE>493</MODE>
      </PERMISSION_STATUS>
    </DATA>
  </RECORD>
- <RECORD>
    <OPCODE>OP_MKDIR</OPCODE>
  - <DATA>
      <TXID>13</TXID>
      <LENGTH>0</LENGTH>
      <INODEID>16395</INODEID>
      <PATH>/user/hive/warehouse</PATH>
      <TIMESTAMP>1470979518949</TIMESTAMP>
    - <PERMISSION_STATUS>
        <USERNAME>acadgild</USERNAME>
        <GROUPNAME>supergroup</GROUPNAME>
        <MODE>493</MODE>
      </PERMISSION_STATUS>
    </DATA>
  </RECORD>
- <RECORD>
    <OPCODE>OP_SET_PERMISSIONS</OPCODE>
  - <DATA>
      <TXID>14</TXID>
      <SRC>/user/hive/warehouse</SRC>
      <MODE>509</MODE>
    </DATA>
  </RECORD>
</EDITS>
```

HDFS edit/audit file

- Some files/directories are accessed more often than others.
  - E.g., fresh logs, core datasets, dictionary files, etc.
  - Using an edit file to find them.
- **To process it faster**, increase its replication factor while it is “hot”.
- **To save disk space**, decrease its replication factor when it becomes “cold”

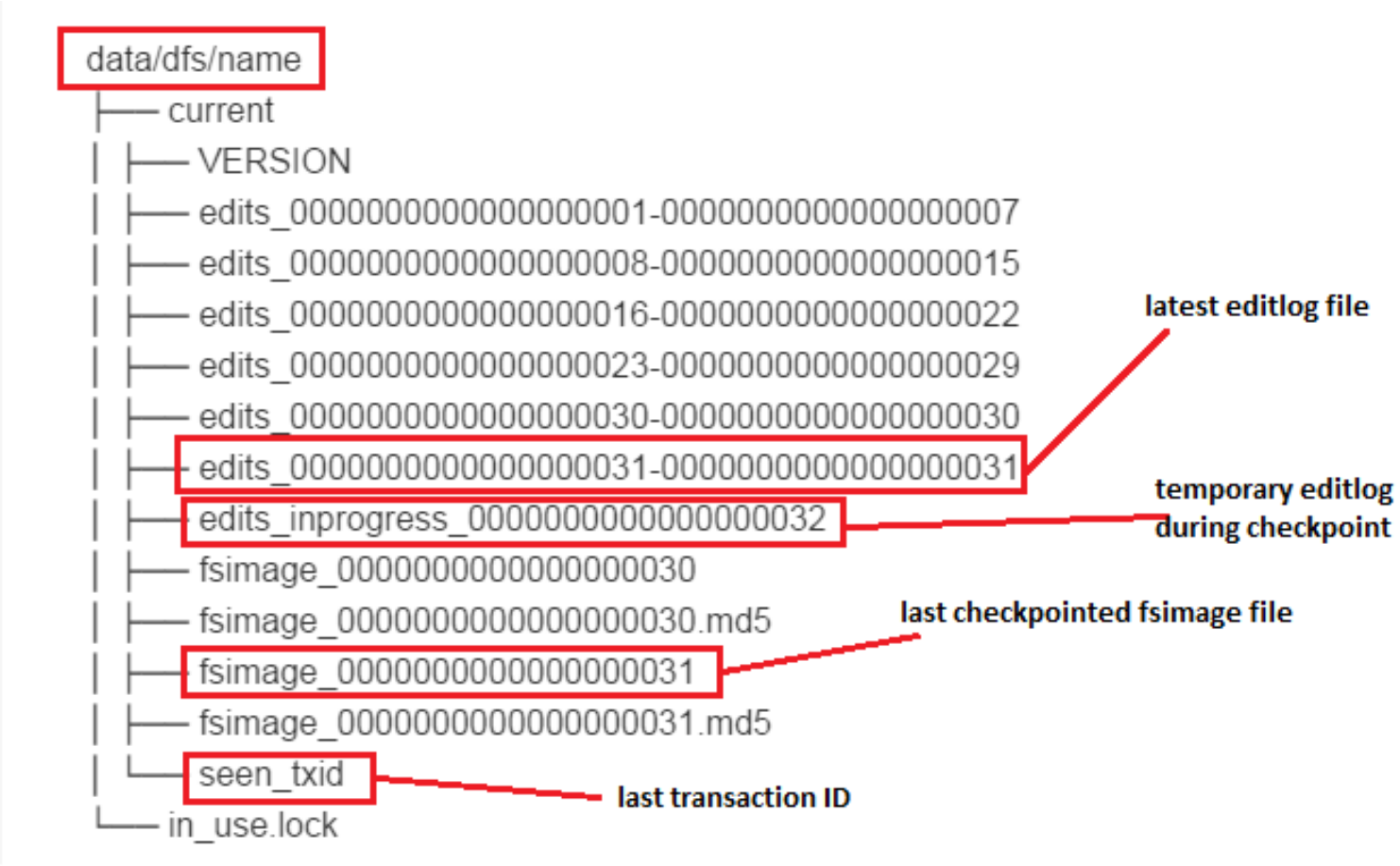
Checkpointing

- A process that takes an **fsimage** and **editLog** and compacts them into a **new fsimage**
- Instead of replaying a potentially unbounded edit log, the NameNode can load the final in-memory state directly from the fsimage.
- More efficient, NameNode startup time reduced.

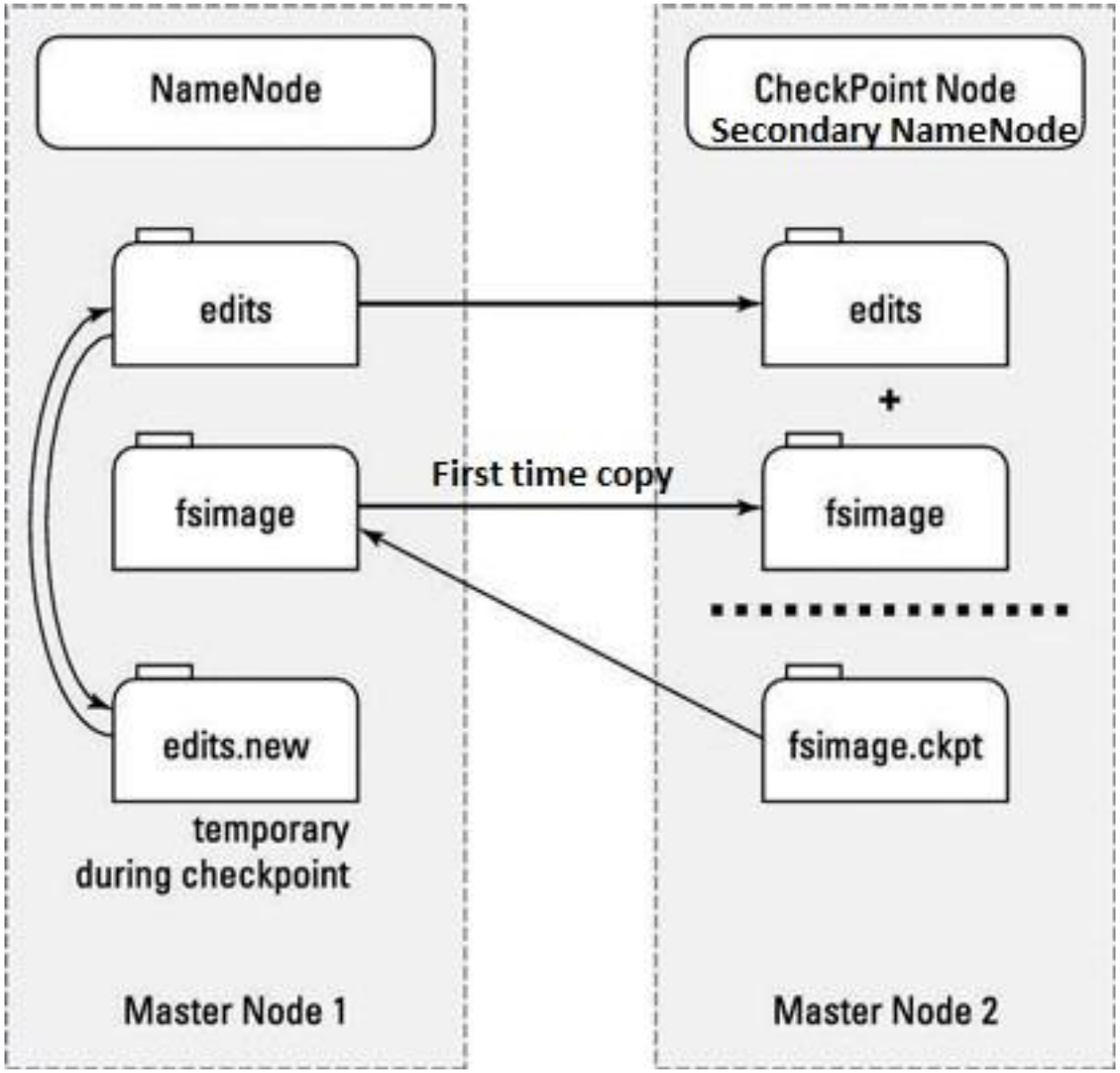


Checkpointing: An example

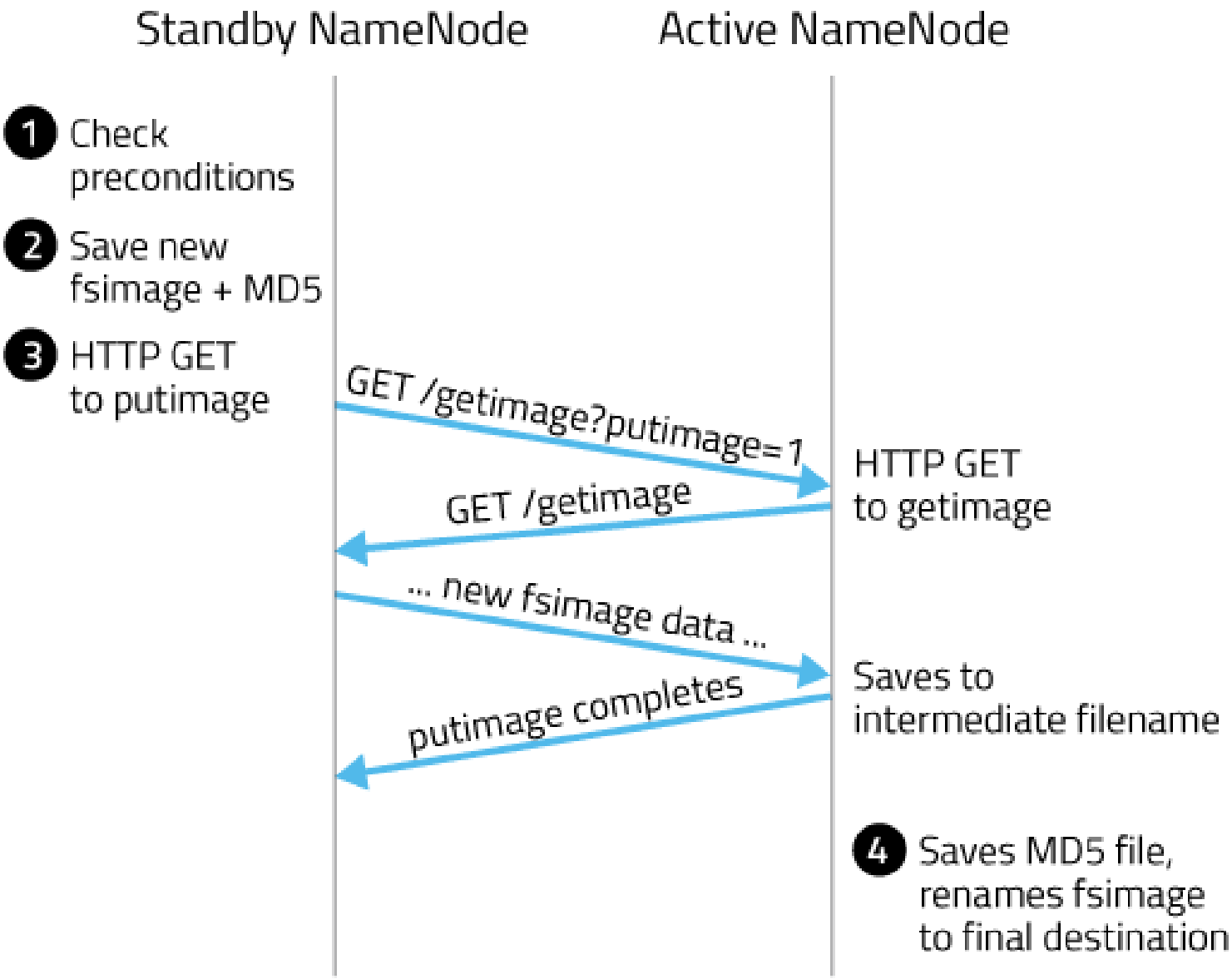
- An example of HDFS metadata directory taken from a NameNode



A typical workflow of checkpointing

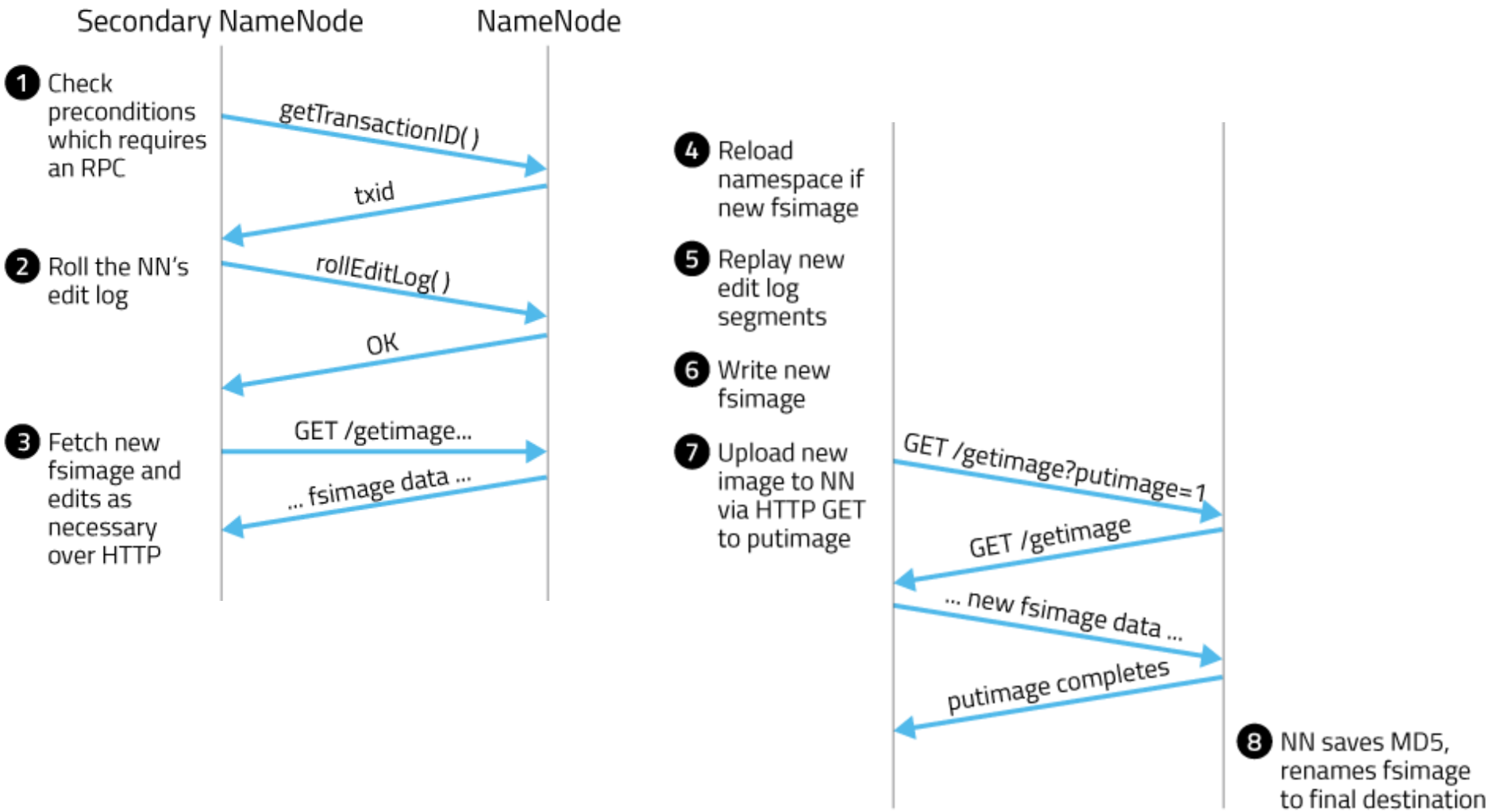


Checkpointing with NameNode HA configured



Checkpointing with NameNode – Secondary NN

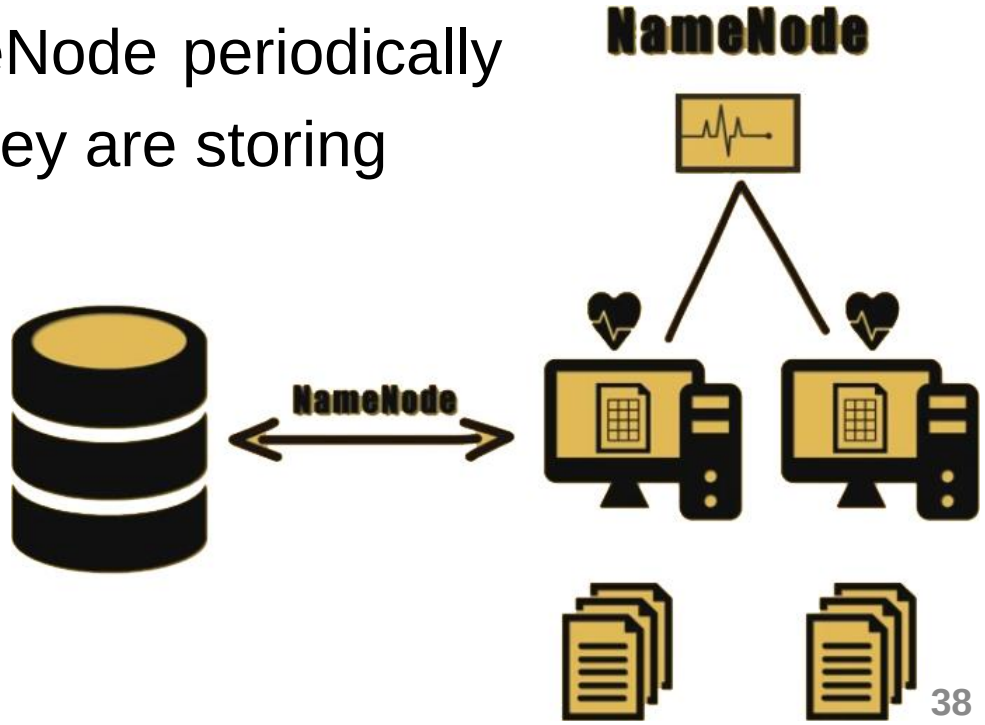
http://blog.cloudera.com/blog/2014/03/a-guide-to-checkpointing-in-hadoop/



DataNode

DataNode

- **DataNodes** are the workhorses of the filesystem.
- Register 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



Functions of a DataNode

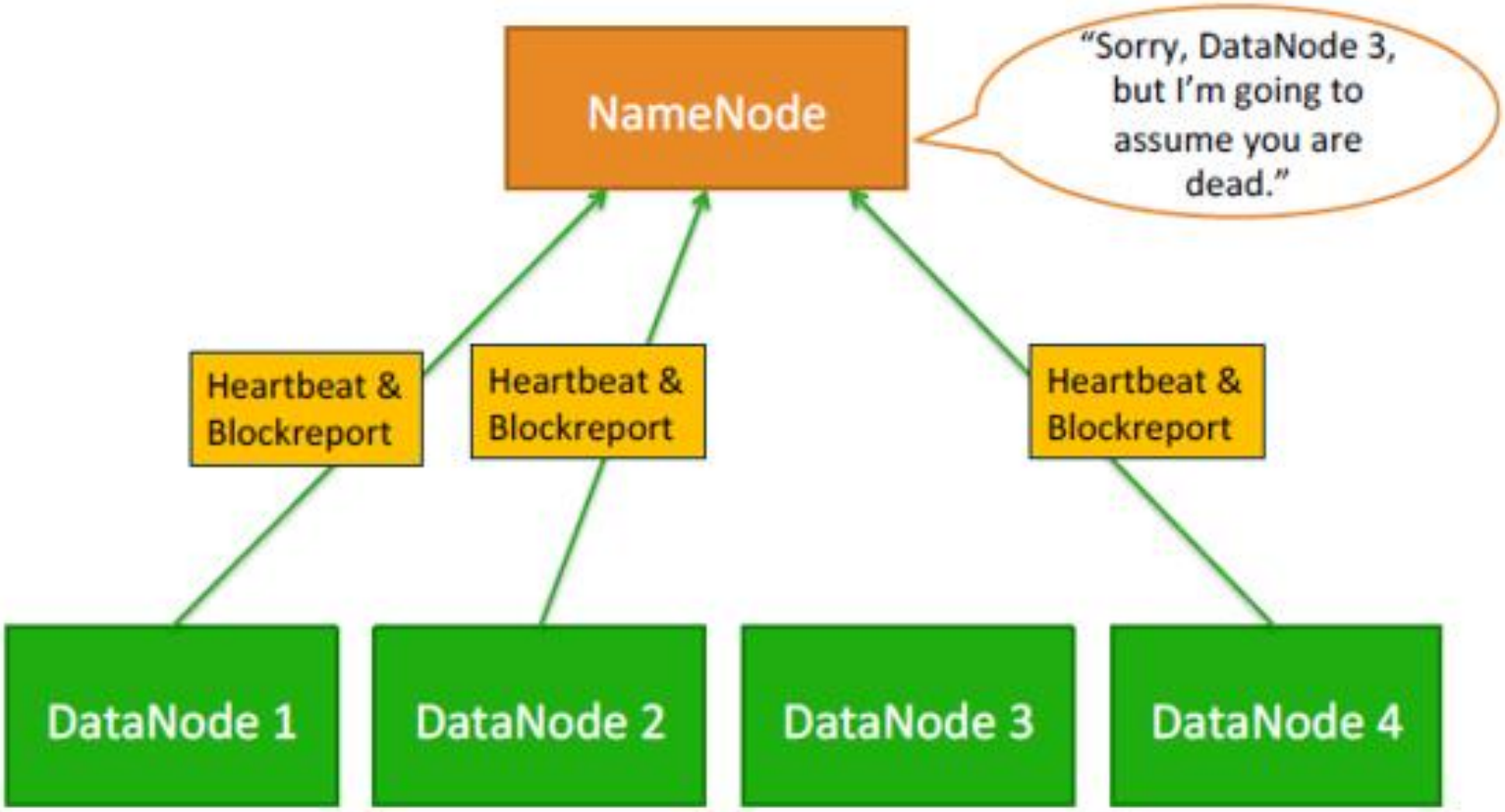
- **Perform low-level read and write requests from clients**
  - Enable pipelining of data, forward data to other specified DataNodes.
- **Responsible for manipulating blocks based on the decisions taken by the NameNode**
  - Create blocks, deleting blocks and replicating the same
  - Store each HDFS data block in separate files in its local filesystem

Functions of a DataNode

- **Send heartbeats to the NameNode every 3 seconds to report the overall health of HDFS**
  - Total storage capacity, fraction of storage in use, and the number of data transfers currently in progress.
- **Periodically send a report on all the blocks present in the cluster to the NameNode**
  - Block ID, the generation stamp and the length for each block replica the server hosts.
  - When getting started, DataNode scans through its local filesystem, creates a list of all data blocks that relate to each of these local files and sends a Block report to the NameNode.

Heartbeat messages

- HDFS uses heartbeat messages to detect connectivity between NameNode and DataNodes.

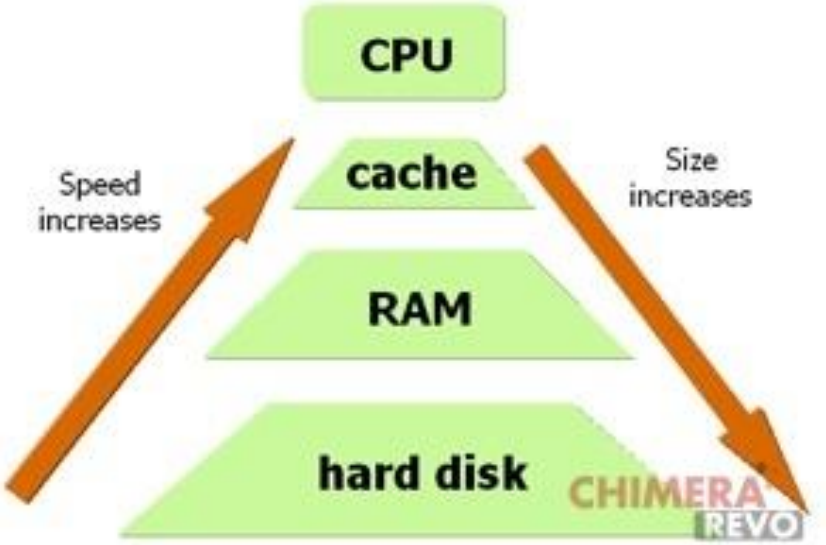


Heartbeat messages

- Periodically sent from each DataNode to NameNode.
- The NameNode marks those from which heartbeats are missing as **dead DataNodes**.
  - Refrain from sending further requests to dead nodes
  - Data stored on a dead node is no longer available, which is effectively removed from the system.
- The death of a node may cause the replication factor of data blocks to drop below their minimum value.
  - The NameNode initiates additional replication to bring the replication factor back to a normalized state.

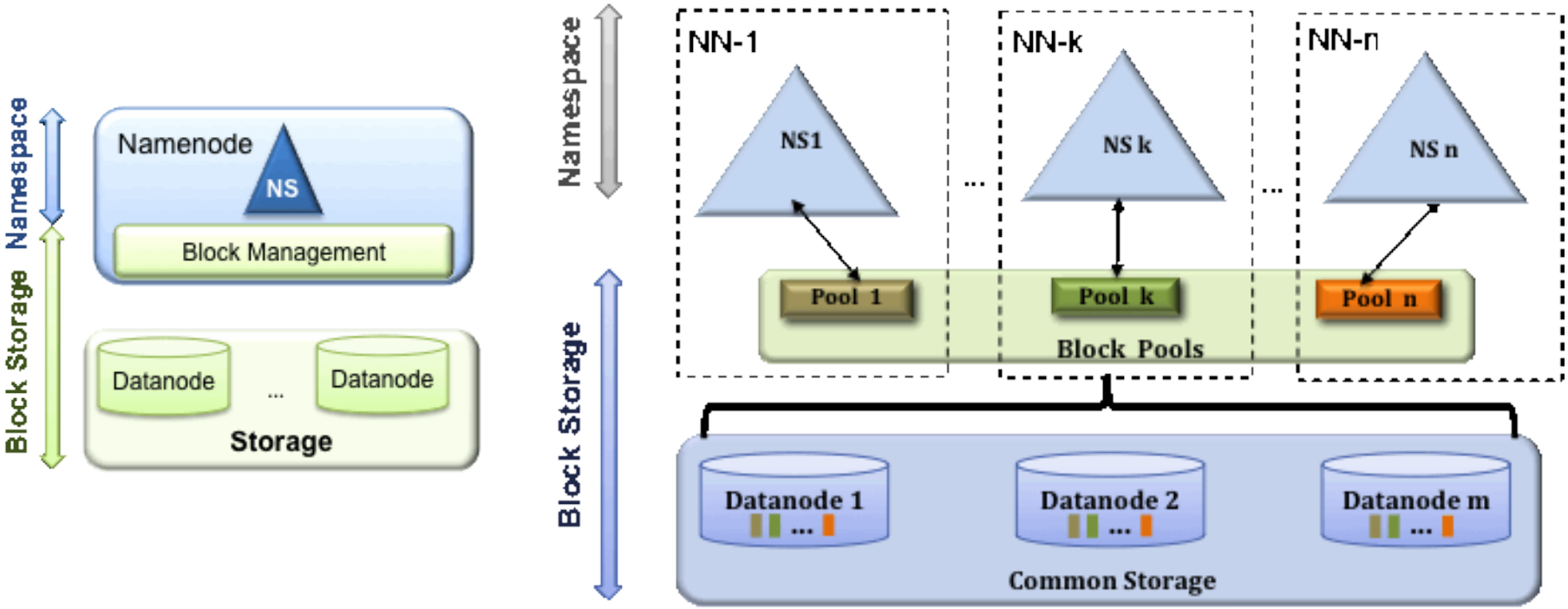
Block caching

- 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.
  - User/application instruct the NameNode which files to cache (and how long) by adding a **cache directive** to a **cache pool**.



HDFS Federation

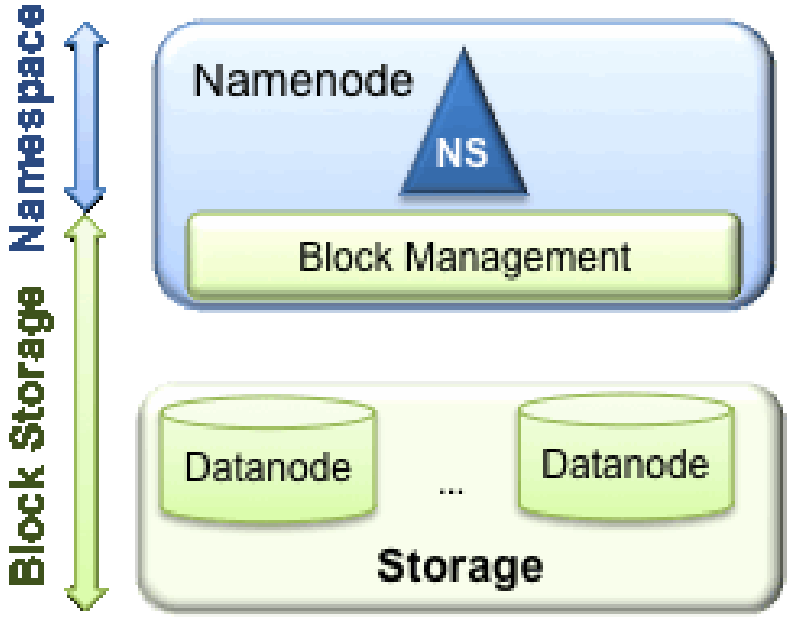
HDFS Federation



Hadoop 1.0 → Hadoop 2.0

HDFS in Hadoop 1.0

- **Namespace**
  - Directories, files and blocks
  - Create, delete, modify and list files or dirs operations



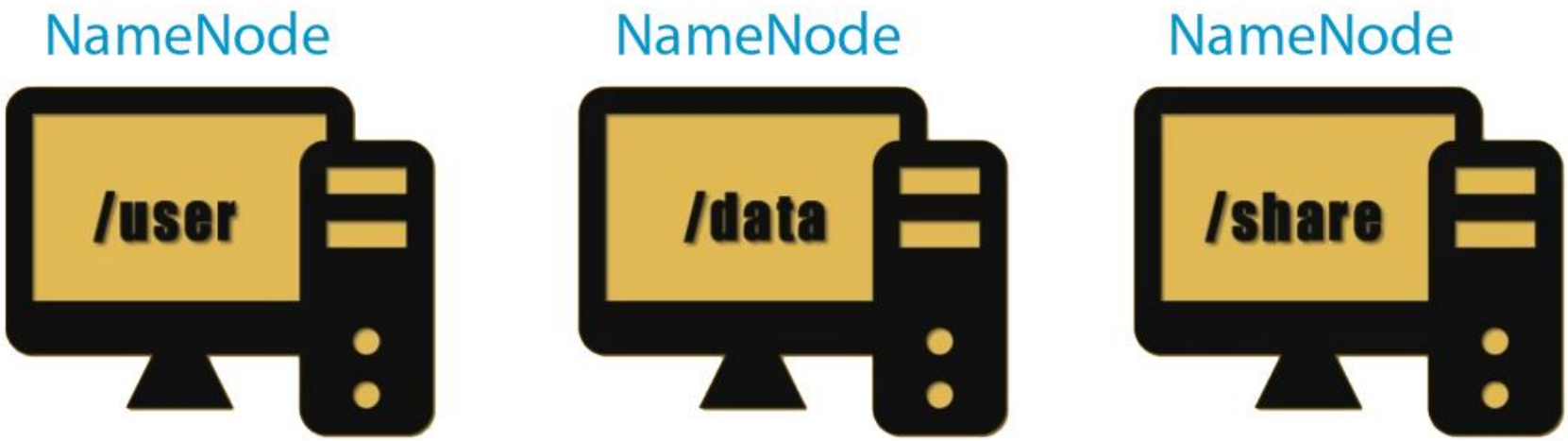
- **Block Storage**
  - **Block Management**
    - DataNode cluster membership
    - Support create/delete/modify/get block location operations
    - Manage replication and replica placement
  - **Storage -** provide read and write access to blocks

HDFS in Hadoop 1.0

- Only **a single namespace** for the entire cluster is allowed.
- **A single NameNode** manages this namespace.
- If cluster is very large with many files, memory becomes the limiting factor of scaling.

HDFS Federation

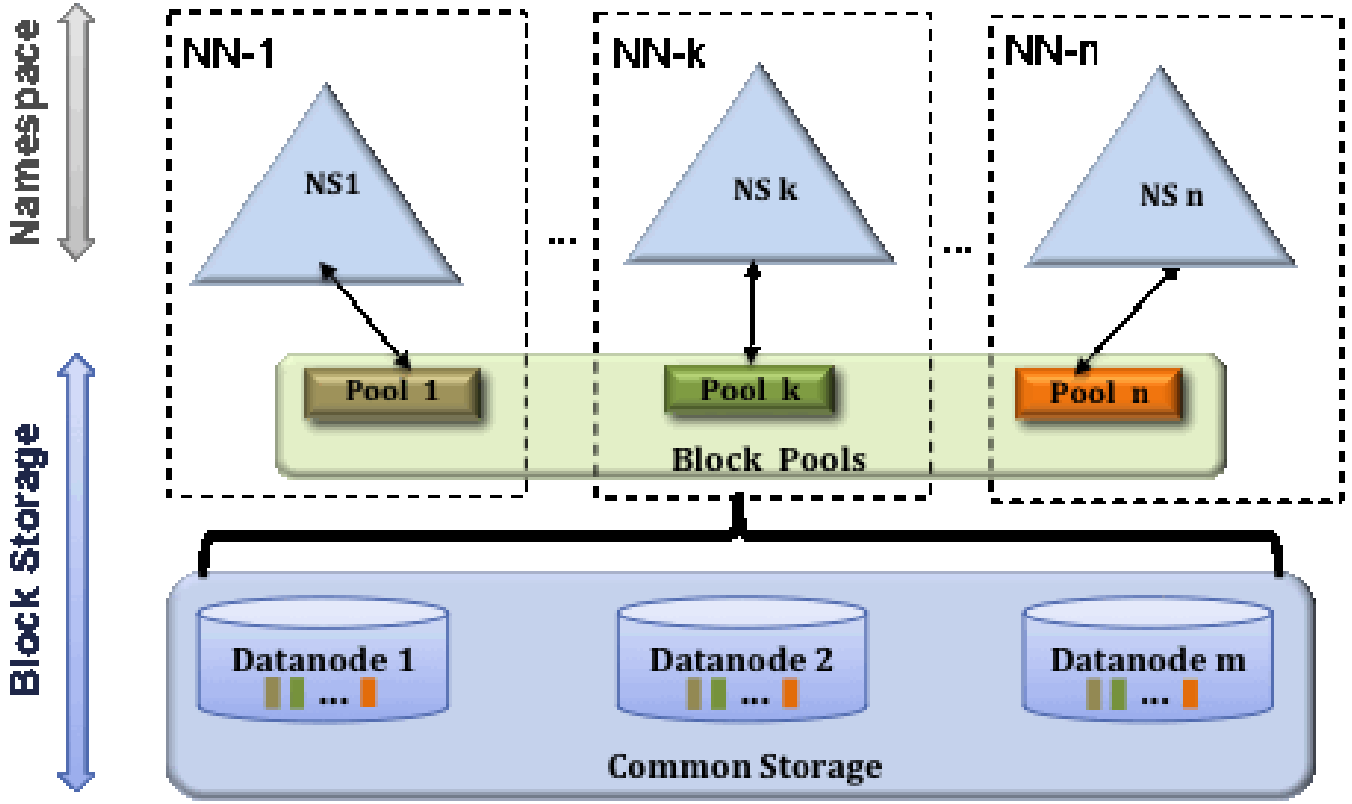
- **HDFS Federation** allows adding multiple NameNodes, and hence multiple namespaces, to the HDFS filesystem.
- For example:
  - One NameNode might manage all files rooted under /user
  - Second NameNode might handle files under /data
  - Thrid NameNode might handle files under /share



HDFS Federation

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

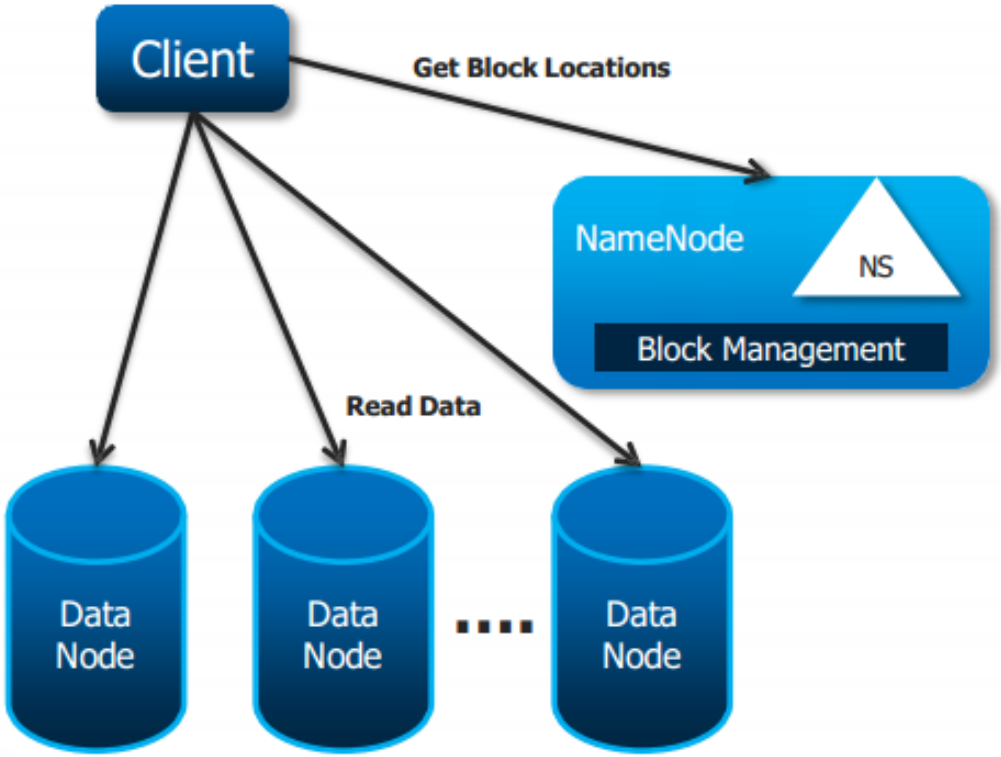
HDFS Federation



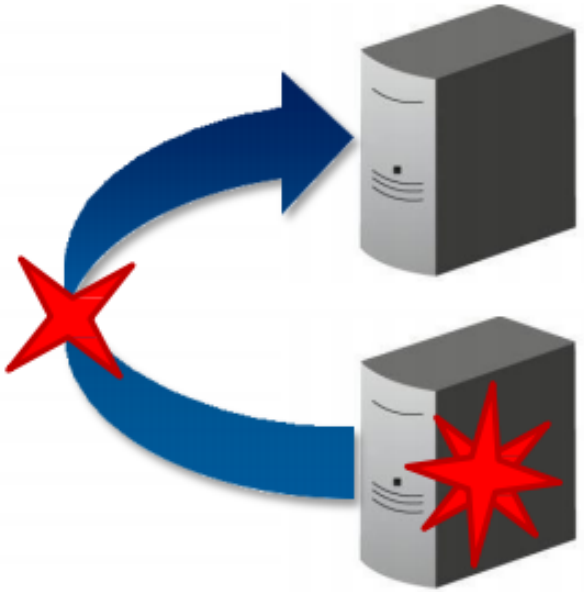
- Namespace Volume = Namespace + Block Pool
- Block Storage as generic storage service
  - Set of blocks for a Namespace Volume is called a **Block Pool**.
  - DataNodes store blocks for all the Namespace Volumes – **no partitioning**.

HDFS High Availability

Single Point of Failure



NameNode - No Horizontal Scale



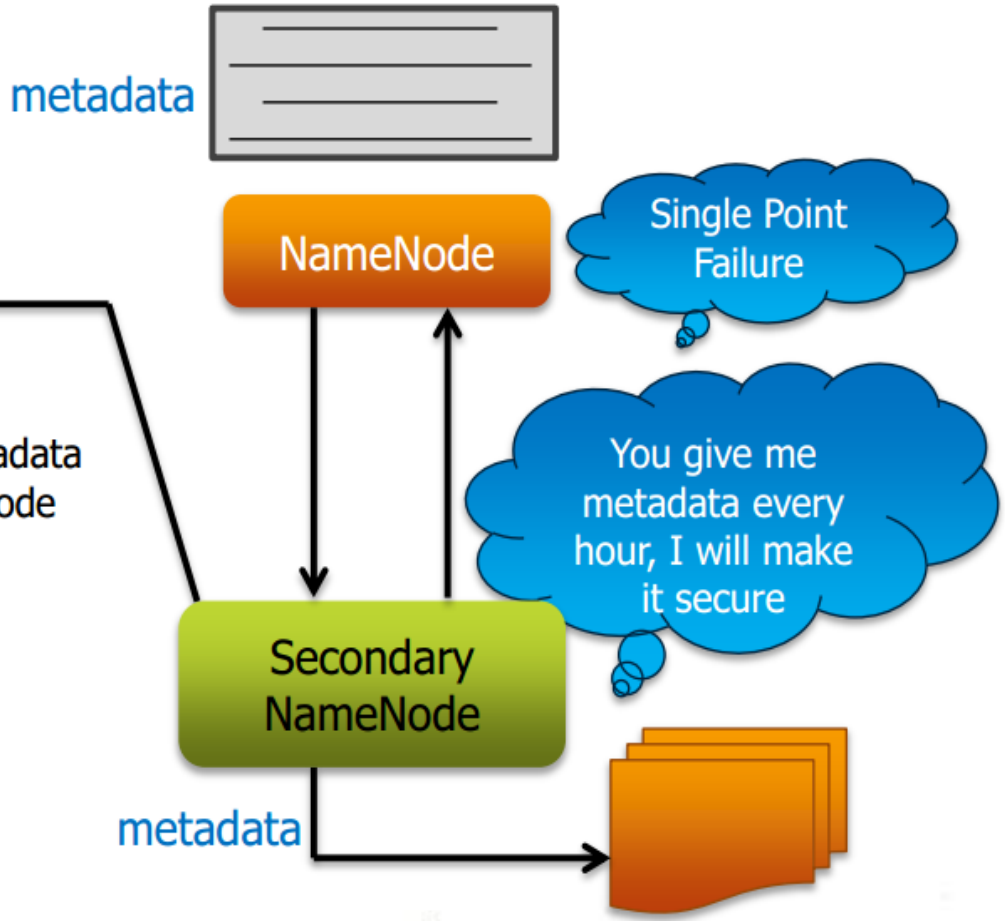
NameNode - No High Availability

Single Point of Failure

- HDFS did:
  - Replicating NameNode metadata on multiple filesystems.
  - Using Secondary NameNode to create checkpoints protects against data loss.
- Yet, it does not provide high availability of the filesystem.
- NameNode is still a **single point of failure** (SPOF).
  - If it did fail, all clients would be unable to read, write,…

Remind Secondary NameNode

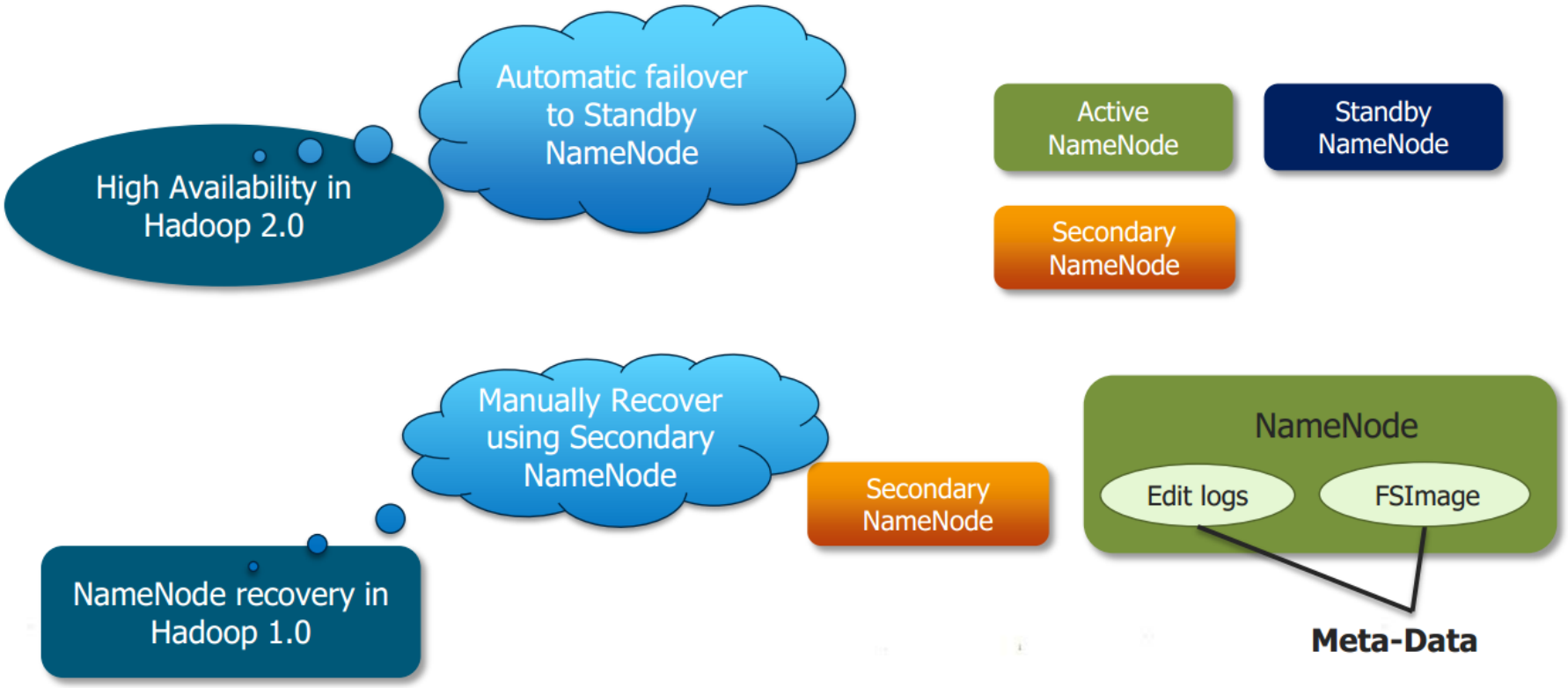
- **Secondary NameNode:**
  - "Not a hot standby" for the NameNode
  - Connects to NameNode every hour*
  - Housekeeping, backup of NameNode metadata
  - Saved metadata can build a failed NameNode



Recover from a failed NameNode

- **Start a new primary NameNode** and configure DataNode and clients to use this new NameNode
- Note that new NameNode is not able to serve requests until
  - It has loaded its namespace image into memory
  - Replayed its edit log
  - Received enough block reports from DataNodes to leave safe mode.
- If clusters is large, it takes **about 30 minutes or more**.

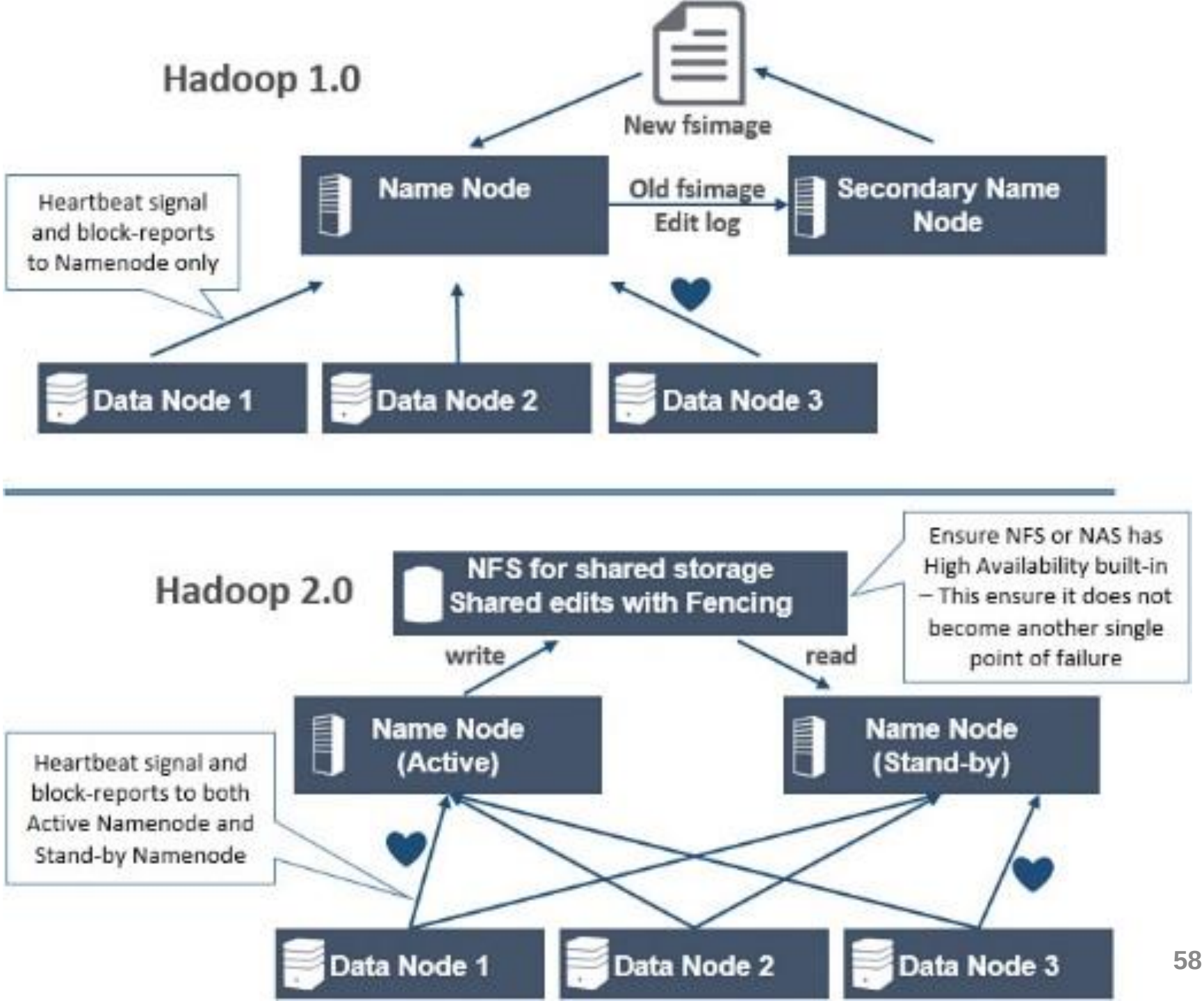
HDFS High Availability



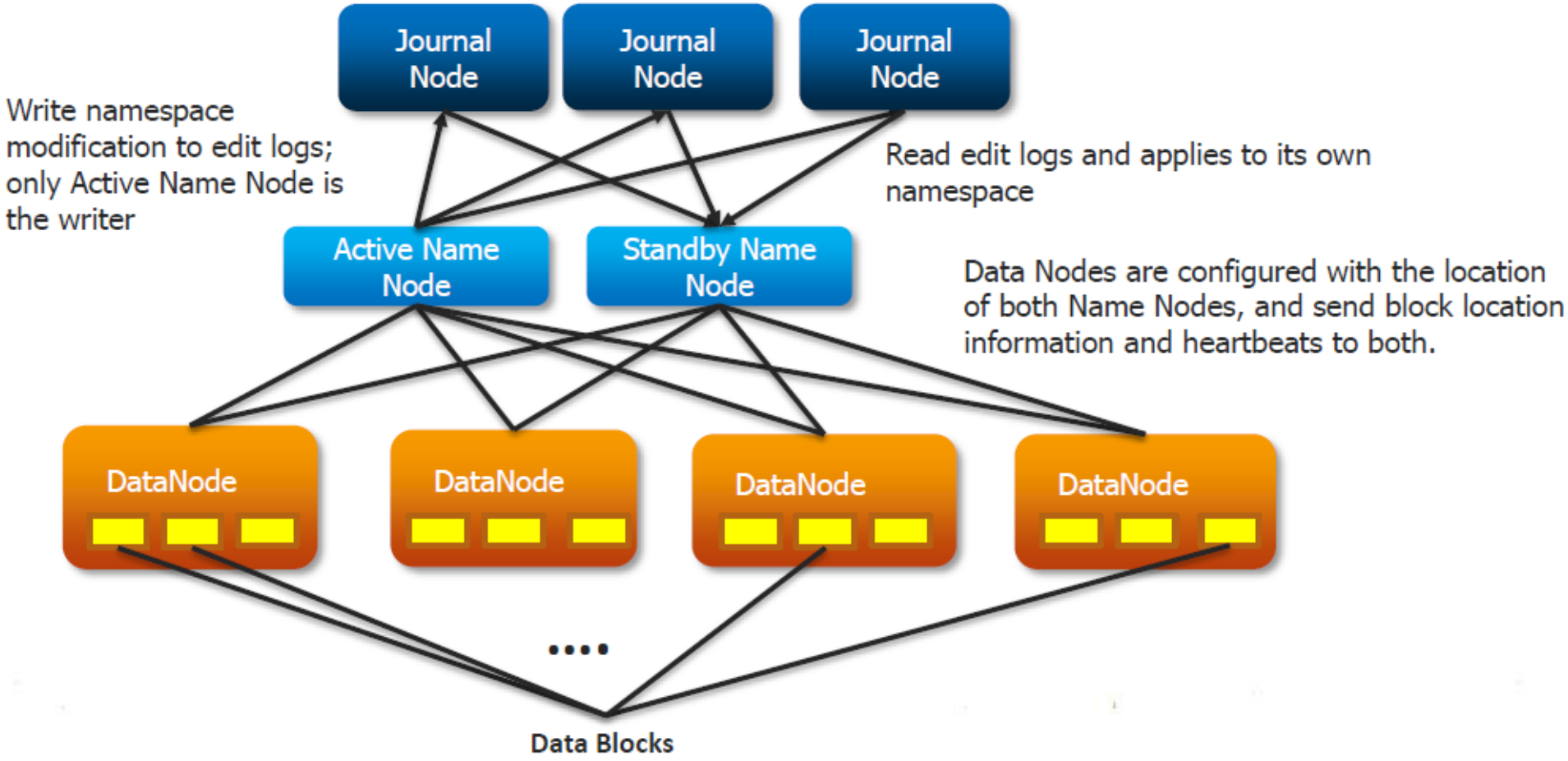
- A pair of NameNodes in an **active-standby** mode
- In failure, the standby takes over its duties to continue servicing client requests without a significant interruption

HDFS High Availability

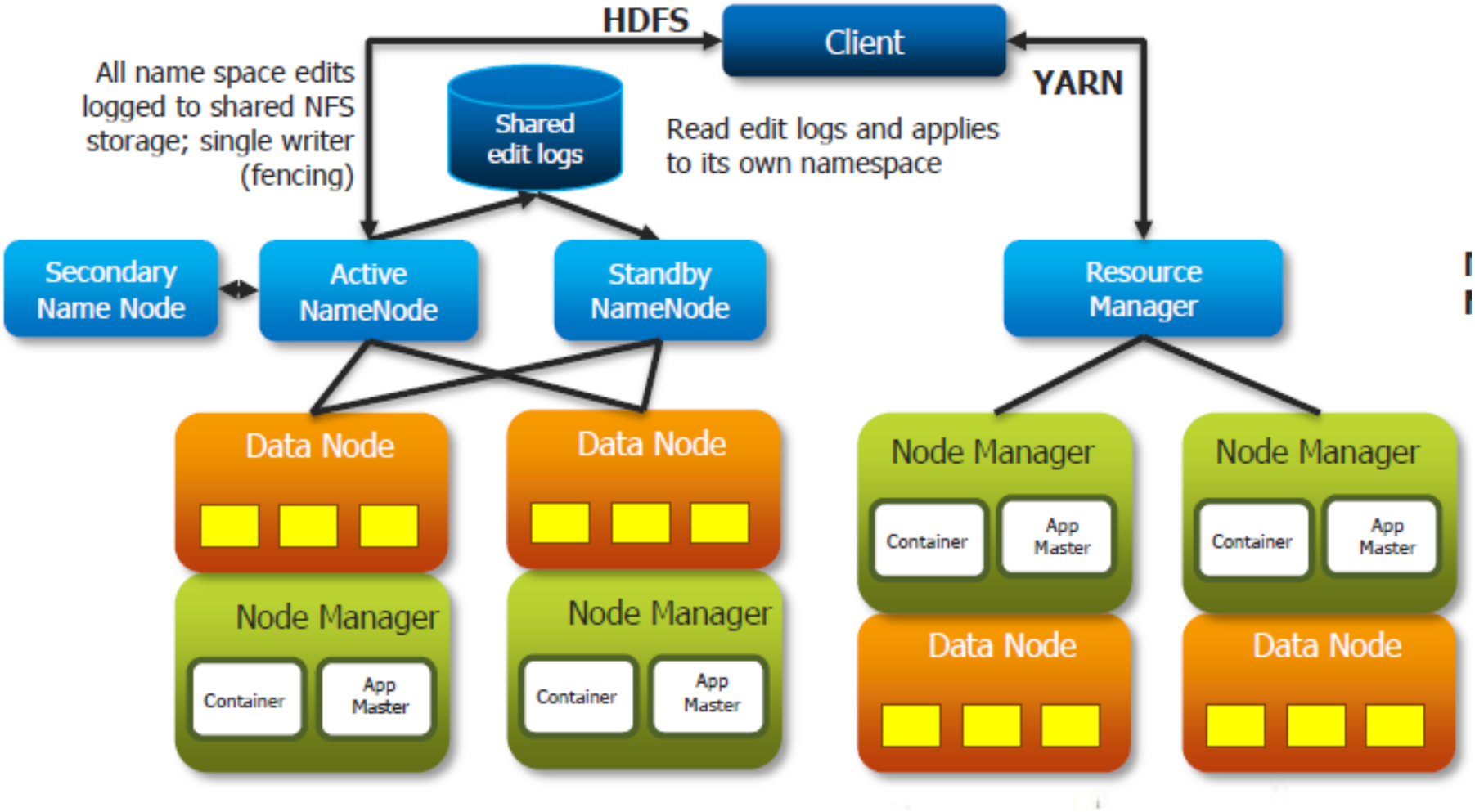
- Use **shared storage** to share edit log
  - Some techniques in shared storage: NFS filer, Quorum journal manager (QJM)
- When coming up, a standby NameNode will
  - Read up to end of the shared edit log to synchronize its state with the active NameNode, then
  - Continue to read new entries as they 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



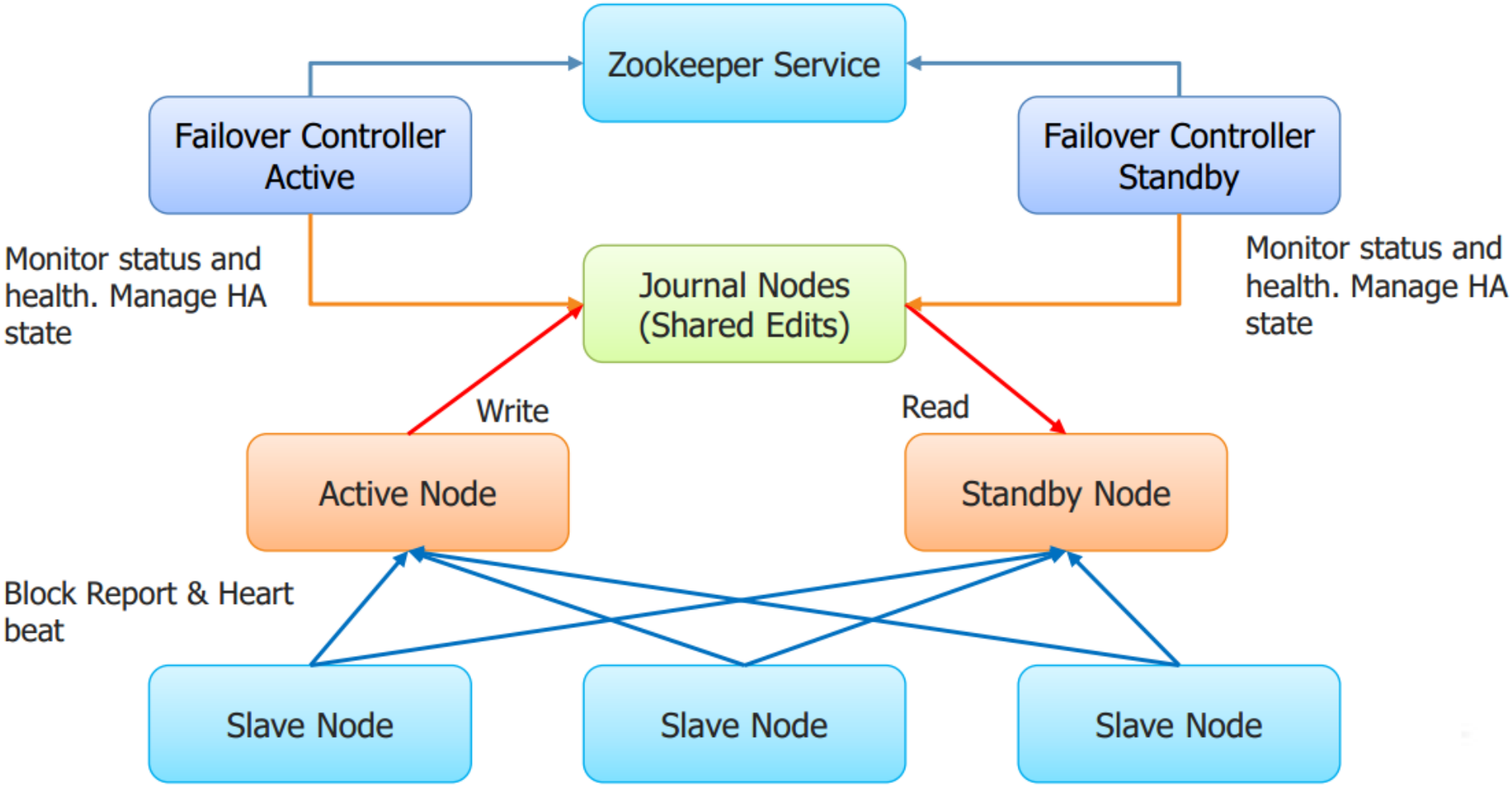
HDFS High Availability with YARN



Failover controller

- The **transition from the active NameNode to the standby** one is managed by an entity integrated the system called the **failover controller**.
  - ZooKeeper Failover Controller (ZKFC), …
- Failover that may be initiated manually by an administrator is call a **graceful failover**.
  - For example, in the case of routine maintenance.
  - The failover controller arranges an orderly transition for both NameNodes to switch roles.

Failover controller

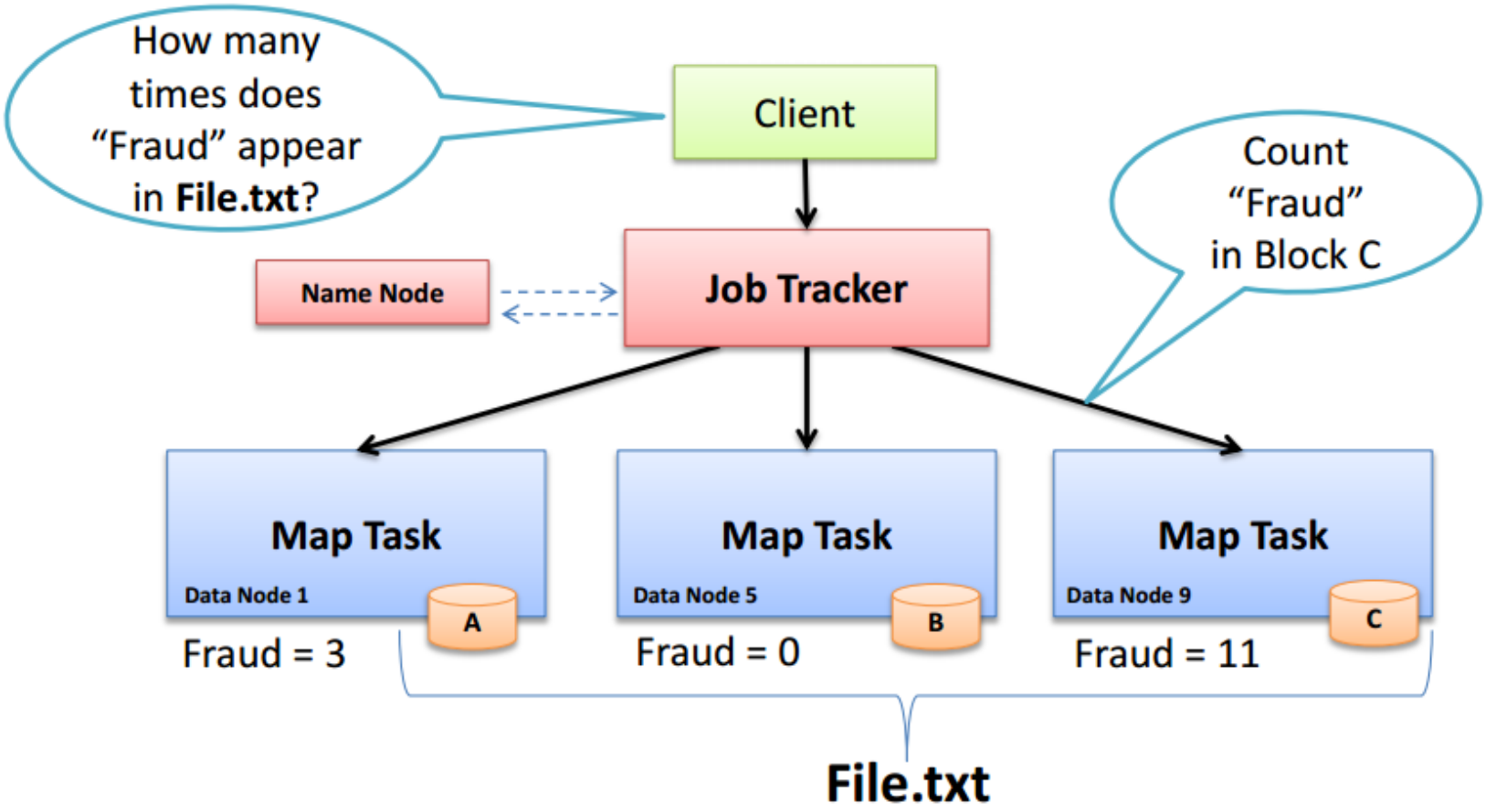


Fencing

- It is impossible to be sure that the failed NameNode has stopped running.
  - E.g., a slow network or a network partition can trigger a failover transition → active NameNode is still running and think it is still the active NameNode.
- The implementation to ensure that the previously active NameNode is prevented from doing any damage and causing corruption is **fencing**.
- This can be controlled by the use of ZooKeeper or something similar.

Data processing on HDFS

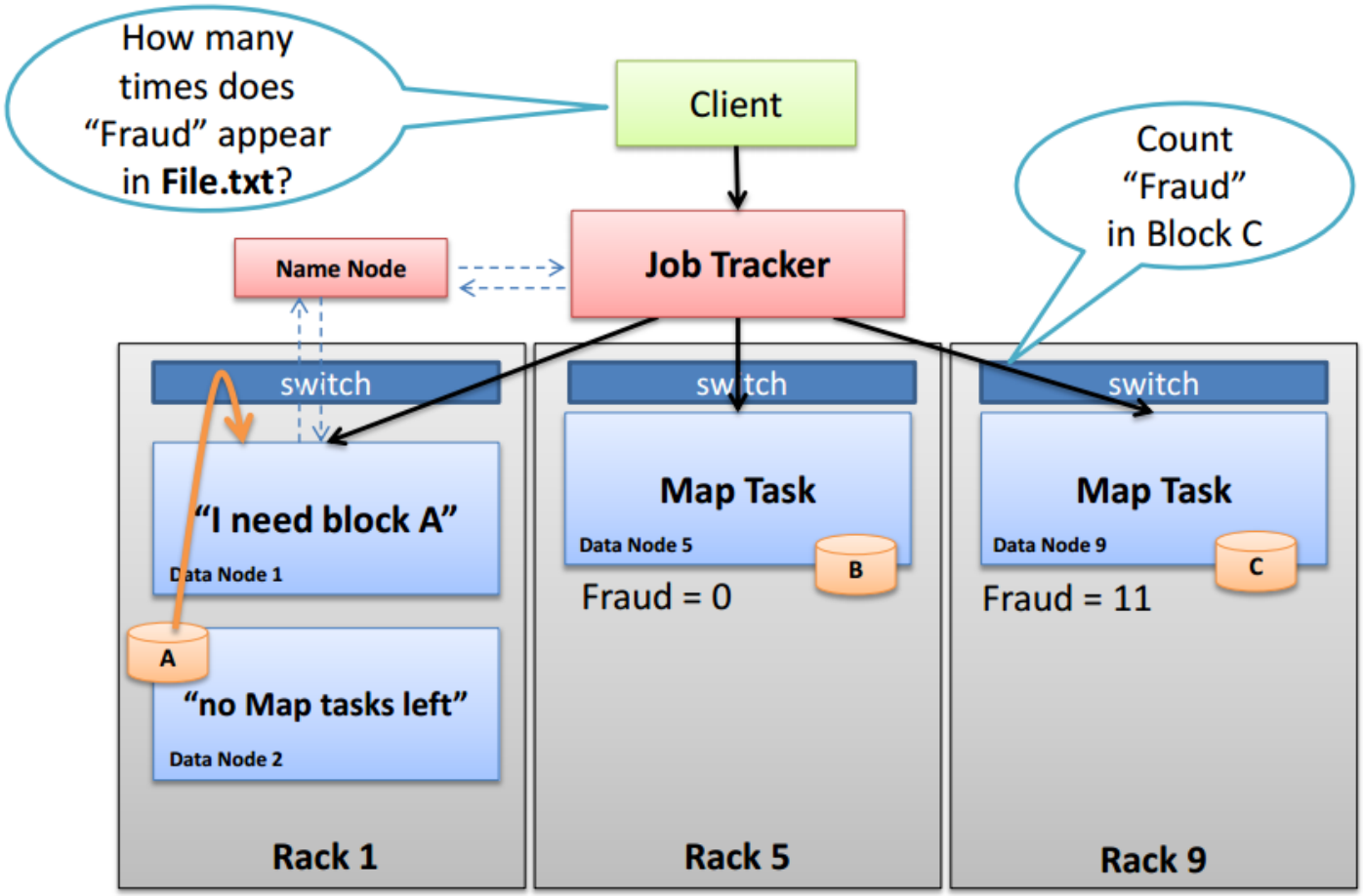
Data processing with Map task



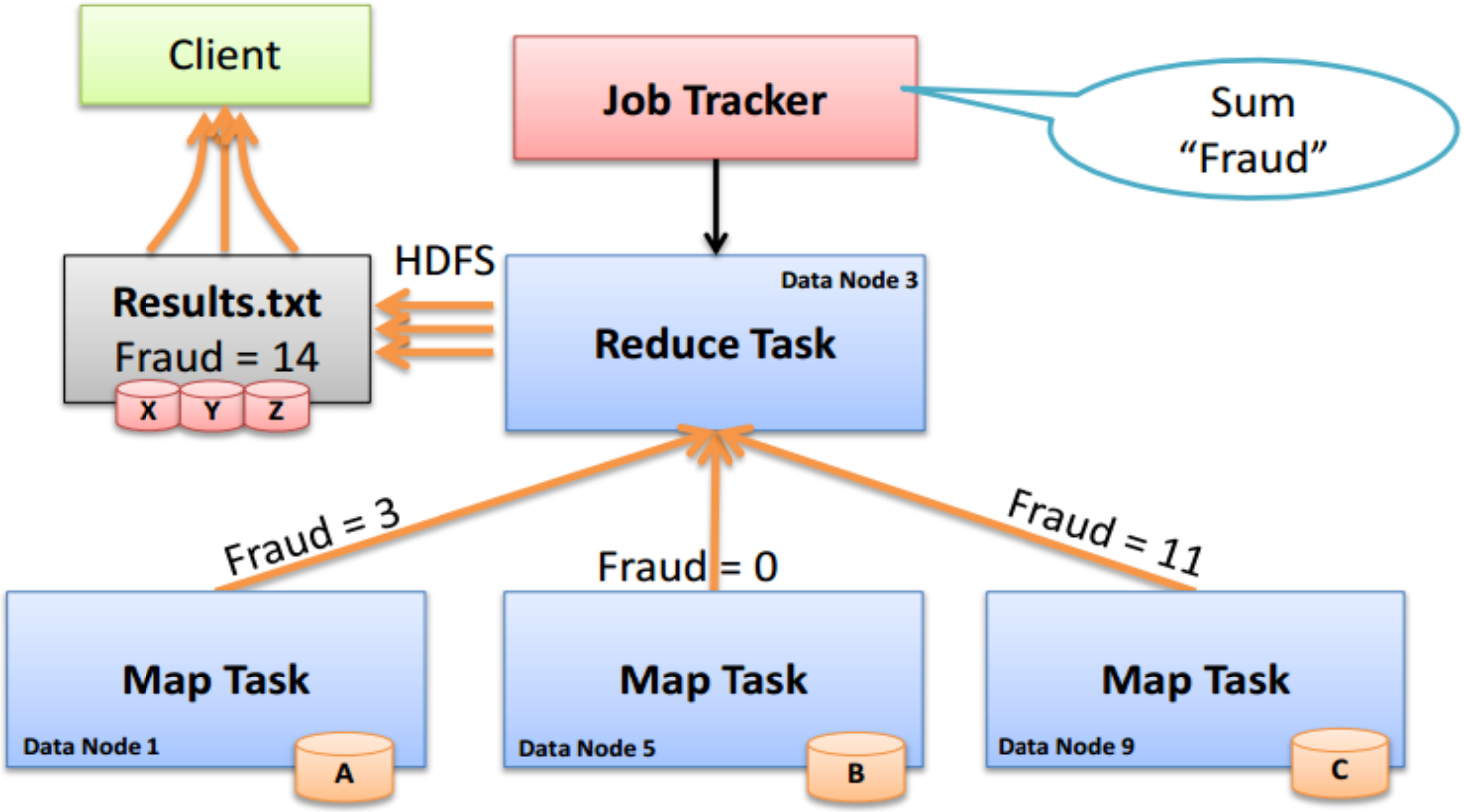
- Map: "Run this computation on your local data".
- JobTracker delivers the code to DataNodes with local data.

What if data is not local?

- **Rack awareness:** The JobTracker tries to select a node in same rack as data.



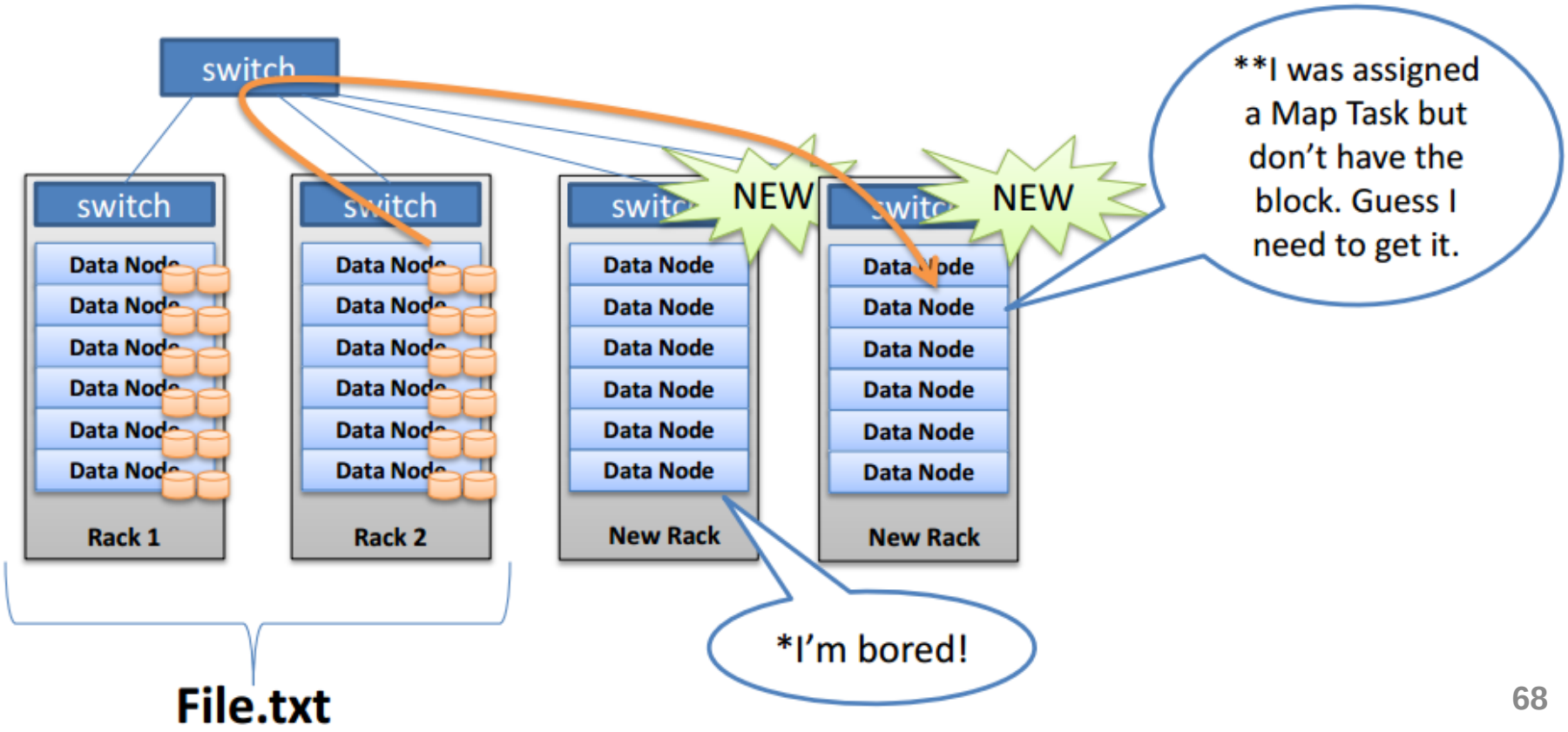
Data processing with Reduce task



- Reduce: "Run this computation across Map results".
- Map tasks deliver output data over the network.
- Reduce task data output written to and read from HDFS.

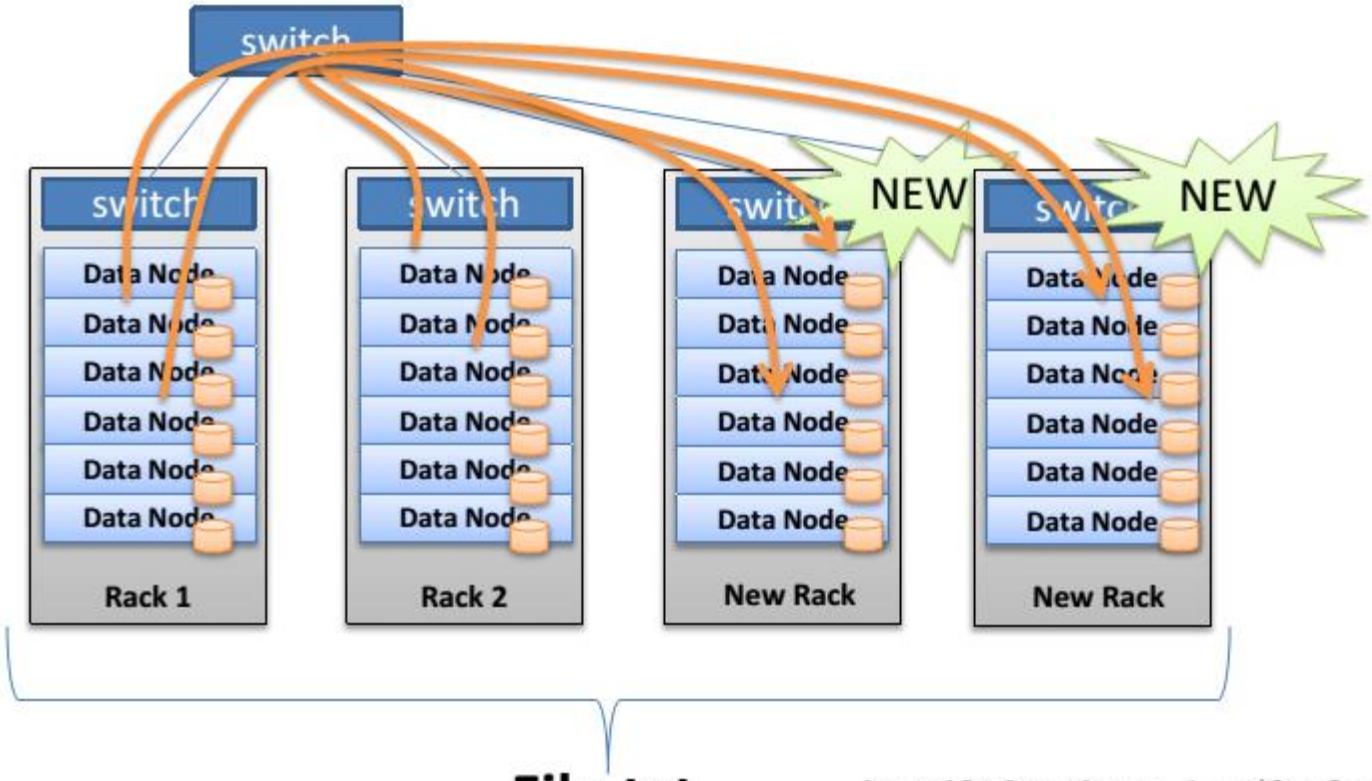
Unbalanced cluster

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

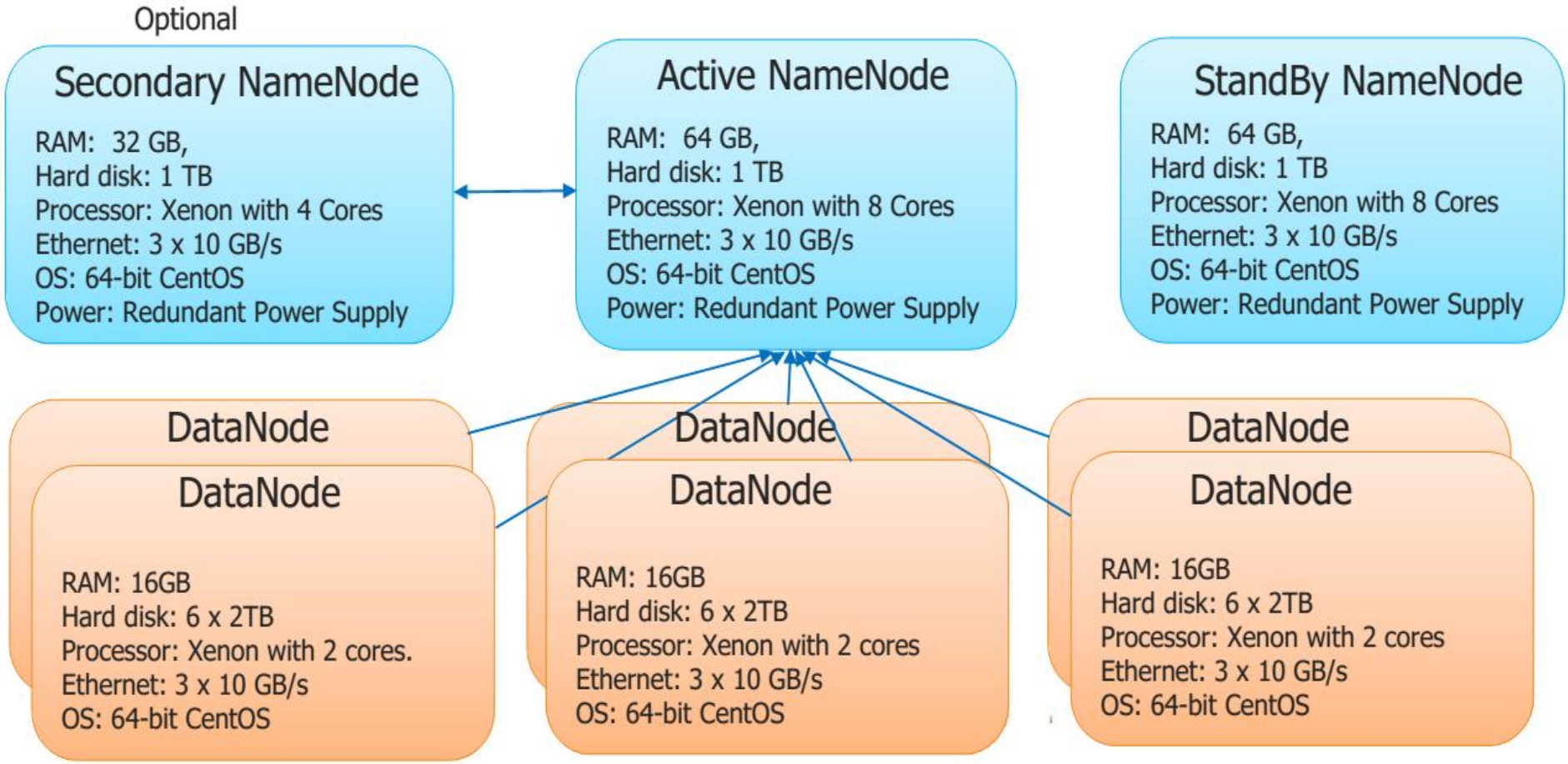


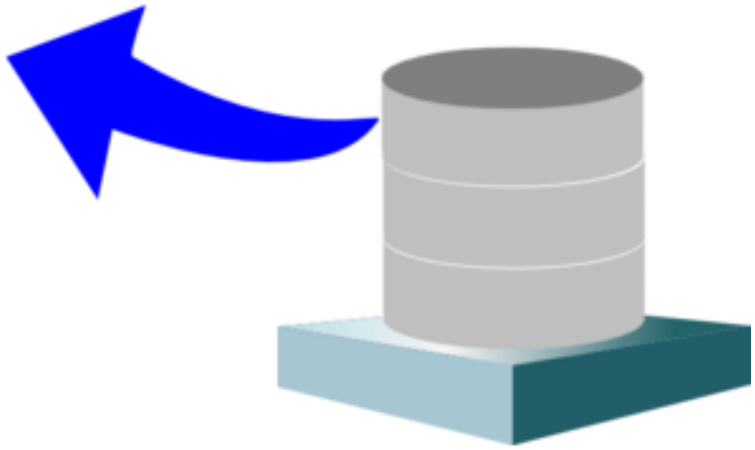
Cluster balancing

- Balancer utility (if used) runs in the background.
- Does not interfere with Map Reduce or HDFS.
- Default speed limit 1 MB/s.



An example of HDFS configurations

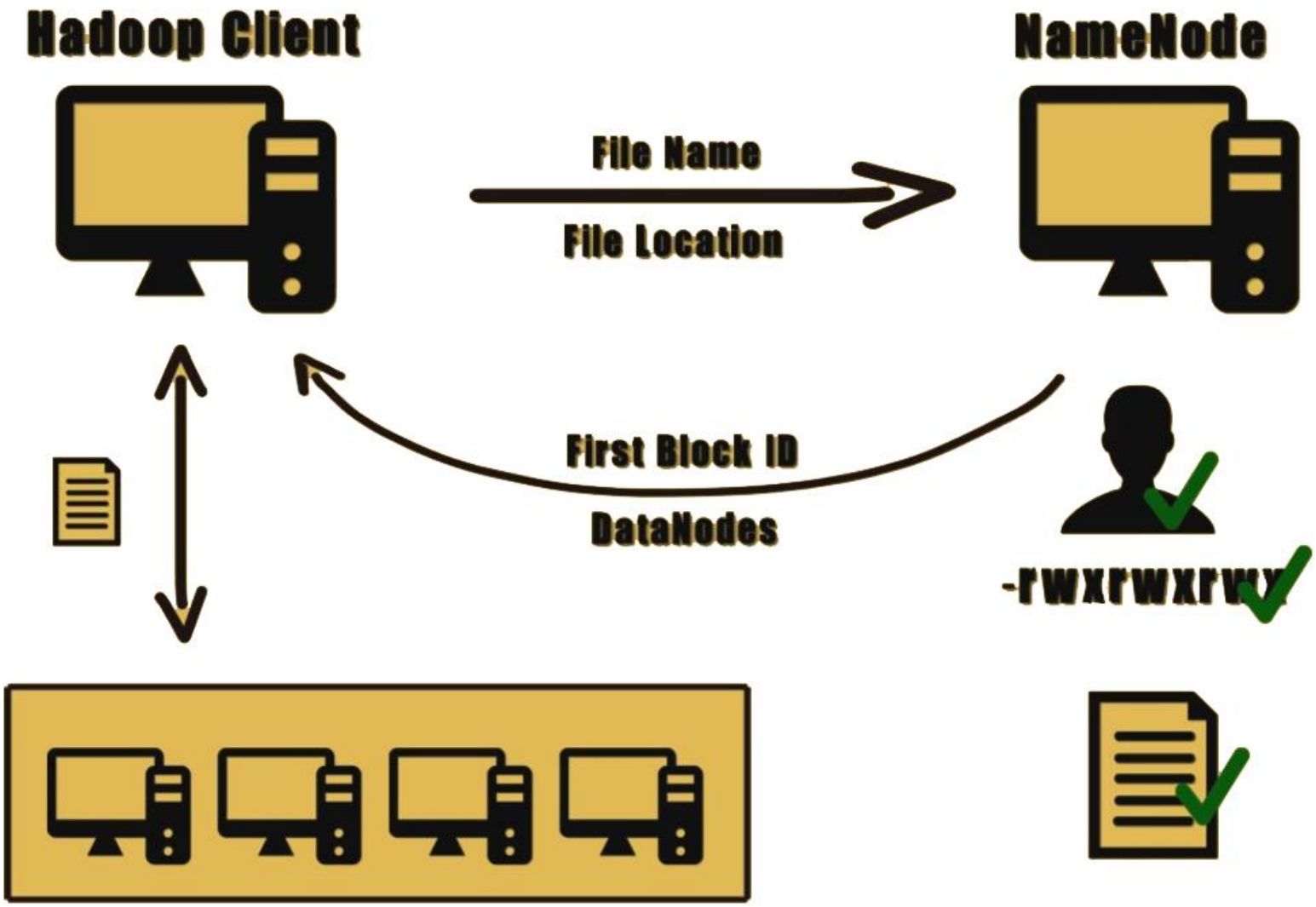




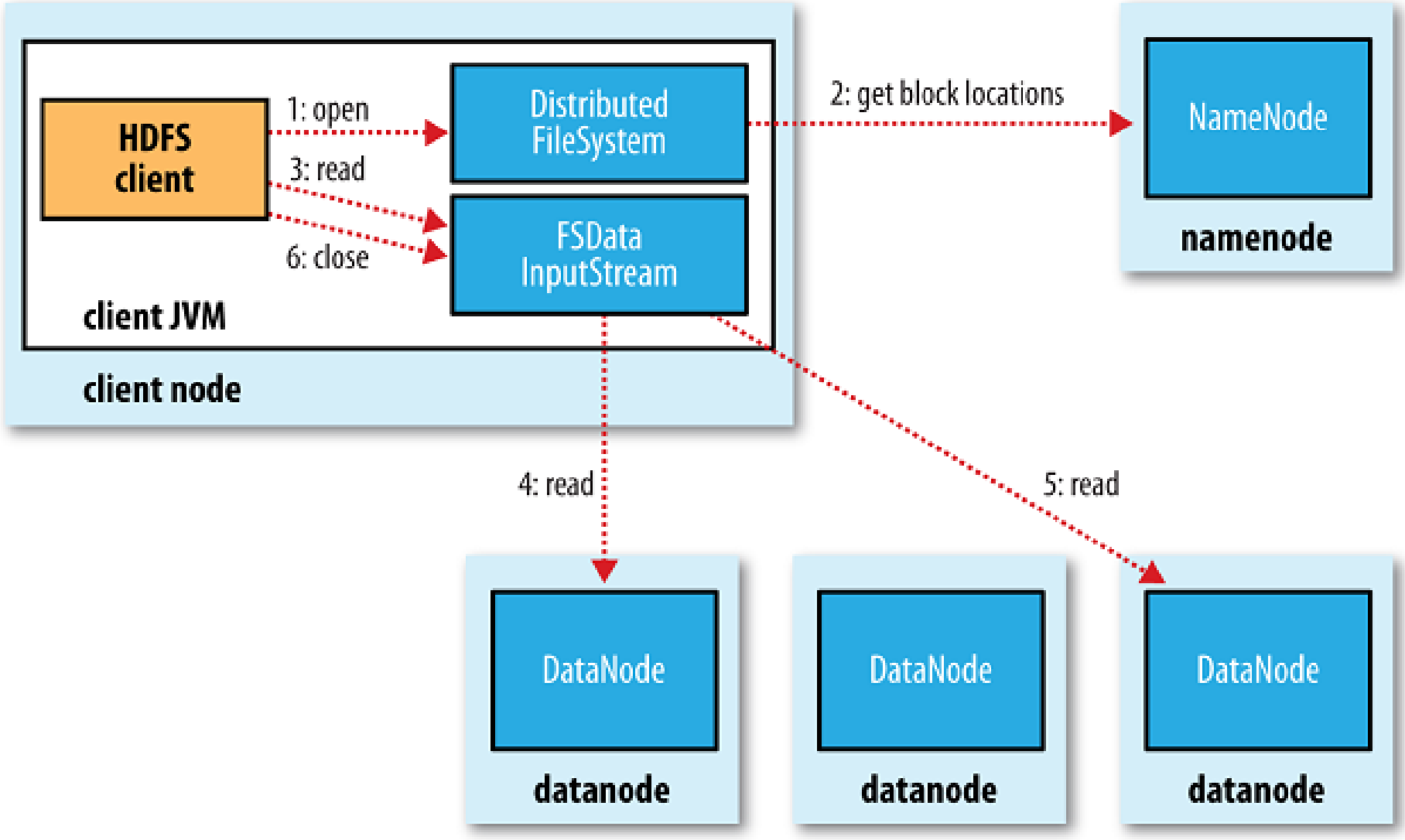
READ DATA FROM HDFS

Anatomy of a file read

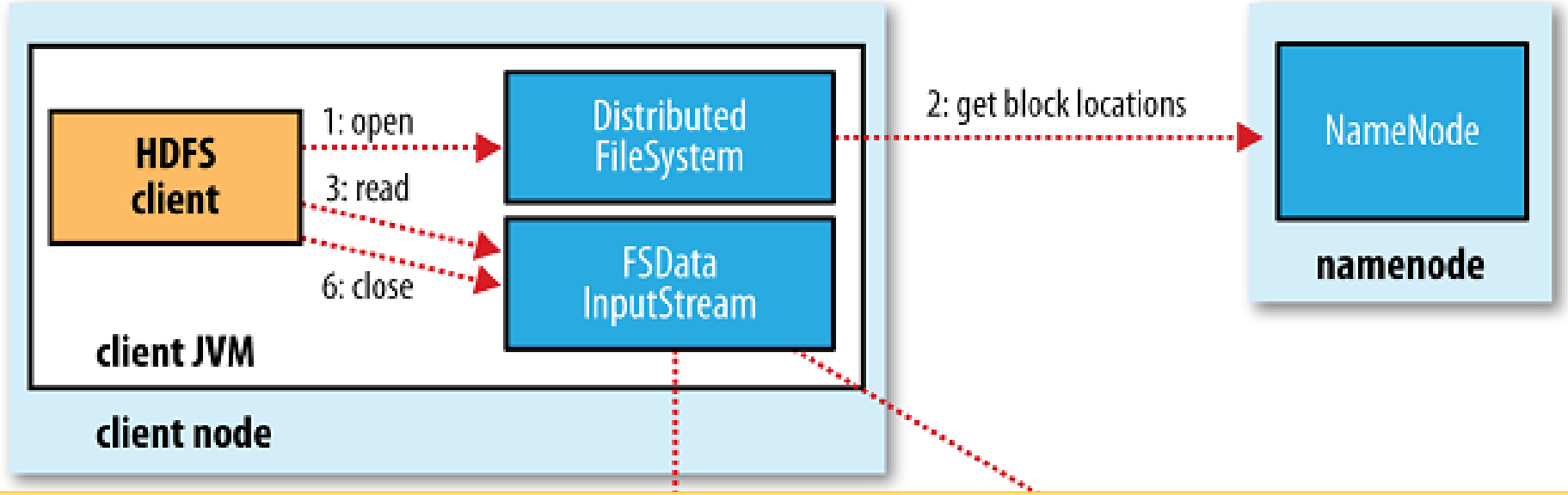
A general workflow of data read



Anatomy of a file read

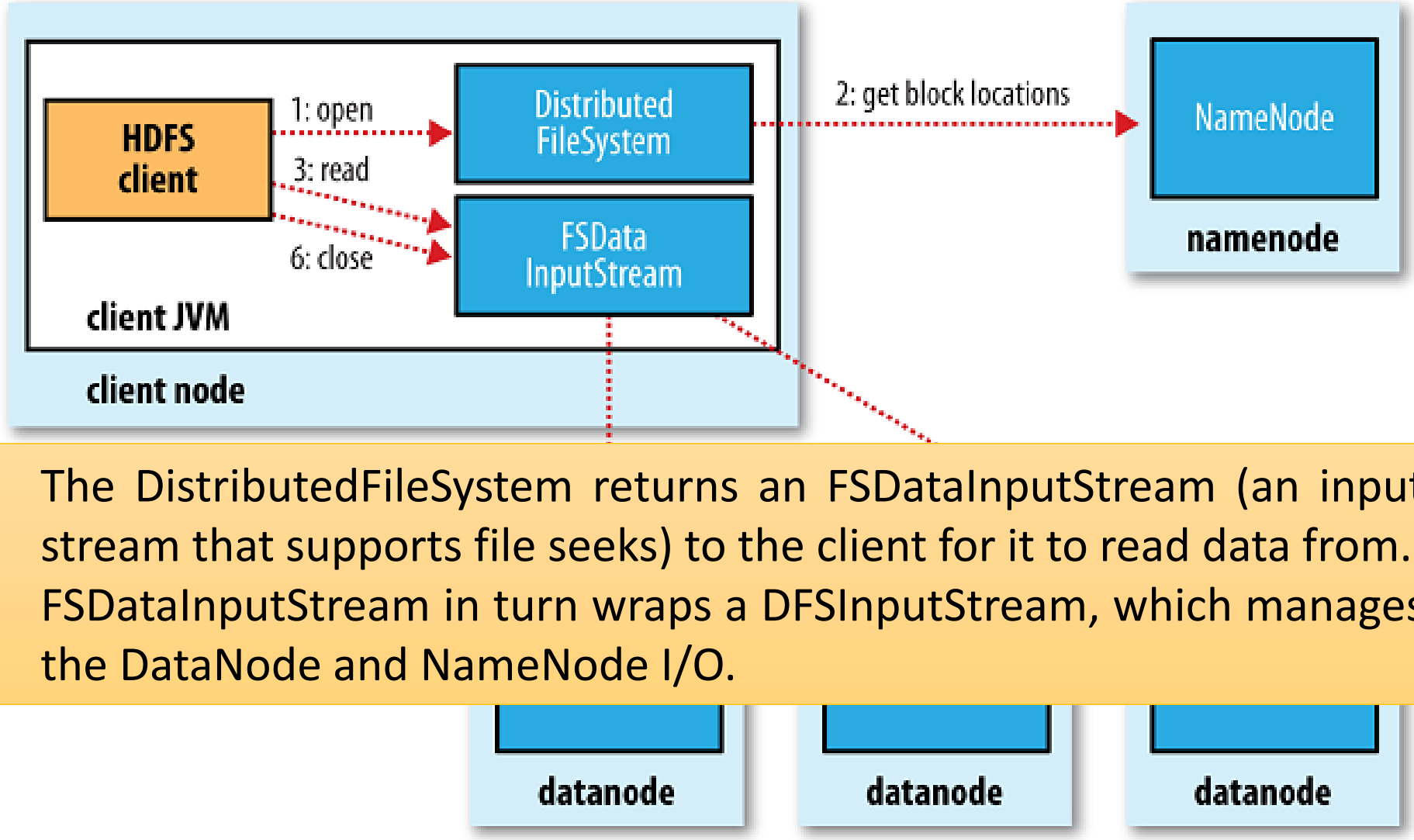


Anatomy of a file read



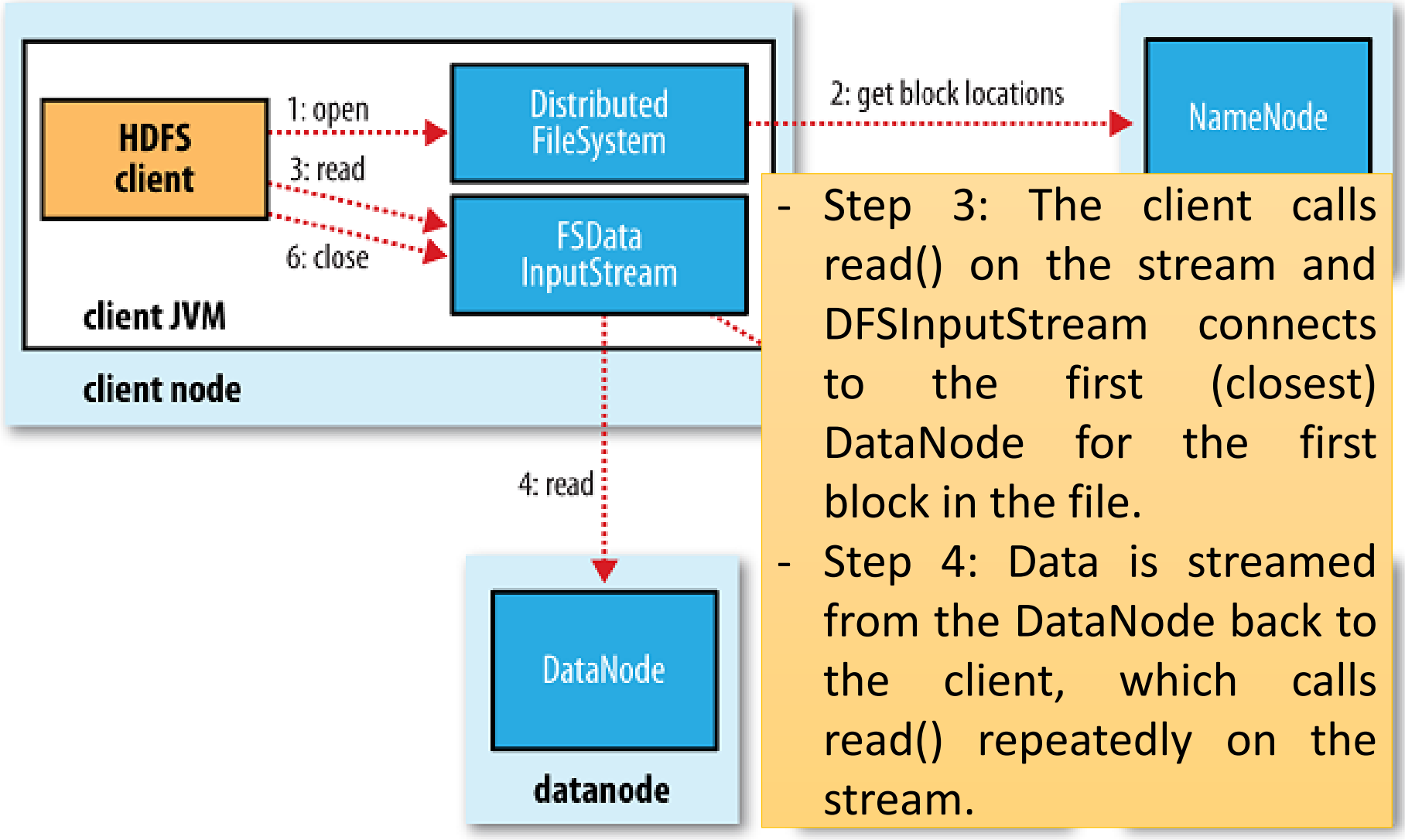
- Step 1: The client calls open() on the FileSystem object.
- Step 2: DistributedFileSystem calls the NameNode to determine the locations of the first few blocks in the file.
  - *For each block, the NameNode returns the addresses of the DataNodes that have a copy of that block, which are sorted according 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.*

Read data from HDFS



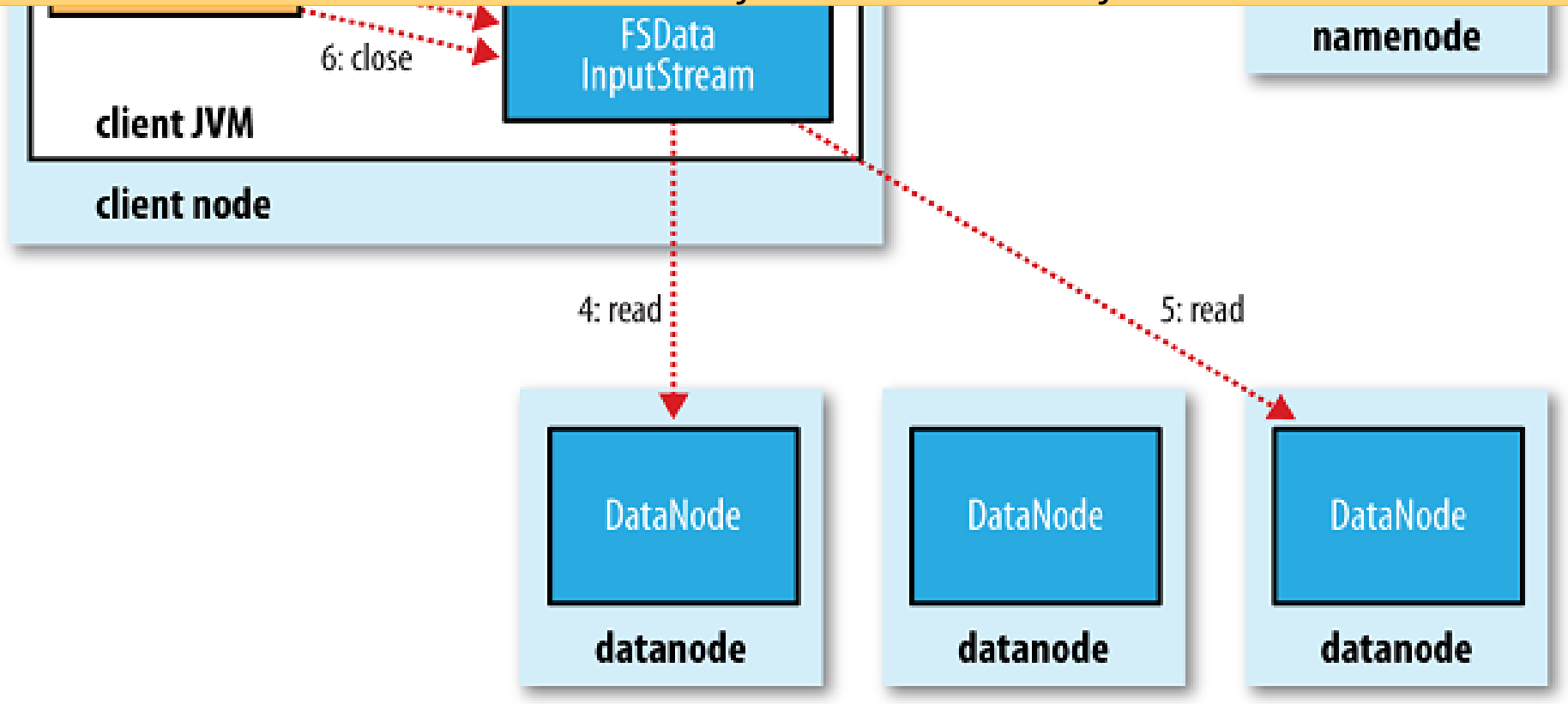
- The DistributedFileSystem returns an FSDataInputStream (an input stream that supports file seeks) to the client for it to read data from.
- FSDataInputStream in turn wraps a DFSInputStream, which manages the DataNode and NameNode I/O.

Read data from HDFS

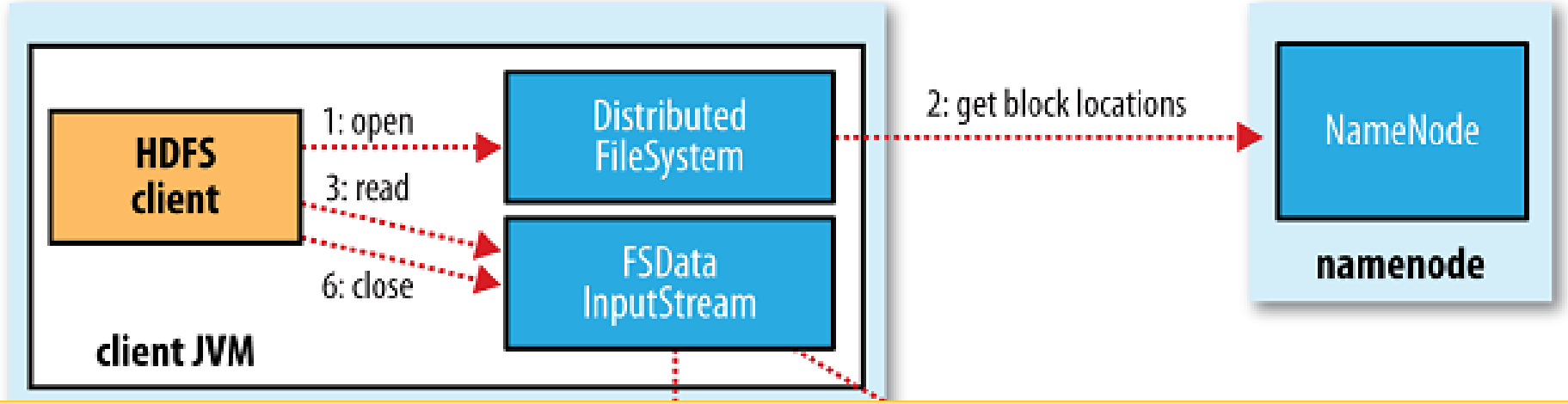


- Step 3: The client calls read() on the stream and DFSInputStream connects to the first (closest) DataNode for the first block in the file.
- Step 4: Data is streamed from the DataNode back to the client, which calls read() repeatedly on the stream.

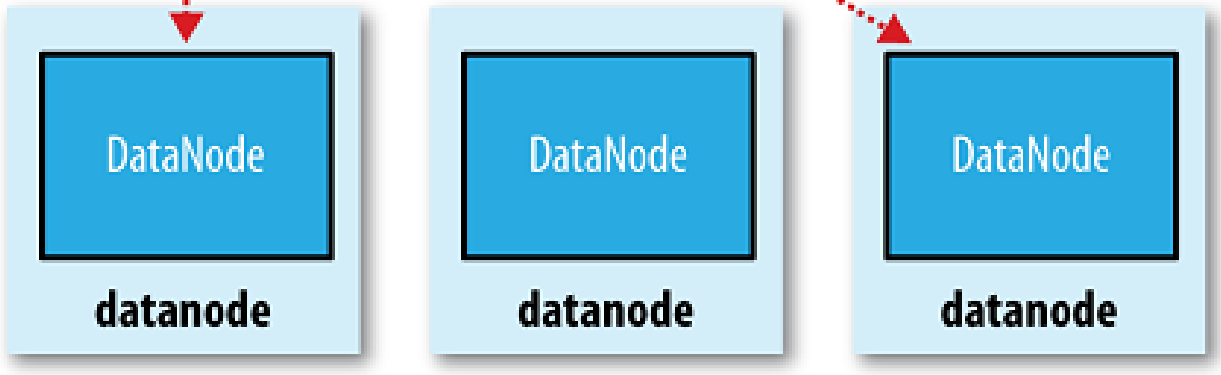
- Step 5: When the end of the block is reached, DFSInputStream will close the connection to the DataNode, then find the best DataNode for the next block.
  - *This happens transparently to the client, which from its point of view is just reading a continuous stream.*
  - *Blocks are read in order. The 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.*



Read data from HDFS



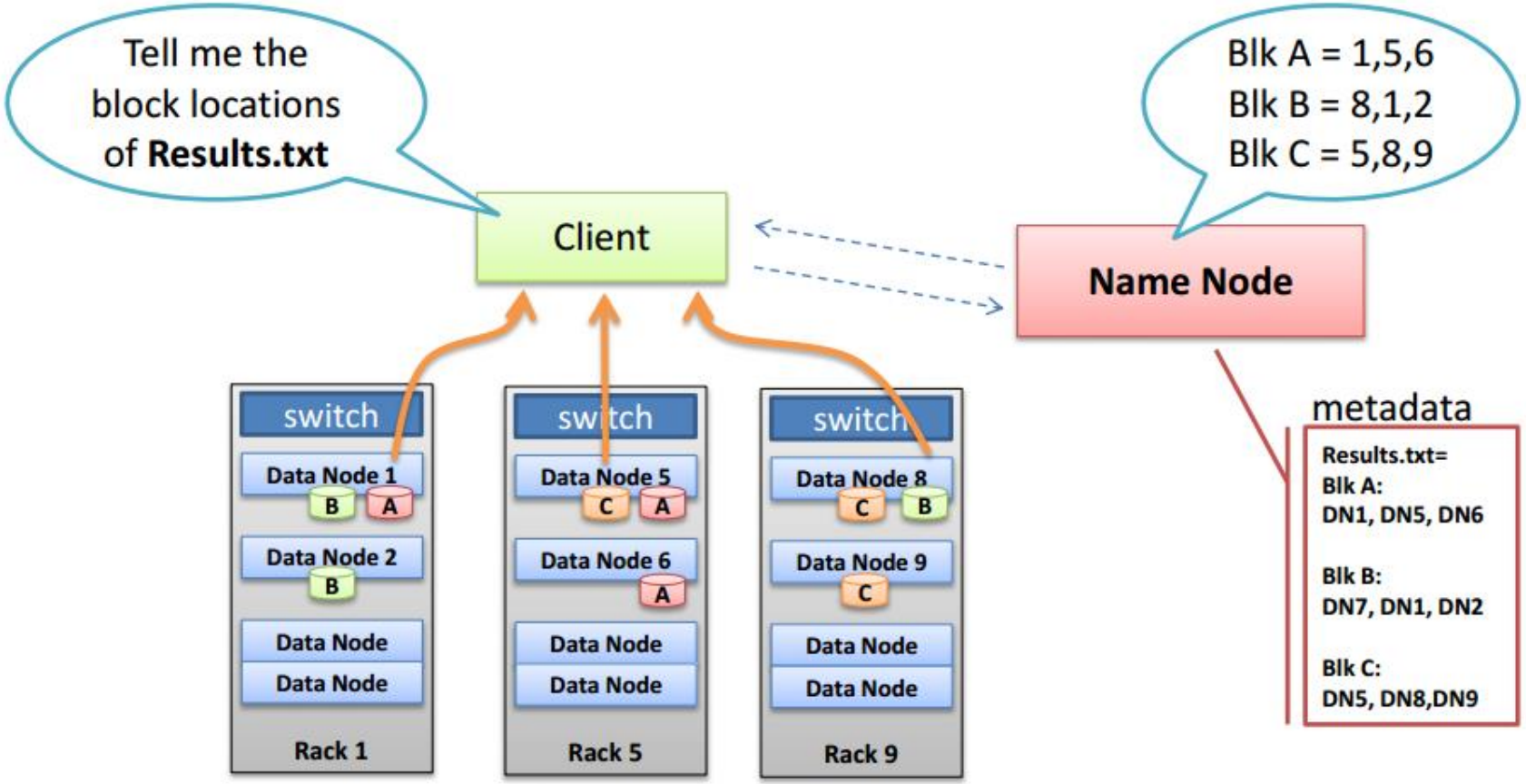
- Step 6: When the client has finished reading, it calls close() on the FSDataInputStream.



Read data from HDFS

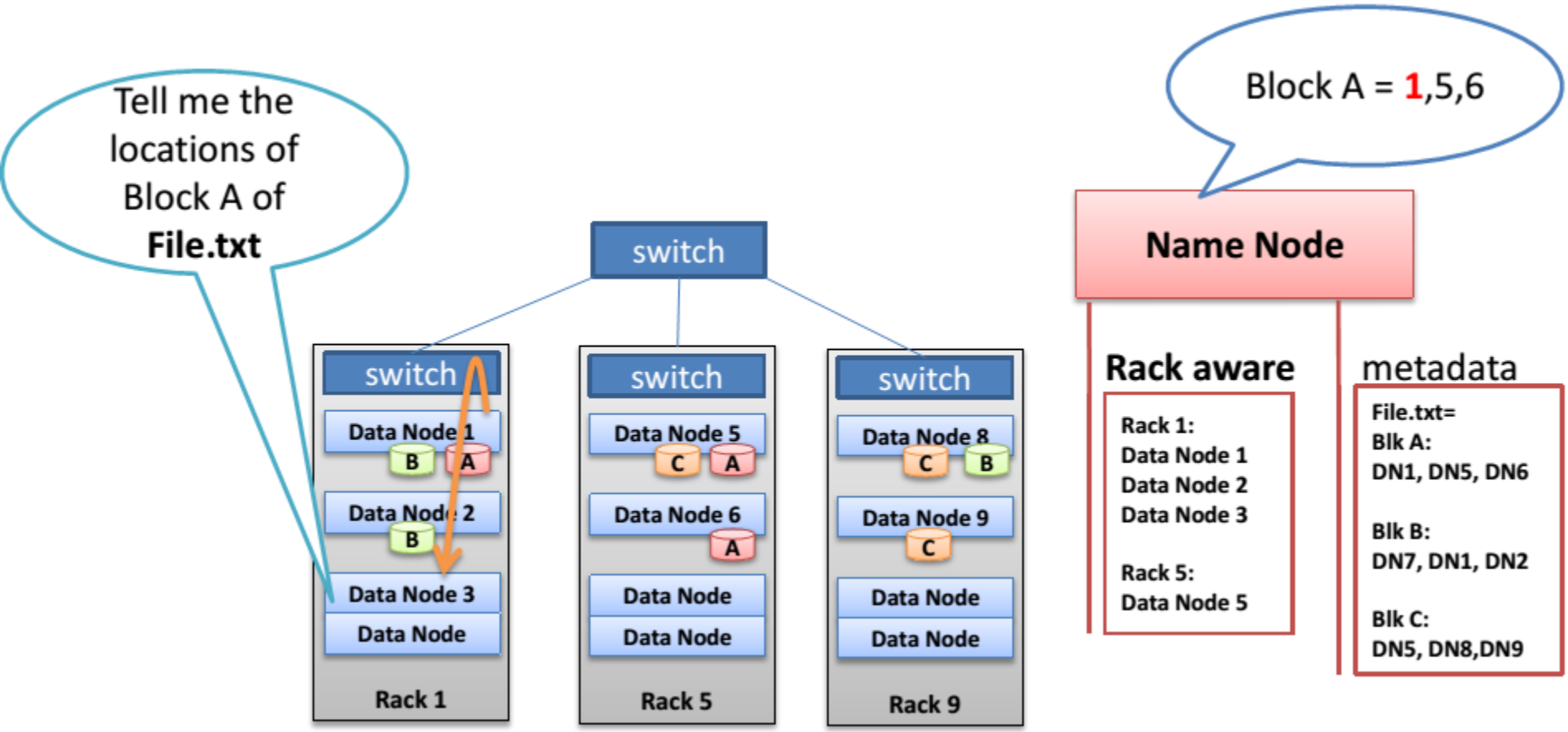
- The DFSInputStream will try the next closest DataNode for a block if it fails to communicate with current node.
  - The failed DataNodes is marked so that the DFSInputStream does not needlessly retry them for later blocks.
- It also verifies checksums for the data transferred from the DataNode.
  -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

Example: Read data from HDFS

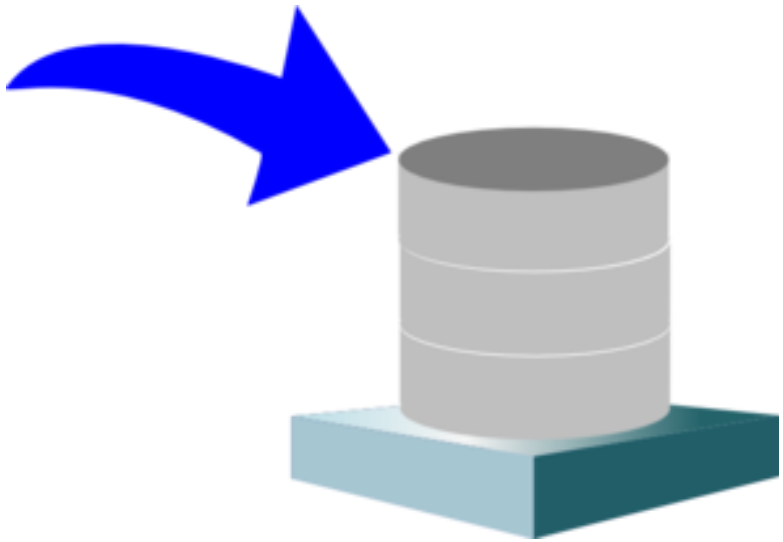


- Client receives DataNode list for each block and picks first Data Node for each block.
- Client reads blocks sequentially.

Example: Read data from HDFS



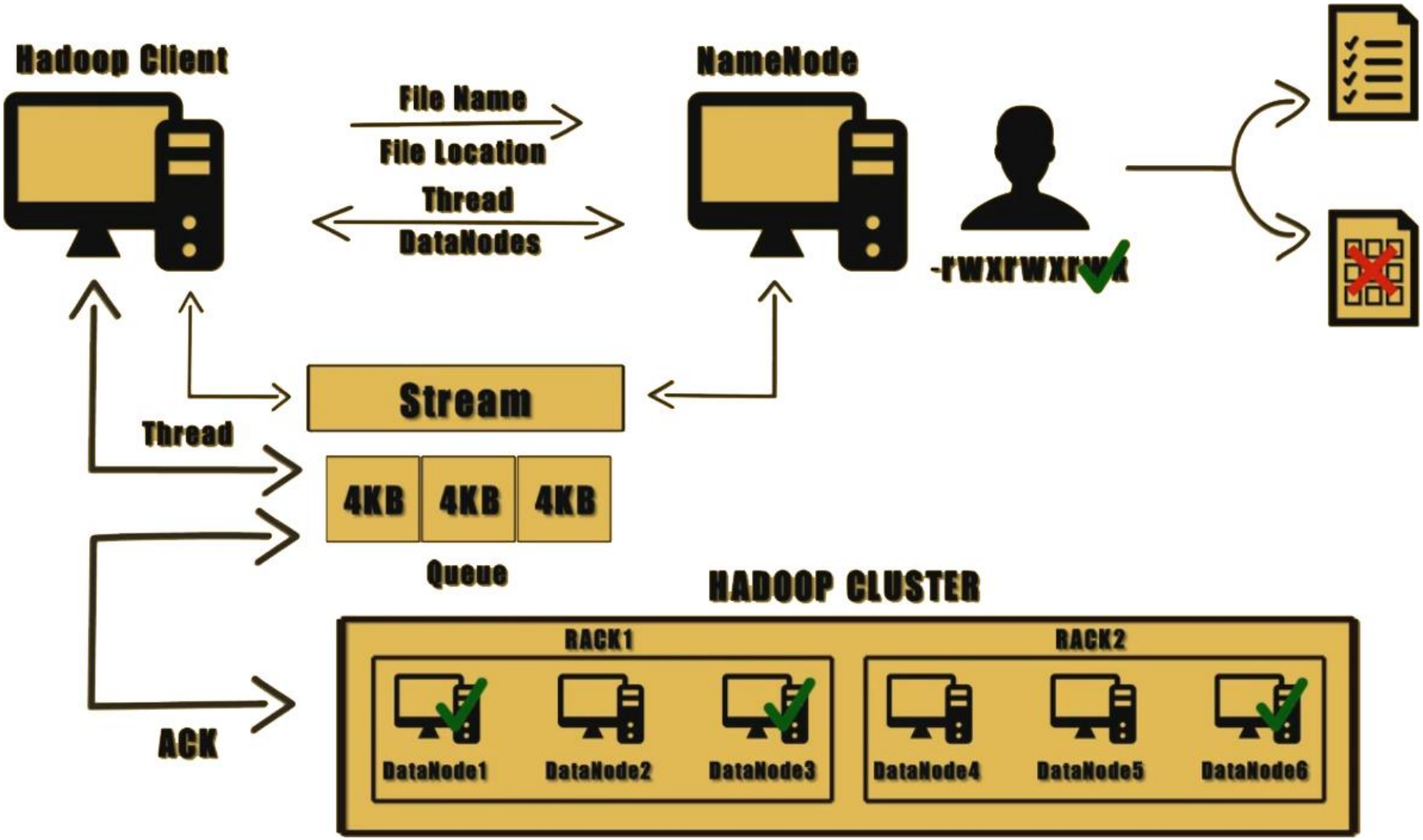
- Name Node provides rack-local nodes first.
- Leverage in-rack bandwidth, single hop.



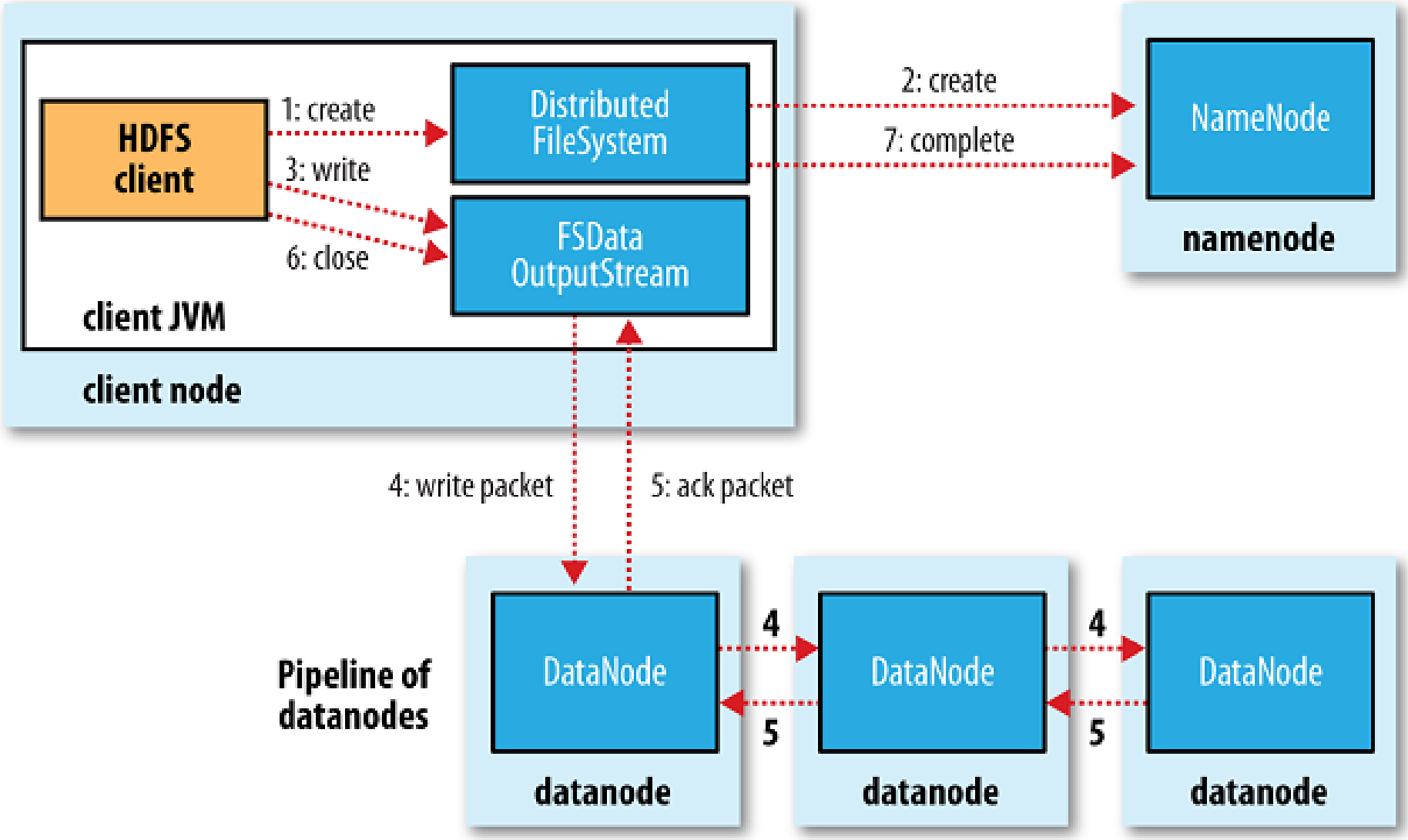
WRITE DATA TO HDFS

Anatomy of a file write

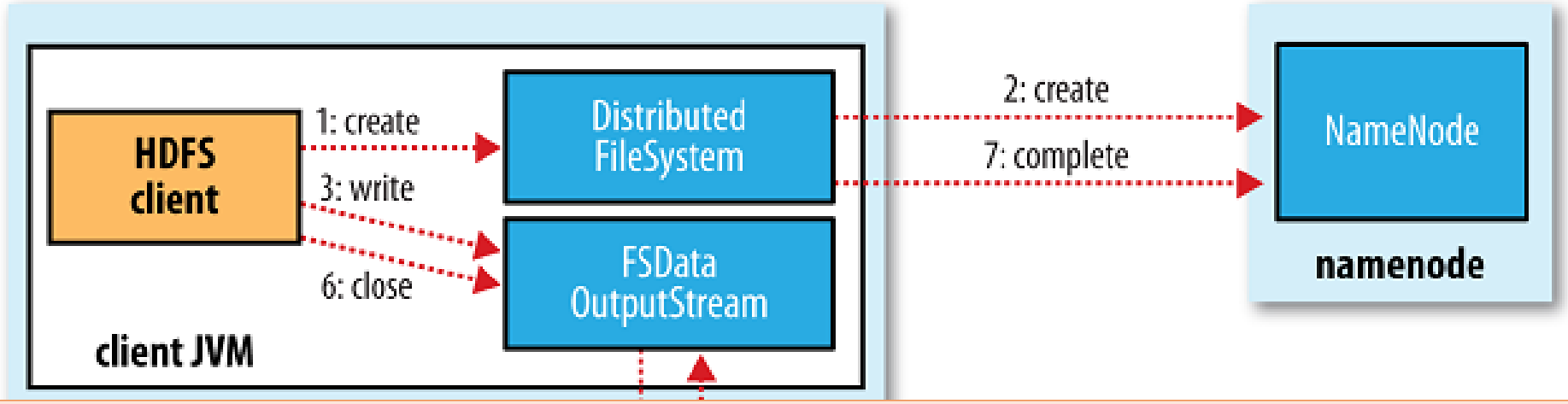
Write data to HDFS



Write data to HDFS

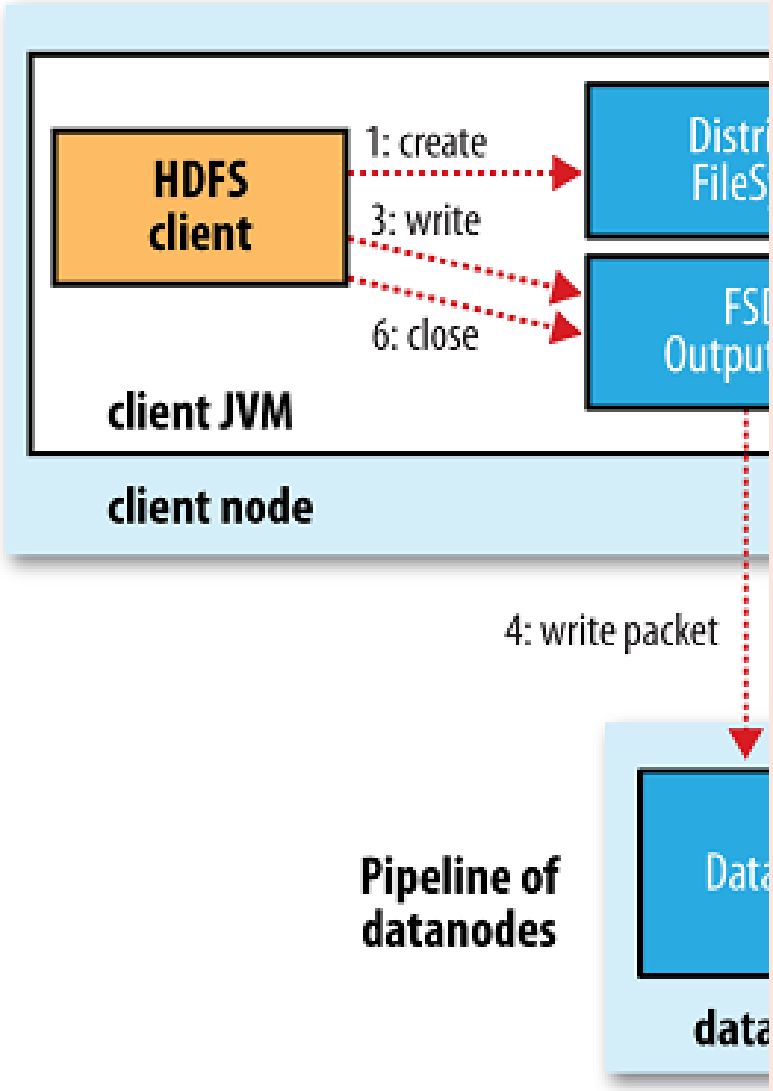


Write data to HDFS



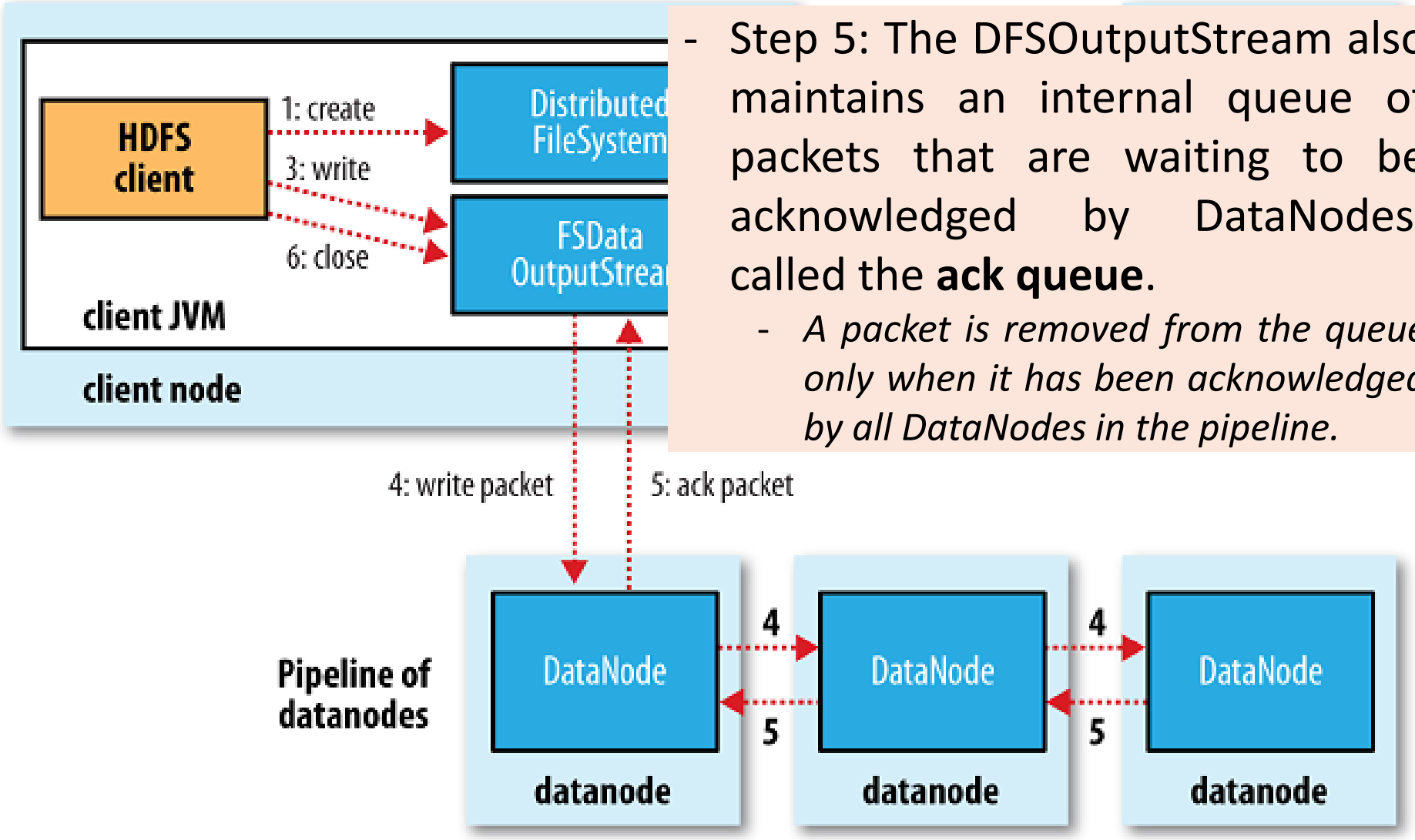
- Step 1: The client calls create() on DistributedFileSystem.
- Step 2: DistributedFileSystem calls the NameNode to create a new file in the filesystem namespace, with no blocks associated with it.
  - *The NameNode performs various checks to make sure the file does not already exist and the client has 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.*

Write data to HDFS



- 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.*

Write data to HDFS



- 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.*

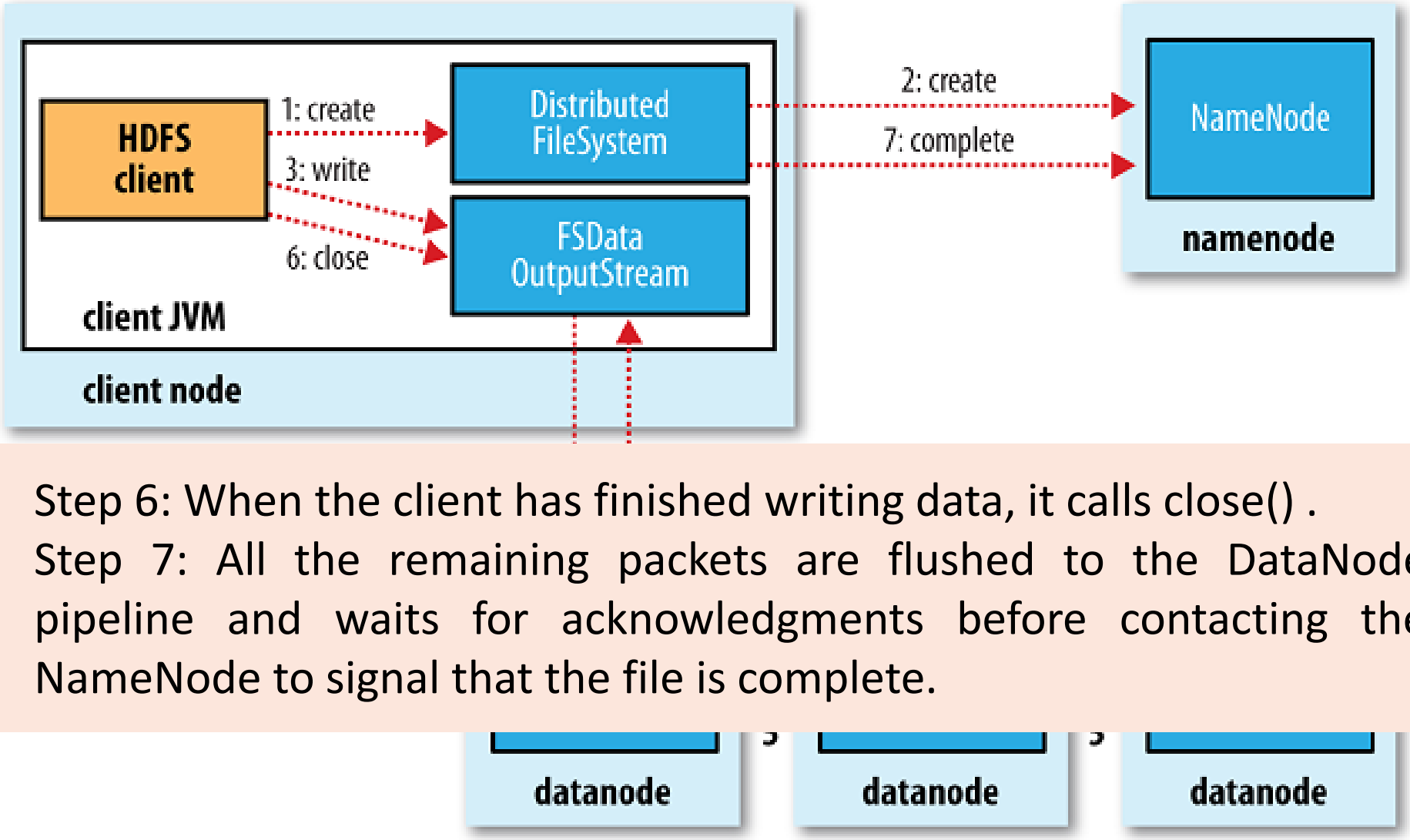
Write data to HDFS: Failure

- First, the pipeline is closed, and any packets in the ack queue are added to the front of the data queue so that DataNodes that are downstream from the failed node will not miss any packets.
- The current block on the good DataNodes is given a new identity, which is communicated to the NameNode, so that the partial block on the failed DataNode will be deleted if the failed DataNode recovers later on.
- The failed DataNode is removed from the pipeline, and a new pipeline is constructed from the two good DataNodes.
- The remainder of the block's data is written to the good DataNodes in the pipeline.

Write data to HDFS: Failure

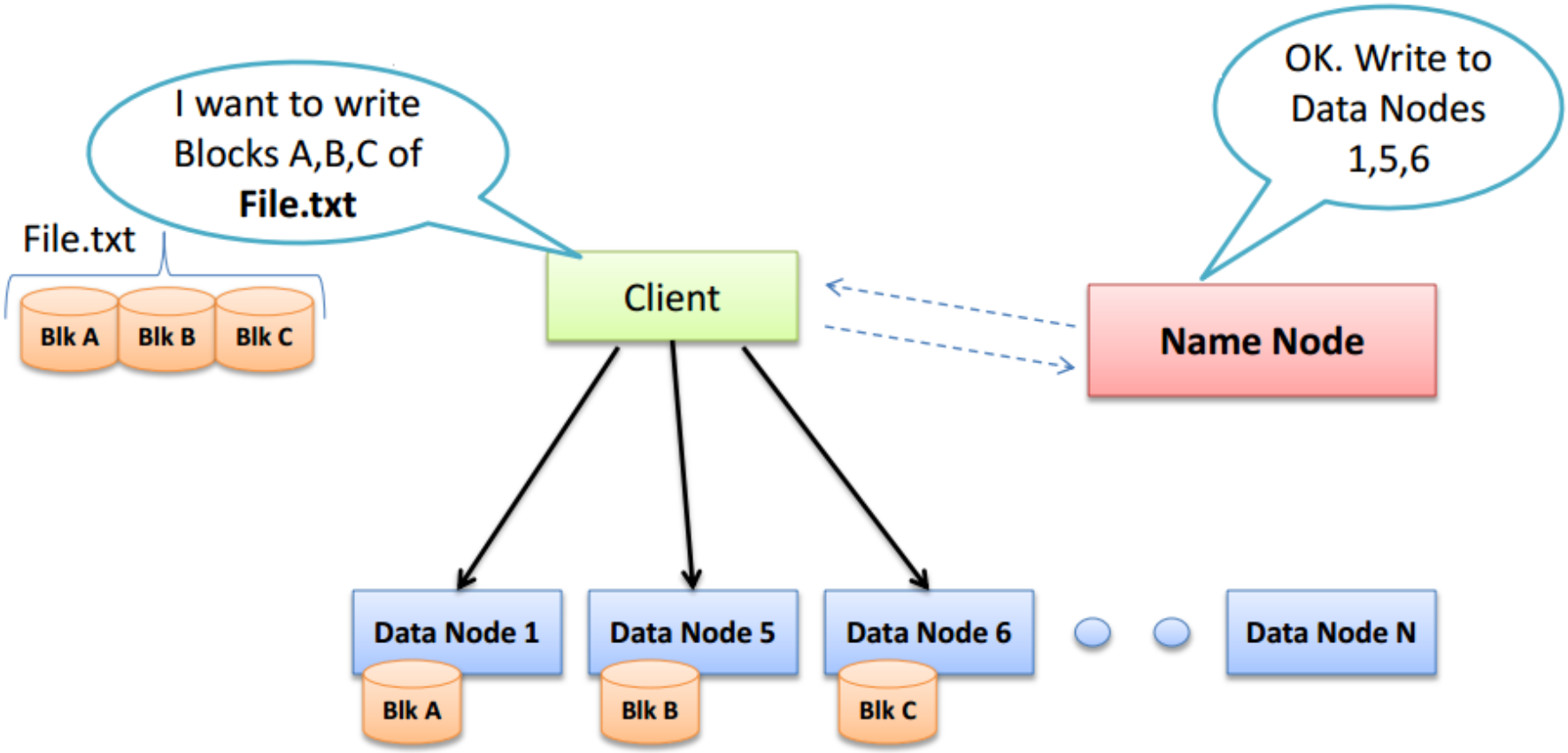
- The aforementioned actions are transparent to the client writing the data.
- The NameNode notices that the block is under-replicated, and it arranges for a further replica to be created on another node. Subsequent blocks are then treated as normal.

Write data to HDFS



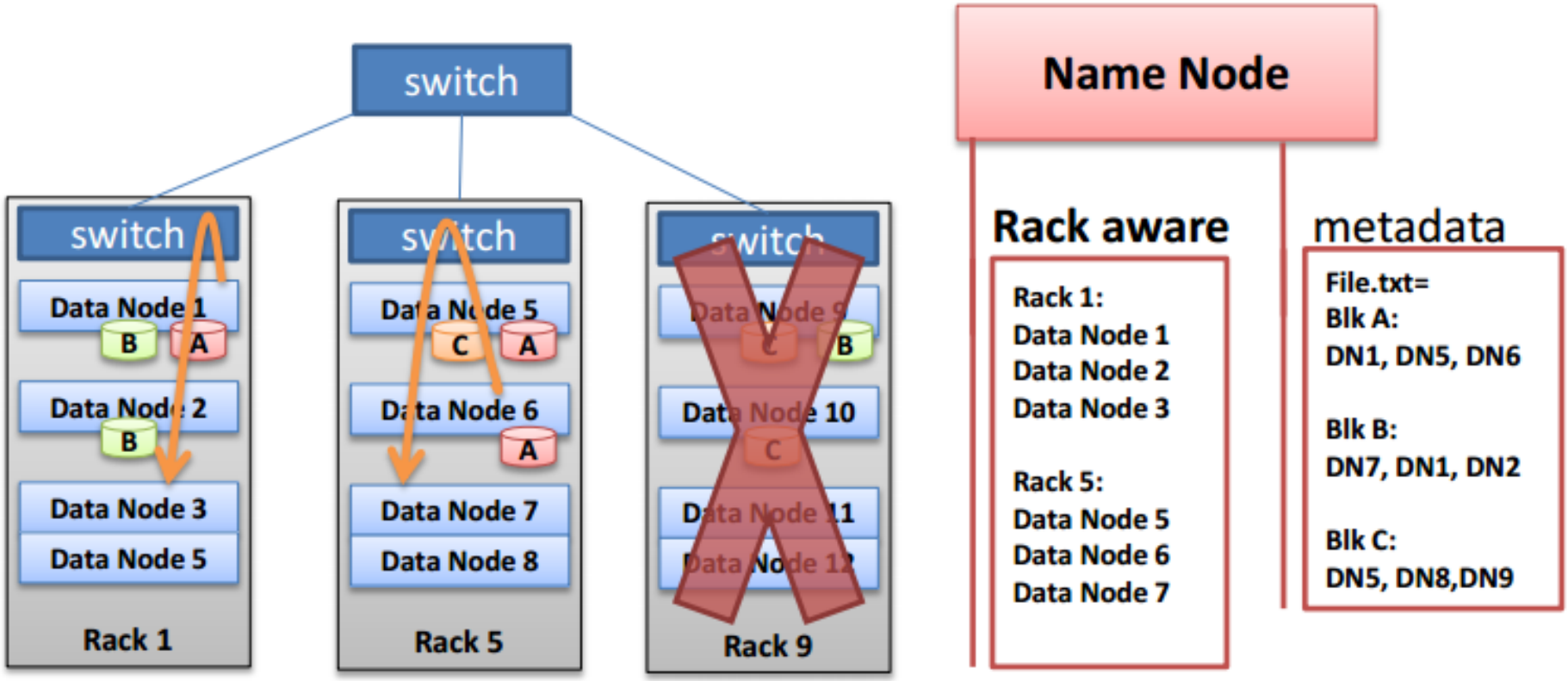
- Step 6: When the client has finished writing data, it calls close() .
- Step 7: All the remaining packets are flushed to the DataNode pipeline and waits for acknowledgments before contacting the NameNode to signal that the file is complete.

An example of writing data to HDFS



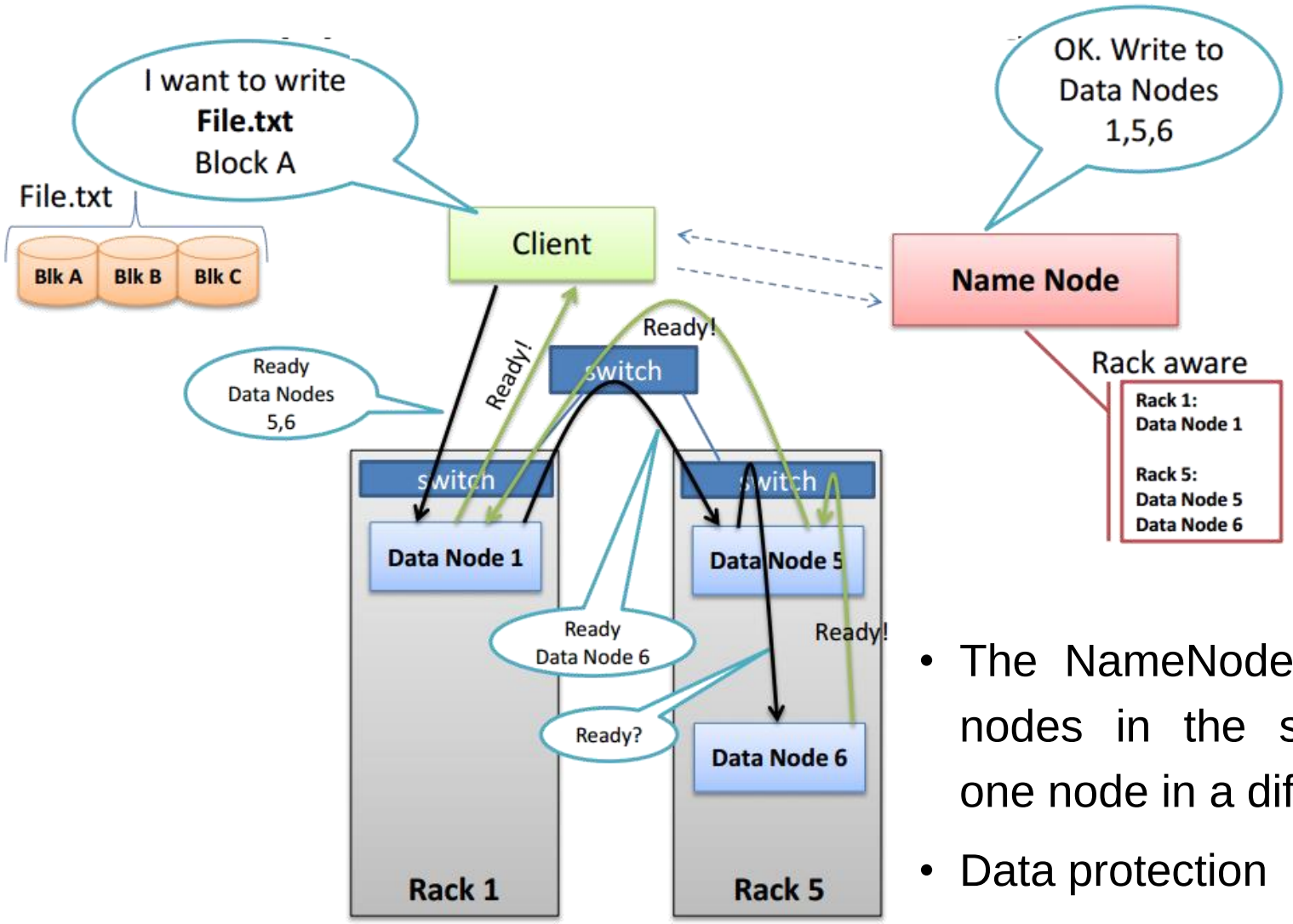
- Client consults NameNode and writes the block directly to one DataNode.
- DataNodes replicates block
- Cycle repeats for next block

Hadoop Rack Awareness



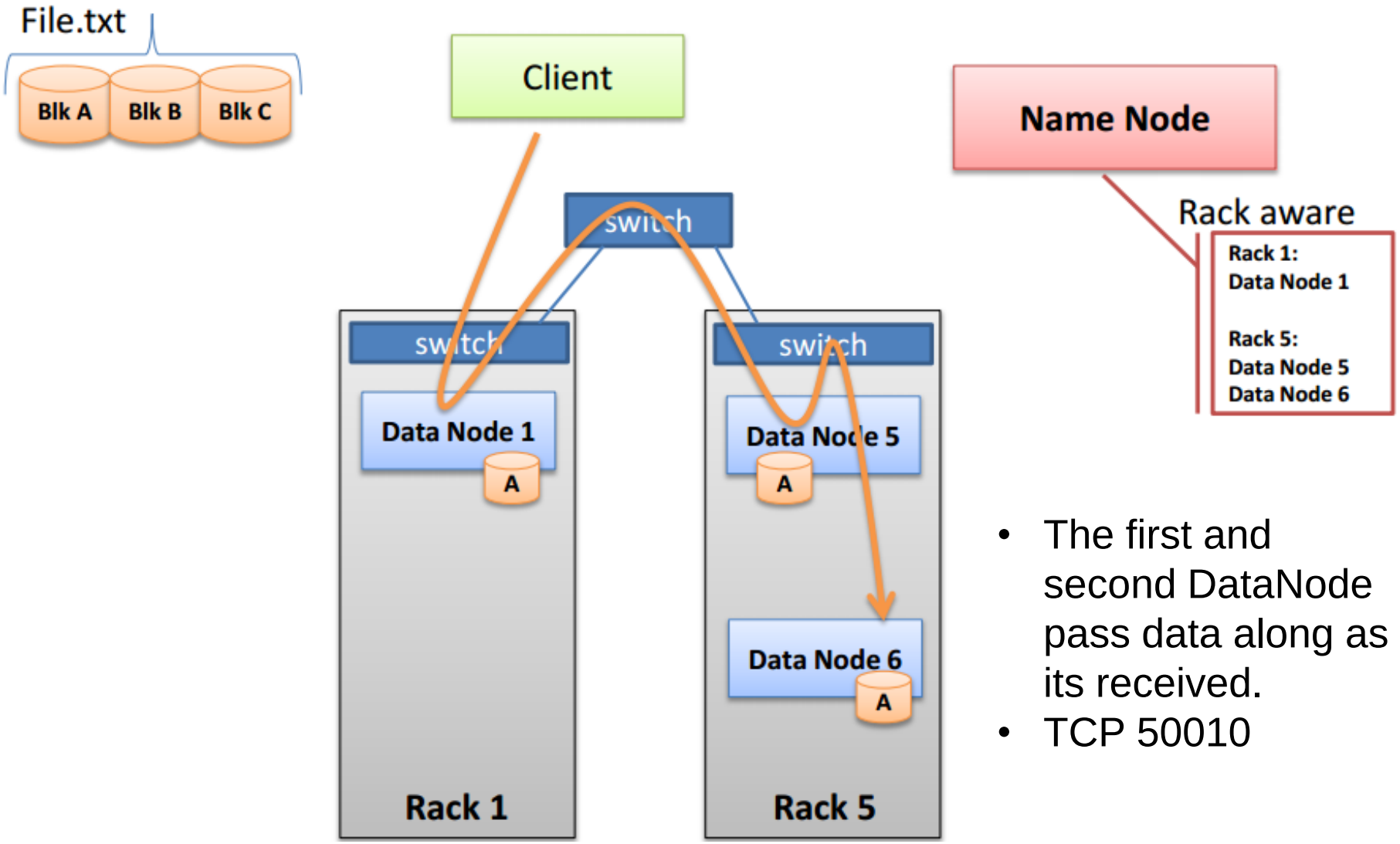
- Never loose all data if entire rack fails.
- Keep bulky flows in-rack when possible.
- Assumption that in-rack is higher bandwidth, lower latency.

HDFS Write Pipeline



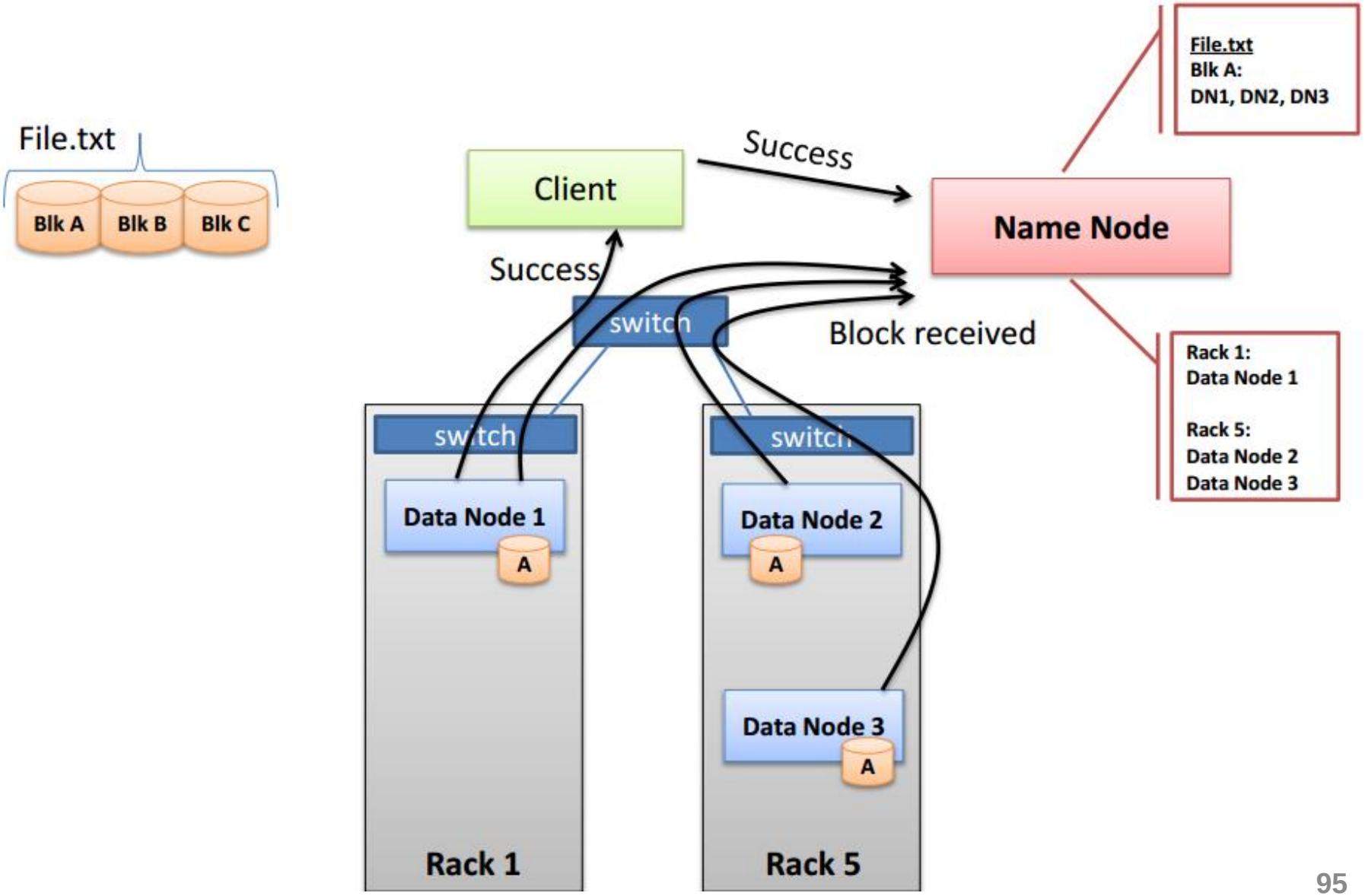
- The NameNode picks two nodes in the same rack, one node in a different rack.
- Data protection
- Locality for M/R

HDFS Write Pipeline

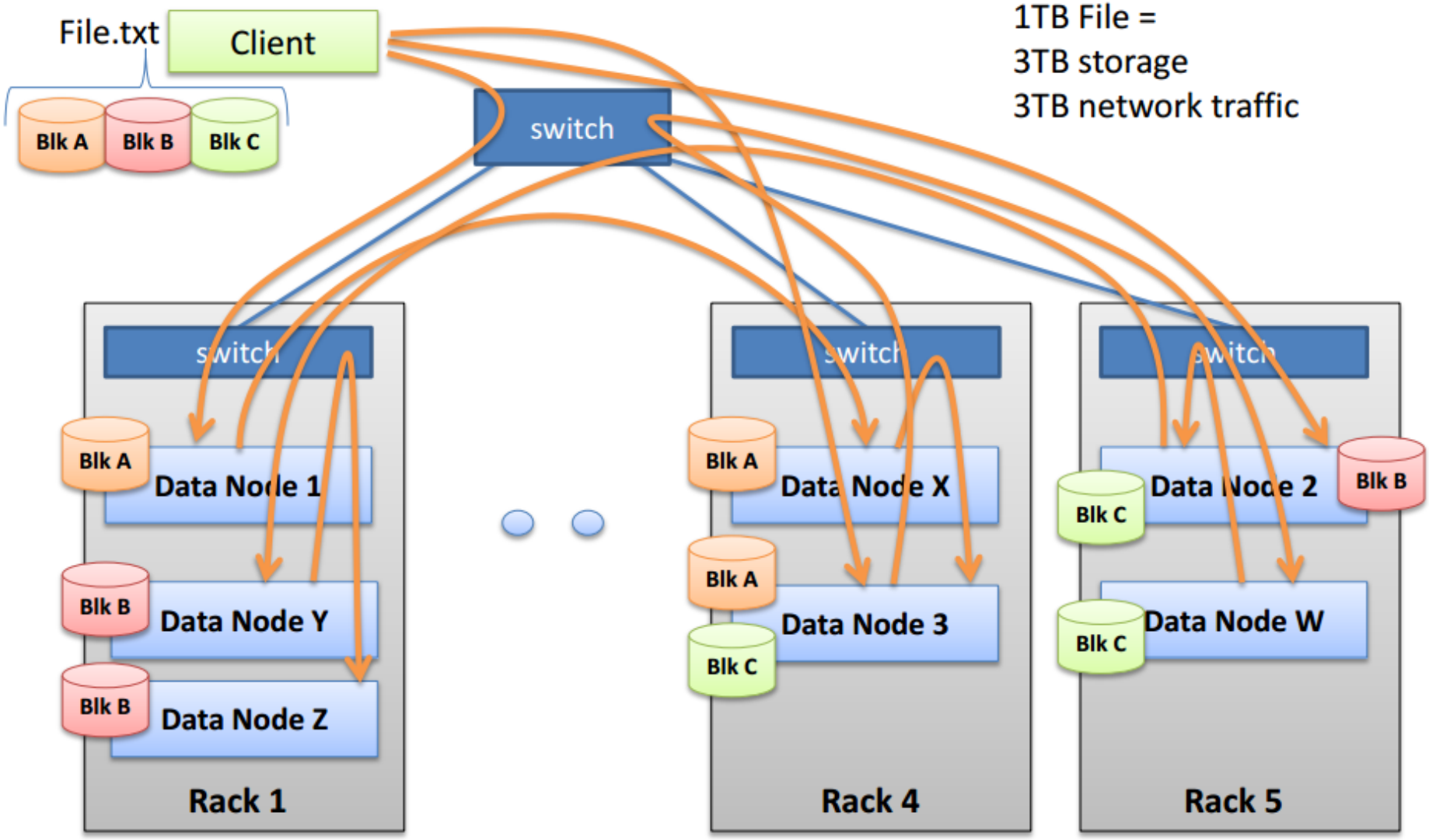


- The first and second DataNode pass data along as its received.
- TCP 50010

HDFS Write Pipeline



HDFS Write Pipeline



THE END