HDFS
Hadoop Distributed File System

Agenda

- ✓ Introduction to HDFS
- ✓ Assumptions and Goals
  - ➢ Hardware Failure
  - ➢ Streaming Data Access
  - ➢ Large Data Sets
  - ➢ Simple Coherency Model
  - ➢ “Moving Computation is Cheaper than Moving Data”
  - ➢ Portability Across Heterogeneous Hardware and Software Platforms
- ✓ HDFS Architecture

Agenda

- ✓ NameNode
  - ➢ Overview
  - ➢ Responsibility
  - ➢ The File System Namespace
  - ➢ fsImage and editLogs
  - ➢ NameNode Checkpointing
  - ➢ NameNode Recovery
- ✓ HDFS - saveNameSpace
  - ➢ Functionality
  - ➢ Execution
- ✓ HDFS – Safe Mode
  - ➢ Functionality
  - ➢ Monitoring
- ✓ Secondary NameNode
  - ➢ Overview
  - ➢ Responsibility
  - ➢ ‘NOT’ a backup of NameNode

Agenda

✓ DataNodes

- ➢ Overview
- ➢ Responsibility
- ➢ Data storage

✓ Data Replication

- ➢ Replica Placement: The First Baby Steps
- ➢ Replica Selection
- ➢ Safemode

✓ HDFS - Balancer

- ➢ Feature
- ➢ Under replication
- ➢ Over replication

✓ The Persistence of File System Metadata

- ➢ Overview
- ➢ Why and How

Agenda

✓ Data Flow

- ➢ Anatomy of a File Read
- ➢ Anatomy of a File Write

✓ Robustness

- ➢ Data Disk Failure, Heartbeats and Re-Replication
- ➢ Data Integrity
- ➢ Metadata Disk Failure
- ➢ Snapshots

✓ Accessibility

- ➢ FS Shell
- ➢ DFSAdmin
- ➢ Browser Interface

✓ Space Reclamation

- ➢ File Deletes and Undeletes
- ➢ Decrease Replication Factor

Agenda

✓ Hadoop Cluster Configuration + Demo

- ➢ Core-site.xml
- ➢ Hdfs-site.xml
- ➢ Masters
- ➢ Slaves
- ➢ Mapred-site.xml
- ➢ Hadoop-env.sh

✓ Data Loading Techniques

- ➢ HDFS Commands - Demo
- ➢ Sqoop – Demo
- ➢ Flume

Introduction To HDFS

✓ Overview

- ➢ HDFS is a file system written in Java
  - Based on Google's GFS
- ➢ Sits on top of a native file system
  - Such as ext3, ext4 or xfs
- ➢ Provides redundant storage for massive amounts of data
  - Using readily-available, industry-standard computers
- ➢ HDFS performs best with a number of large files
  - Each file typically 100MB or more
- ➢ Files in HDFS are 'write once'
  - No random writes to files are allowed
- ➢ HDFS is optimized for large, streaming reads of files
  - Rather than random reads

Introduction To HDFS

✓ Overview

- ➢ Although files are split into 64MB or 128MB blocks, if a file is smaller than this the full 64MB/128MB will not be used
- ➢ Blocks are stored as standard files on the DataNodes, in a set of directories specified in Hadoop's configuration files
  - This will be set by the system administrator
- ➢ Without the metadata on the NameNode, there is no way to access the files in the HDFS cluster
- ➢ When a client application wants to read a file:
  - It communicates with the NameNode to determine which blocks make up the file, and which DataNodes those blocks reside on
  - It then communicates directly with the DataNodes to read the data
  - The NameNode will not be a bottleneck

Assumptions and Goals

✓ Hardware Failure

- ➢ Hardware failure is the norm rather than the exception.
- ➢ An HDFS instance may consist of hundreds or thousands of server machines, each storing part of the file system’s data.
- ➢ The fact that there are a huge number of components and that each component has a non-trivial probability of failure means that some component of HDFS is always non-functional.
- ➢ Therefore, detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.

✓ Streaming Data Access

- ➢ HDFS build around the idea of having most efficient pattern of Distriubute processing that is to have write once and read multiple times pattern.
- ➢ HDSF is designed to have Higher throughput(how many records can be read in a given time) , it does not care much on latency(time to read one record).
- ➢ Why Block in HDFS is so large 64MB or 128MB?
  - The reason is to minimize the cost of seeks.A seek time is to actually locate the records in a block in a disk. which means which loaction or sector that block resides in the disk.Once the block is more size , it will be a contiguous records and disk arm or computer will be able to read it faster.If block size is small , seek time for the first record in a block will be same but after then you dont need to do seeks multiple time and you can read more amount of data faster.

Assumptions and Goals

✓ Large Data Sets

- ➢ Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size. Thus, HDFS is tuned to support large files. It should provide high aggregate data bandwidth and scale to hundreds of nodes in a single cluster. It should support tens of millions of files in a single instance.

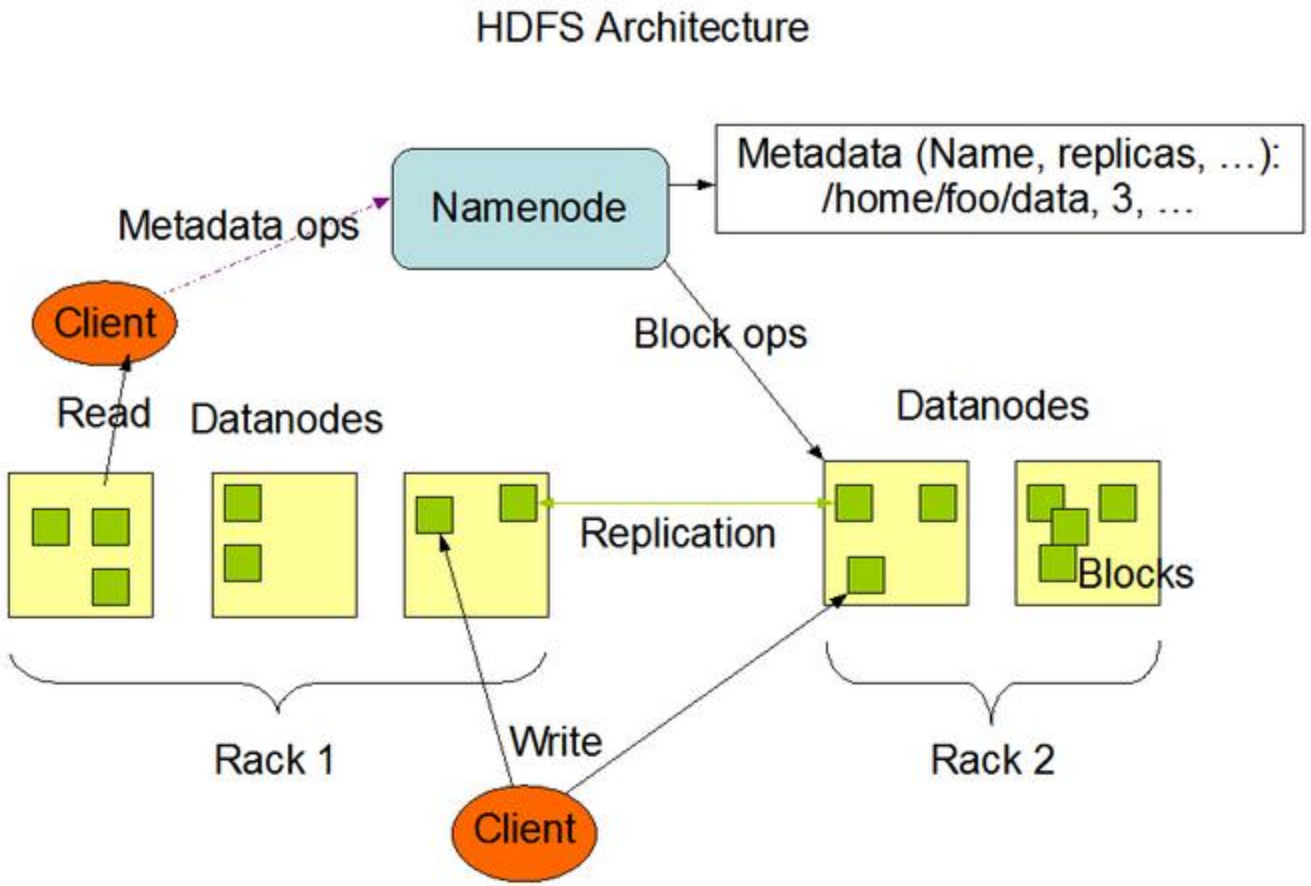
✓ "Moving Computation is Cheaper than Moving Data"

- ➢ A computation requested by an application is much more efficient if it is executed near the data it operates on. This is especially true when the size of the data set is huge. This minimizes network congestion and increases the overall throughput of the system.
- ➢ HDFS provides interfaces for applications to move themselves closer to where the data is located.

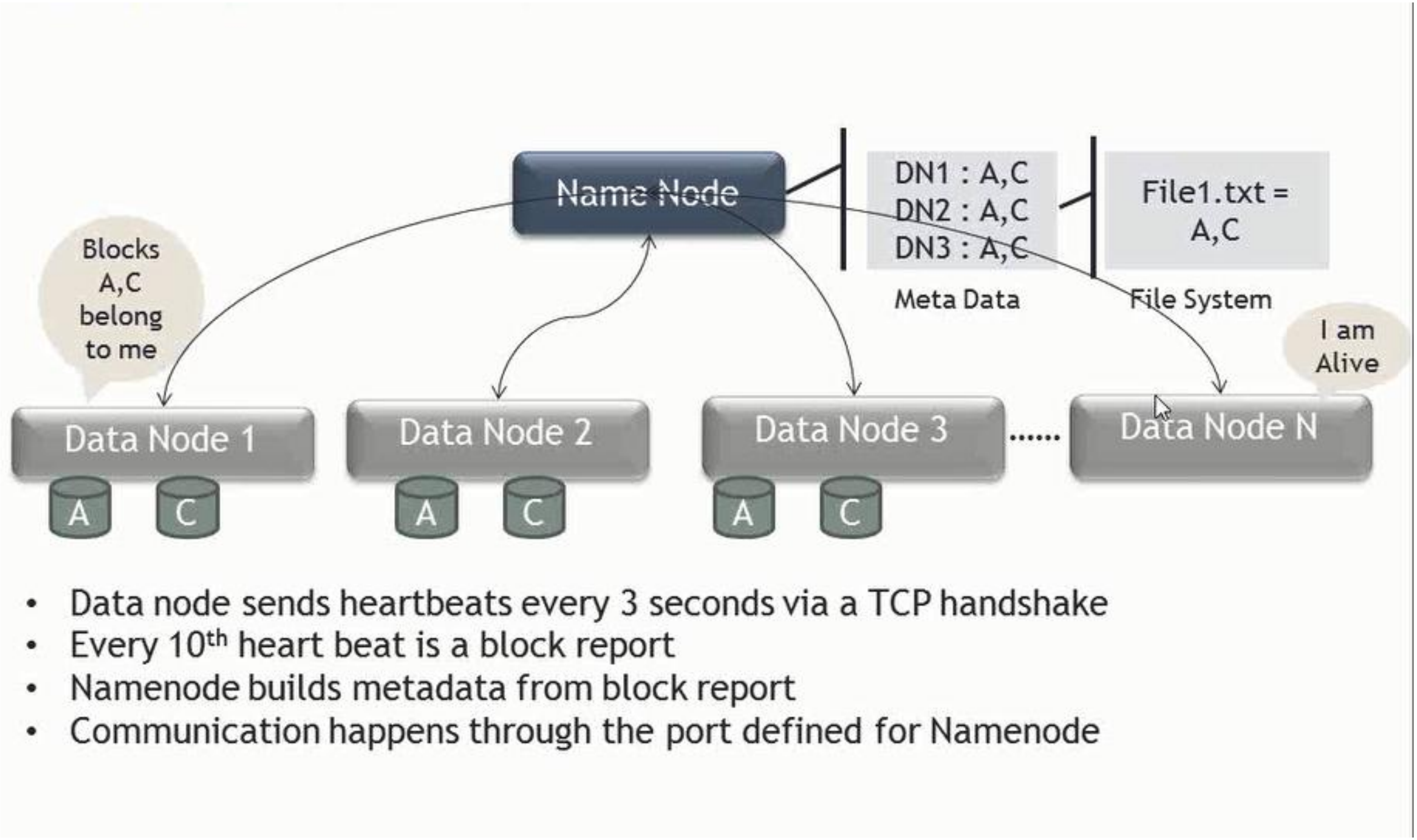
✓ Portability Across Heterogeneous Hardware and Software Platforms

- ➢ HDFS has been designed to be easily portable from one platform to another. This facilitates widespread adoption of HDFS as a platform of choice for a large set of applications.

HDFS Architecture



Let us Zoom into HDFS



- Data node sends heartbeats every 3 seconds via a TCP handshake
- Every 10th heart beat is a block report
- Namenode builds metadata from block report
- Communication happens through the port defined for Namenode

NameNode

✓ Functionalities

➢ what gets stored in main memory and what gets stored in secondary memory (hard disk).

- File names, Permissions, The file to block mapping, locations of blocks on data nodes, active data nodes, a bunch of other metadata is all stored in memory on the NameNode.
- The thing stored on disk is the fsimage, edit log, and status logs. It's interesting to note that the NameNode never really uses these files on disk, except for when it starts. The fsimage and edits file pretty much only exist to be able to bring the NameNode back up if it needs to be stopped or it crashes

➢ What we mean by "file to block mapping" ?

- When a file is put into HDFS, it is split into blocks (of configurable size). Let's say you have a file called "file.txt" that is 201MB and your block size is 64MB. You will end up with three 64MB blocks and a 9MB block (64+64+64+9 = 201). The NameNode keeps track of the fact that "file.txt" in HDFS maps to these four blocks. DataNodes store blocks, not files, so the mapping is important to understanding where your data is and what your data is.

➢ What exactly is fsimage and edit logs ?

- A recent checkpoint of the memory of the NameNode is stored in the fsimage. The NameNode's state (i.e. file->block mapping, file properties, etc.) from that checkpoint can be restored from this file.
- The edits file are all the new updates from the fsimage since the last checkpoint. These are things like a file being deleted or added. This is important for if your NameNode goes down, as it has the most recent changes since the last checkpoint stored in fsimage. The way the NameNode comes up is it materializes the fsimage into memory, and then applies the edits in the order it sees them in the edits file.
- fsimage and edits exist the way they do because editing the potentially massive fsimage file every time a HDFS operation is done can be hard on the system. Instead, the edits file is simply appended to.

HDFS- saveNameSpace

✓ Functionalities

- ➢ NameNode writes current in-memory image file to new fsimage file and reset the edit file. Its done periodically using following properties.
  - Fs.checkpoint.size (default 64MB) – when editLog reaches this size.
  - Fs.checkpoint.period

- ➢ Can also be forced by administrator using following command.
  - Hadoop dfsadmin -saveNameSpace

HDFS- Safe Mode

✓ Functionalities

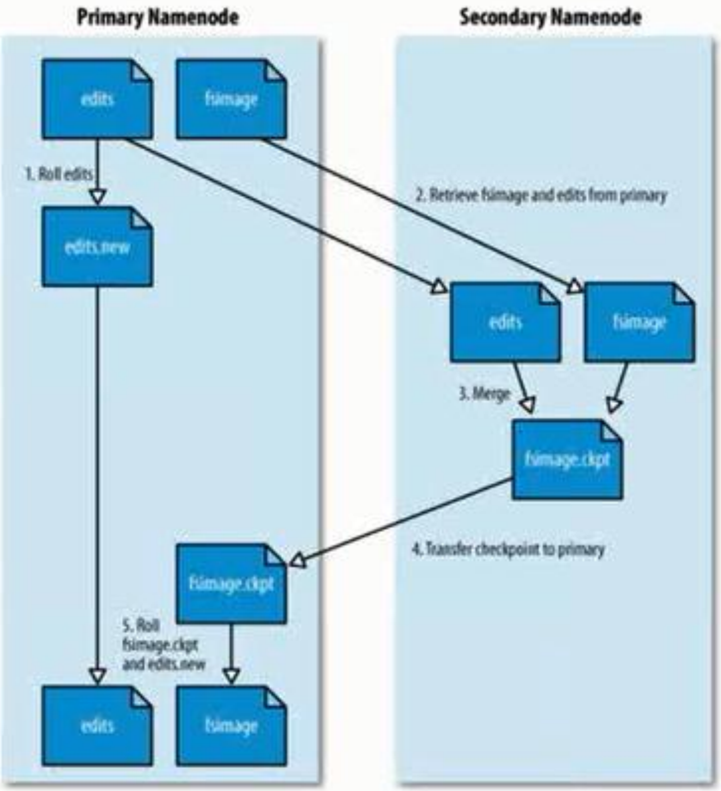
- ➢ NameNode enters safe mode while its starting up.
  - It does not listen to requests as long as it is in safe mode.
  - It loads the fsImage file into memory and apply latest changes from editLog file.
- ➢ Check whether NameNode is in safe mode.
  - Hadoop dfsadmin –safemode get
- ➢ Wait till safe mode is off
  - Hadoop dfsadmin –safemode wait
- ➢ Enter or leave safe mode.
  - Hadoop dfsadmin –safemode enter/leave

Secondary NameNode

✓ Functionalities

- ➢ The SecondaryNameNode is the process that periodically takes the fsimage and edits file and merges them together, into a new checkpointed fsimage file.This is an important process to prevent edits from getting huge.
- ➢ It is also possible to run a secondary namenode, which despite its name does not act as a namenode.
- ➢ Its main role is to periodically merge the fsimage with the edit log to prevent the edit log from becoming too large.
- ➢ The secondary namenode usually runs on a separate physical machine, since it requires plenty of CPU and as much memory as the namenode to perform the merge. It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing.

How Checkpoint Process Work



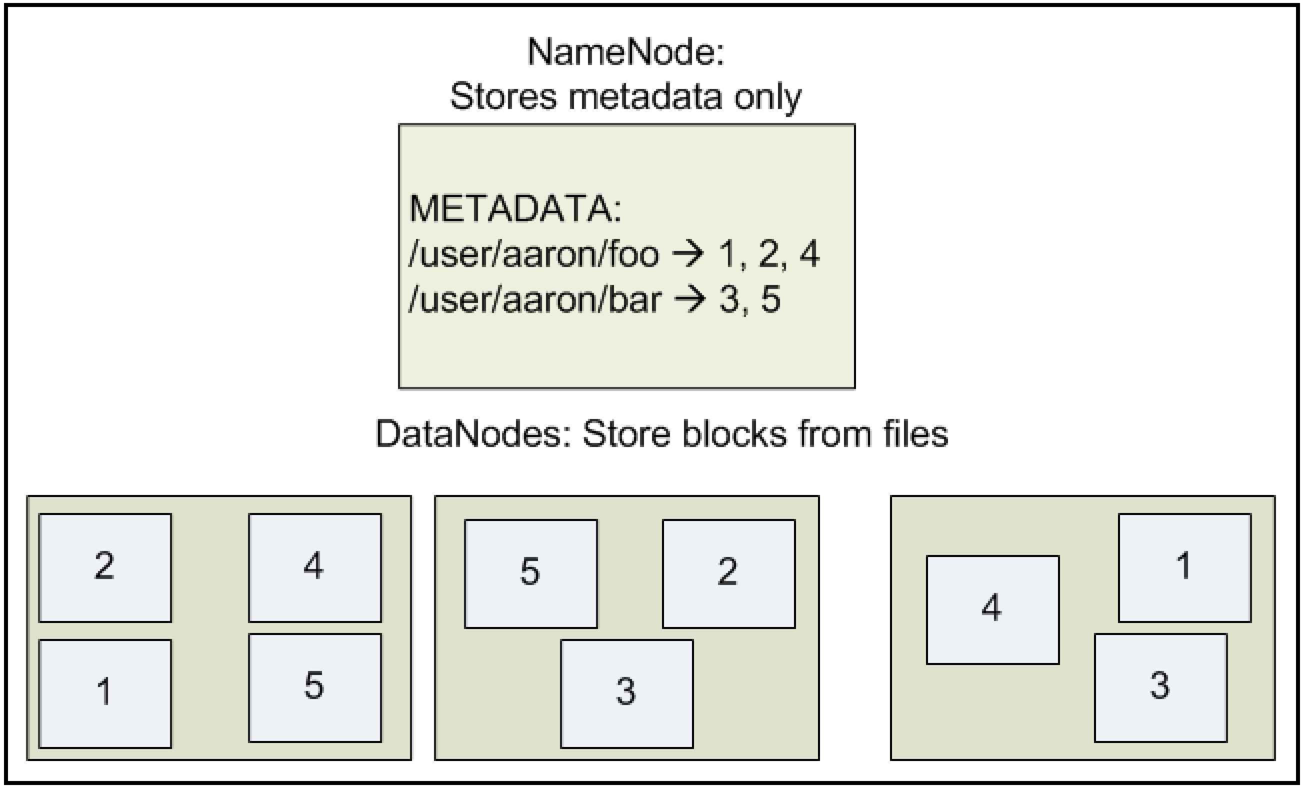
1. The secondary asks the primary to roll its *edits* file, so new edits go to a new file.
2. The secondary retrieves *fsimage* and *edits* from the primary (using HTTP GET).
3. The secondary loads *fsimage* into memory, applies each operation from *edits*, then
creates a new consolidated *fsimage* file.
4. The secondary sends the new *fsimage* back to the primary (using HTTP POST).
5. The primary replaces the old *fsimage* with the new one from the secondary, and the old *edits* file with the new one it started in step 1. It also updates the *fstime* file
to record the time that the checkpoint was taken.

DataNode

✓ Functionalities

- ➢ Datanodes are the workhorses of the filesystem. They store and retrieve blocks when they are told to (by clients or the namenode).
- ➢ They report back to the namenode periodically with lists of blocks that they are storing
- ➢ DataNodes periodically sends a Heartbeat and a Blockreport to the NameNode in the cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly. A Blockreport contains a list of all blocks on a DataNode.

NameNode and DataNodes



NameNode:
Stores metadata only

METADATA:
/user/aaron/foo → 1, 2, 4
/user/aaron/bar → 3, 5

DataNodes: Store blocks from files

Data Replication

✓ Replica Placement: The First Baby Steps

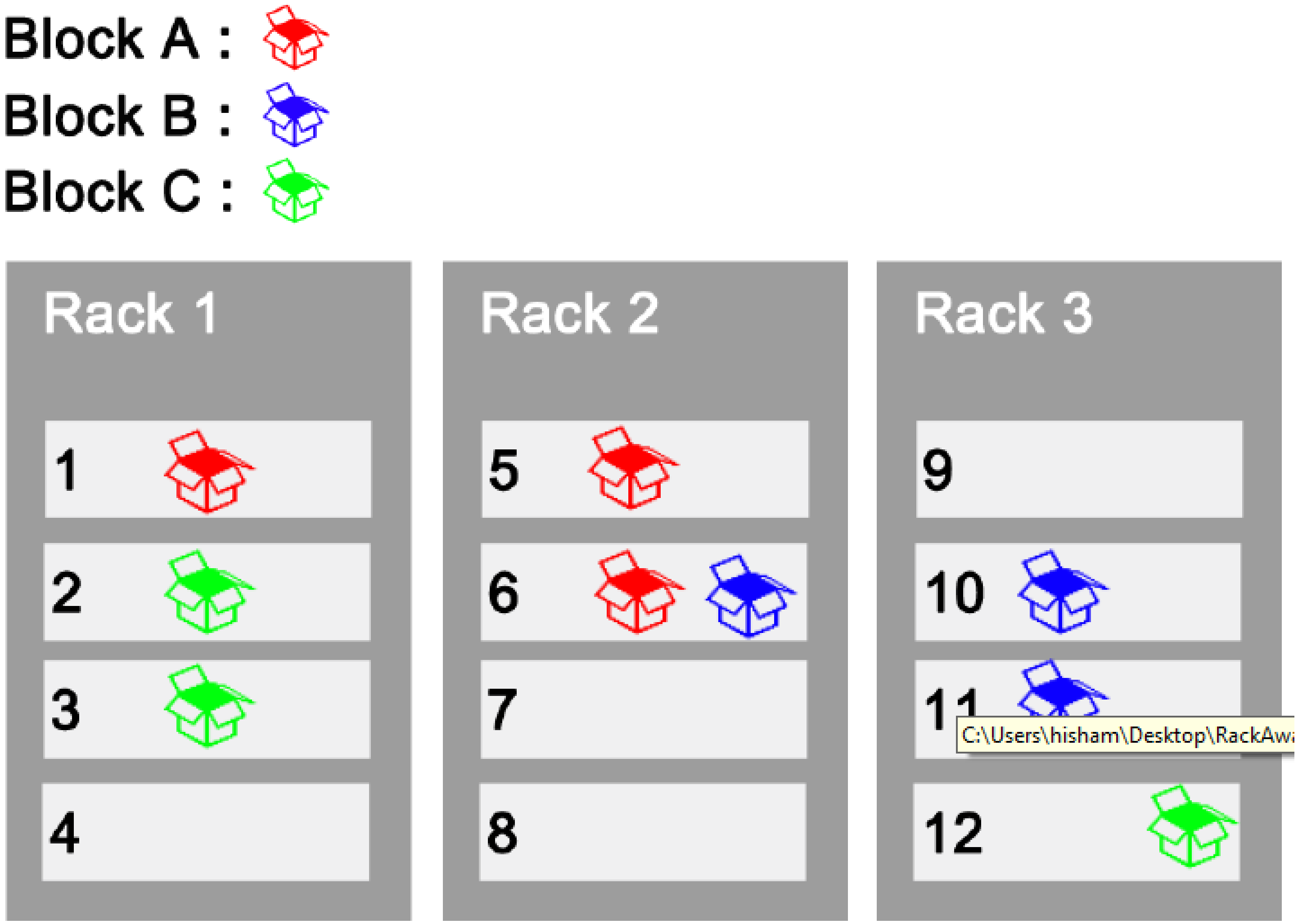
- ➢ The placement of replicas is critical to HDFS reliability and performance. Optimizing replica placement distinguishes HDFS from most other distributed file systems. This is a feature that needs lots of tuning and experience. The purpose of a rack-aware replica placement policy is to improve data reliability, availability, and network bandwidth utilization. The current implementation for the replica placement policy is a first effort in this direction. The short-term goals of implementing this policy are to validate it on production systems, learn more about its behavior, and build a foundation to test and research more sophisticated policies.

- ➢ Large HDFS instances run on a cluster of computers that commonly spread across many racks. Communication between two nodes in different racks has to go through switches. In most cases, network bandwidth between machines in the same rack is greater than network bandwidth between machines in different racks.

- ➢ The NameNode determines the rack id each DataNode belongs to via the process outlined in Hadoop Rack Awareness. A simple but non-optimal policy is to place replicas on unique racks. This prevents losing data when an entire rack fails and allows use of bandwidth from multiple racks when reading data. This policy evenly distributes replicas in the cluster which makes it easy to balance load on component failure. However, this policy increases the cost of writes because a write needs to transfer blocks to multiple racks.

- ➢ For the common case, when the replication factor is three, HDFS's placement policy is to put one replica on one node in the local rack, another on a node in a different (remote) rack, and the last on a different node in the same remote rack. This policy cuts the inter-rack write traffic which generally improves write performance. The chance of rack failure is far less than that of node failure; this policy does not impact data reliability and availability guarantees. However, it does reduce the aggregate network bandwidth used when reading data since a block is placed in only two unique racks rather than three. With this policy, the replicas of a file do not evenly distribute across the racks. One third of replicas are on one node, two thirds of replicas are on one rack, and the other third are evenly distributed across the remaining racks. This policy improves write performance without compromising data reliability or read performance.

Data Replication & Rack Awareness



Data Replication

✓ Replica Selection

➢ To minimize global bandwidth consumption and read latency, HDFS tries to satisfy a read request from a replica that is closest to the reader. If there exists a replica on the same rack as the reader node, then that replica is preferred to satisfy the read request. If angg/ HDFS cluster spans multiple data centers, then a replica that is resident in the local data center is preferred over any remote replica.

HDFS - Balancer

✓ Functionalities

- ➢ Distribution of blocks across data nodes becomes unbalanced over a period of time.
- ➢ Affects the performance and put strain on highly utilized data nodes
- ➢ Re destribute the blocks from highly utilized data nodes to under utilized data nodes
  - Bin/start-balancer.sh

✓ Under Replication

- ➢ If default factor is 3 means 3 copies are there and one of the node goes down in the cluster the total number of copies remains is 2. Which is called under replication. Namenode does the normalization and makes it 3 by taking some time. It happens automatically and users are not aware of this.

✓ Over Replication

- ➢ If Node comes back to the cluster, then the copy of the block becomes 4 but default is 3. So Namenode does the normalization and make it 3 otherwise it will use extra storage space.

The Persistence of File System Metadata

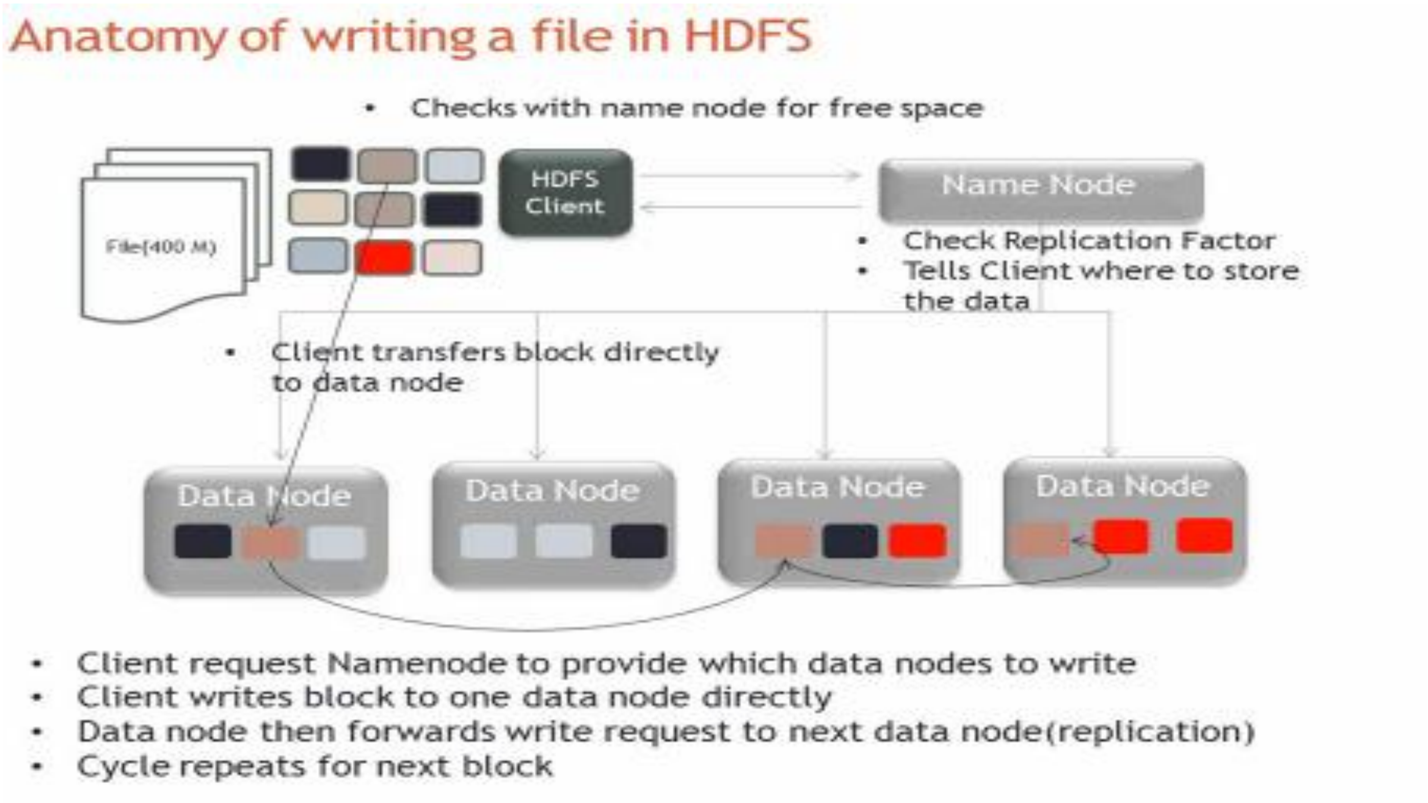
✓ Functionalities

- ➢ The HDFS namespace is stored by the NameNode. The NameNode uses a transaction log called the EditLog to persistently record every change that occurs to file system metadata. For example, creating a new file in HDFS causes the NameNode to insert a record into the EditLog indicating this. Similarly, changing the replication factor of a file causes a new record to be inserted into the EditLog. The NameNode uses a file in its local host OS file system to store the EditLog. The entire file system namespace, including the mapping of blocks to files and file system properties, is stored in a file called the FsImage. The FsImage is stored as a file in the NameNode's local file system too.

- ➢ The NameNode keeps an image of the entire file system namespace and file Blockmap in memory. This key metadata item is designed to be compact, such that a NameNode with 4 GB of RAM is plenty to support a huge number of files and directories. When the NameNode starts up, it reads the FsImage and EditLog from disk, applies all the transactions from the EditLog to the in-memory representation of the FsImage, and flushes out this new version into a new FsImage on disk. It can then truncate the old EditLog because its transactions have been applied to the persistent FsImage. This process is called a checkpoint. In the current implementation, a checkpoint only occurs when the NameNode starts up. Work is in progress to support periodic checkpointing in the near future.

- ➢ The DataNode stores HDFS data in files in its local file system. The DataNode has no knowledge about HDFS files. It stores each block of HDFS data in a separate file in its local file system. The DataNode does not create all files in the same directory. Instead, it uses a heuristic to determine the optimal number of files per directory and creates subdirectories appropriately. It is not optimal to create all local files in the same directory because the local file system might not be able to efficiently support a huge number of files in a single directory. When a DataNode starts up, it scans through its local file system, generates a list of all HDFS data blocks that correspond to each of these local files and sends this report to the NameNode: this is the Blockreport.

Data Flow (Anatomy of a File Write)



Anatomy of writing a file in HDFS

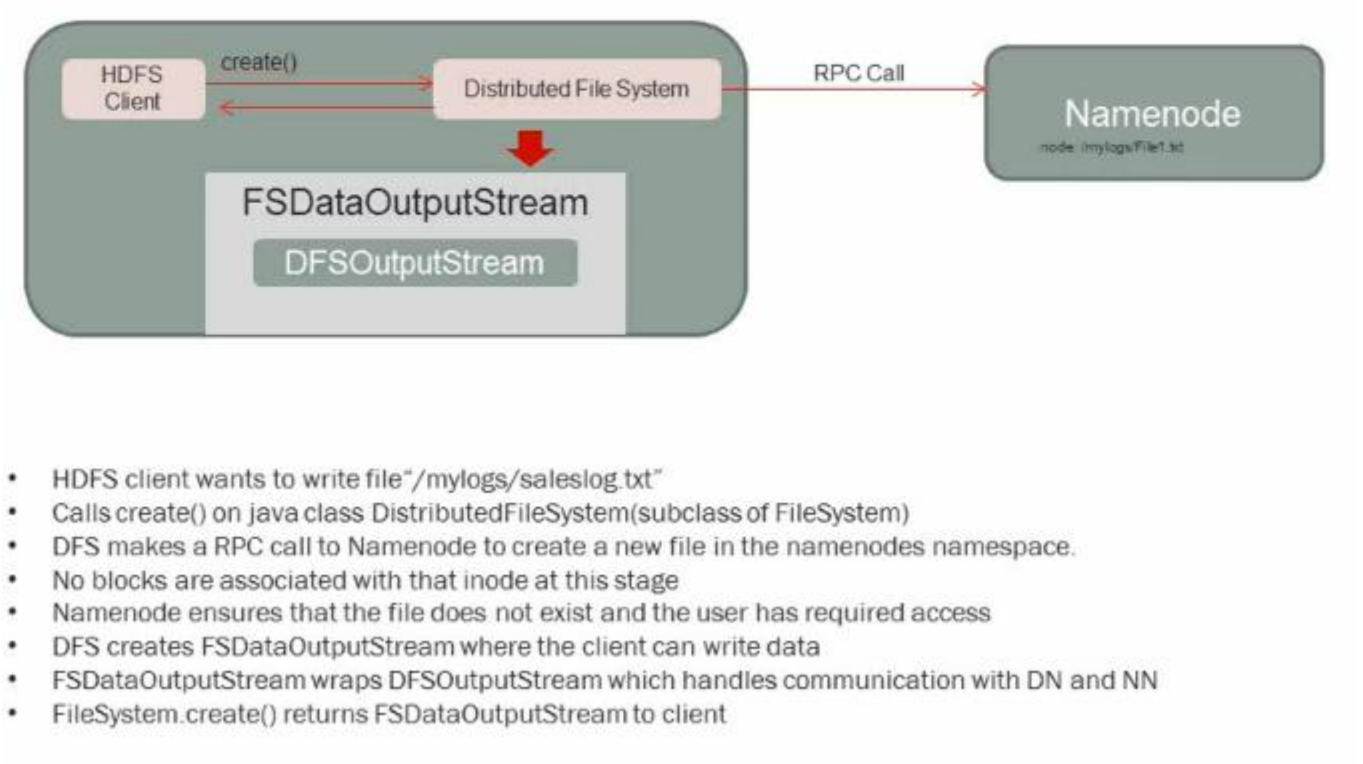
- Checks with name node for free space

- Check Replication Factor
- Tells Client where to store the data

- Client transfers block directly to data node

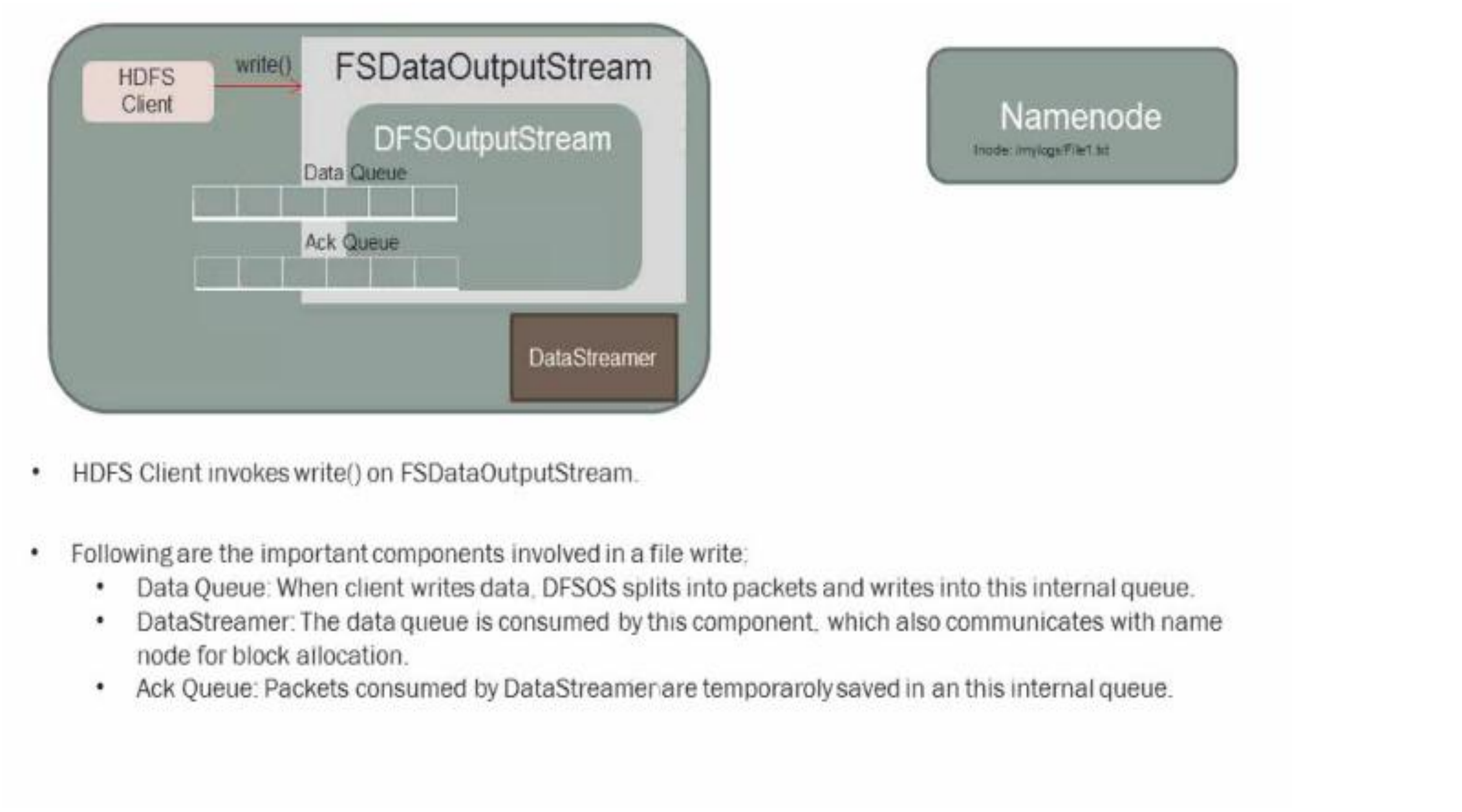
- Client request Namenode to provide which data nodes to write
- Client writes block to one data node directly
- Data node then forwards write request to next data node(replication)
- Cycle repeats for next block

Data Flow (Anatomy of a File Write)



- HDFS client wants to write file"/mylogs/saleslog.txt"
- Calls create() on java class DistributedFileSystem(subclass of FileSystem)
- DFS makes a RPC call to Namenode to create a new file in the namenodes namespace.
- No blocks are associated with that inode at this stage
- Namenode ensures that the file does not exist and the user has required access
- DFS creates FSDataOutputStream where the client can write data
- FSDataOutputStream wraps DFSOutputStream which handles communication with DN and NN
- FileSystem.create() returns FSDataOutputStream to client

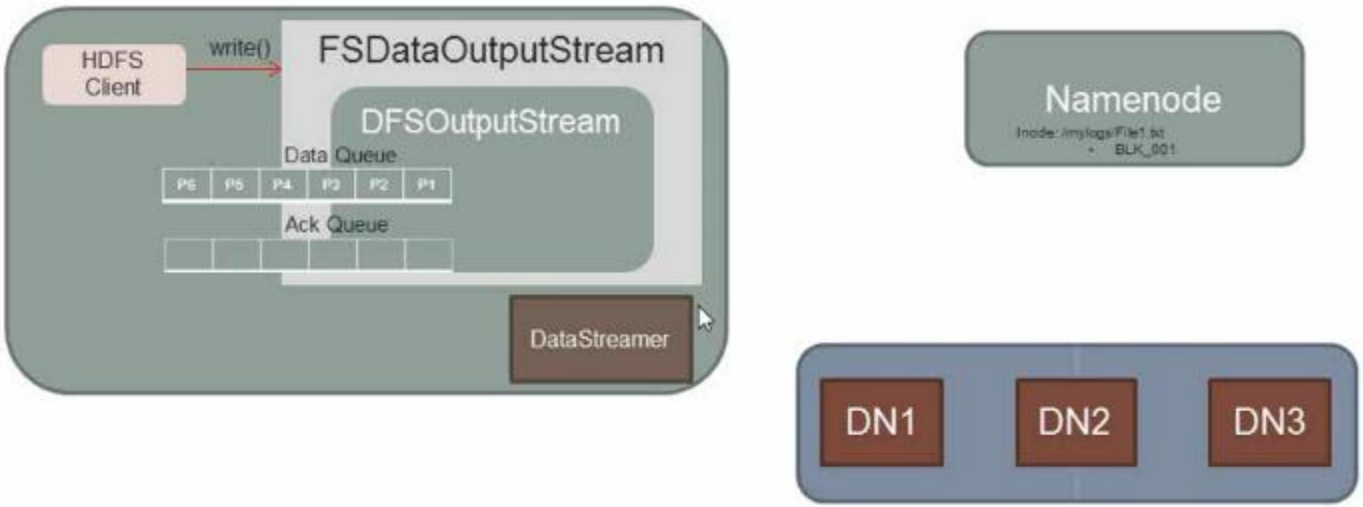
Data Flow (Anatomy of a File Write)



- HDFS Client invokes write() on FSDataOutputStream.

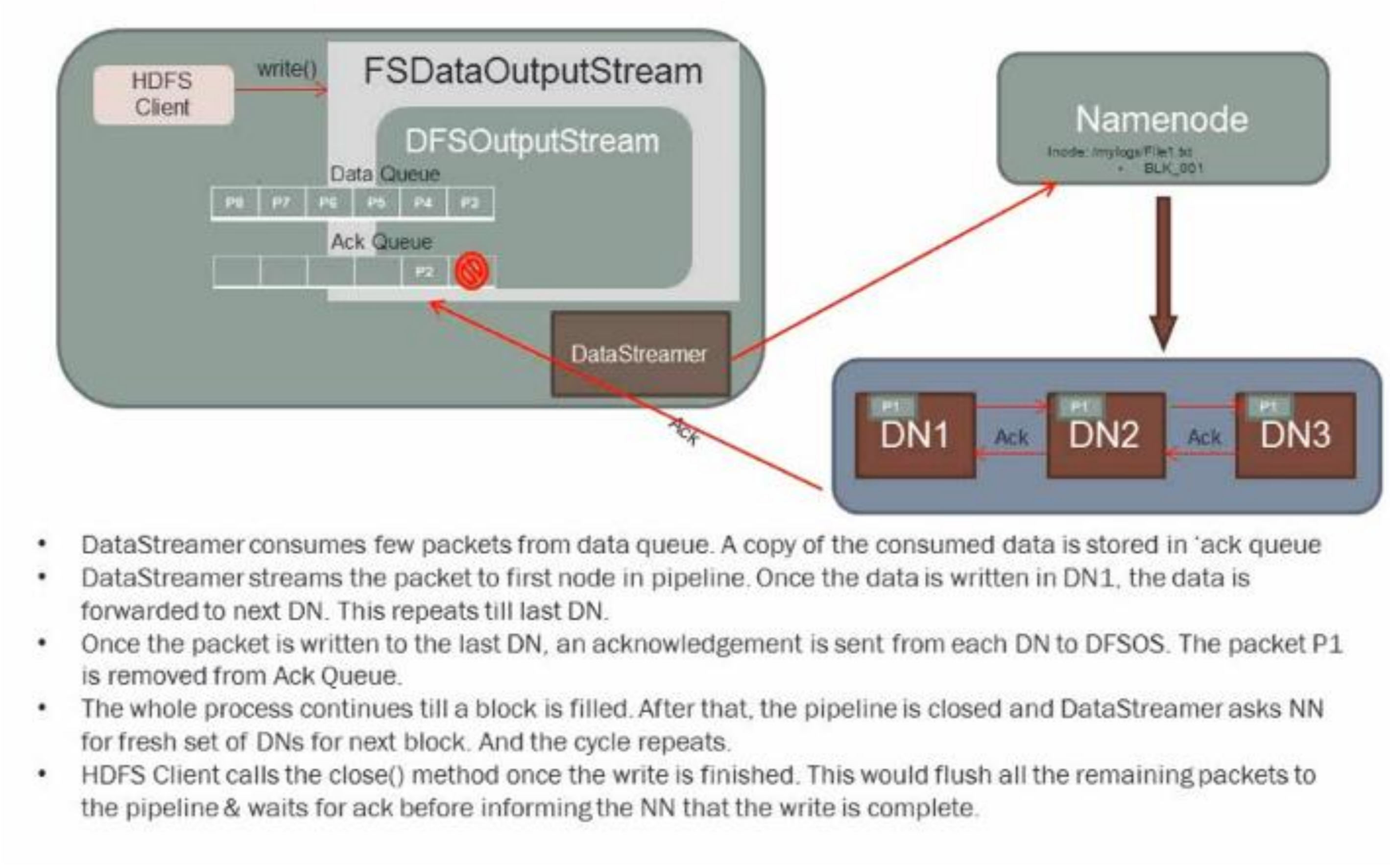
- Following are the important components involved in a file write:
  - Data Queue: When client writes data, DFSOS splits into packets and writes into this internal queue.
  - DataStreamer: The data queue is consumed by this component, which also communicates with name node for block allocation.
  - Ack Queue: Packets consumed by DataStreamer are temporaroly saved in an this internal queue.

Data Flow (Anatomy of a File Write)



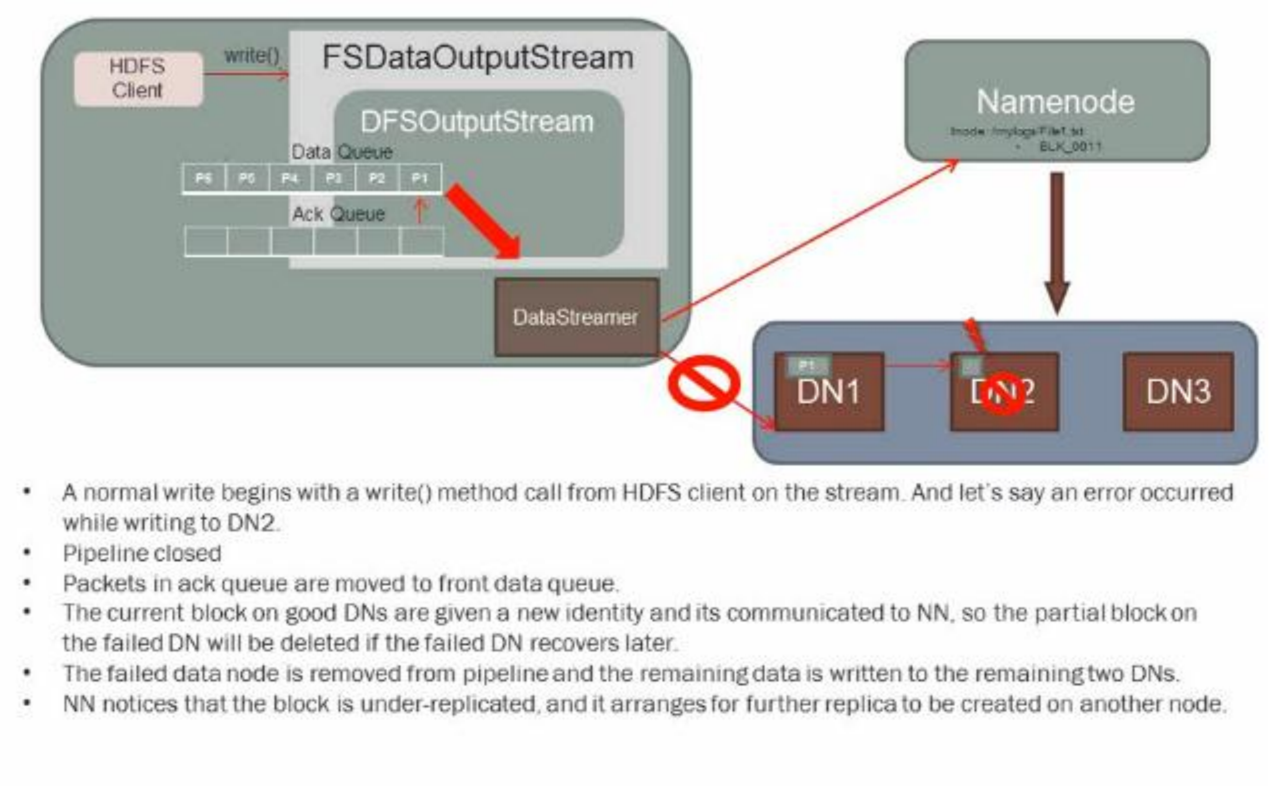
- Data will be converted into packets(**dfs.write.packet.size**) and stored in data queue.
- DataStreamer communicates with NN to allocate new blocks by picking a list of suitable DNs to store the replicas. NN uses 'Replica Placement' as a strategy to pick DNs for a block.
- The list of DNs form a pipeline. Since the replication factor is assumed as 3, there are 3 nodes picked by NN.

Data Flow (Anatomy of a File Write)



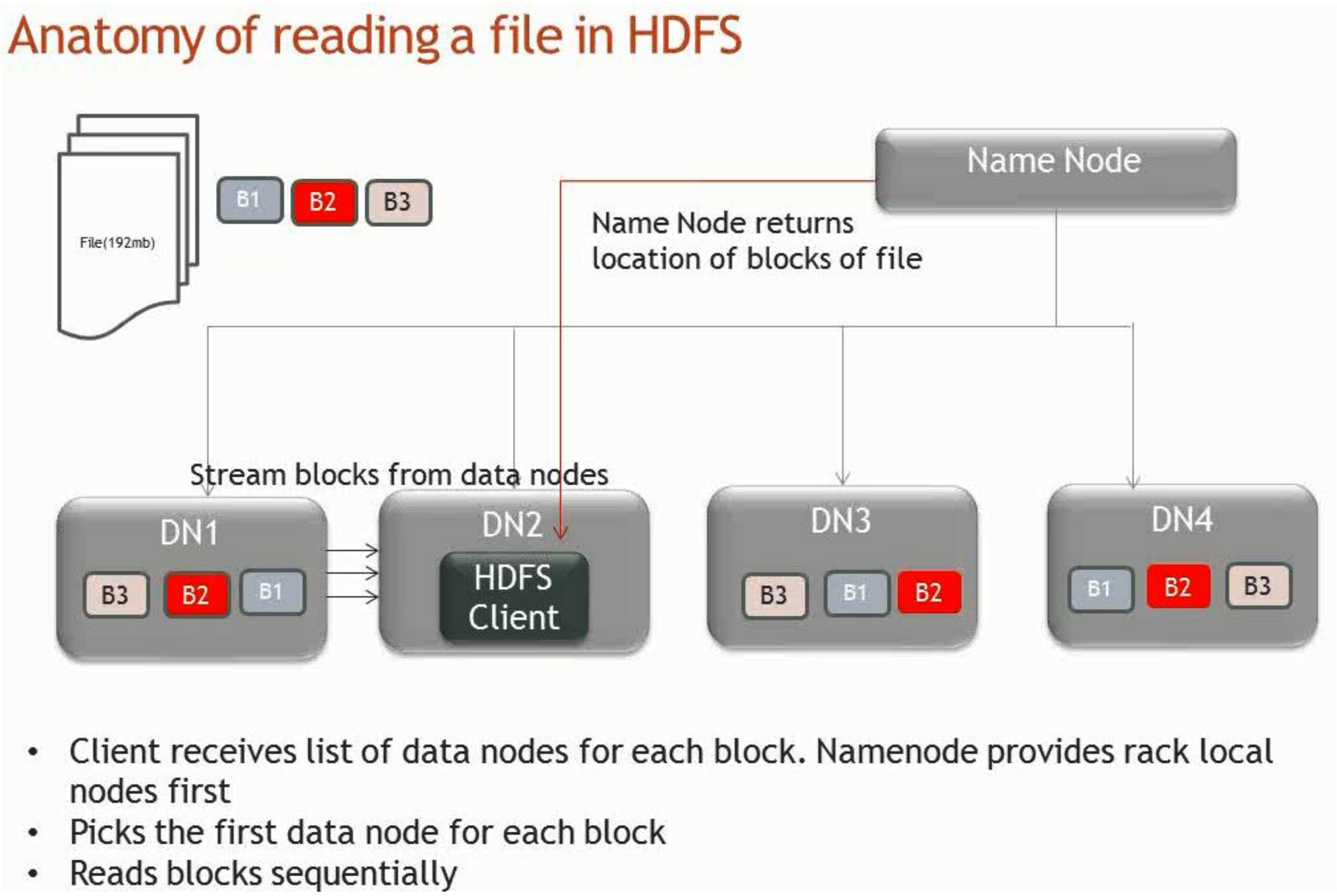
- DataStreamer consumes few packets from data queue. A copy of the consumed data is stored in 'ack queue
- DataStreamer streams the packet to first node in pipeline. Once the data is written in DN1, the data is forwarded to next DN. This repeats till last DN.
- Once the packet is written to the last DN, an acknowledgement is sent from each DN to DFSOS. The packet P1 is removed from Ack Queue.
- The whole process continues till a block is filled. After that, the pipeline is closed and DataStreamer asks NN for fresh set of DNs for next block. And the cycle repeats.
- HDFS Client calls the close() method once the write is finished. This would flush all the remaining packets to the pipeline & waits for ack before informing the NN that the write is complete.

Anatomy of a File Write – Unhappy Path



- A normal write begins with a write() method call from HDFS client on the stream. And let's say an error occurred while writing to DN2.
- Pipeline closed
- Packets in ack queue are moved to front data queue.
- The current block on good DNs are given a new identity and its communicated to NN, so the partial block on the failed DN will be deleted if the failed DN recovers later.
- The failed data node is removed from pipeline and the remaining data is written to the remaining two DNs.
- NN notices that the block is under-replicated, and it arranges for further replica to be created on another node.

Anatomy of a File Read



Anatomy of reading a file in HDFS

File(192mb)

B1 B2 B3

Name Node

Name Node returns location of blocks of file

Stream blocks from data nodes

DN1: B3 B2 B1

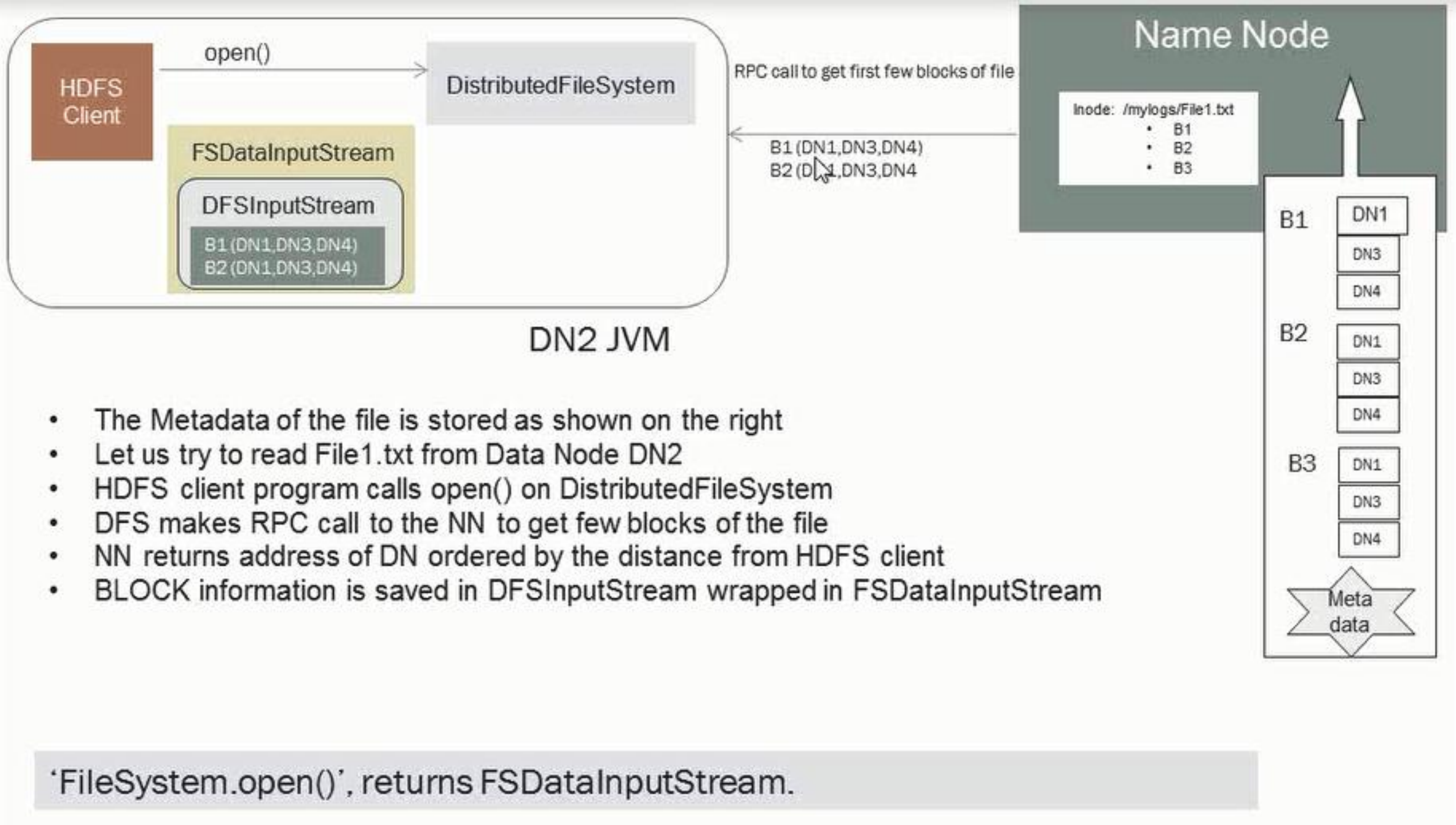
DN2: HDFS Client

DN3: B3 B1 B2

DN4: B1 B2 B3

- Client receives list of data nodes for each block. Namenode provides rack local nodes first
- Picks the first data node for each block
- Reads blocks sequentially

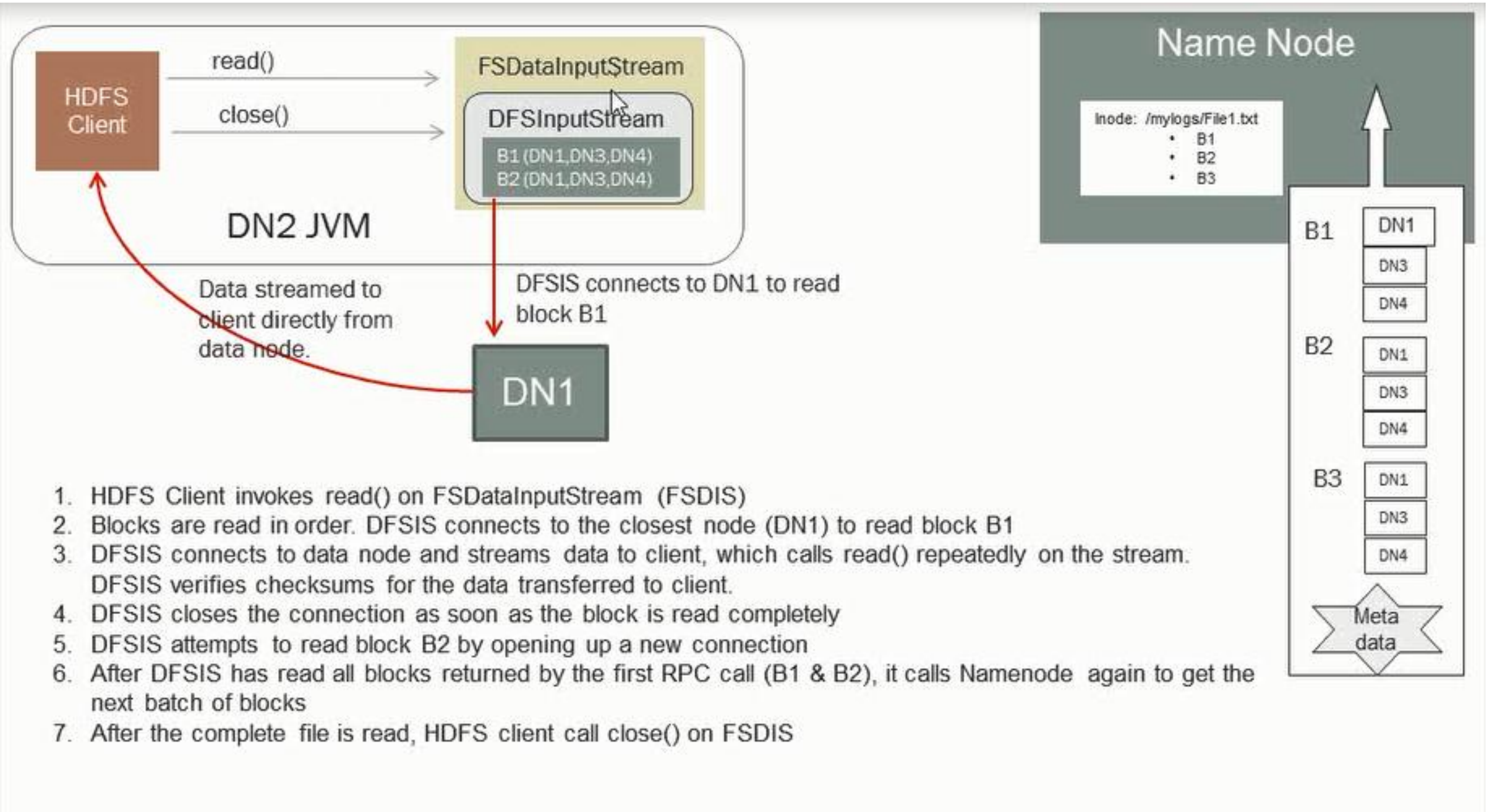
Anatomy of a File Read



- The Metadata of the file is stored as shown on the right
- Let us try to read File1.txt from Data Node DN2
- HDFS client program calls open() on DistributedFileSystem
- DFS makes RPC call to the NN to get few blocks of the file
- NN returns address of DN ordered by the distance from HDFS client
- BLOCK information is saved in DFSInputStream wrapped in FSDataInputStream

'FileSystem.open()', returns FSDataInputStream.

Anatomy of a File Read

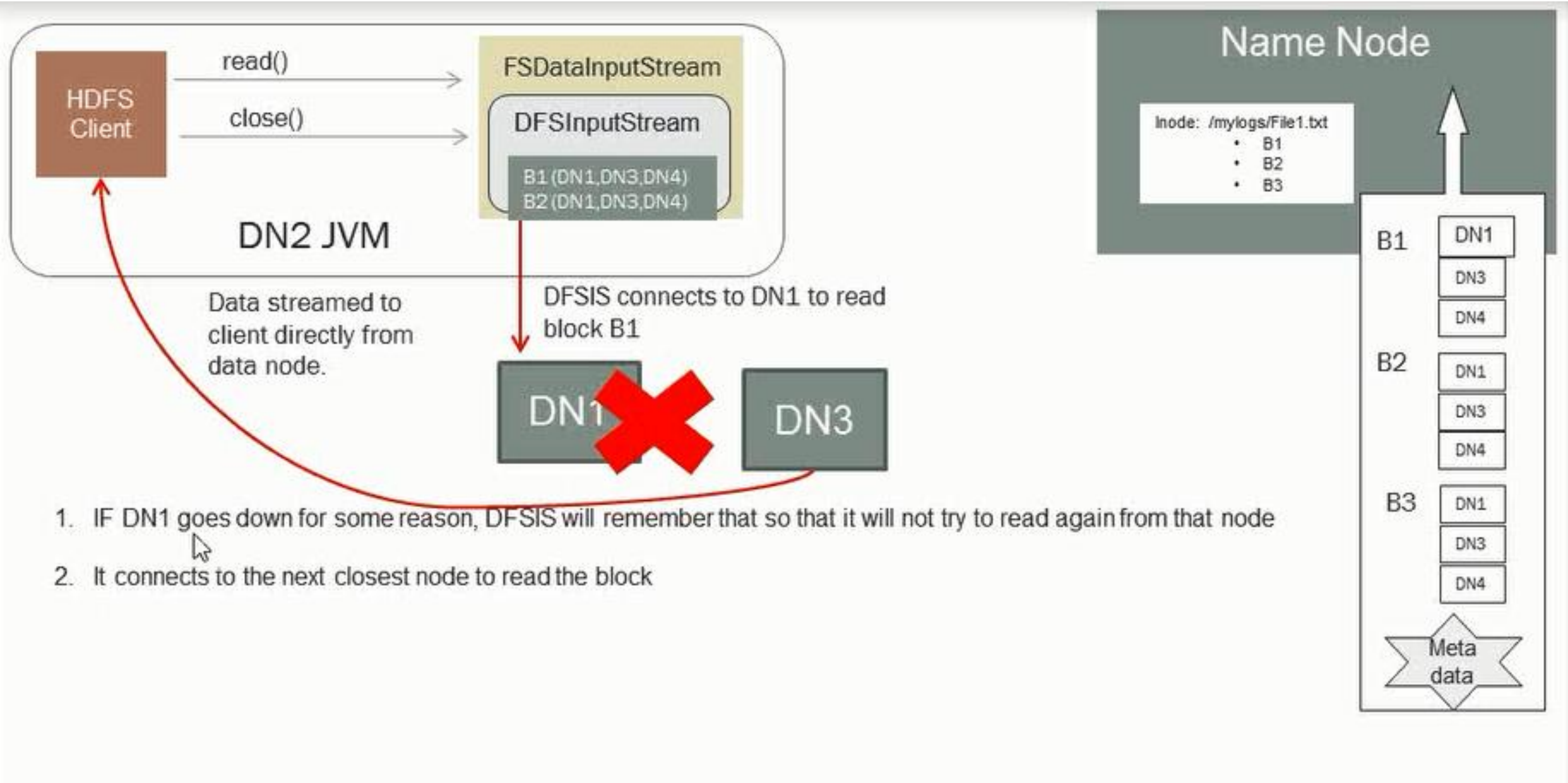


Data streamed to client directly from data node.

DFSIS connects to DN1 to read block B1

1. HDFS Client invokes read() on FSDataInputStream (FSDIS)
2. Blocks are read in order. DFSIS connects to the closest node (DN1) to read block B1
3. DFSIS connects to data node and streams data to client, which calls read() repeatedly on the stream. DFSIS verifies checksums for the data transferred to client.
4. DFSIS closes the connection as soon as the block is read completely
5. DFSIS attempts to read block B2 by opening up a new connection
6. After DFSIS has read all blocks returned by the first RPC call (B1 & B2), it calls Namenode again to get the next batch of blocks
7. After the complete file is read, HDFS client call close() on FSDIS

Anatomy of a File Read–Unhappy Path

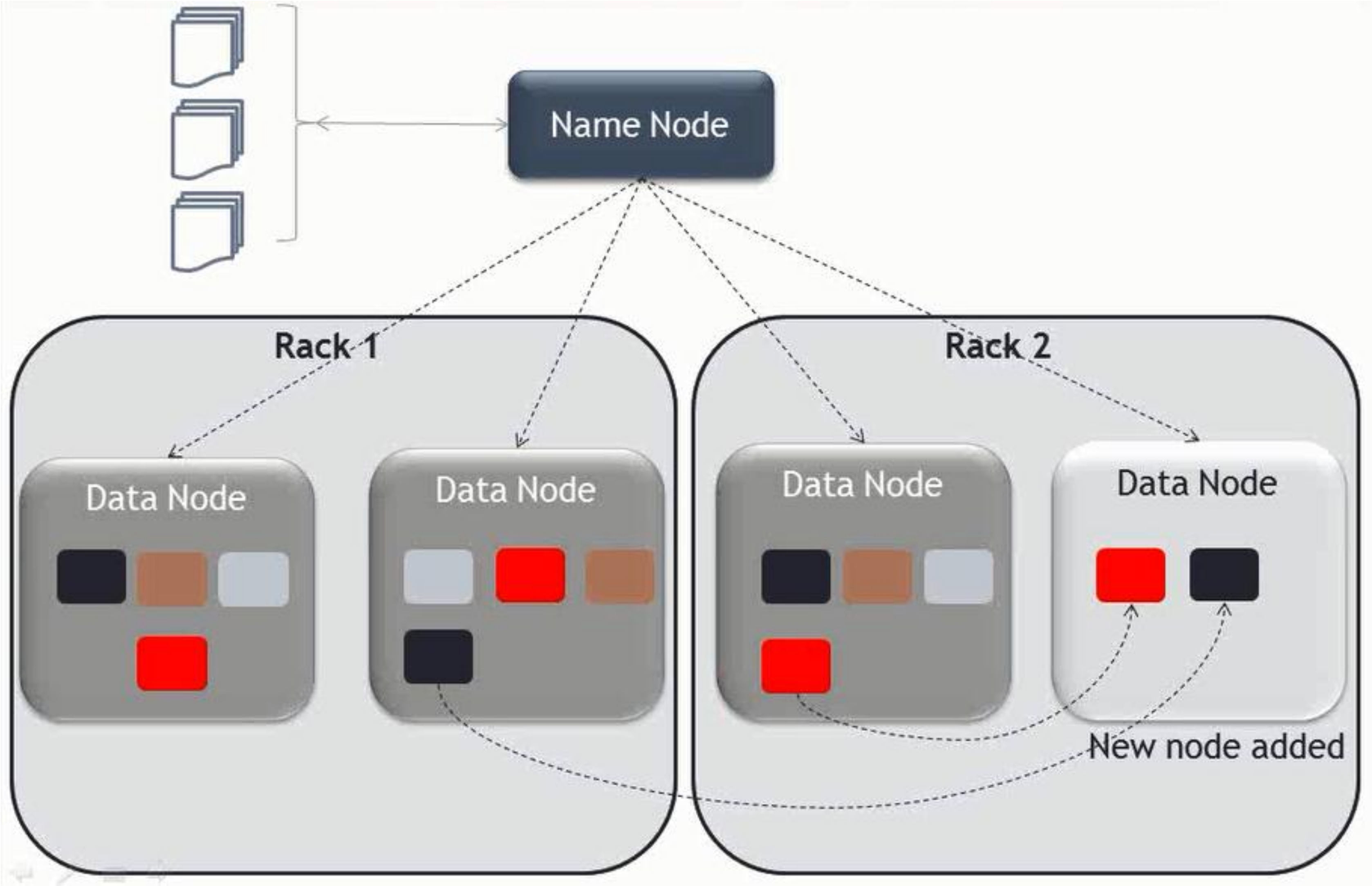


Data streamed to client directly from data node.

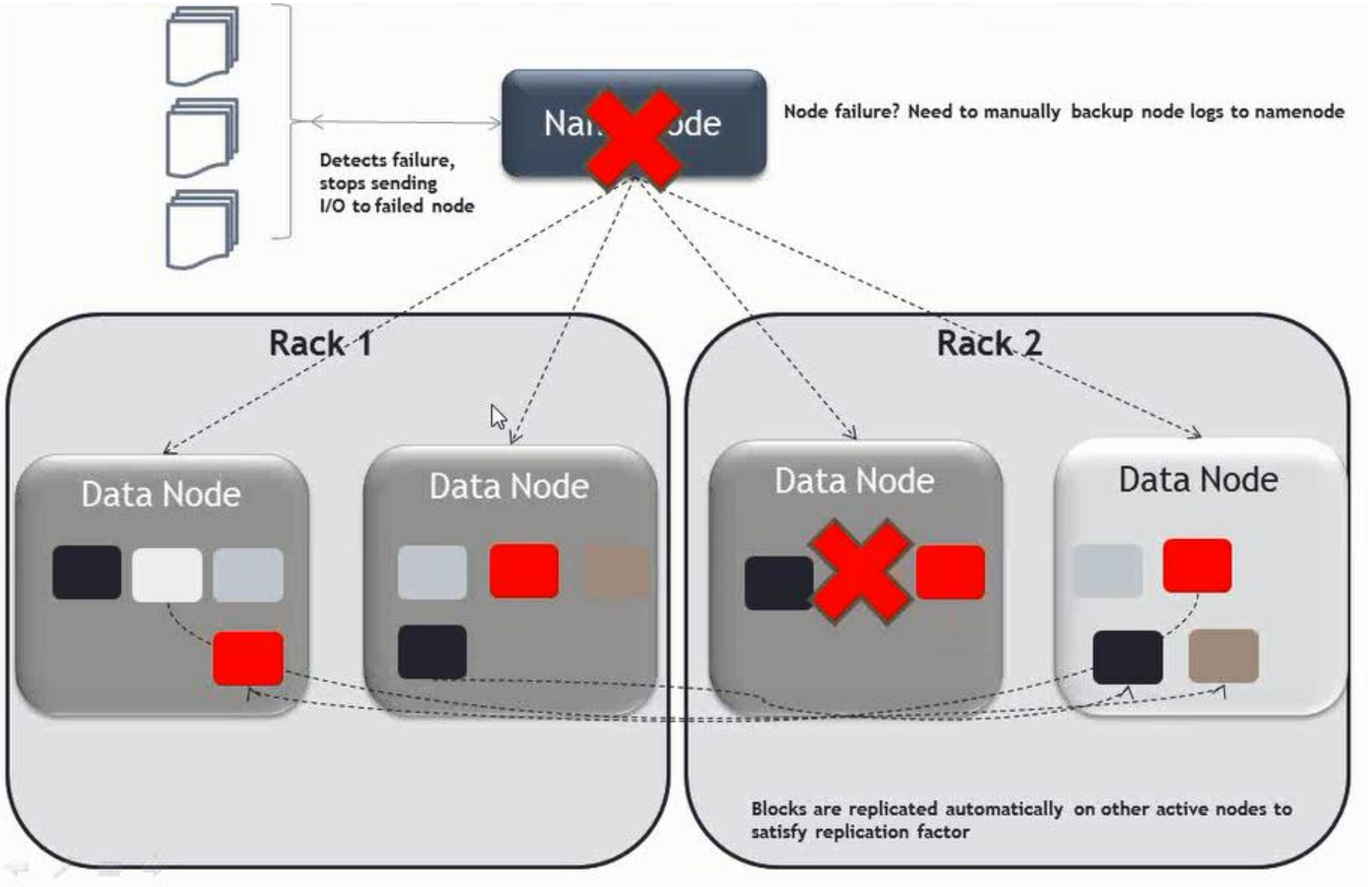
DFSIS connects to DN1 to read block B1

1. IF DN1 goes down for some reason, DFSIS will remember that so that it will not try to read again from that node
2. It connects to the next closest node to read the block

How HDFS Balancer works



How Hadoop manages Node failures

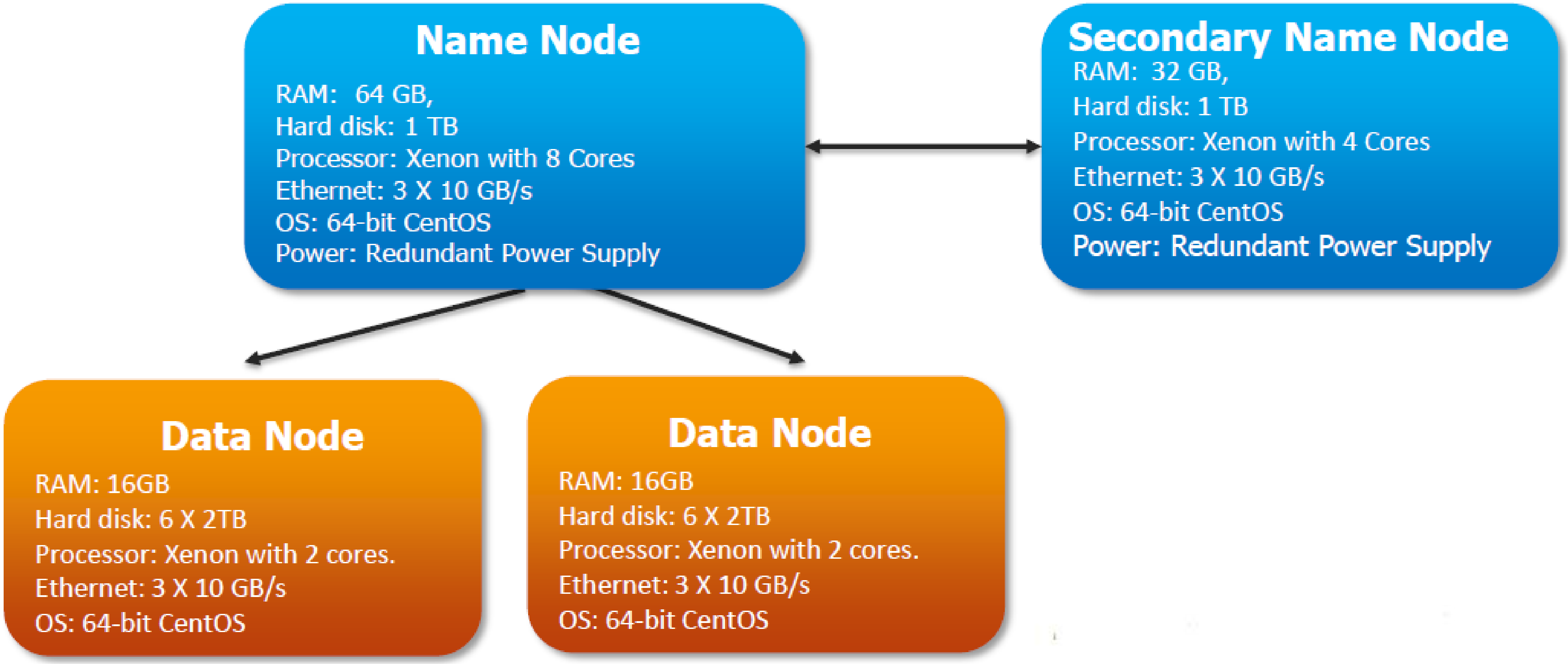


Node failure? Need to manually backup node logs to namenode

Detects failure, stops sending I/O to failed node

Blocks are replicated automatically on other active nodes to satisfy replication factor

Hadoop Cluster – A Typical Scenario



Name Node

RAM: 64 GB,
Hard disk: 1 TB
Processor: Xenon with 8 Cores
Ethernet: 3 X 10 GB/s
OS: 64-bit CentOS
Power: Redundant Power Supply

Secondary Name Node

RAM: 32 GB,
Hard disk: 1 TB
Processor: Xenon with 4 Cores
Ethernet: 3 X 10 GB/s
OS: 64-bit CentOS
Power: Redundant Power Supply

Data Node

RAM: 16GB
Hard disk: 6 X 2TB
Processor: Xenon with 2 cores.
Ethernet: 3 X 10 GB/s
OS: 64-bit CentOS

Data Node

RAM: 16GB
Hard disk: 6 X 2TB
Processor: Xenon with 2 cores.
Ethernet: 3 X 10 GB/s
OS: 64-bit CentOS

Robustness

✓ Data Disk Failure, Heartbeats and Re-Replication

- ➢ Each DataNode sends a Heartbeat message to the NameNode periodically. A network partition can cause a subset of DataNodes to lose connectivity with the NameNode. The NameNode detects this condition by the absence of a Heartbeat message. The NameNode marks DataNodes without recent Heartbeats as dead and does not forward any new IO requests to them.
- ➢ Any data that was registered to a dead DataNode is not available to HDFS any more. DataNode death may cause the replication factor of some blocks to fall below their specified value. The NameNode constantly tracks which blocks need to be replicated and initiates replication whenever necessary. The necessity for re-replication may arise due to many reasons: a DataNode may become unavailable, a replica may become corrupted, a hard disk on a DataNode may fail, or the replication factor of a file may be increased.

✓ Data Integrity

- ➢ It is possible that a block of data fetched from a DataNode arrives corrupted. This corruption can occur because of faults in a storage device, network faults, or buggy software. The HDFS client software implements checksum checking on the contents of HDFS files. When a client creates an HDFS file, it computes a checksum of each block of the file and stores these checksums in a separate hidden file in the same HDFS namespace.
- ➢ When a client retrieves file contents it verifies that the data it received from each DataNode matches the checksum stored in the associated checksum file. If not, then the client can opt to retrieve that block from another DataNode that has a replica of that block

Robustness

✓ Metadata Disk Failure

- ➢ The FsImage and the EditLog are central data structures of HDFS. A corruption of these files can cause the HDFS instance to be non-functional. For this reason, the NameNode can be configured to support maintaining multiple copies of the FsImage and EditLog. Any update to either the FsImage or EditLog causes each of the FsImages and EditLogs to get updated synchronously. This synchronous updating of multiple copies of the FsImage and EditLog may degrade the rate of namespace transactions per second that a NameNode can support. However, this degradation is acceptable because even though HDFS applications are very data intensive in nature, they are not metadata intensive. When a NameNode restarts, it selects the latest consistent FsImage and EditLog to use.

- ➢ The NameNode machine is a single point of failure for an HDFS cluster. If the NameNode machine fails, manual intervention is necessary. Currently, automatic restart and failover of the NameNode software to another machine is not supported.

Accessibility

✓ FS Shell

- ➢ HDFS allows user data to be organized in the form of files and directories. It provides a commandline interface called FS shell that lets a user interact with the data in HDFS. The syntax of this command set is similar to other shells (e.g. bash, csh) that users are already familiar with.

✓ DFSAdmin

- ➢ The DFSAdmin command set is used for administering an HDFS cluster. These are commands that are used only by an HDFS administrator..

✓ Browser Interface

- ➢ A typical HDFS install configures a web server to expose the HDFS namespace through a configurable TCP port. This allows a user to navigate the HDFS namespace and view the contents of its files using a web browser

Space Reclamation

✓ File Deletes and Undeletes

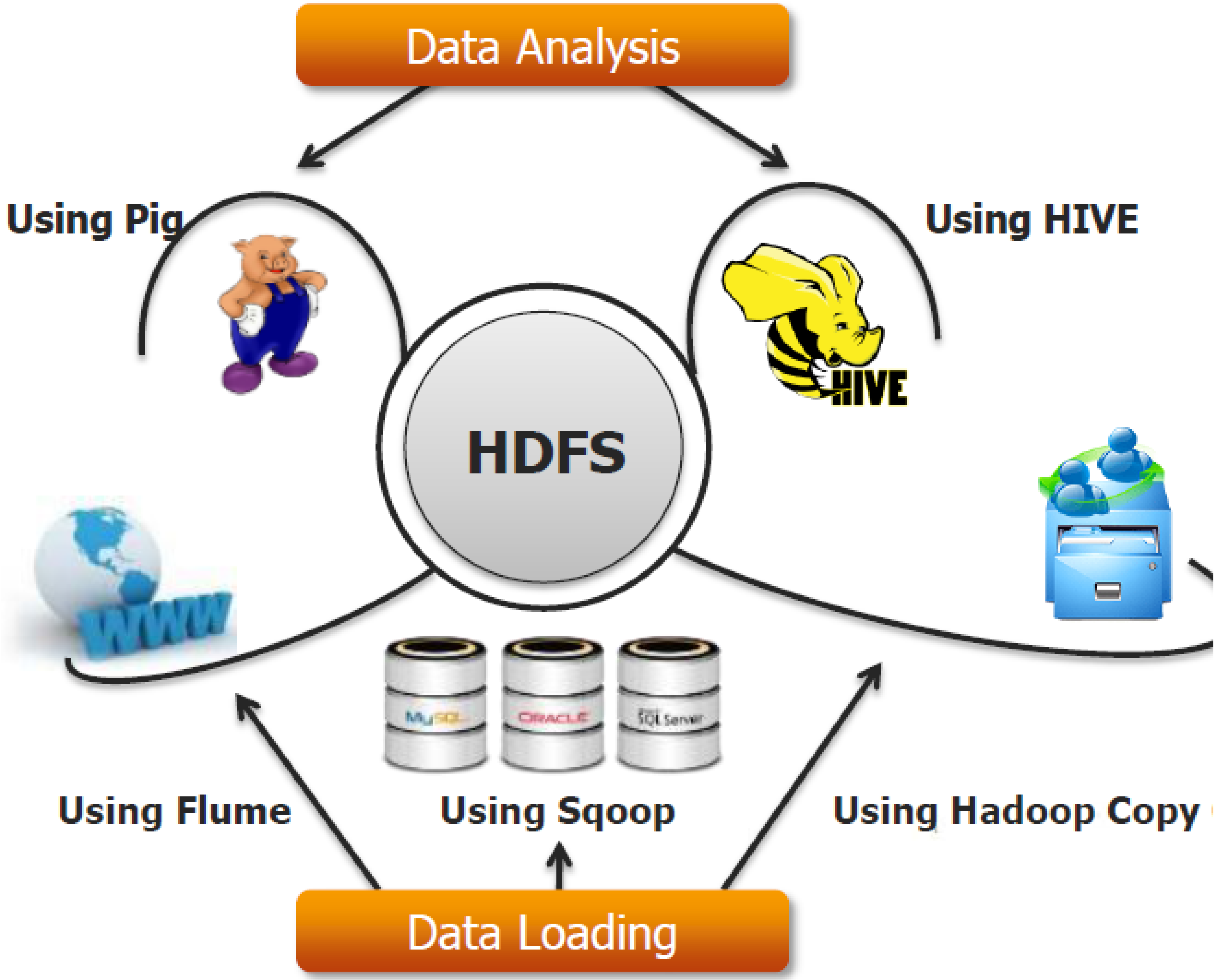
- When a file is deleted by a user or an application, it is not immediately removed from HDFS. Instead, HDFS first renames it to a file in the /trash directory. The file can be restored quickly as long as it remains in /trash. A file remains in /trash for a configurable amount of time. After the expiry of its life in /trash, the NameNode deletes the file from the HDFS namespace. The deletion of a file causes the blocks associated with the file to be freed. Note that there could be an appreciable time delay between the time a file is deleted by a user and the time of the corresponding increase in free space in HDFS.

- A user can Undelete a file after deleting it as long as it remains in the /trash directory. If a user wants to undelete a file that he/she has deleted, he/she can navigate the /trash directory and retrieve the file. The /trash directory contains only the latest copy of the file that was deleted. The /trash directory is just like any other directory with one special feature: HDFS applies specified policies to automatically delete files from this directory. The current default policy is to delete files from /trash that are more than 6 hours old. In the future, this policy will be configurable through a well defined interface.

✓ Decrease Replication Factor

- When the replication factor of a file is reduced, the NameNode selects excess replicas that can be deleted. The next Heartbeat transfers this information to the DataNode. The DataNode then removes the corresponding blocks and the corresponding free space appears in the cluster. Once again, there might be a time delay between the completion of the setReplication API call and the appearance of free space in the cluster

Data Loading Techniques and Analysis



Data Loading Using Sqoop

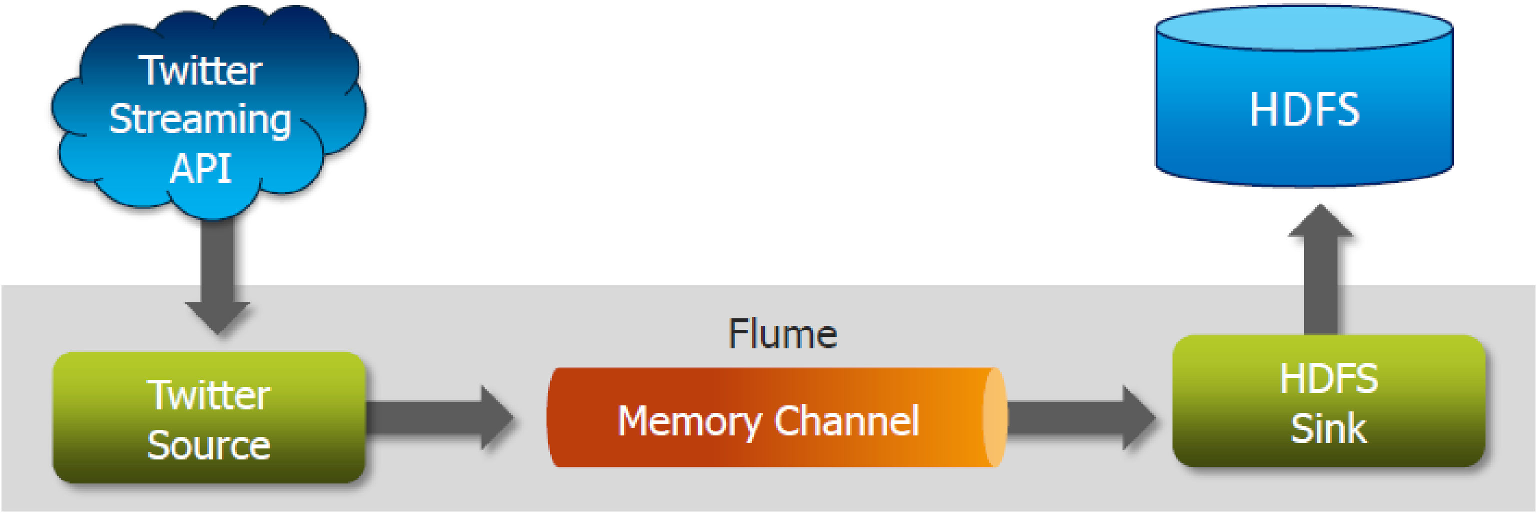
Apache Sqoop (TM) is a tool designed for efficiently transferring bulk data between Apache Hadoop and structured data stores such as relational databases.

- ✓ Imports individual tables or entire databases to HDFS.
- ✓ Generates Java classes to allow you to interact with your imported data.
- ✓ Provides the ability to import from SQL databases straight into your Hive data warehouse.

hadoop
SQOOP
SQL

Data Loading Using Flume

Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of streaming event data.



Data Loading Techniques

Demo for Data Load

HDFS commands and Sqoop

HDFS Interaction

- ✓ Demo hive example to insert a table
  - Create table (file) in HDFS
  - Load data to this table using hive queries in HDFS
  - View data from this table/file in HDFS using hive queries
- ✓ View NameNode logs after executing above steps of demo
- ✓ HDFS Interaction using HDFS commands
- ✓ How File is devided into Blocks and The location of these Blocks

Thank You