BITS Pilani
Pilani | Dubai | Goa | Hyderabad

# DSECL ZG 522: Big Data Systems

## Session 6: Hadoop architecture and filesystem


**Dr. Anindya Neogi**

**Associate Professor**

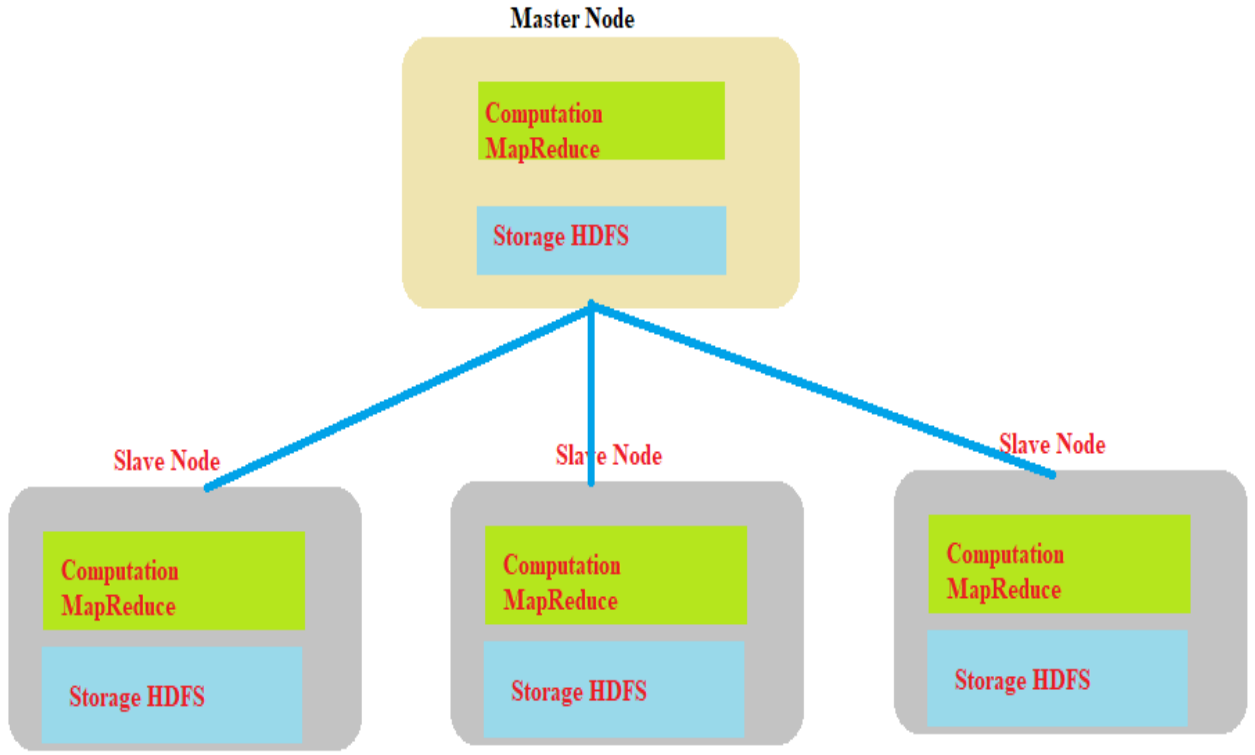
anindya.neogi@pilani.bits-pilani.ac.in

# Topics for today

- **Hadoop architecture overview**
  - ✓ Components
  - ✓ Hadoop 1 vs Hadoop 2
- HDFS
  - ✓ Architecture
  - ✓ Robustness
  - ✓ Blocks and replication strategy
  - ✓ Read and write operations
  - ✓ File formats
  - ✓ Commands

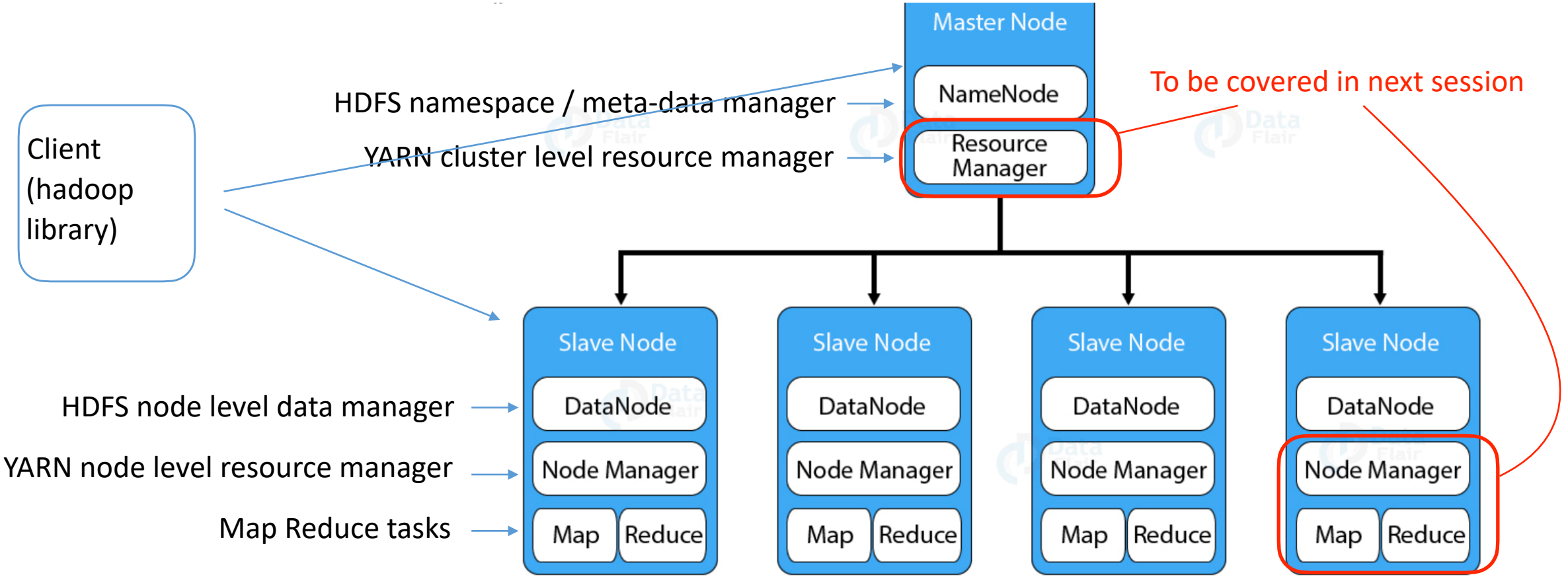
# Hadoop - Data and Compute layers

- A data storage layer
  - A Distributed File System - HDFS
- A data processing layer
  - MapReduce programming



# Hadoop 2 - Architecture

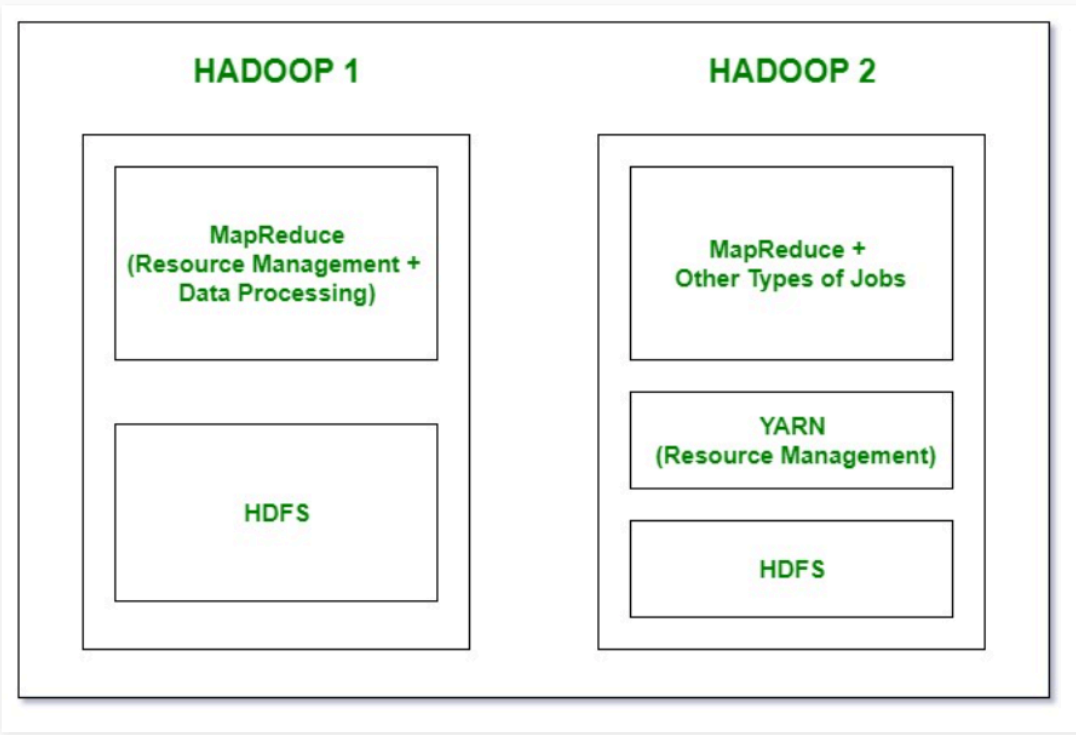
- Master-slave architecture for overall compute and data management
- Slaves implement peer-to-peer communication




***Note**: YARN Resource Manager also uses application level App Master processes on slave nodes for application specific resource management*

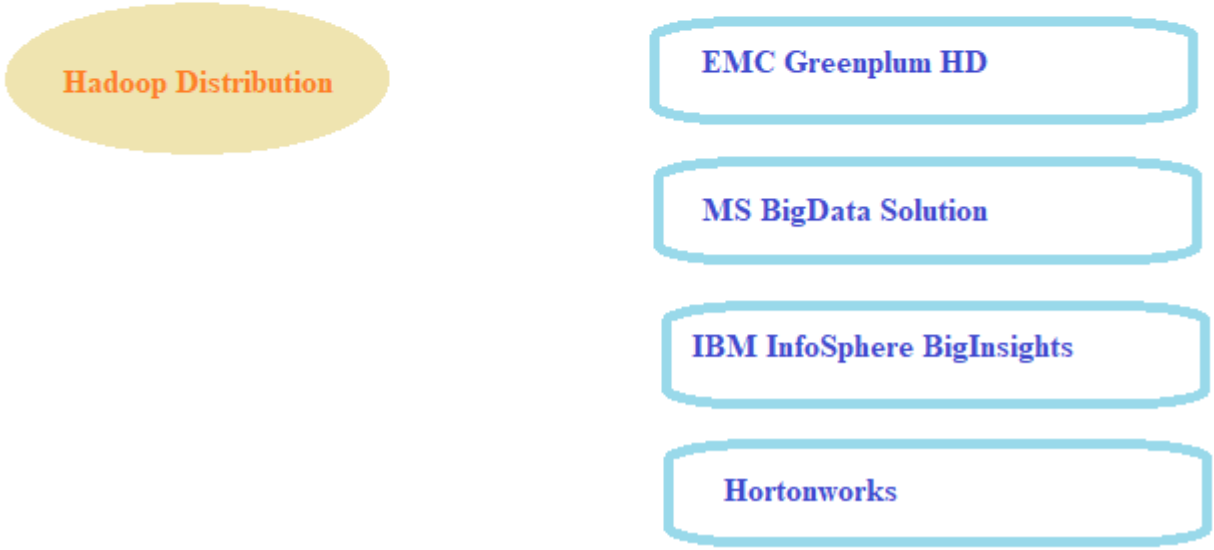
# What changed from Hadoop 1 to Hadoop 2

- Hadoop 1: MapReduce was coupled with resource management
  - Hadoop 2 brought in **YARN** as a resource management capability and **MapReduce** is only about data processing.
- Hadoop 1: Single Master node with NameNode is a SPOF
  - Hadoop 2 introduced active-passive and other **HA** configurations besides **secondary NameNodes**
- Hadoop 1: Only MapReduce programs
  - In Hadoop 2, **non MR programs** can be run by YARN on slave nodes (since decoupled from MapReduce) as well support for **non-HDFS storage**, e.g. Amazon S3 etc.



# Hadoop Distributions

- Open source Apache project
- Core components :
  - ✓ Hadoop Common
  - ✓ Hadoop Distributed File System
  - ✓ Hadoop YARN
  - ✓ Hadoop MapReduce



# Topics for today

- Hadoop architecture overview
  - ✓ Components
  - ✓ Hadoop 1 vs Hadoop 2
- **HDFS**
  - ✓ Architecture
  - ✓ Robustness
  - ✓ Blocks and replication strategy
  - ✓ Read and write operations
  - ✓ File formats
  - ✓ Commands

# HDFS Features (1)

- A **DFS** stores data over multiple nodes in a cluster and allows multi-user access
  - ✓ Gives a feeling to the user that the data is on single machine
  - ✓ HDFS is a **Java based DFS** that sits on **top of native FS**
  - ✓ Enables storage of very large files across nodes of a Hadoop cluster
  - ✓ Data is split into large blocks : 128MB
- **Scale** through parallel data processing
  - ✓ 1 node with 1TB storage can have an IO bandwidth of 400MBps across 4 IO channels = 43 min
  - ✓ 10 nodes with partitioned 1 TB data can access in parallel that data in 4.3 min

# HDFS Features (2)

- **Fault tolerance** through replication
  - ✓ Default replication factor = 3 for every block (Hadoop 3 has some optimisations)
  - ✓ So 1 GB data can actually take 3GB storage
- **Consistency**
  - ✓ Write once and read many time workload
  - ✓ Files can be appended, truncated but not updated at any arbitrary point

# HDFS Features (3)

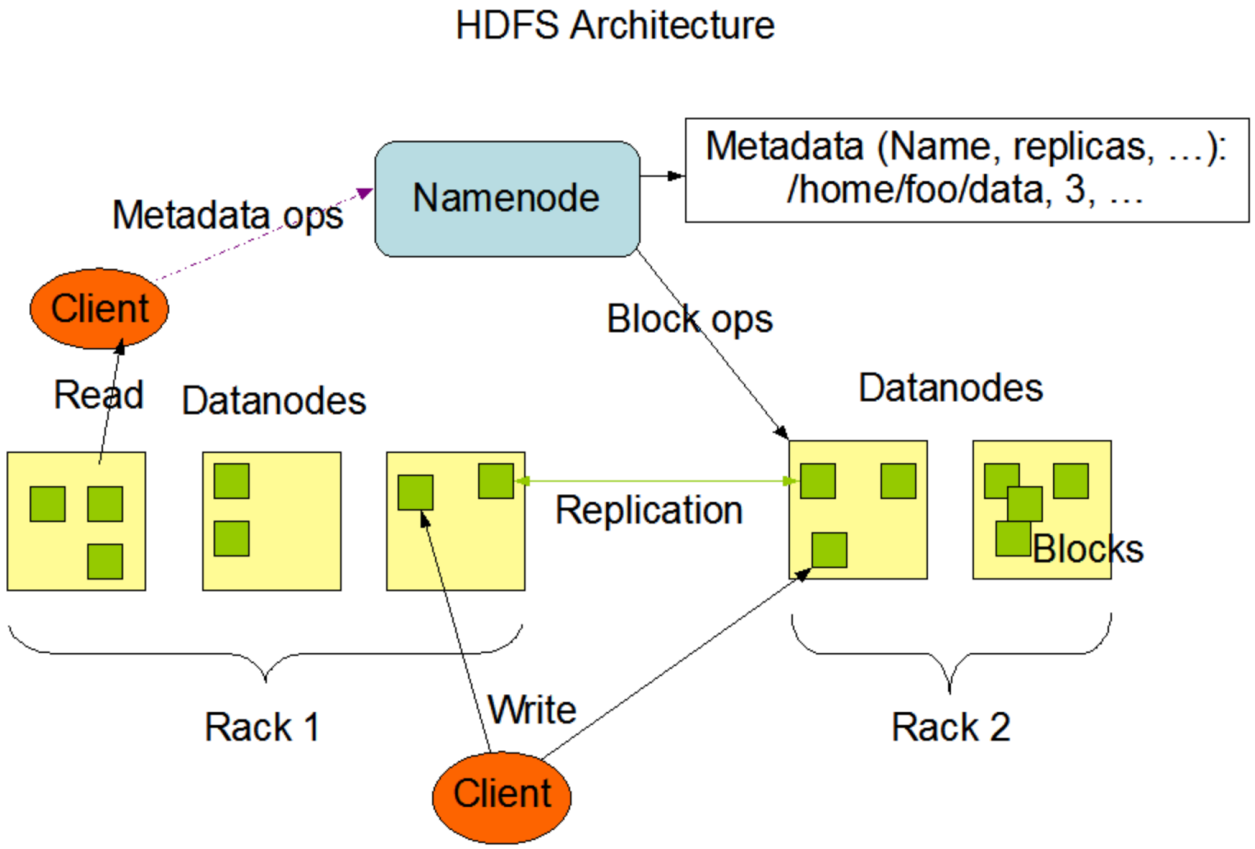
- **Cost:** Typically deployed using commodity hardware for low TCO - so adding more nodes is cost-effective
- **Variety and Volume of Data:** Huge data i.e. Terabytes & petabytes of data and different kinds of data - structured, unstructured or semi structured.

# HDFS Features (4)

- **Data Integrity:** HDFS nodes constantly verify checksums to preserve data integrity. On error, new copies are created and old copies are deleted.
- **Data Locality:** Data locality talks about moving processing unit to data rather than the data to processing unit. Bring the computation part to the data nodes where the data is residing. Hence, you are not moving the data, you are bringing the program or processing part to the data.

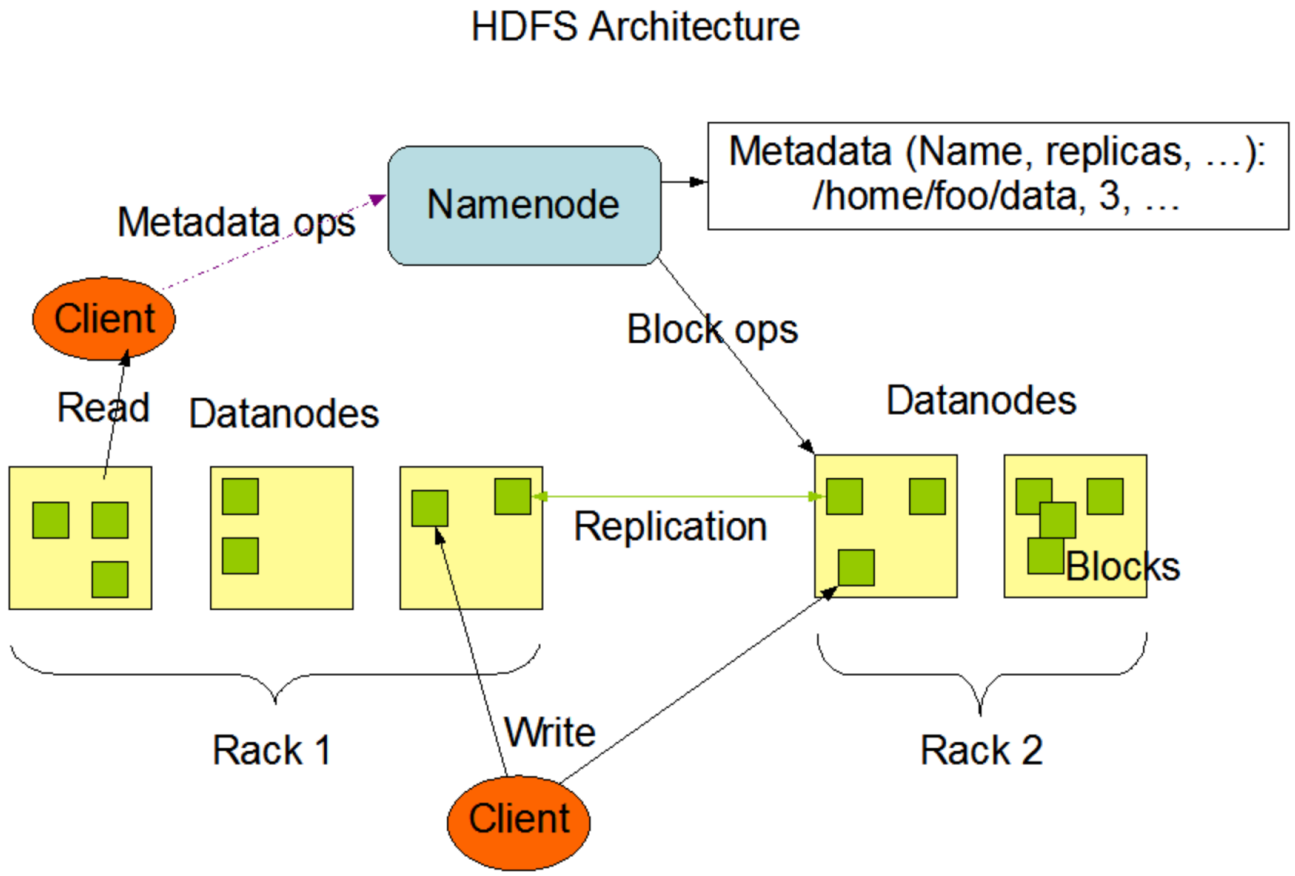
# HDFS Architecture - Master node

- Master slave architecture within a HDFS cluster
- One master node with NameNode
  - Maintains namespace - Filename to blocks and their replica mappings
  - Serves as arbitrator and doesn't handle actual data flow
  - HDFS client app interacts with NameNode for metadata



# HDFS Architecture - Slave nodes

- Multiple slave nodes with one DataNode per slave
  - Serves block R/W from Clients
  - Serves Create/Delete/Replicate requests from NameNode
  - DataNodes interact with each other for pipeline reads and writes.



# Functions of a NameNode

- Maintains namespace in HDFS with 2 files
  - **FsImage**: Contains mapping of blocks to file, hierarchy, file properties / permissions
  - **EditLog**: Transaction log of changes to metadata in FsImage
- Does not store any data - only meta-data about files
- Runs on **Master** node while DataNodes run on Slave nodes
- HA can be configured (discussed later)
- **Records each change** that takes place to the meta-data. e.g. if a file is deleted in HDFS, the NameNode will immediately record this in the **EditLog**.
- Receives periodic **Heartbeat** and a **block report** from all the DataNodes in the cluster to ensure that the DataNodes are live.
- Ensure **replication factor** is maintained across DataNode failures
  - In **case of the DataNode failure**, the NameNode chooses new DataNodes for new replicas, balance disk usage and manages the communication traffic to the DataNodes

# Where are fsimage and edit logs ?

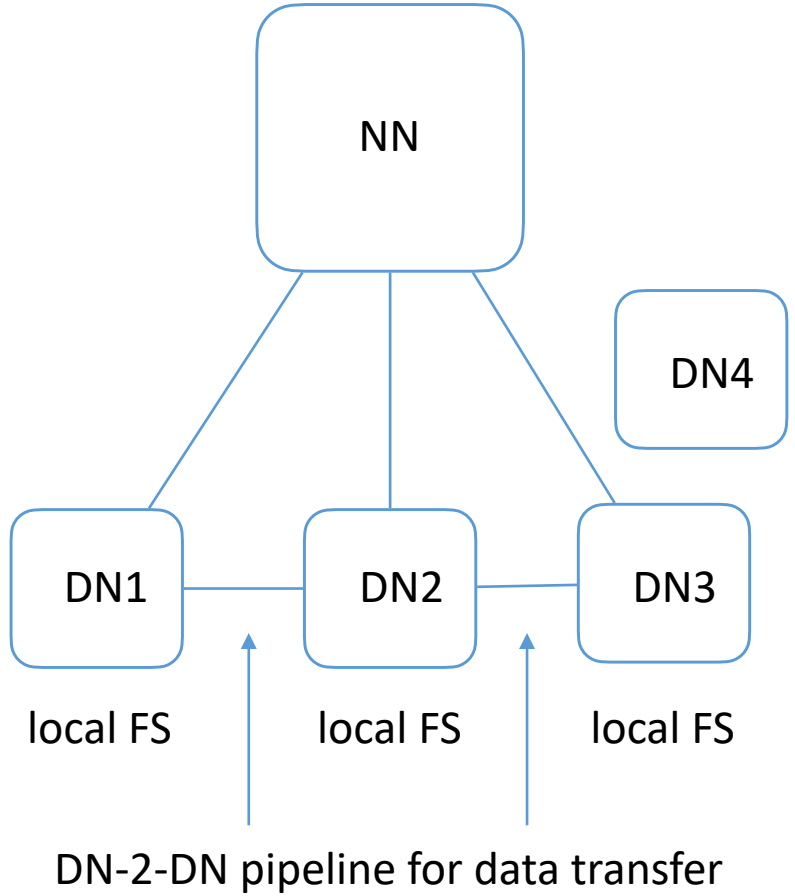
```
[root@centos-s-4vcpu-8gb-blr1-01 current]# pwd
/home/hadoop/hadoopdata/hdfs/namenode/current
[root@centos-s-4vcpu-8gb-blr1-01 current]# ls
VERSION                                     edits_0000000000000000205-0000000000000000206  edits_0000000000000000983-0000000000000000984
edits_0000000000000000001-0000000000000000002  edits_0000000000000000207-0000000000000000216  edits_0000000000000000985-0000000000000000986
edits_0000000000000000003-0000000000000000100  edits_0000000000000000217-0000000000000000298  edits_0000000000000000987-0000000000000000988
edits_0000000000000000101-0000000000000000102  edits_0000000000000000299-0000000000000000381  edits_0000000000000000989-0000000000000000990
edits_0000000000000000103-0000000000000000104  edits_0000000000000000382-0000000000000000383  edits_0000000000000000991-0000000000000000992
edits_0000000000000000105-0000000000000000106  edits_0000000000000000384-0000000000000000384  edits_0000000000000000993-0000000000000000994
edits_0000000000000000107-0000000000000000107  edits_0000000000000000385-0000000000000000386  edits_0000000000000000995-0000000000000000996
edits_0000000000000000108-0000000000000000108  edits_0000000000000000387-0000000000000000558  edits_0000000000000000997-0000000000000000998
edits_0000000000000000109-0000000000000000109  edits_0000000000000000559-0000000000000000560  edits_0000000000000000999-0000000000000001000
edits_0000000000000000110-0000000000000000111  edits_0000000000000000561-0000000000000000683  edits_0000000000000001001-0000000000000001002
edits_0000000000000000112-0000000000000000112  edits_0000000000000000684-0000000000000000685  edits_0000000000000001003-0000000000000001004
edits_0000000000000000113-0000000000000000114  edits_0000000000000000686-0000000000000000776  edits_0000000000000001005-0000000000000001006
edits_0000000000000000115-0000000000000000115  edits_0000000000000000777-0000000000000000893  edits_inprogress_0000000000000001007
edits_0000000000000000116-0000000000000000117  edits_0000000000000000894-0000000000000000971  fsimage_0000000000000001004
edits_0000000000000000118-0000000000000000199  edits_0000000000000000972-0000000000000000973  fsimage_0000000000000001004.md5
edits_0000000000000000200-0000000000000000200  edits_0000000000000000974-0000000000000000978  fsimage_0000000000000001006
edits_0000000000000000201-0000000000000000202  edits_0000000000000000979-0000000000000000980  fsimage_0000000000000001006.md5
edits_0000000000000000203-0000000000000000204  edits_0000000000000000981-0000000000000000982  seen_txid
You have mail in /var/spool/mail/root
[root@centos-s-4vcpu-8gb-blr1-01 current]#
```

# Namenode - What happens on start-up

1. Enters into **safe mode**
   - ✓ Check for status of Data nodes on slaves
     - Does **not allow** any Datanode **replications** in this mode
     - Gets **heartbeat** and **block report** from Datanodes
     - Checks for **minimum Replication Factor** needed for configurable majority of blocks
   - ✓ Updates meta-data (this is also done at checkpoint time)
     - Reads **FsImage** and **EditLog** from disk **into memory**
     - Applies all transactions from the EditLog to the in-memory version of FsImage
     - Flushes out new version of FsImage on disk
     - Keeps **latest FsImage in memory** for client requests
     - Truncates the old EditLog as its changes are applied on the new FsImage
2. Exits safe mode
3. Continues with further replications needed and client requests

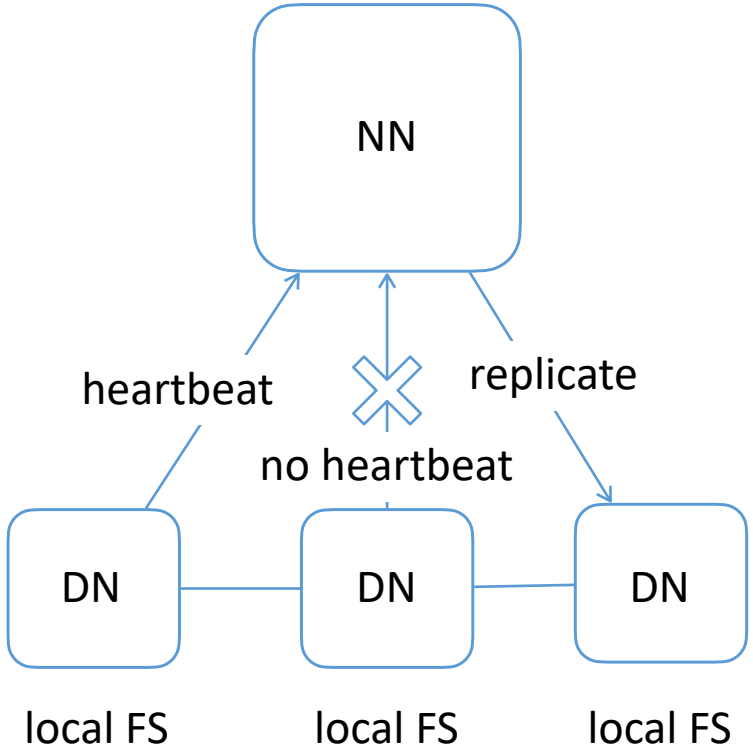
# Functions of a DataNode (1)

- Each slave in cluster runs a DataNode
- Nodes **store actual data blocks** and R/W data for the HDFS clients as **regular files** on the native file system, e.g. ext2 or ext3
- During **pipeline read and write**, DataNodes communicate with each other
  - We will discuss what's a pipeline
- No additional HA because blocks are anyway replicated



# Functions of a DataNode (2)

- DataNode continuously sends **heartbeat** to NameNode (default 3 sec)
  - ✓ To ensure the connectivity with NameNode
- If no heartbeat message from DataNode, NameNode replicates that DataNode within the cluster and removes the DN from the meta-data records
- DataNodes also send a **BlockReport** on start-up / periodically containing file list
- Applies some **heuristic to subdivide files** into directories based on limits of local FS but has no knowledge of HDFS level files

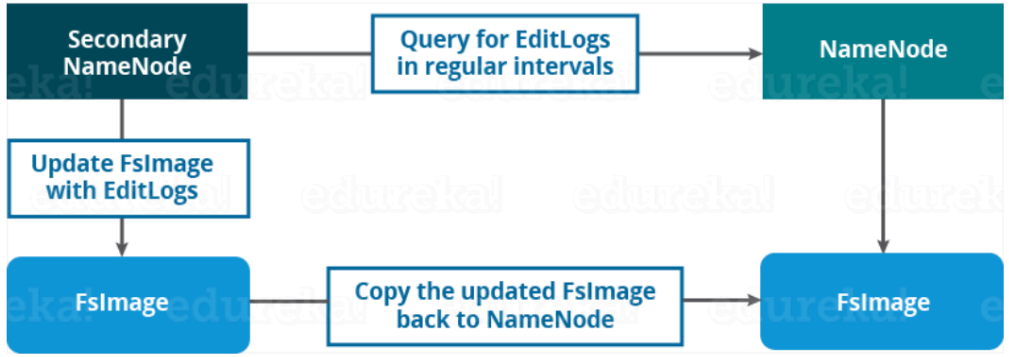


# Topics for today

- Hadoop architecture overview
  - ✓ Components
  - ✓ Hadoop 1 vs Hadoop 2
- **HDFS**
  - ✓ Architecture
  - ✓ **Robustness**
  - ✓ Blocks and replication strategy
  - ✓ Read and write operations
  - ✓ File formats
  - ✓ Commands

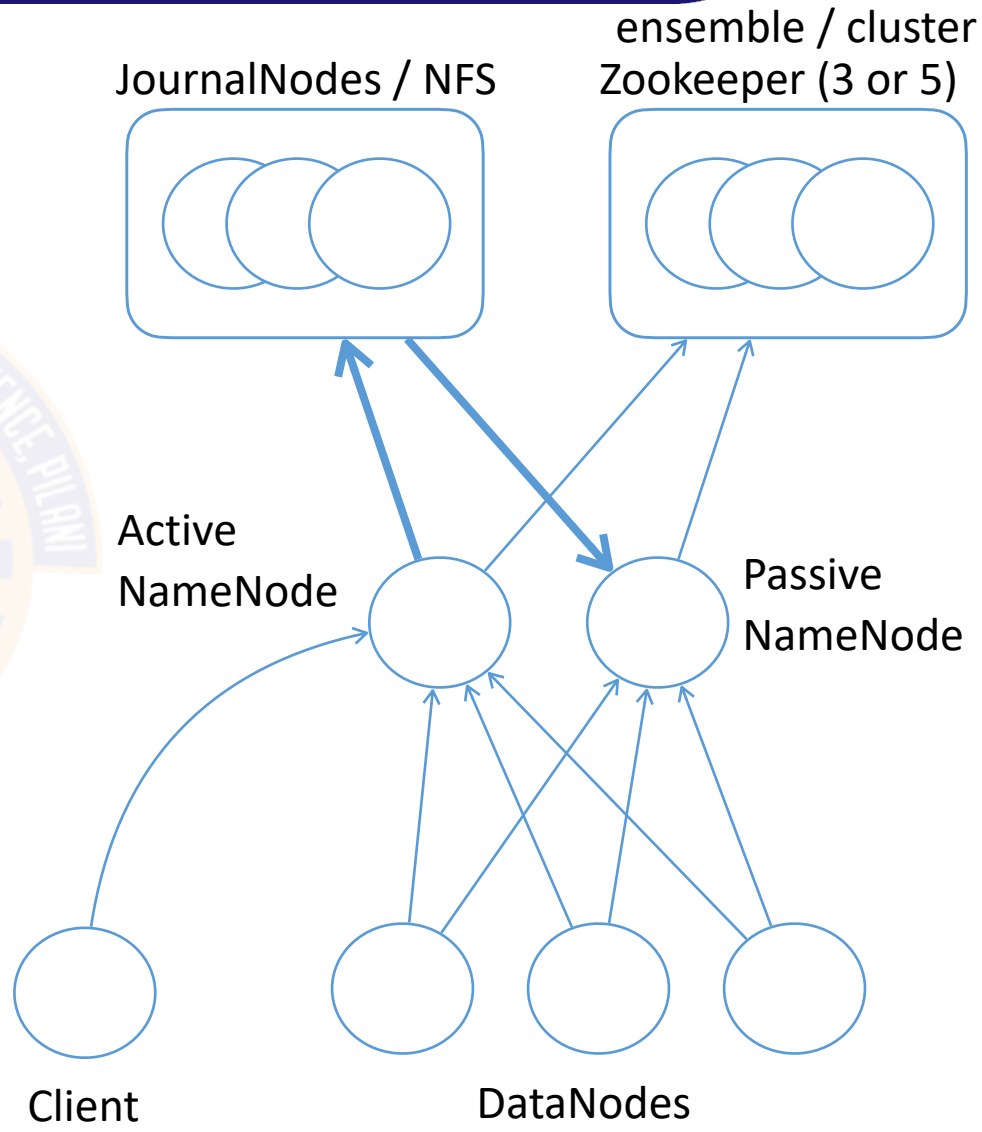
# Hadoop 2: Introduction of Secondary NameNode

- In the case of failure of NameNode,
  - ✓ The secondary NameNode can be configured **manually** to bring up the cluster
  - ✓ But it **does not record any real-time changes** that happen to the HDFS metadata
- The Secondary NameNode constantly reads all the file systems and metadata from the RAM of the NameNode (snapshot) and writes to its local file system.
- It is responsible for combining the EditLogs with FsImage from the NameNode.
- It downloads the EditLogs from the NameNode at regular intervals and applies to FsImage.
- Hence, Secondary NameNode performs regular checkpoints in HDFS. Therefore, it is also called **CheckpointNode**.
- After recover from a failure, the new FsImage is copied back to the NameNode, which is used whenever the NameNode is started the next time.



# HA configuration of NameNode

- Active-Passive configuration can also be setup with a standby NameNode
- Can use a Quorum Journal Manager (QJM) or NFS to maintain shared state
- DataNodes send **heartbeats and updates to both NameNodes**.
- Writes to JournalNodes only happens via Active NameNode - avoids "split brain" scenario of network partitions
- Standby reads from JournalNodes to keep updated on state as well as latest updates from DataNodes
- **Zookeeper session** may be used for failure detection and election of new Active



# Other robustness mechanisms

- **Types of failures** - DataNode, NameNode failures and network partitions
- **Heartbeat** from DataNode to NameNode for handling DN failures
  - ✓ When data node heartbeat times out (10min) NameNode updates state and starts pointing clients to other replicas.
  - ✓ Timeout (10min) is high to avoid replication storms but can be set lower especially if clients want to read recent data and avoid stale replicas.
- **Cluster rebalancing** by keeping track of RF per block and node usage
- **Checksums** stored in NameNode for blocks written to DataNodes to check data integrity on corruption on node / link and software bugs

# Communication

- TCP/IP at network level
- RPC abstraction on HDFS specific protocols
  - Clients talk to HDFS using **Client protocol**
  - DataNode and NameNode talk using **DataNode protocol**
- RPC is always **initiated by DataNode to NameNode** and not vice-versa
  - For better fault tolerance and NameNode state maintenance

# Topics for today

- Hadoop architecture overview
  - ✓ Components
  - ✓ Hadoop 1 vs Hadoop 2
- **HDFS**
  - ✓ Architecture
  - ✓ Robustness
  - ✓ **Blocks and replication strategy**
  - ✓ Read and write operations
  - ✓ File formats
  - ✓ Commands

# Blocks in HDFS

- HDFS stores each file as blocks which are scattered throughout the Apache Hadoop cluster.
- The default size of each block is 128 MB in Apache Hadoop 2.x (64 MB in Apache Hadoop 1.x) which you can configure as per your requirement.
- It is not necessary that in HDFS, each file is stored in exact multiple of the configured block size (128 MB, 256 MB etc.).
  - A file of size 514 MB can have 4 x 128MB and 1 x 2MB blocks
- Why large block of 128MB ?
  - HDFS is used for TB/PB size files and small block size will create too much meta-data
  - Spatial locality
  - Larger blocks will further reduce the "indexing" at block level, impact load balancing across nodes etc.

# How to see blocks of a file in HDFS - fsck

```
[root@centos-s-4vcpu-8gb-blr1-01 bds]# hdfs fsck /SalesJan2009.csv -files -blocks
21/06/12 20:46:46 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using
plicable
Connecting to namenode via http://localhost:50070/fsck?ugi=root&files=1&blocks=1&path=%2FSalesJan2009.csv
FSCK started by root (auth:SIMPLE) from /127.0.0.1 for path /SalesJan2009.csv at Sat Jun 12 20:46:47 IST 2021
/SalesJan2009.csv 123637 bytes, 1 block(s):  OK
0. BP-235233240-127.0.0.1-1618685260802:blk_1073741825_1001 len=123637 Live_repl=1

Status: HEALTHY
 Total size:    123637 B
 Total dirs:    0
 Total files:   1
 Total symlinks:                0
 Total blocks (validated):      1 (avg. block size 123637 B)
 Minimally replicated blocks:   1 (100.0 %)
 Over-replicated blocks:        0 (0.0 %)
 Under-replicated blocks:       0 (0.0 %)
 Mis-replicated blocks:         0 (0.0 %)
 Default replication factor:    1
 Average block replication:     1.0
 Corrupt blocks:                0
 Missing replicas:              0 (0.0 %)
 Number of data-nodes:          1
 Number of racks:               1
FSCK ended at Sat Jun 12 20:46:47 IST 2021 in 5 milliseconds


The filesystem under path '/SalesJan2009.csv' is HEALTHY
```

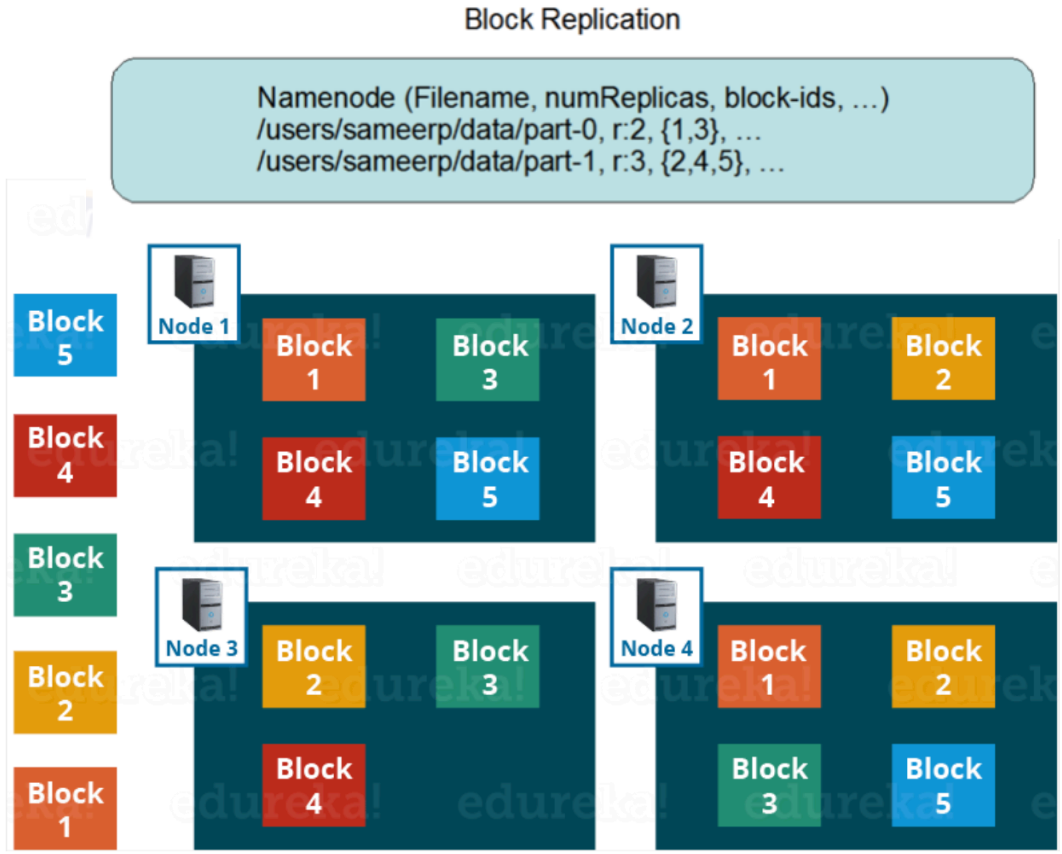
# HDFS on local FS

- Find / configure the root of HDFS in hdfs-site.xml - > dfs.data.dir property
  - e.g. $HADOOP_HOME/data/dfs/data/hadoop-${user.name}/current
- If you want to see the files in local FS that store blocks of HDFS :
  - cd to the HDFS root dir specified in dfs.data.dir
  - go inside the sub-dir with name you got from fsck command
  - navigate into further sub-directories to find the block files
  - All this mapping is stored on the NameNode to map HDFS files to blocks (local FS files) on DataNodes

```
[root@centos-s-4vcpu-8gb-blr1-01 subdir0]#
[root@centos-s-4vcpu-8gb-blr1-01 subdir0]# pwd
/home/hadoop/hadoopdata/hdfs/datanode/current/BP-235233240-127.0.0.1-1618685260802/current/finalized/subdir0/subdir0
[root@centos-s-4vcpu-8gb-blr1-01 subdir0]# ls -l
total 2712
-rw-r--r--. 1 root root 123637 Apr 18 01:18 blk_1073741825
-rw-r--r--. 1 root root    975 Apr 18 01:18 blk_1073741825_1001.meta
-rw-r--r--. 1 root root    661 Apr 18 01:20 blk_1073741832
-rw-r--r--. 1 root root     15 Apr 18 01:20 blk_1073741832_1008.meta
-rw-r--r--. 1 root root    352 Apr 18 01:20 blk_1073741833
-rw-r--r--. 1 root root     11 Apr 18 01:20 blk_1073741833_1009.meta
-rw-r--r--. 1 root root  40183 Apr 18 01:20 blk_1073741834
-rw-r--r--. 1 root root    323 Apr 18 01:20 blk_1073741834_1010.meta
-rw-r--r--. 1 root root 206080 Apr 18 01:20 blk_1073741835
-rw-r--r--. 1 root root   1619 Apr 18 01:20 blk_1073741835_1011.meta
-rw-r--r--. 1 root root    661 Apr 18 12:22 blk_1073741842
-rw-r--r--. 1 root root     15 Apr 18 12:22 blk_1073741842_1018.meta
-rw-r--r--. 1 root root    353 Apr 18 12:22 blk_1073741843
-rw-r--r--. 1 root root     11 Apr 18 12:22 blk_1073741843_1019.meta
```

# Replica Placement Strategy - with Rack awareness

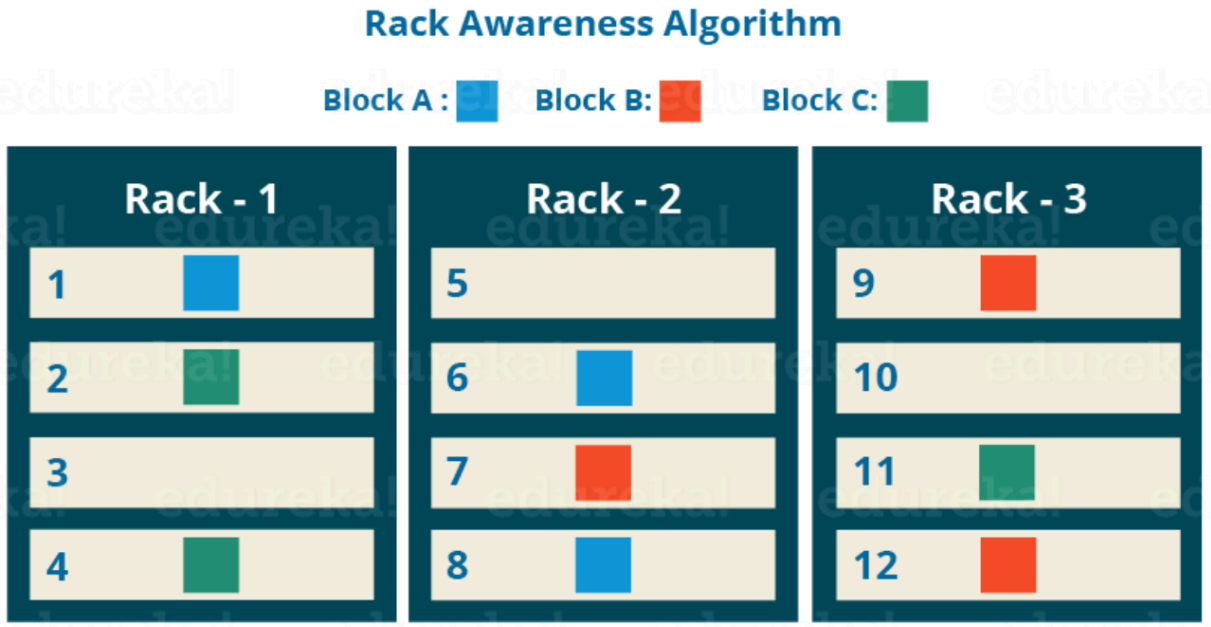
- First replica is placed on the same node as the client
- Second replica is placed on a node that is present on different rack
- Third replica is placed on same rack as second but on a different node
- Putting each replica on a different rack is expensive write operation
- **For replicas > 3, nodes are randomly picked for 4th replica without violating upper limit per rack as (replicas-1) / racks + 2.**
- **Total replicas <= #DataNodes with no 2 replicas on same DN**
- Once the replica locations are set, pipeline is built
- Shows good reliability
- NameNode collects block report from DataNodes to balance the blocks across nodes and control over/under replication of blocks



# Why rack awareness ?

- **To improve the network performance:** The communication between nodes residing on different racks is directed via switch. In general, you will find *greater network bandwidth* between machines in the same rack than the machines residing in different rack. So, the **Rack Awareness helps you to have reduce write traffic in between different racks** and thus providing a better write performance. Also, you will be gaining increased read performance because you are using the bandwidth of multiple racks.

- **To prevent loss of data:** We **don't have to worry about the data even if an entire rack fails** because of the switch failure or power failure. And if you think about it, it will make sense, as it is said that *never put all your eggs in the same basket.*



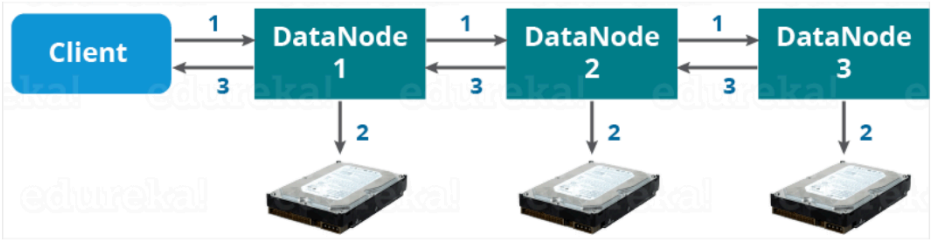
# Topics for today

- Hadoop architecture overview
  - ✓ Components
  - ✓ Hadoop 1 vs Hadoop 2
- **HDFS**
  - ✓ Architecture
  - ✓ Robustness
  - ✓ Blocks and replication strategy
  - ✓ **Read and write operations**
  - ✓ File formats
  - ✓ Commands

# HDFS data writes

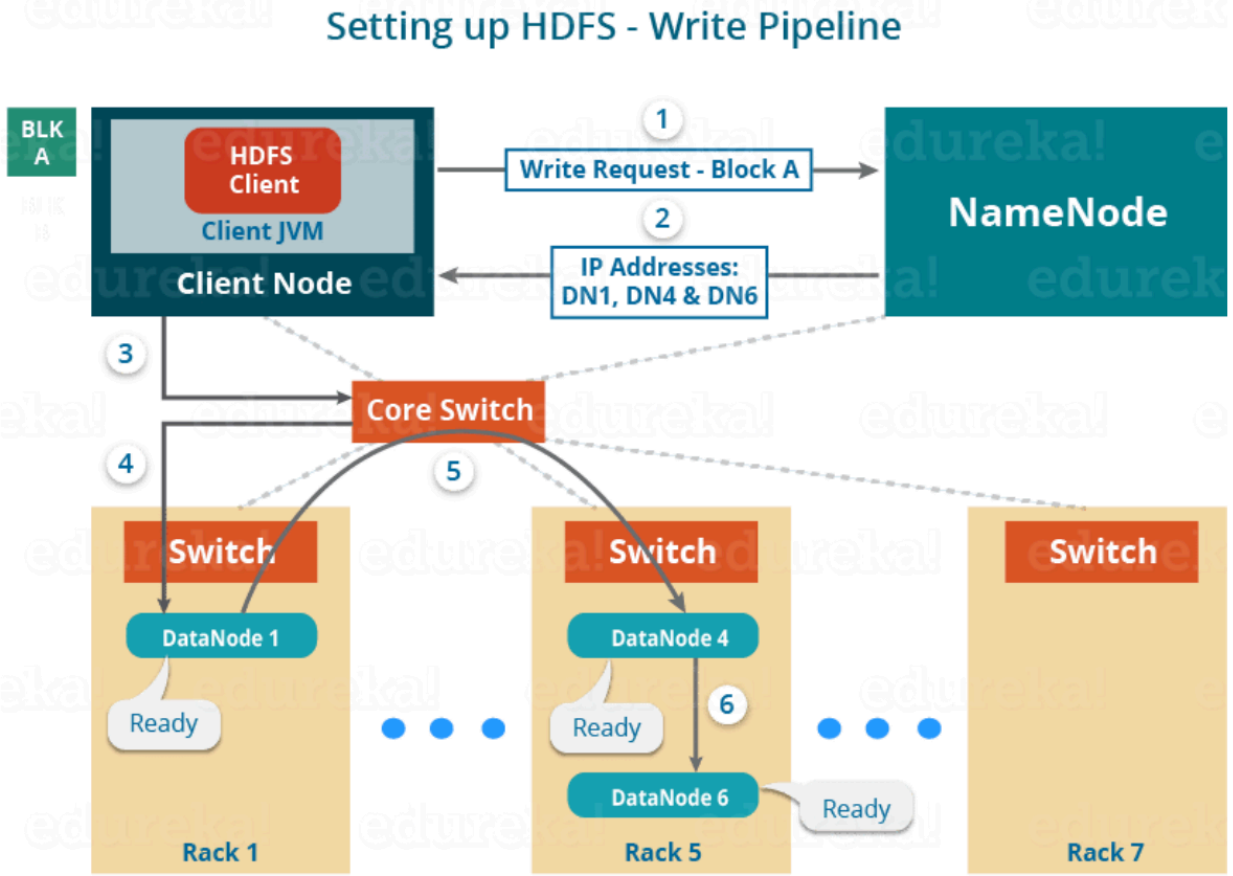
Now, the following protocol will be followed whenever the data is written into HDFS:

- HDFS client contacts NameNode for Write Request against the two blocks, say, Block A & Block B.
- NameNode grants permission to client with IP addresses of the DataNodes to copy blocks
- Selection of DataNodes is randomized but factoring in availability, RF, and rack awareness
- For 3 copies, 3 unique DNs needed, if possible, for each block.
  - For Block A, list A = {DN1, DN4, DN6}
  - For Block B, set B = {DN3, DN7, DN9}
- Each block will be copied in three different DataNodes to maintain the replication factor consistent throughout the cluster.
- Now the whole data copy process will happen in three stages:
  1. **Set up of Pipeline**
  2. **Data streaming and replication**
  3. **Shutdown of Pipeline (Acknowledgement stage)**



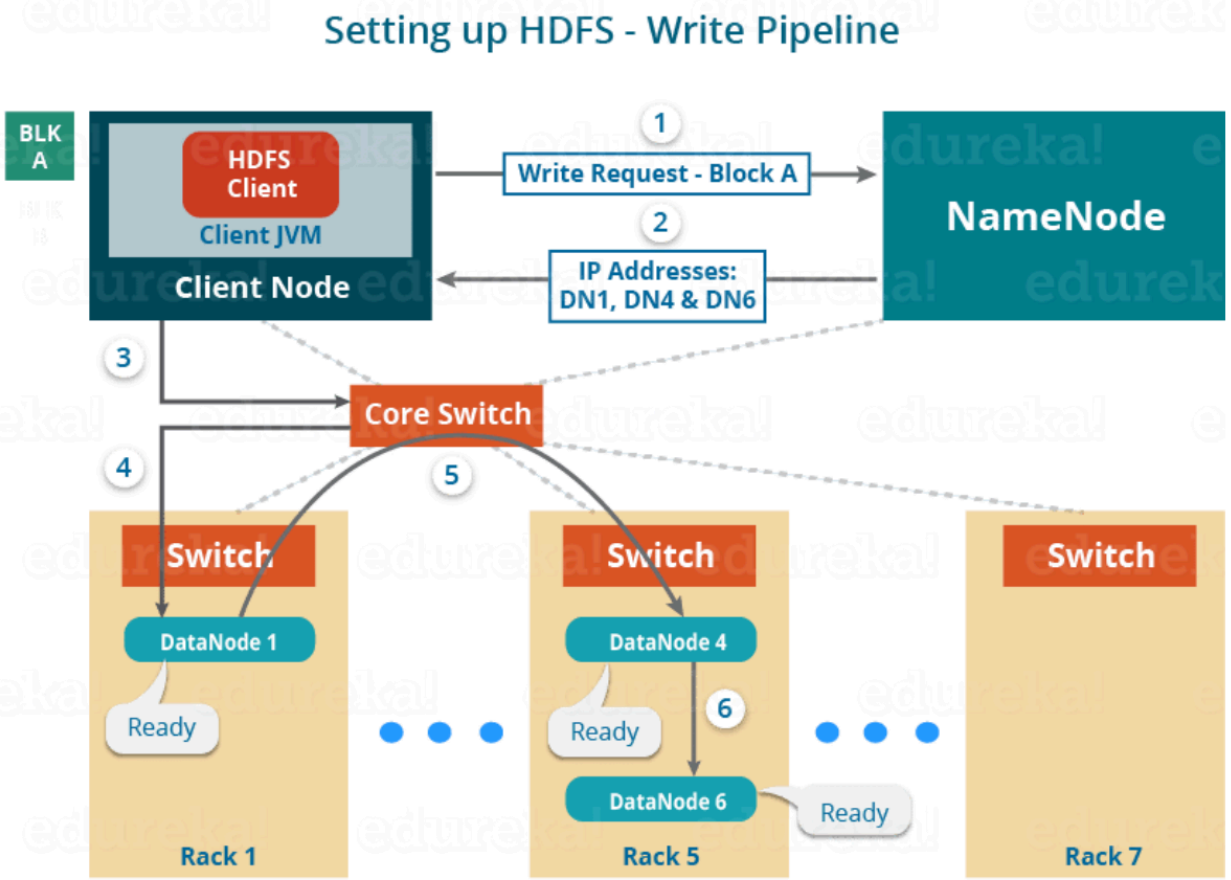
# HDFS Write: Step 1. Setup pipeline

Client creates a pipeline for each of the blocks by connecting the individual DataNodes in the respective list for that block. Let us consider Block A. The list of DataNodes provided by the NameNode is DN1, DN4, DN6



# HDFS Write: Step 1. Setup pipeline for a block

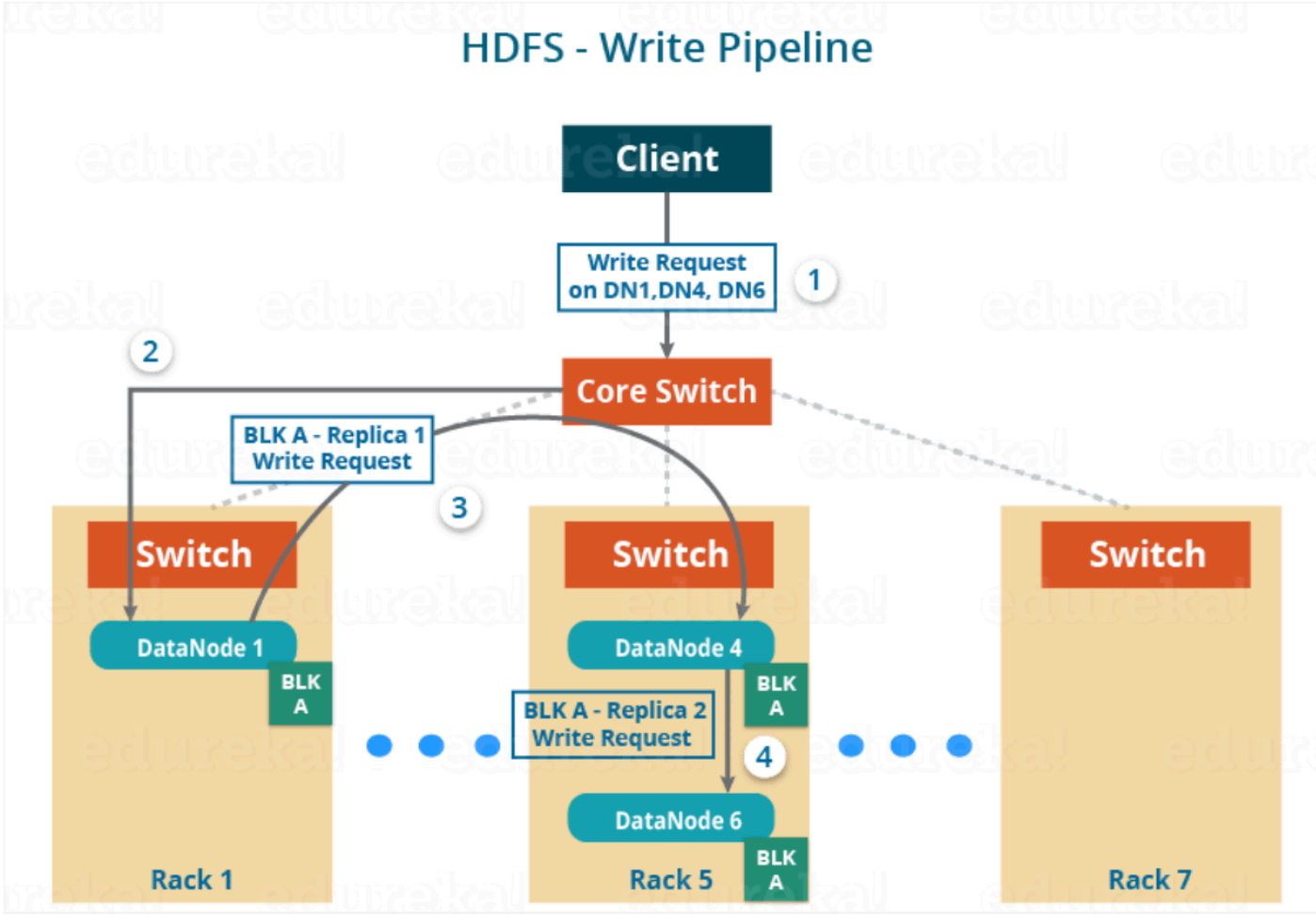
1. Client chooses the first DataNode (DN1) and will establish a TCP/IP connection.
2. Client informs DN1 to be ready to receive the block.
3. Provides IPs of next two DNs (4, 6) to DN1 for replication.
4. The DN1 connects to DN4 and informs it to be ready and gives IP of DN6. DN4 asks DN6 to be ready for data.
5. Ack of readiness follows the reverse sequence, i.e. from the DN6 to DN4 to DN1.
6. At last DN1 will inform the client that all the DNs are ready and a pipeline will be formed between the client, DataNode 1, 4 and 6.
7. Now pipeline set up is complete and the client will finally begin the data copy or streaming process.



# HDFS Write: Step 2. Data streaming

Client pushes the data into the pipeline.

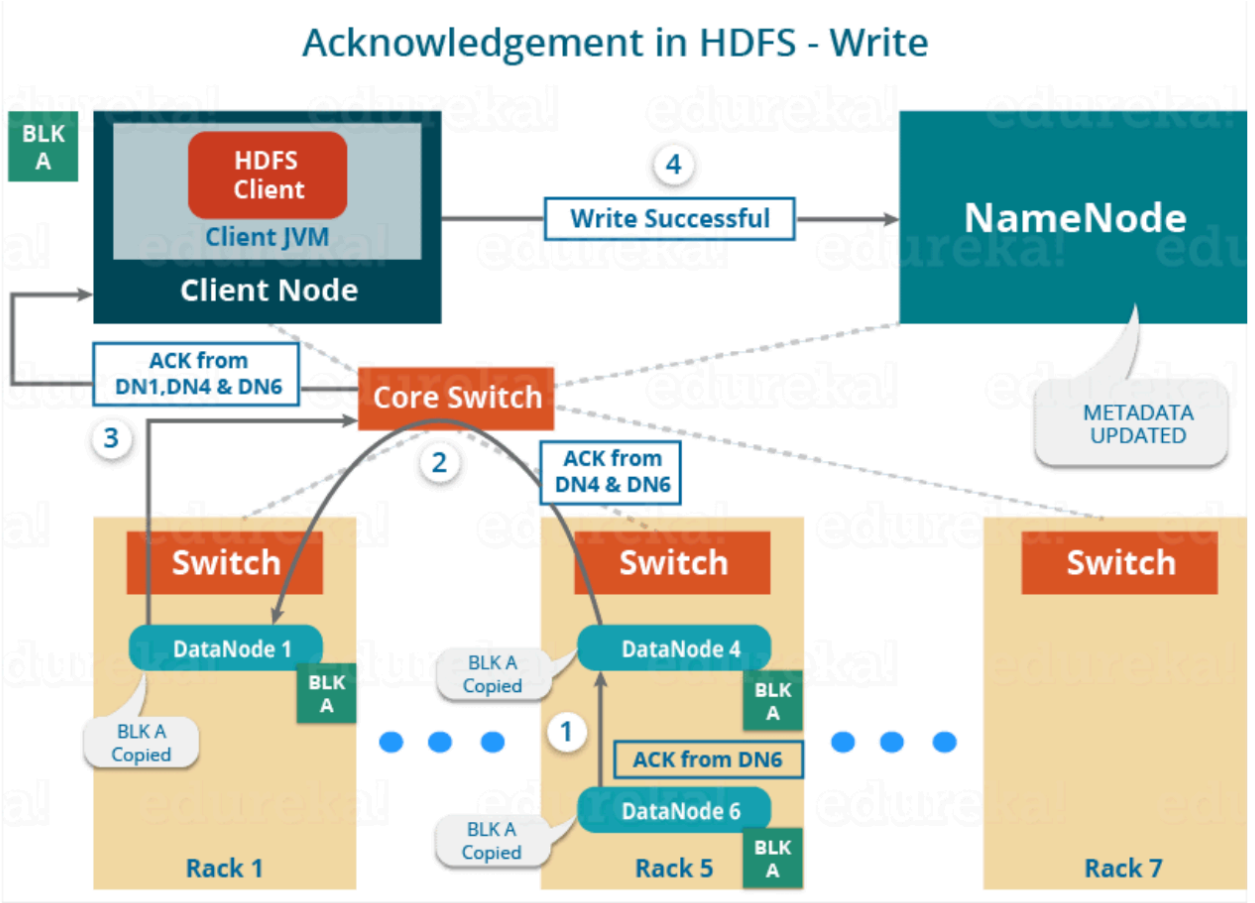
1. Once the block has been written to DataNode 1 by the client, DataNode 1 will connect to DataNode 4.
2. Then, DataNode 1 will push the block in the pipeline and data will be copied to DataNode 4.
3. Again, DataNode 4 will connect to DataNode 6 and will copy the last replica of the block.



# HDFS Write: Step 3. Shutdown pipeline / ack

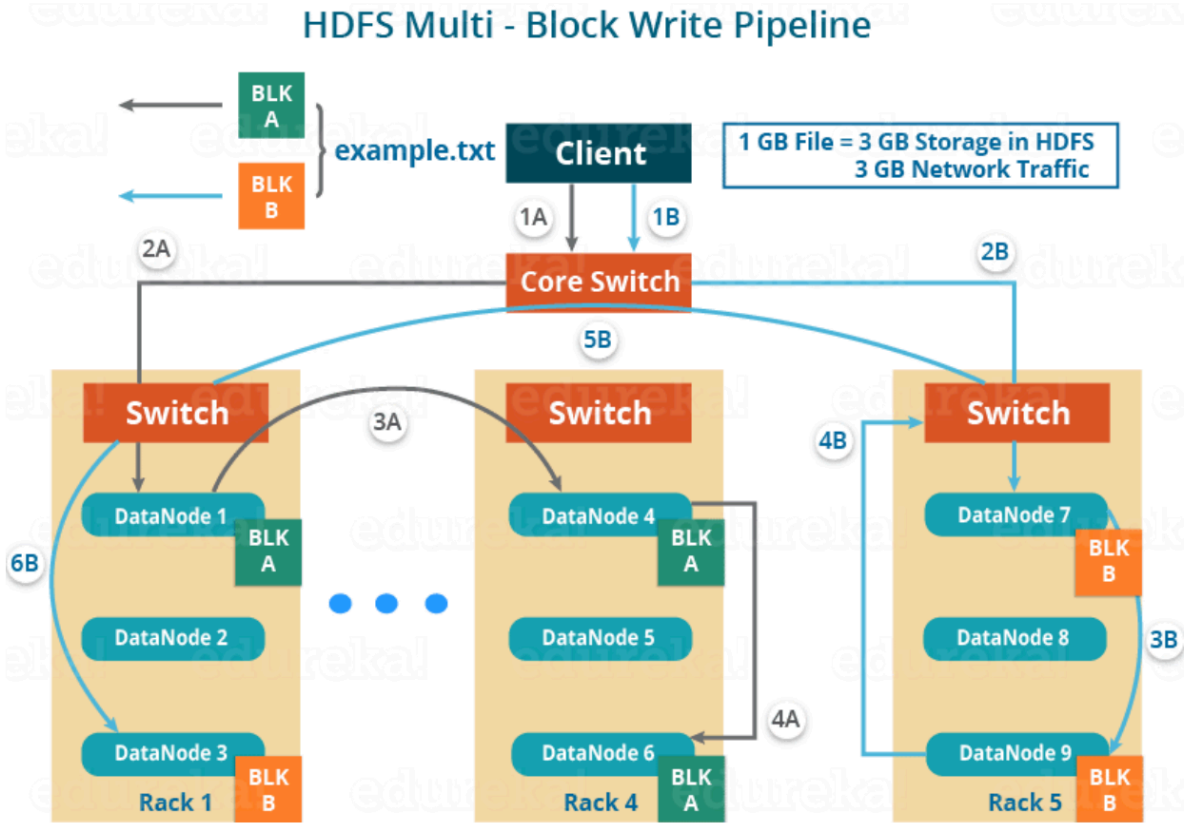
Block is now copied to all DNs. Client and NameNode need to be updated. Client needs to close pipeline and end TCP session.

1. Acknowledgement happens in the reverse sequence i.e. from DN 6 to 4 and then to 1.
2. DN1 pushes three acknowledgements (including its own) into pipeline and client.
3. Client informs NameNode that data has been written successfully.
4. NameNode updates metadata.
5. Client shuts down the pipeline.



# Multi-block writes

- The client will copy Block A and Block B to the first DataNode **simultaneously**.
- Parallel pipelines for each block
- Pipeline process for a block is same as discussed.
- E.g. 1A, 2A, 3A, … and 1B, 2B, 3B, … work in parallel



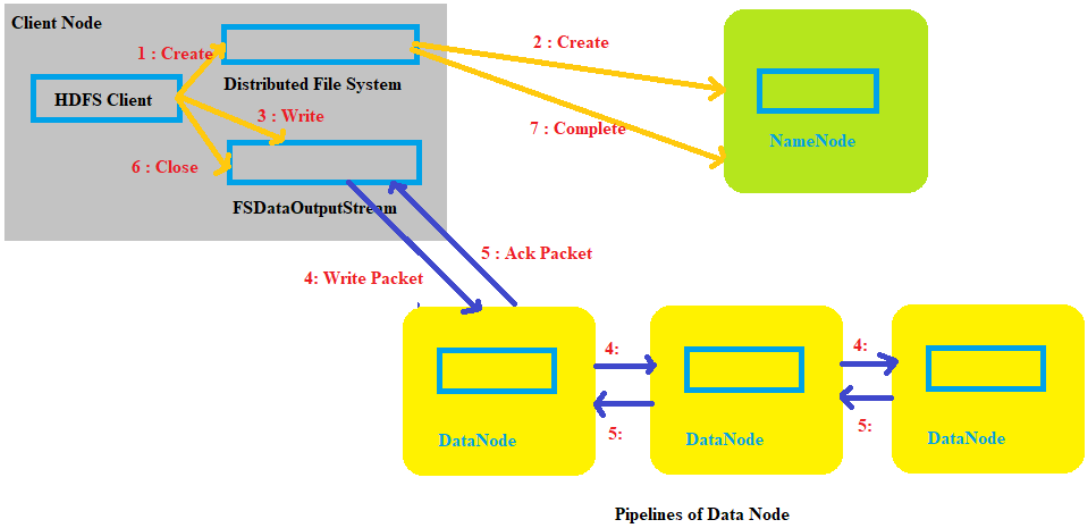
# Sample write code

```
public static void writeFileToHDFS() throws IOException {
        Configuration configuration = new Configuration();
        configuration.set("fs.defaultFS", "hdfs://<host>:9000");
        FileSystem fileSystem = FileSystem.get(configuration);
        //Create a path
        String fileName = "sample file.txt";
        Path hdfsWritePath = new Path("/sample/" + fileName);
        FSDataOutputStream fsDataOutputStream =
              fileSystem.create(hdfsWritePath,true);

        BufferedWriter bufferedWriter = new BufferedWriter(new
              OutputStreamWriter(fsDataOutputStream, StandardCharsets.UTF_8));
        bufferedWriter.write("Sample text");
        bufferedWriter.newLine();
        bufferedWriter.close();
        fileSystem.close();
    }
```

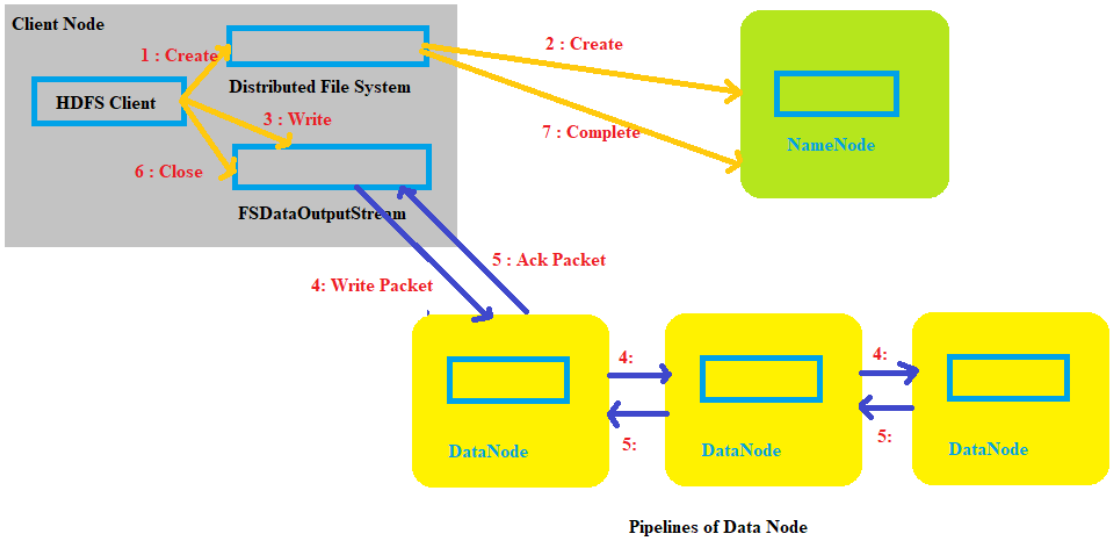
# HDFS Create / Write - Call sequence in code

1) Client calls **create()** on **FileSystem** to create a file
    1) RPC call to **NameNode** happens through **FileSystem** to create new file.
    2) **NameNode** performs checks to create a new file. Initially, **NameNode** creates a file without associating any data blocks to the file.
    3) The **FileSystem.create()** returns an **FSDataOutputStream** to client to perform write.
2) Client creates a **BufferedWriter** using **FSDataOutputStream** to write data to a pipeline
    1) Data is split into packets by **FSDataOutputStream**, which is then written to the internal queue.
    2) **DataStreamer** consumes the data queue
    3) **DataStreamer** requests **NameNode** to allocate new blocks by selecting a list of suitable **DataNodes** to store replicas. This is pipeline.
    4) **DataStreamer** streams packets to first **DataNode** in the pipeline.



# HDFS Create / Write - Call sequence in code

3) The first **DataNode** stores packet and forwards it to Second **DataNode** and then Second node transfer it to Third **DataNode**.
4) **FSDataOutputStream** also manages a **"Ack queue"** of packets that are waiting for the acknowledgement by **DataNodes**.
5) A packet is removed from the queue only if it is acknowledged by all the **DataNodes** in the pipeline
6) When the client finishes writing to the file, it calls **close()** on the stream
7) This flushes all the remaining packets to **DataNode** pipeline and waits for relevant acknowledgements before communicating the **NameNode** to inform the client that the writing of the file is complete.

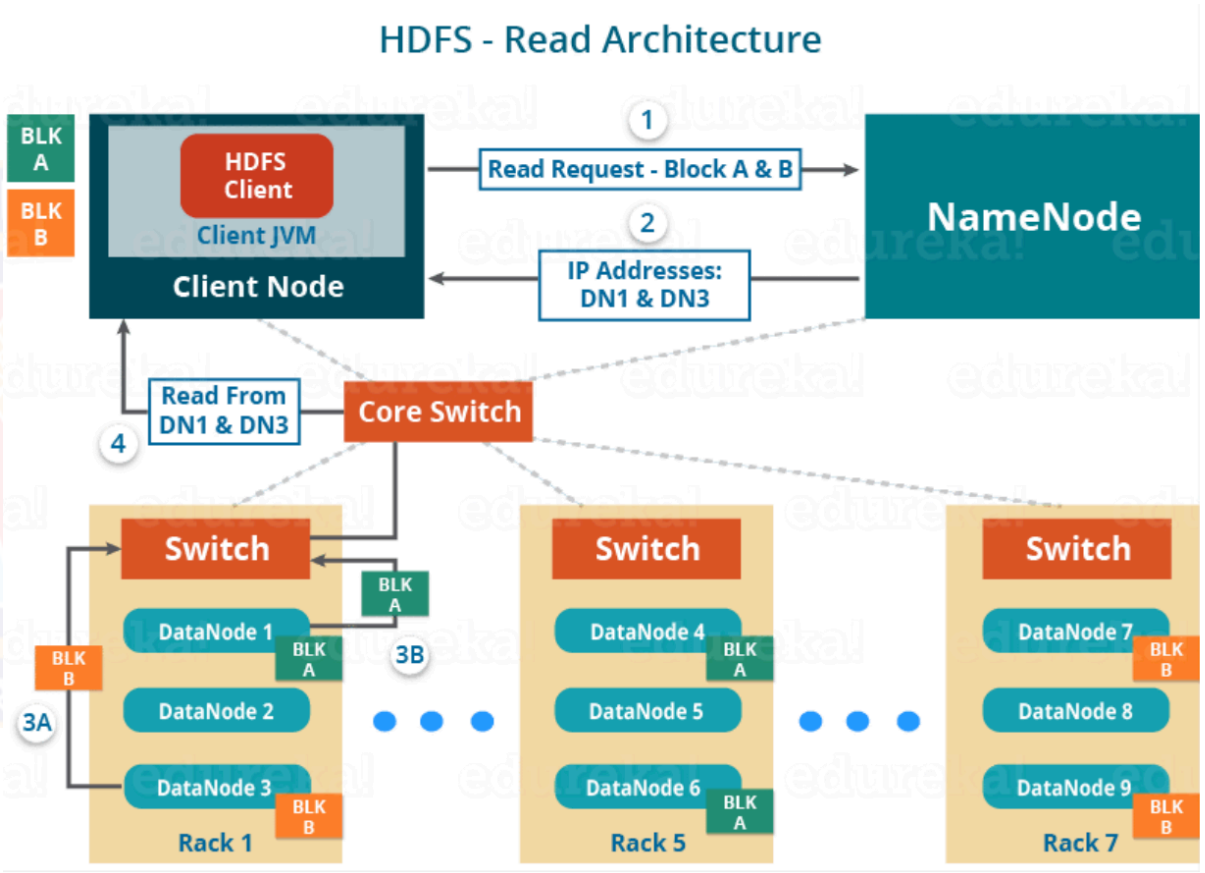


# HDFS Read

1. Client contacts NameNode asking for the block metadata for a file
2. NameNode returns list of DNs where each block is stored
3. Client connects to the DNs where blocks are stored
4. The client starts reading data parallel from the DNs (e.g. Block A from DN1, Block B from DN3)
5. Once the client gets all the required file blocks, it will combine these blocks to form a file.

**How are blocks chosen by NameNode ?**

While serving read request of the client, HDFS *selects the replica which is closest to the client*. This reduces the read latency and the bandwidth consumption. Therefore, that replica is selected which resides on the *same rack* as the reader node, if possible.

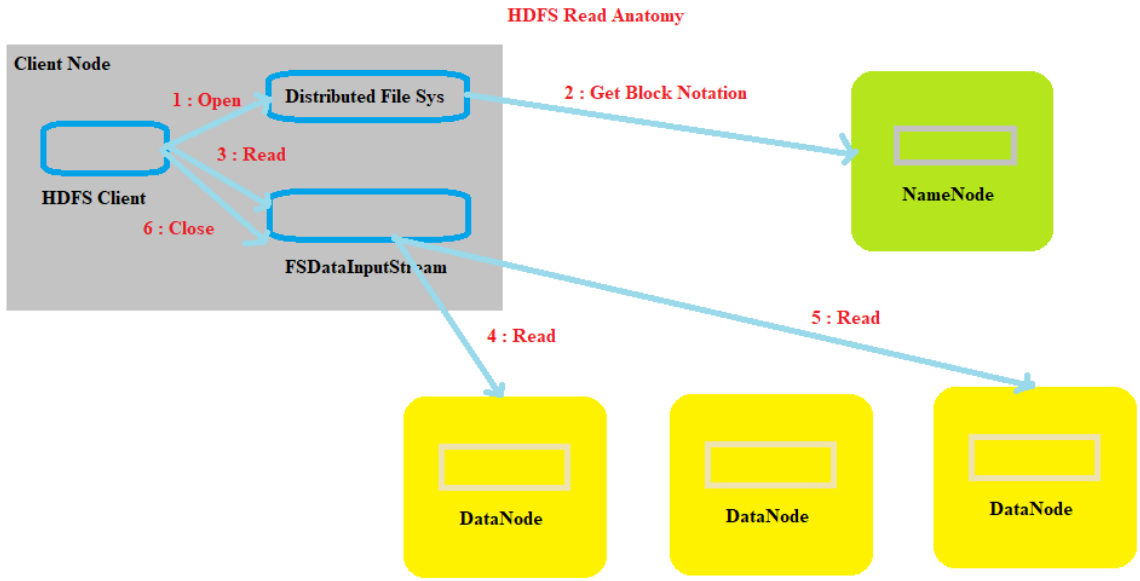


# Sample read code

```
public static void readFileFromHDFS() throws IOException
      Configuration conf = new Configuration();
      FileSystem fs = FileSystem.get(conf);
      Path inFile = new Path(args[0])
      FSDataInputStream in = fs.open(inFile);
      OutputStream out = System.out;
      byte buffer[] = new byte[256];
      int bytesRead = 0;
      while ((bytesRead = in.read(buffer)) > 0) {
          out.write(buffer, 0, bytesRead);
     }
     in.close();
     out.close();
}
```

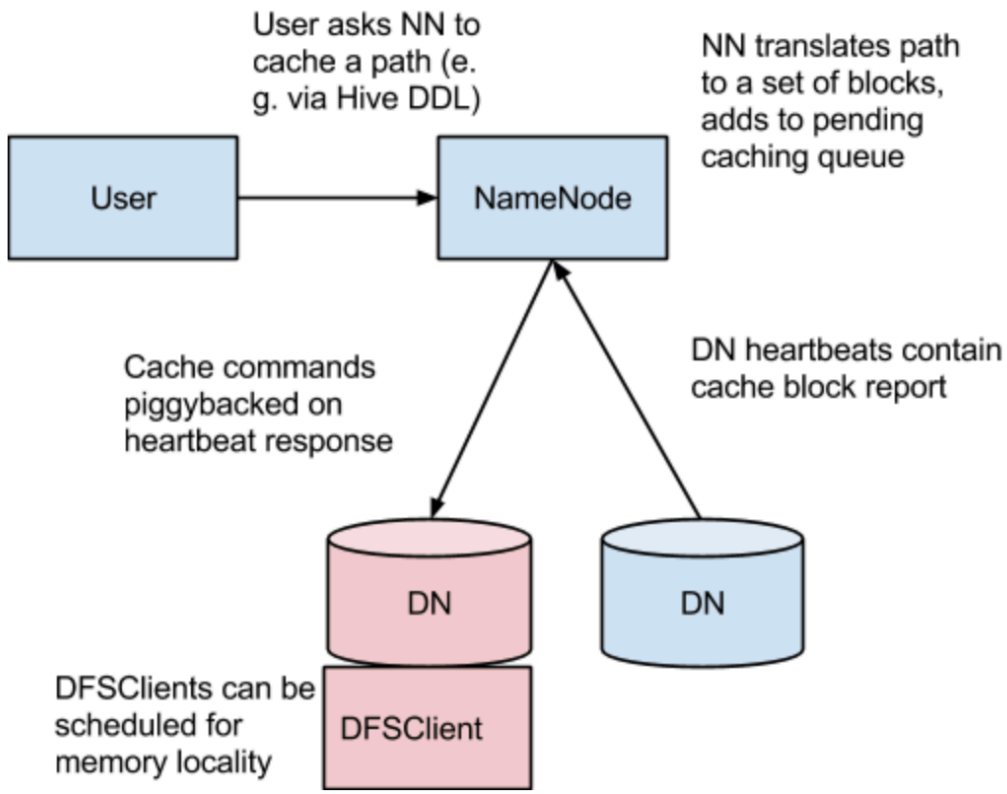
# HDFS Read - Call sequence in code

1) Client opens the file that it wishes to read from by calling **open()** on **FileSystem**
    1) **FileSystem** communicates with **NameNode** to get location of data blocks.
    2) **NameNode** returns the addresses of **DataNodes** on which blocks are stored.
    3) **FileSystem** returns **FSDataInputStream** to client to read from file.
2) Client then calls **read()** on the stream, which has addresses of **DataNodes** for first few blocks of file, connects to the closest **DataNode** for the first block in file
    1) Client calls **read()** repeatedly to stream the data from DataNode
    2) When end of block is reached, stream closes the connection with **DataNode**.
    3) The stream repeats the steps to find the best **DataNode** for the next blocks.
3) When the client completes the reading of file, it calls **close()** on the stream to close the connection.



# Read optimizations

- Short-circuit reads can also be made by client to **local file bypassing DataNode** using /dev/shm for better performance.
- Client can ask certain **HDFS paths to be cached**
  - ✓ NameNode asks DNs to cache corresponding blocks and send cache block report along with heartbeat
- Clients can be scheduled for data locality so that it can use local reads or caches better



# Security

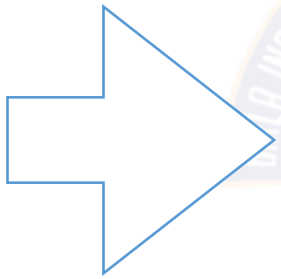
- POSIX style file permissions
- Choice of transparent encryption/decryption
  - ✓ HDFS encryption is between file system and DB level encryption
- Create hierarchical encryption zones so that any file within the zone (a path) has the same key
- Hadoop Key Management Server (KMS) does the following
  - ✓ Provides access to stored Encryption Zone Keys
  - ✓ Create encrypted data encryption keys (EDEK) for storage by NameNode
  - ✓ Decrypting EDEK for use by clients

# Topics for Session 6 (18 Dec)

- Hadoop architecture overview
  - ✓ Components
  - ✓ Hadoop 1 vs Hadoop 2
- **HDFS**
  - ✓ Architecture
  - ✓ Robustness
  - ✓ Blocks and replication strategy
  - ✓ Read and write operations
  - ✓ **File formats**
  - ✓ Commands

# File formats

- Text (JSON, CSV, ..)
- AVRO
- Parquet
- RC
- ORC
- Sequence



- Many ways to evaluate formats
  - ✓ Write performance
  - ✓ Read performance
  - ✓ Partial read performance
  - ✓ Block compression
  - ✓ Columnar support
  - ✓ Schema change

# File formats - text based

- Text-based (JSON, CSV …)
  - ✓ Easily splittable
  - ✓ Can’t split compressed files
    - so large Map tasks because you have to give the entire file to a Map task
  - ✓ Simplest to start putting in structured / semi-structured data

# File formats - sequence files

- A flat file consisting of binary key/value pairs
- Extensively used in MapReduce as input/output formats as well as internal temporary outputs of maps
- 3 types
  1. Uncompressed key/value records.
  2. Record compressed key/value records - only 'values' are compressed here.
  3. Block compressed key/value records - both keys and values are collected in configurable 'blocks' and compressed. Sync markers added for random access and splitting.
- Header includes information about :
  - key, value class
  - whether compression is enabled and whether at block level
  - compressor codec used

ref: https:/cwiki.apache.org/confluence/display/HADOOP2/SequenceFile

# Example: Storing images in HDFS

```
public class ImageToSeq {
   public static void main(String args[]) throws Exception {

      Configuration confHadoop = new Configuration();
       ... details of conf ...
      FileSystem fs = FileSystem.get(confHadoop);
      Path inPath = new Path("/mapin/1.png");
      Path outPath = new Path("/mapin/11.png");
      FSDataInputStream in = null;
      Text key = new Text();
      BytesWritable value = new BytesWritable();
      SequenceFile.Writer writer = null;
      try{
         in = fs.open(inPath);
         byte buffer[] = new byte[in.available()];
         in.read(buffer);
         writer = SequenceFile.createWriter(fs, confHadoop, outPath,
                              key.getClass(),value.getClass());
         writer.append(new Text(inPath.getName()), new BytesWritable(buffer));
      }catch (Exception e) {
         System.out.println("Exception MESSAGES = "+e.getMessage());
      }
      finally {
         IOUtils.closeStream(writer);
         System.out.println("last line of the code....!!!!!!!!!!");
      }
   }
}
```

- We want to store a bunch of images as key-value pairs, maybe attach some meta-data as well
- Files are stored as binary format, e.g. sequence file
- We can apply some image processing on the files, as well as use meta-data to retrieve specific images

Ref: https://stackoverflow.com/questions/16546040/store-images-videos-into-hadoop-hdfs

# File formats - Optimized Row Columnar (ORC) *

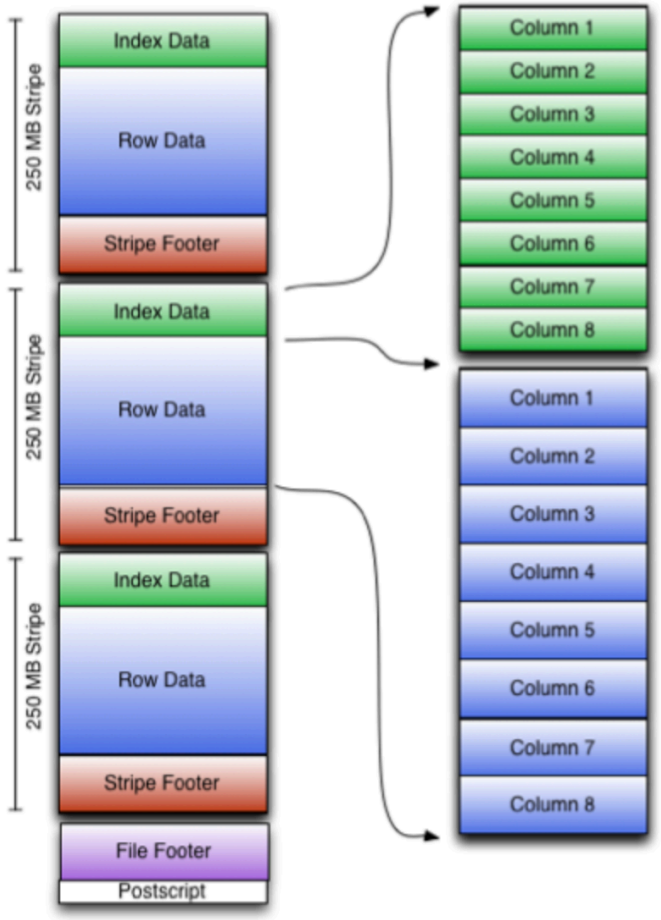
Improves performance when **Hive** is reading, writing, and processing data

**Index data** used to skip rows
Includes min/max for each column and their row positions

**Row data** used for table scans

```
create table Addresses (
  name string,
  street string,
  city string,
  state string,
  zip int
) stored as orc tblproperties ("orc.compress"="ZLIB");
```

| Key | Default | Notes |
|---|---|---|
| orc.compress | ZLIB | high level compression (one of NONE, ZLIB, SNAPPY) |
| orc.compress.size | 262,144 | number of bytes in each compression chunk |
| orc.stripe.size | 67,108,864 | number of bytes in each stripe |
| orc.row.index.stride | 10,000 | number of rows between index entries (must be >= 1000) |
| orc.create.index | true | whether to create row indexes |
| orc.bloom.filter.columns | "" | comma separated list of column names for which bloom filter should be created |
| orc.bloom.filter.fpp | 0.05 | false positive probability for bloom filter (must >0.0 and <1.0) |




* more in Hive session

ref: https://cwiki.apache.org/confluence/display/hive/languagemanual+orc

# File formats - Parquet

- Columnar format
- Pros
  - ✓ Good for compression because data in a column tends to be similar
    - Support block level compression as well as file level
  - ✓ Good query performance when query is for specific columns
  - ✓ Compared to ORC - more flexible to add columns
  - ✓ Good for Hive and Spark workloads if working on specific columns at a time
- Cons
  - ✓ Expensive write due to columnar
    - So if use case is more about reading entire rows, then not a good choice
    - Anyway - Hadoop systems are more about write once and read many times - so read performance is paramount
- Note: Avro is another option for workloads with full row scans because it is row major storage

# Topics for today

- Hadoop architecture overview
  - ✓ Components
  - ✓ Hadoop 1 vs Hadoop 2
- **HDFS**
  - ✓ Architecture
  - ✓ Robustness
  - ✓ Blocks and replication strategy
  - ✓ Read and write operations
  - ✓ File formats
  - ✓ **Commands**

# Basic command reference

- list files in the path of the file system
  - ✓ **hadoop fs -ls <path>**
- alters the permissions of a file where <arg> is the binary argument e.g. 777
  - ✓ **hadoop fs -chmod <arg> <file-or-dir>**
- change the owner of a file
  - ✓ **hadoop fs -chown <owner>:<group> <file-or-dir>**
- make a directory on the file system
  - ✓ **hadoop fs -mkdir <path>**
- copy a file from the local storage onto file system
  - ✓ **hadoop fs -put <local-origin> <destination>**
- copy a file to the local storage from the file system
  - ✓ **hadoop fs -get <origin> <local-destination>**
- similar to the put command but the source is restricted to a local file reference
  - ✓ **hadoop fs -copyFromLocal <local-origin> <destination>**
- similar to the get command but the destination is restricted to a local file reference
  - ✓ **hadoop fs -copyToLocal <origin> <local-destination>**
- create an empty file on the file system
  - ✓ **hadoop fs -touchz**
- copy files to stdout
  - ✓ **hadoop fs -cat <file>**

# More HDFS commands - config and usage

Get configuration data in general, about name nodes, about any specific attribute

- **hdfs getconf**
  - return various configuration settings in effect
- **hdfs getconf -namenodes**
  - returns namenodes in the cluster
- **hdfs getconf -confkey <a.value>**
  - return the value of a particular setting (e.g. dfs.replication)
- **hdfs dfsadmin -report**
  - find out how much disk space us used, free, under-replicated, etc.

# Summary

- High level architecture of Hadoop 2 and differences with earlier Hadoop 1
- HDFS
  - ✓ Architecture
  - ✓ Read and write flows
  - ✓ File formats
  - ✓ Commands commonly used
  - ✓ Additional reading about HDFS: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html


# Next Session:

Hadoop MapReduce and YARN