BITS Pilani
Pilani | Dubai | Goa | Hyderabad

DSECL ZG 522: Big Data Systems

Session 6: Hadoop Architecture and Distributed Filesystem

Janardhanan PS

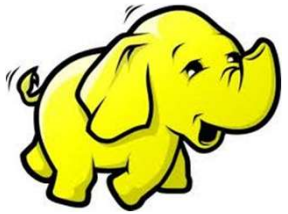
janardhanan.ps@wilp.bits-pilani.ac.in

Topics for today

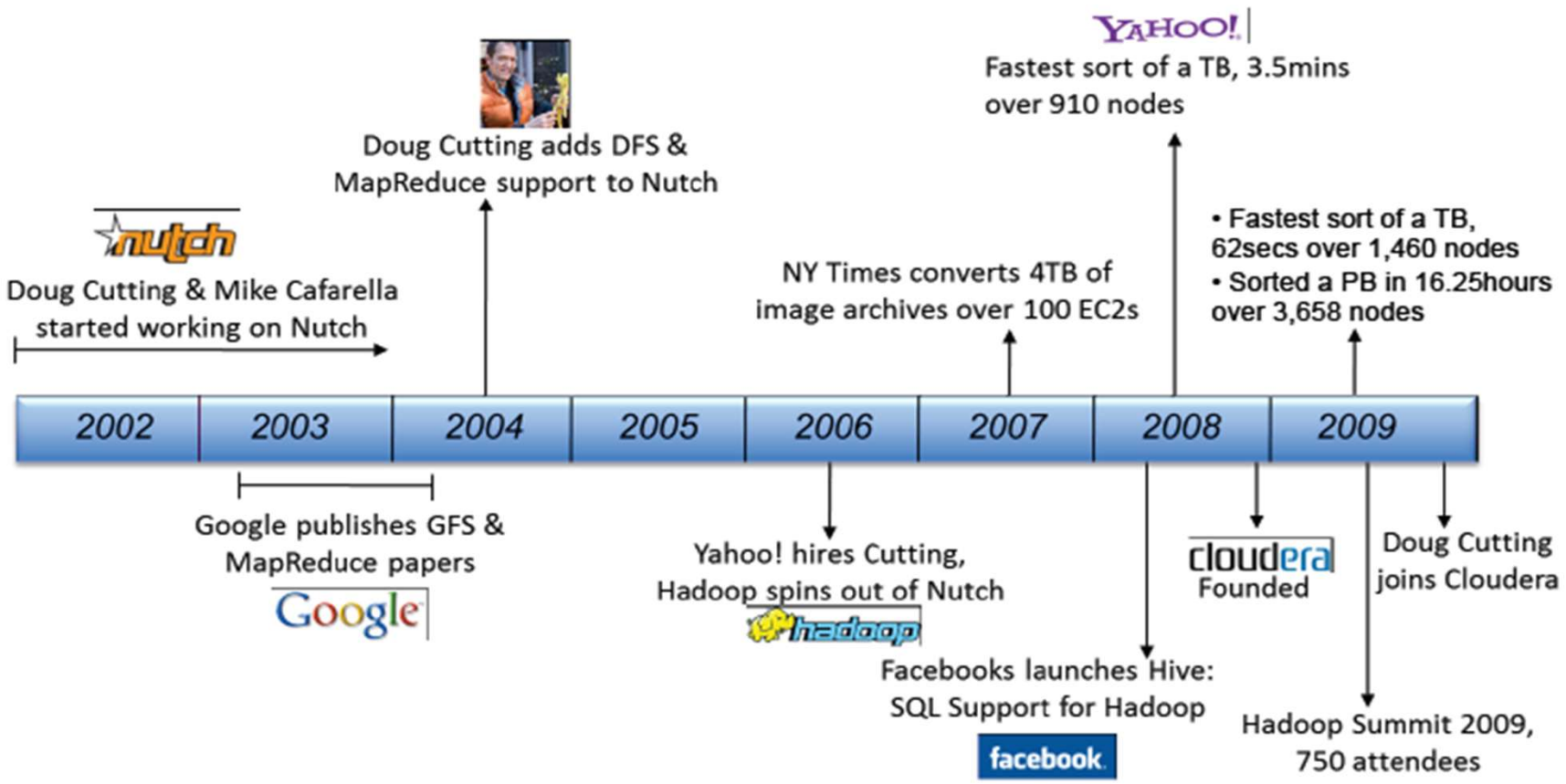
- **Hadoop architecture overview**
  - ✓ Components
  - ✓ Hadoop 2
- HDFS
  - ✓ Architecture
  - ✓ Robustness
  - ✓ Blocks and replication strategy
  - ✓ Read and write operations
  - ✓ Big Data File formats
  - ✓ Commands

The Hadoop Project

- Originally based on 2 papers published by Google in 2003 and 2004
  1. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. Appeared in 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003.
  2. Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. Appeared in OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December 2004.
- Hadoop started in 2006 at Yahoo!
- Top level Apache Foundation project
- Large, active user base, user groups
- Very active development, strong development team

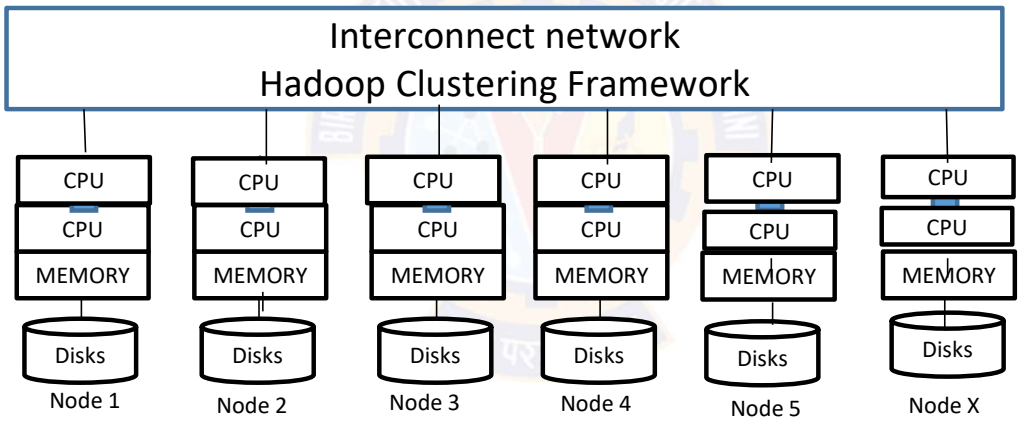


History of Hadoop



Hadoop High level Concept

- Shared nothing cluster – scales out !
- Split and distribute data across many machines (sharding)
  - ✓ Fault tolerant - keeps multiple copies of data (typically 3)



- Distribute data processing
  - ✓ sequential data scanning happens concurrently on multiple nodes
  - ✓ profit from data locality => high throughput between storage and CPU and Memory

What is Hadoop ?

- Hadoop is a complete, open-source ecosystem for capturing, organizing, storing, searching, sharing, analyzing and otherwise processing disparate data sources :
  - Structured
  - Semi-structured
  - Unstructured
- High Scalability
- High Fault-tolerance
- Impressive price/performance ratio :
  - ✓ Hadoop uses commodity hardware which results in a remarkably low cost.

Hadoop – Set of independently deployable components

Zookeeper
Coordination of distributed systems

Kafka
Data streaming

Flume
Data collector

Impala
Low latency SQL

YARN
Cluster resource manager

Spark
Large scale data proceesing

MapReduce

Oozie
Workflow manager

Sqoop
Data exchange with RDBMS

Pig
Scripting

Hive
SQL

HDFS
Hadoop Distributed File System

HBase
NoSql columnar store

Solr
Full-text search, real-time indexing

Hue
Web UI for Hadoop

Hadoop for Data Lakes

- Hadoop is changing the data warehousing paradigm.
  - ✓ Data warehouse is expensive but gives quick access to specific data records (DBMS based)
  - ✓ Data lakes are low-cost implementations which gives slow access to specific data records
  - ✓ Data lakes are ideal for applications in which the entire data set is to be scanned in every cycle of processing - (Hadoop based implementation is easy)

Read the Blog - What are Data Lakes ? - DataScienceCentral.com

Real-World Hadoop Use Cases

- Risk modeling - Banks use Hadoop cluster to construct a new and more accurate score of the risk in its customer portfolios.
- Customer Churn Analysis - Why do companies really lose customers?
- Recommendation Engine - How can companies predict customer preferences?
- Ad Targeting - How can companies increase campaign efficiency?
- Point of sale transaction analysis - How do retailers target promotions guaranteed to make you buy?
- Analyzing network data to predict failure - How can organizations use machine generated data to identify potential trouble?
- Threat analysis - How can companies detect threats and fraudulent activity?
- Trade surveillance - How can a bank spot the rogue trader?
- Search quality - A major online retailer meets the challenge of delivering good search results by building its indexing infrastructure on Hadoop.

Hadoop components

Storage

HDFS

Self-healing
high-bandwidth
clustered storage

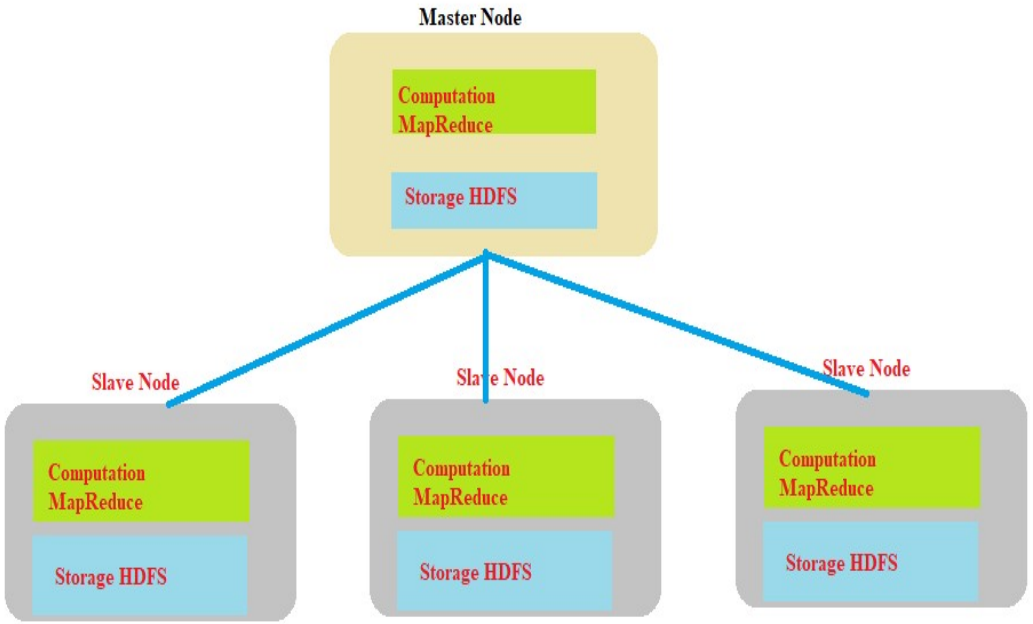
Processing

YARN

Framework for
Fault-tolerant
Distributed processing

Hadoop - Data and Compute layers

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



HDFS – Hadoop Distributed File System

Features of HDFS

- HDFS is derived from GFS (Google File System).
- HDFS forms the 'Infrastructural' part of Hadoop
- HDFS is a distributed file system with single root (/)
- It is designed to run on commodity hardware.
- It is a low-cost scalable data platform ideal for machine learning projects
- HDFS is fault tolerant and scalable
- It can magically scale from one-node cluster to thousand-nodes cluster .

  (*Yahoo! has 4,500-node cluster managing 40 petabytes of enterprise data*).

HDFS in a Nutshell

- Distributed file system for Hadoop
  - ✓Fault tolerant -> multiple replicas of data spread across a cluster
  - ✓Scalable -> designed to deliver high throughputs, sacrificing on access latency
  - ✓Immutable -> Files cannot be modified in place
  - ✓Permissions on files and folders like in POSIX + additional ACLs can be set
- Architecture
  - ✓**NameNode** -> maintains and manages file system metadata (in RAM)
  - ✓**DataNodes** -> store and manipulate the data (blocks)

HDFS Daemons (1)

NameNode

- → The namenode stores the HDFS filesystem information in a file named fsimage.
  → updates to the filesystem are not updating the fsimage but instead are logging in to a file, so the I/O is fast
  → Client applications talk to the NameNode whenever they want to add/copy/move/delete a file.
- → When restarting, reads the fsimage and then applies log file to bring the filesystem state up to date in memory.

Secondary Namenode

→ periodically read the filesystem changes (edit logs) and apply them into the fsimage file.

HDFS Daemons (2)

Datanode

→ A DataNode stores data in the HDFS

→ The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.

→ Client talks directly to datanode once namenode provides location of the data

→ Map / Reduce tasks run on Data nodes, so that processing is performed close to the data.

→ DataNode instances can talk to each other, (done during data replication)

Adding Datanode as online

→ Add new node's IP inside 'worker' file of namenode

→ start datanode of new slave nodenodes

- hdfs --daemon start datanode

→ Refresh the nodes

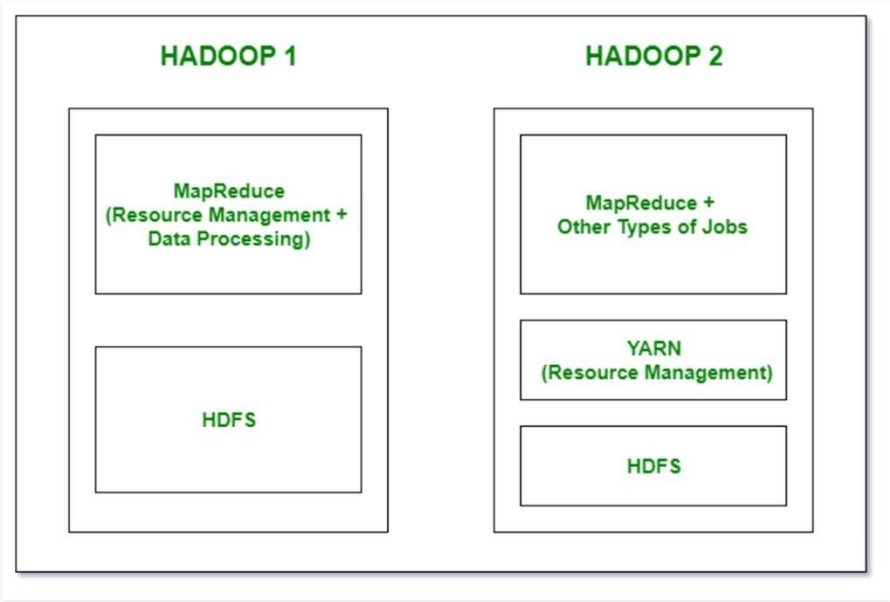
- hdfs dfsadmin -refreshNodes

What changed from Hadoop 1 to Hadoop 2

- **Hadoop 1:**

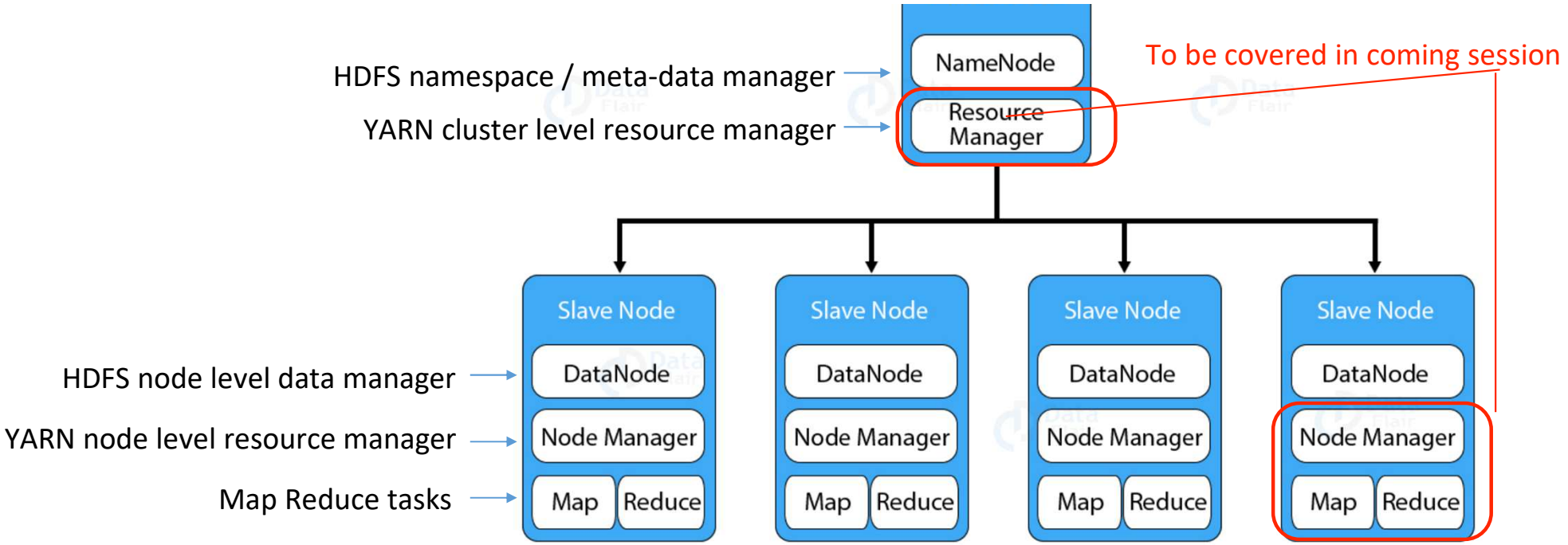
  MapReduce was coupled with resource management

  - 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 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 (AM) processes on slave nodes for application specific resource management*

Hadoop Distributions

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

- Apache Hadoop
- Intel Distribution
- Cloudera CDH
- MapR
- EMC Greenplum HD
- MS BigData Solution
- IBM InfoSphere BigInsights

Amazon Web Services
Elastic Map Reduce

- Hortonworks

Topics for today

- Hadoop architecture overview
  - ✓ Components
  - ✓ Hadoop 2.0
- **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 (Default)

- Scales 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
  - ✓ 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.

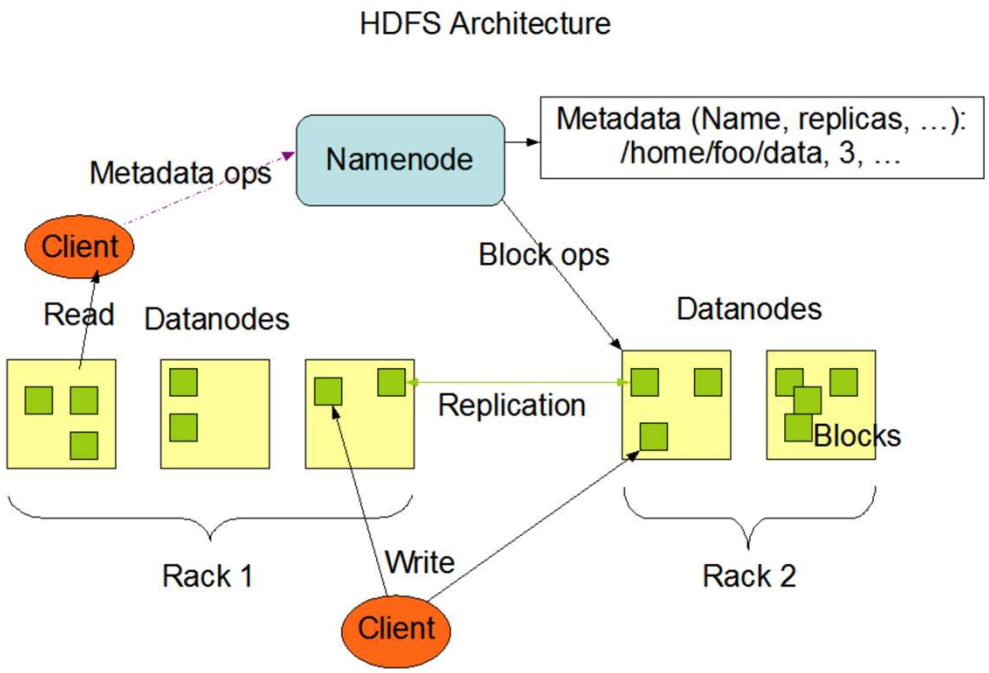
HDFS Features (4)

- 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 near to the data.

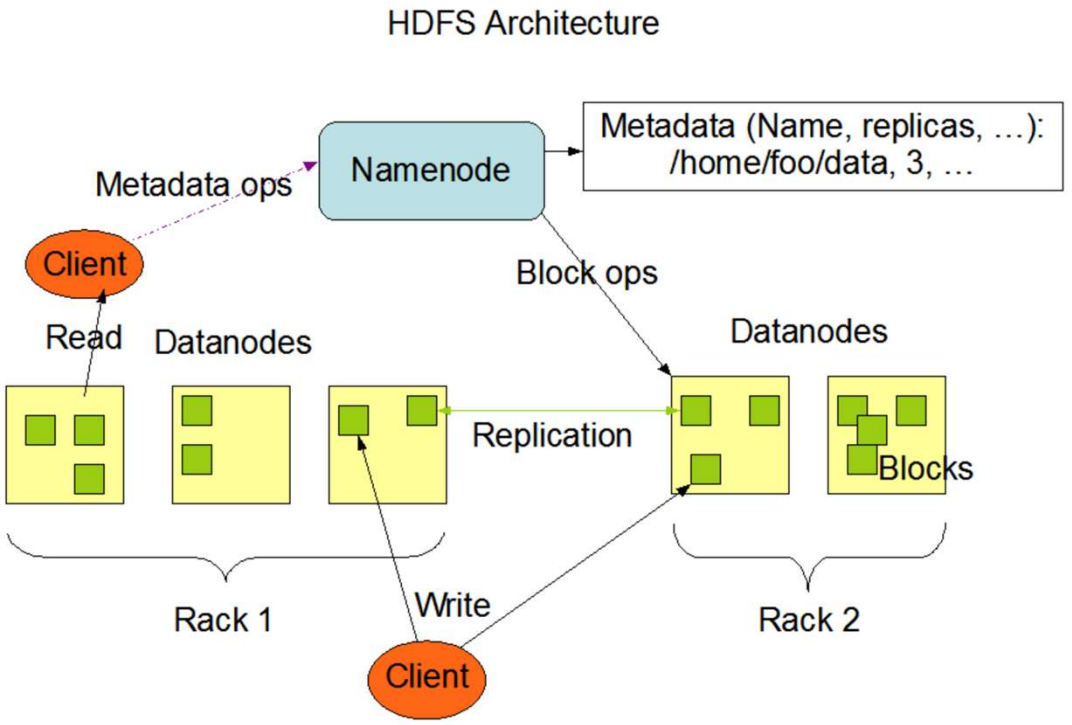
HDFS Architecture - Master node

- 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 - Worker 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 of 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 worker 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 replicas (replication factor times) are 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]#
```

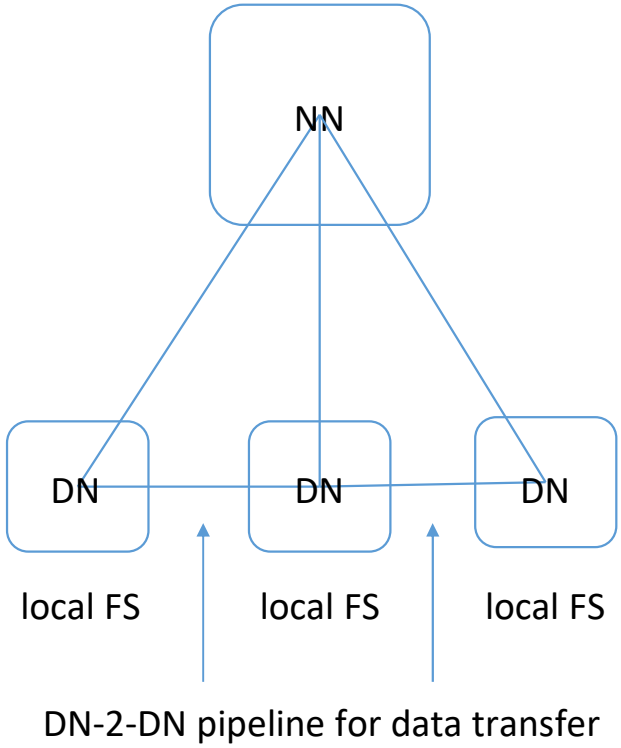
Namenode - What happens on start-up

1. Enters into safe mode
   - ✓ Check for status of Data nodes on workers
     - 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

* *[hdfs dfsadmin -safemode enter | leave | get | wait | forceExit]*

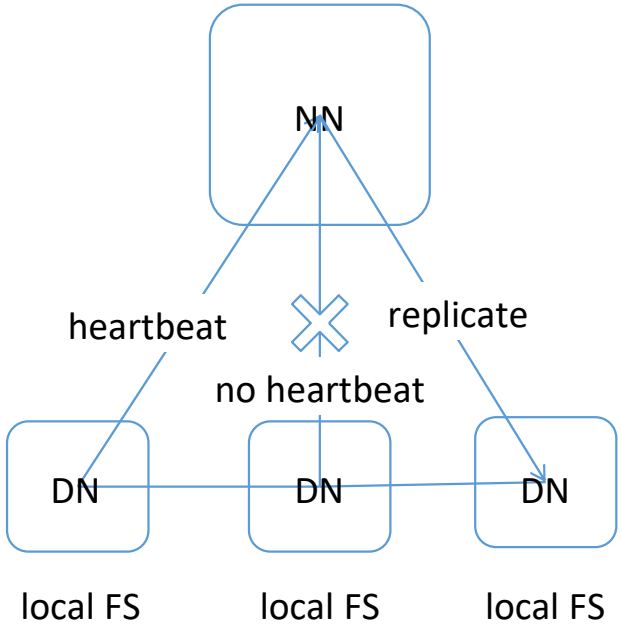
Functions of a DataNode (1)

- Each worker 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
- Default block size for ext2 and ext3 – 4096 bytes
- 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 Block Report 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

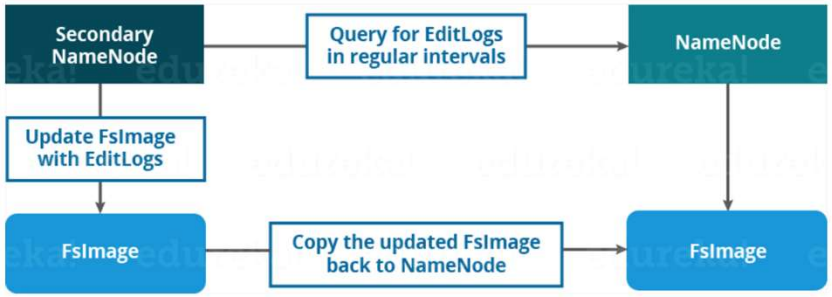


Topics for today

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

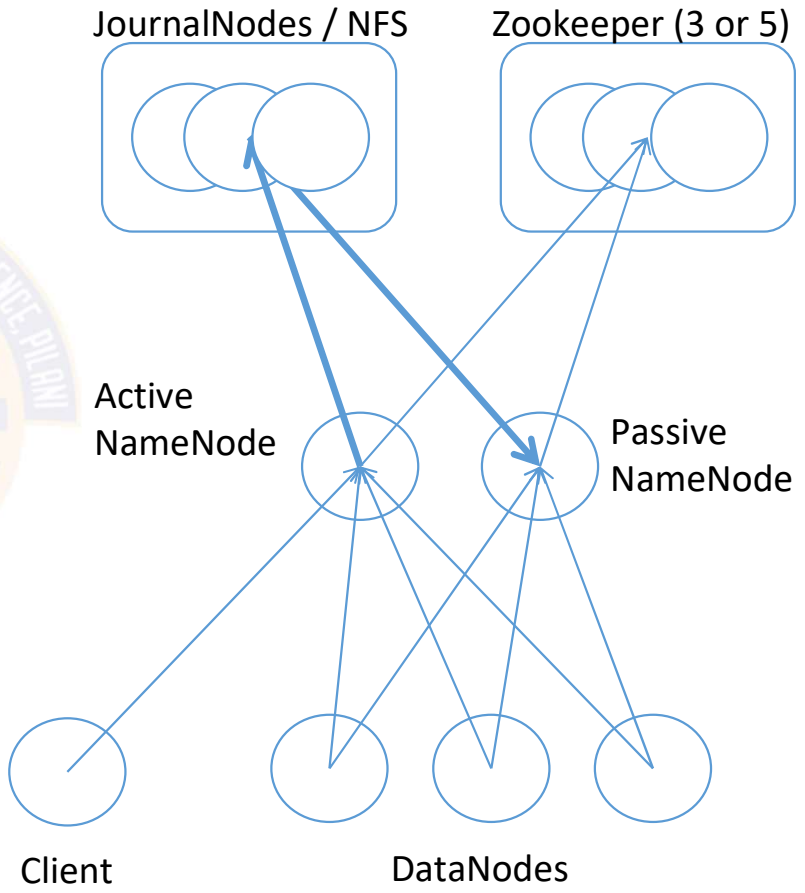
Hadoop 2: Introduction of Secondary NameNode

- 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 protocols

- 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 2
- HDFS
  - ✓ Architecture
  - ✓ Robustness
  - ✓ **Blocks and replication strategy**
  - ✓ Read and write operations
  - ✓ Big Data File formats
  - ✓ Commands

Block SIzes in HDFS

- HDFS stores each file as blocks which are scattered throughout the Hadoop cluster nodes.
- 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
  - Specifying block size when a file is copied to HDFS:
    hadoop fs -D dfs.block.size=268435456 -put pg5000book.txt /Input256MB
- Why choose large block size for HDFS files ?
  - HDFS is used for TB/PB size files and small block size will create too much meta-data
  - 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

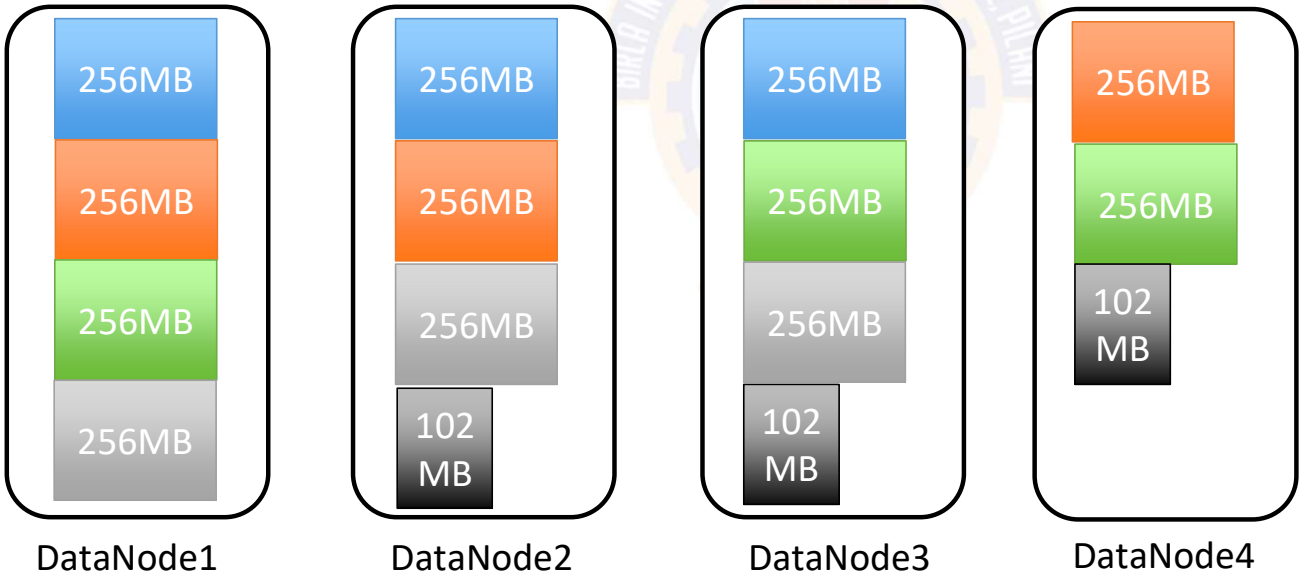
```
[root@centos-s-4vcpu-8gb-blr1-01 bds]# hdfs fsck /SalesJan2009.csv -files -blocks
21/06/12 20:46:46 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using
plicable
Connecting to namenode via http://localhost:50070/fsck?ugi=root&files=1&blocks=1&path=%2FSalesJan2009.csv
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
```

How HDFS Stores Replicated Data

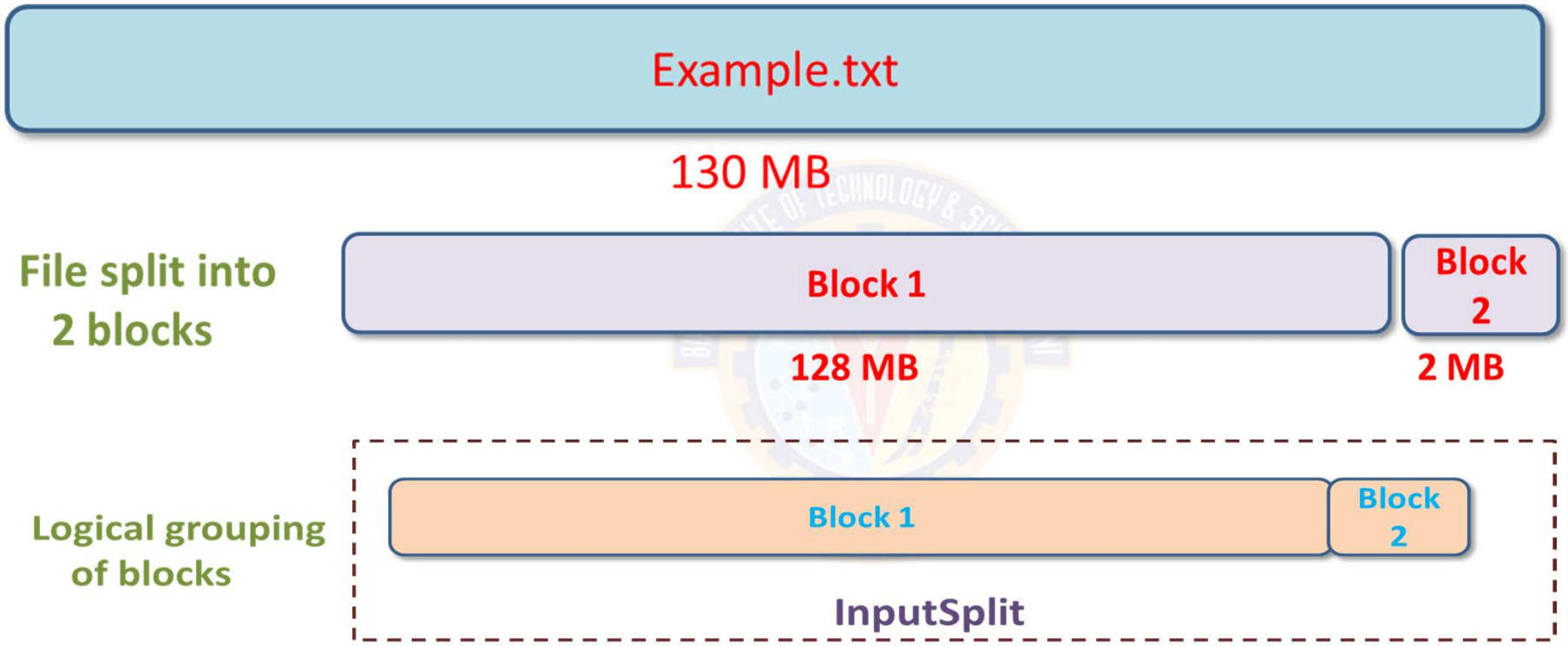
1) File to be stored on HDFS – File Size 1.1 GB
2) Splitting into 256MB blocks
3) Ask NameNode where to put them

4) Blocks with their replicas (by default 3) are distributed across Data Nodes

Setting Replication factor on HDFS files - hadoop fs **-setrep 5** /HDFSfile



HDFS Blocksize Vs Input Split



Set split size parameters in in mapred-site.xml

mapreduce.input.fileinputformat.split.minsize

mapreduce.input.fileinputformat.split.maxsize

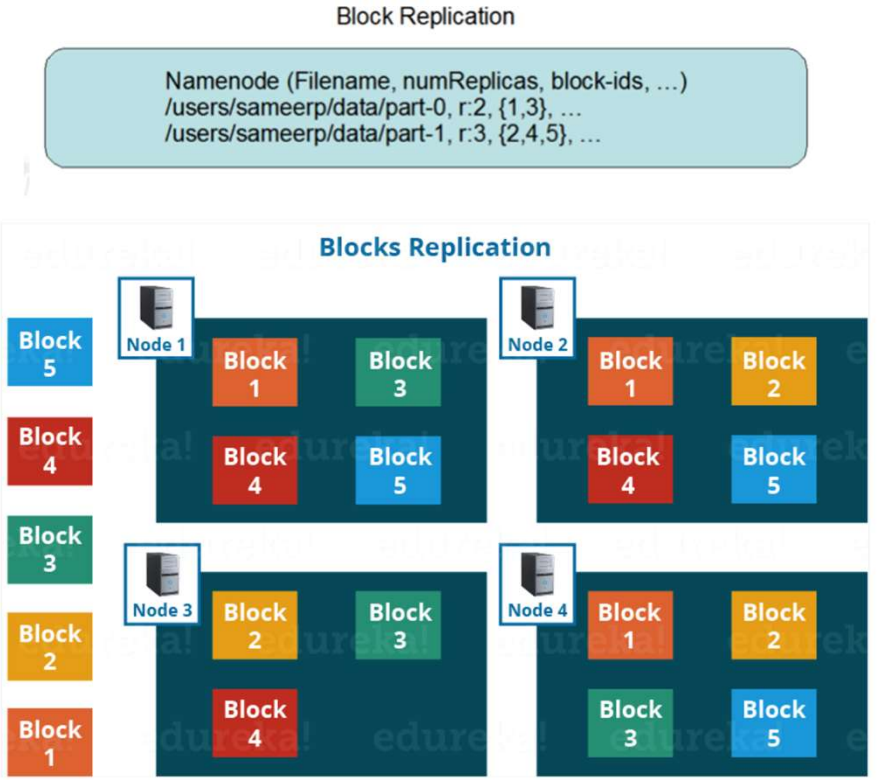
HDFS on local FS

- 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

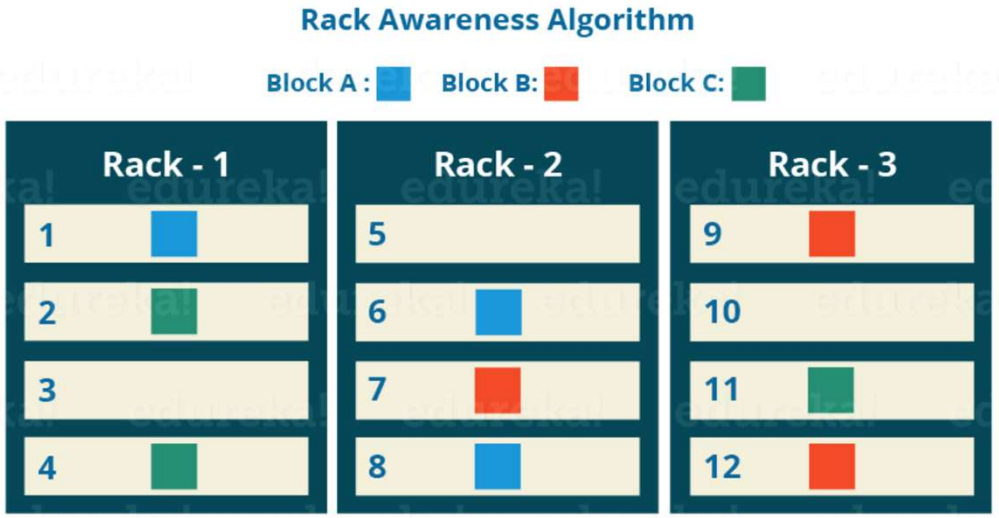
- First replica is placed on the same node as the client
- Second replica is placed on a node that is present on different rack
- Third replica is placed on same rack as second but on a different node
- Putting each replica on a different rack is expensive write operation
- For replicas > 3, nodes are 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 for replication
- 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 reduced 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

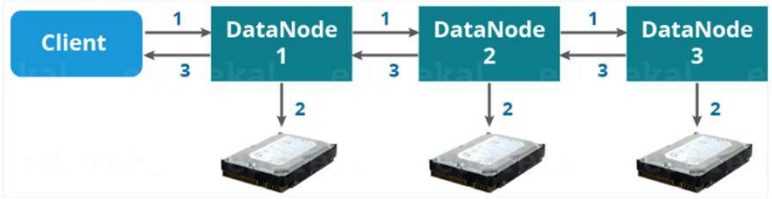
- Hadoop architecture overview
  - ✓ Components
  - ✓ Hadoop 2
- HDFS
  - ✓ Architecture
  - ✓ Robustness
  - ✓ Blocks and replication strategy
  - ✓ **Read and write operations**
  - ✓ Big Data File formats
  - ✓ Commands

HDFS data writes

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

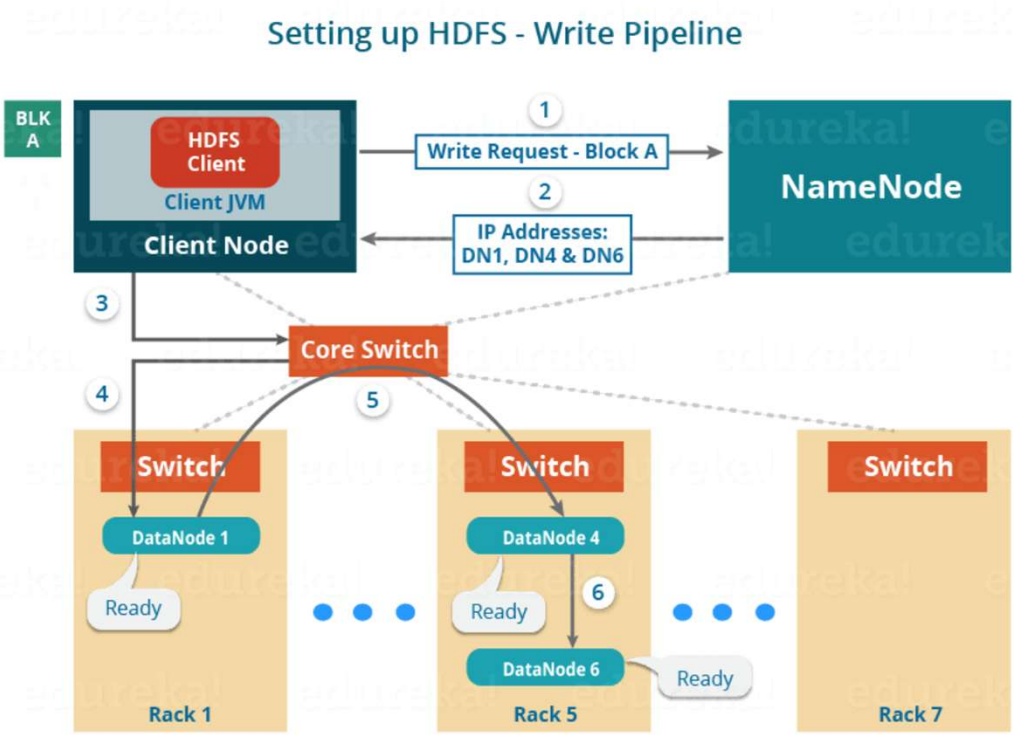
HDFS:

- 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)



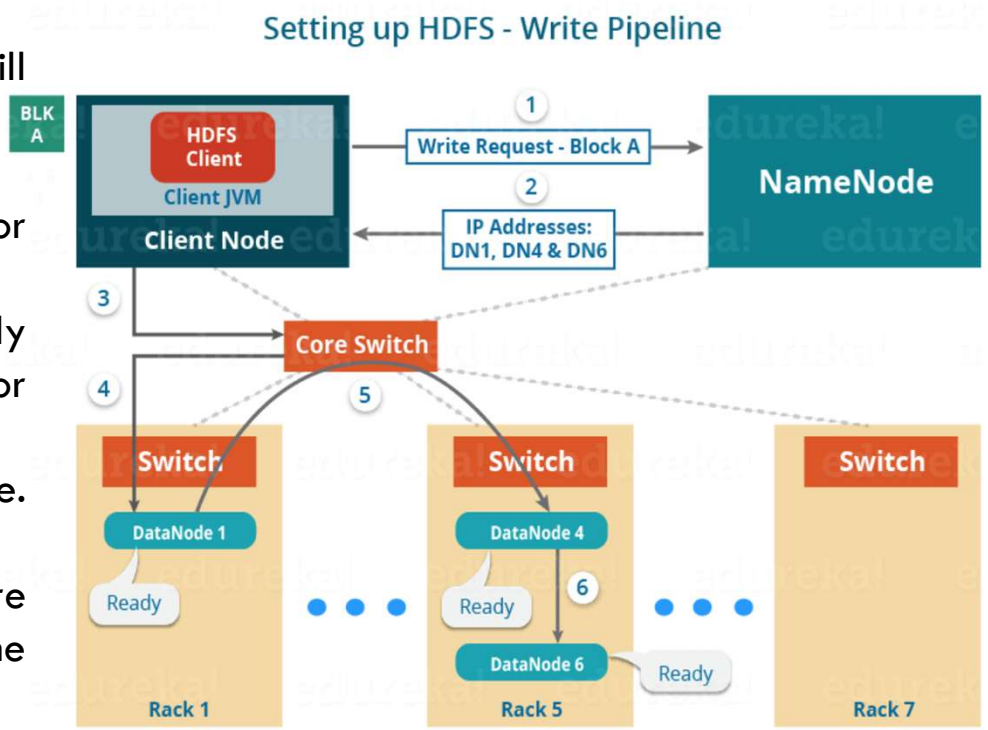
HDFS Write: Step 1. Setup pipeline

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

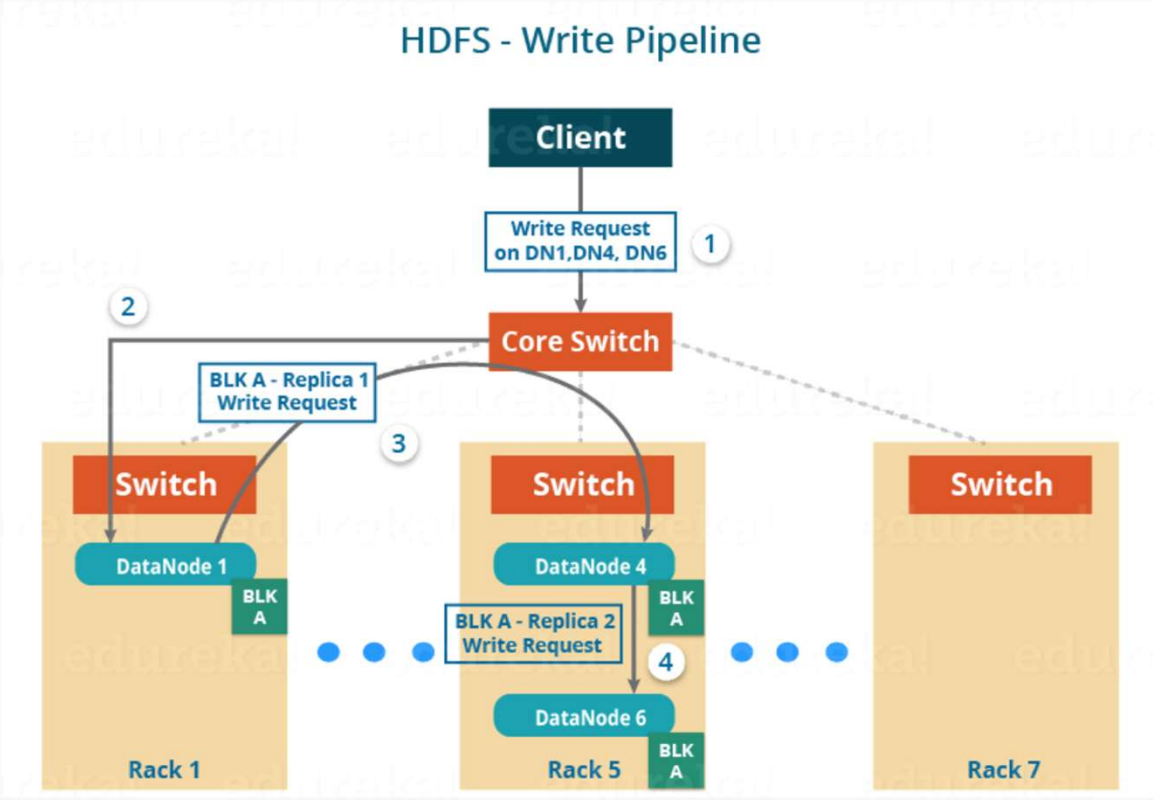
1. Client chooses the first DataNode (DN1) and will establish a TCP/IP connection.
2. Client informs DN1 to be ready to receive the block.
3. Provides IPs of next two DNs (4, 6) to DN1 for replication.
4. The DN1 connects to DN4 and informs it to be ready and gives IP of DN6. DN4 asks DN6 to be ready 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.

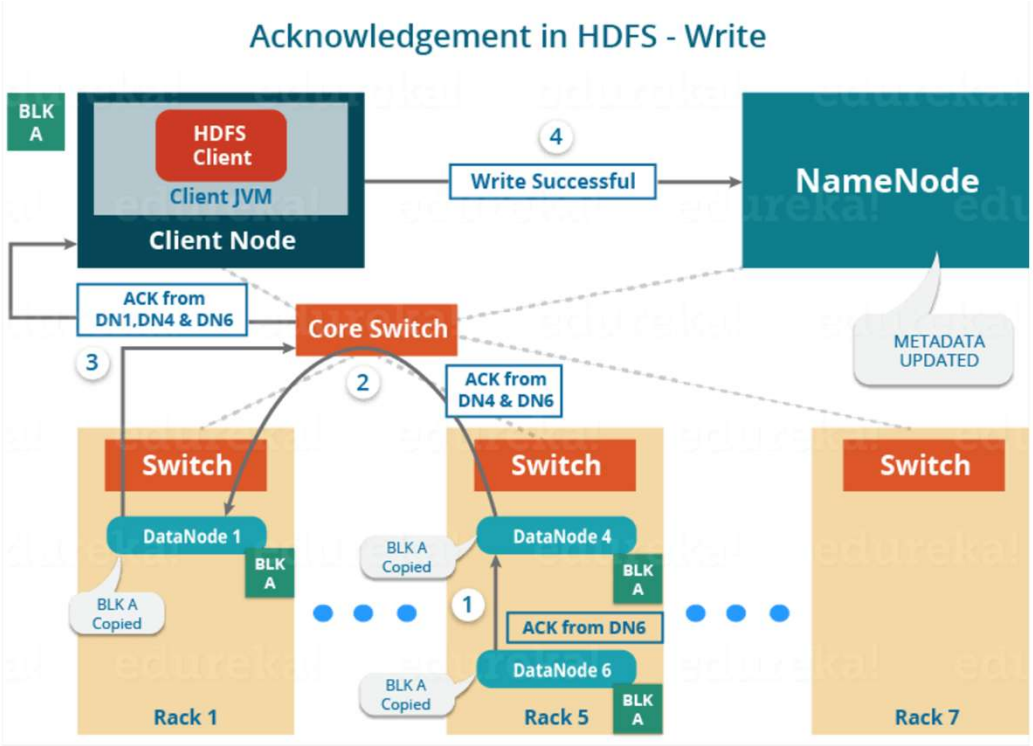
1. Once the block has been written to DataNode 1 by the client, DataNode 1 will connect to DataNode 4.
2. Then, DataNode 1 will push the block in the pipeline and data will be copied to DataNode 4.
3. Again, DataNode 4 will connect to DataNode 6 and will copy the last replica of the block.



HDFS Write: Step 3. Shutdown pipeline / ack

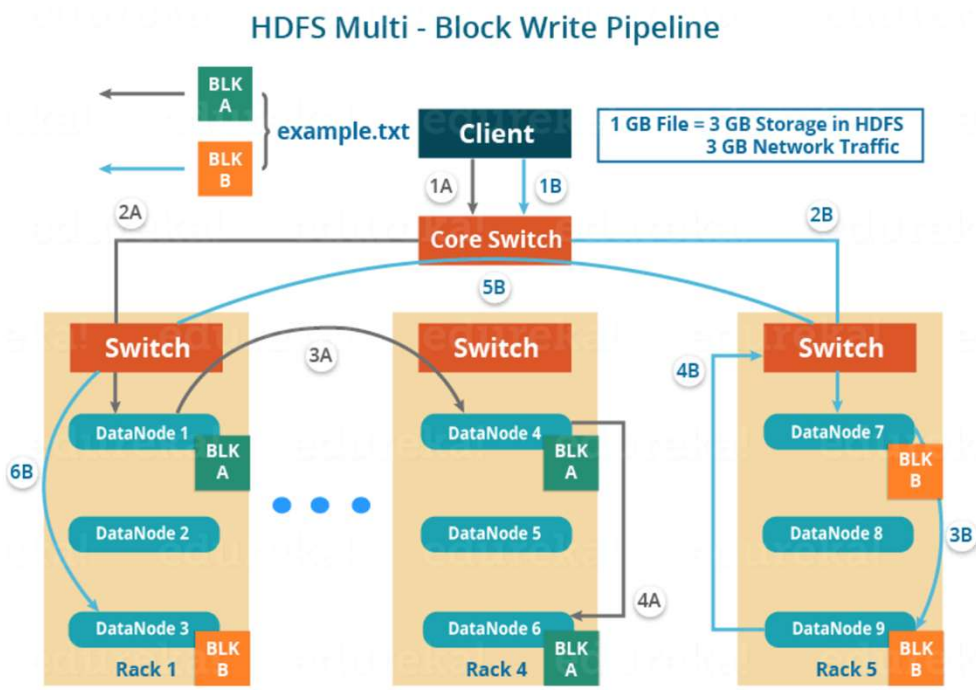
Block is now copied to all DNs. Client and NameNode need to be updated. Client needs to close pipeline and end TCP session.

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

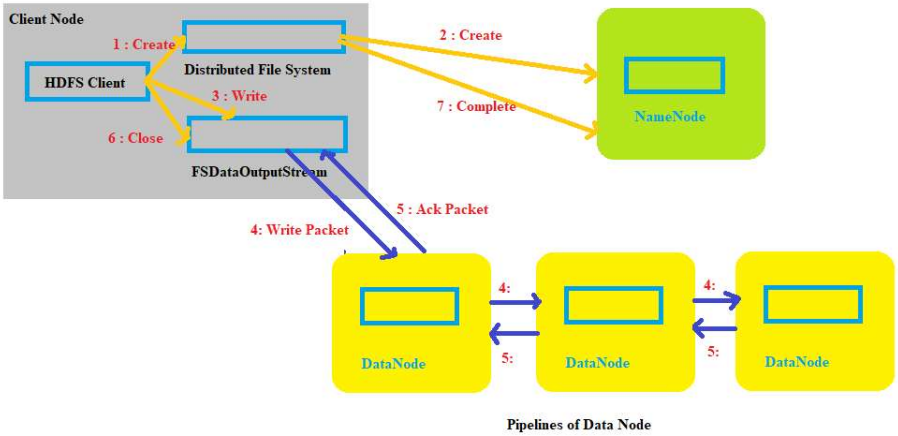


Sample write code

```
public class WriteFileToHDFS{
    public static void main(String[] args) throws IOException {
        WriteFileToHDFS.writeFileToHDFS();
    }
public static void writeFileToHDFS() throws IOException {
        Configuration configuration = new Configuration();
        configuration.set("fs.defaultFS", "hdfs://localhost:9000");
        FileSystem fileSystem = FileSystem.get(configuration);
        String fileName = "read_write_hdfs_example.txt";
        Path hdfsWritePath = new Path("/javareadwriteexample/" + fileName);
        FSDataOutputStream fsDataOutputStream = fileSystem.create(hdfsWritePath,true);
        BufferedWriter bufferedWriter = new BufferedWriter(new OutputStreamWriter(fs
        DataOutputStream,StandardCharsets.UTF_8));
        bufferedWriter.write("Java API to write data in HDFS");
        bufferedWriter.newLine();
        bufferedWriter.close();
        fileSystem.close();
    }
}
```

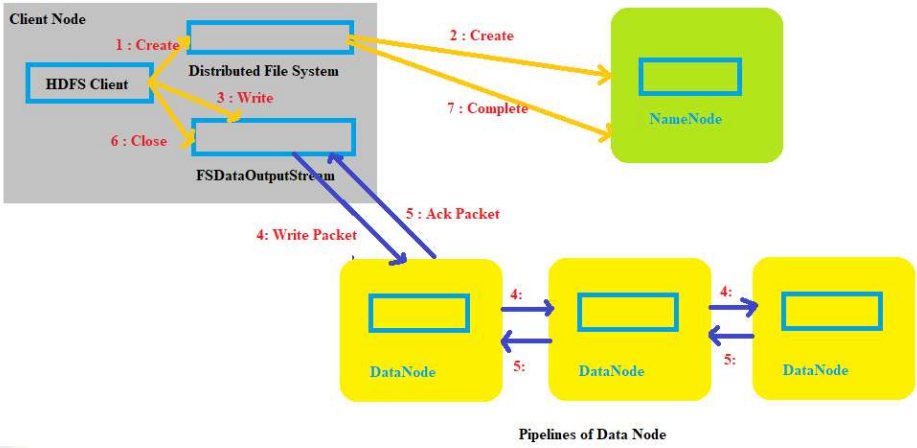
HDFS Create / Write - Call sequence in code

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

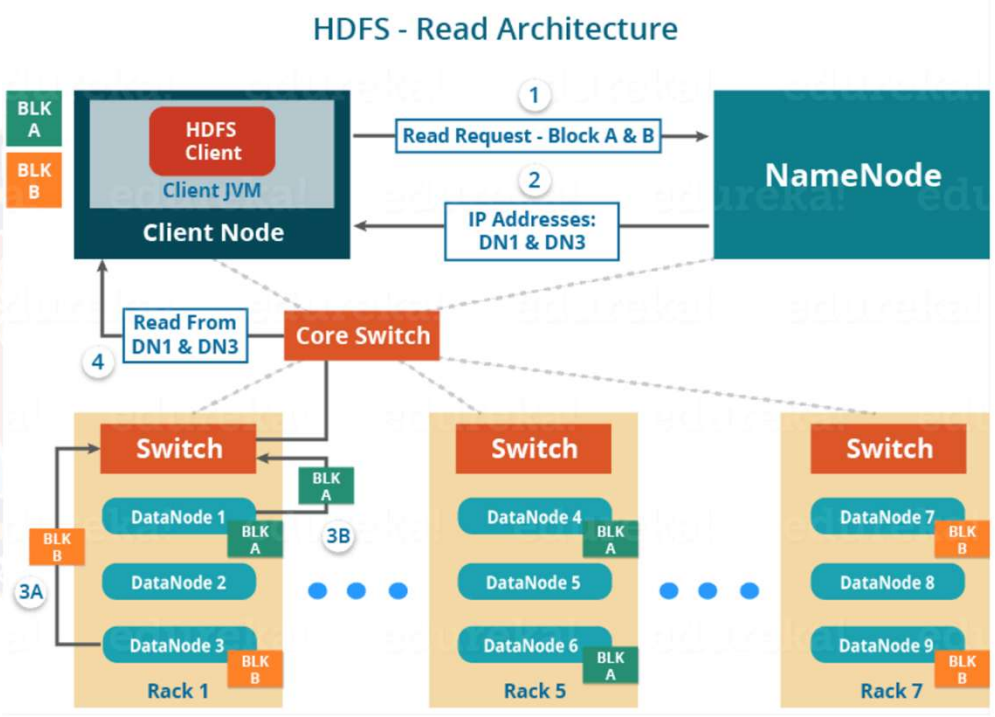
3) The first DataNode stores packet and forwards it to Second DataNode and then Second node transfer it to Third DataNode.
4) FSDataOutputStream also manages a “Ack queue” of packets that are waiting for the acknowledgement by DataNodes.
5) A packet is removed from the queue only if it is acknowledged 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.

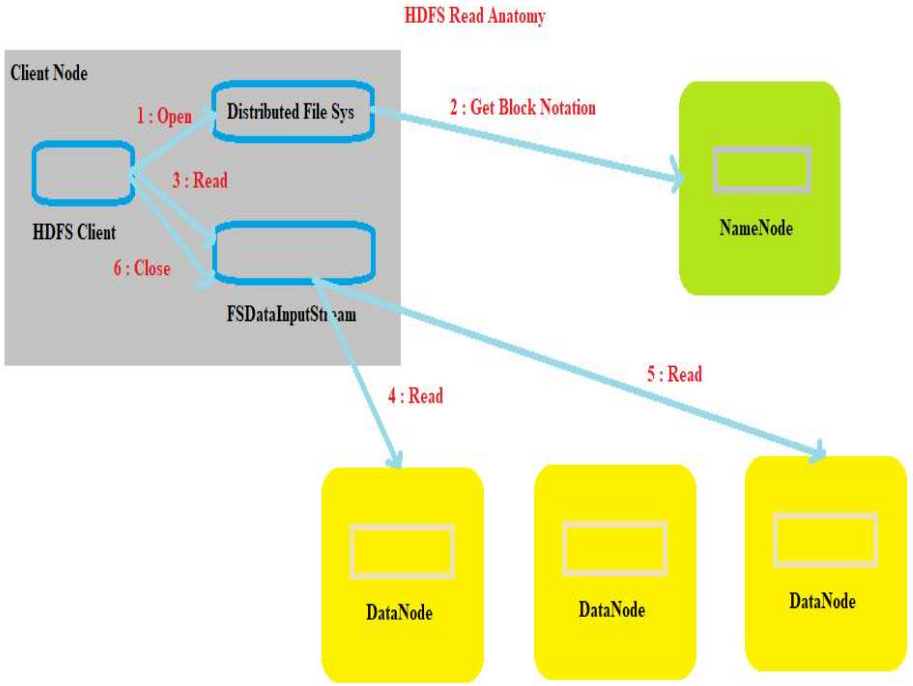


Sample read code

```
public class ReadFileFromHDFS{
    public static void main(String[] args) throws IOException {
        ReadFileFromHDFS.readFileFromHDFS();
    }
    public static void readFileFromHDFS() throws IOException {
        Configuration configuration = new Configuration();
        configuration.set("fs.defaultFS", "hdfs://localhost:9000");
        FileSystem fileSystem = FileSystem.get(configuration);
        String fileName = "read_write_hdfs_example.txt";
        Path hdfsReadPath = new Path("/javareadwriteexample/" + fileName);
        FSDataInputStream inputStream = fileSystem.open(hdfsReadPath);
        //Classical input stream usage
        String out= IOUtils.toString(inputStream, "UTF-8");
        System.out.println(out);
        inputStream.close();
        fileSystem.close();
    }
}
```

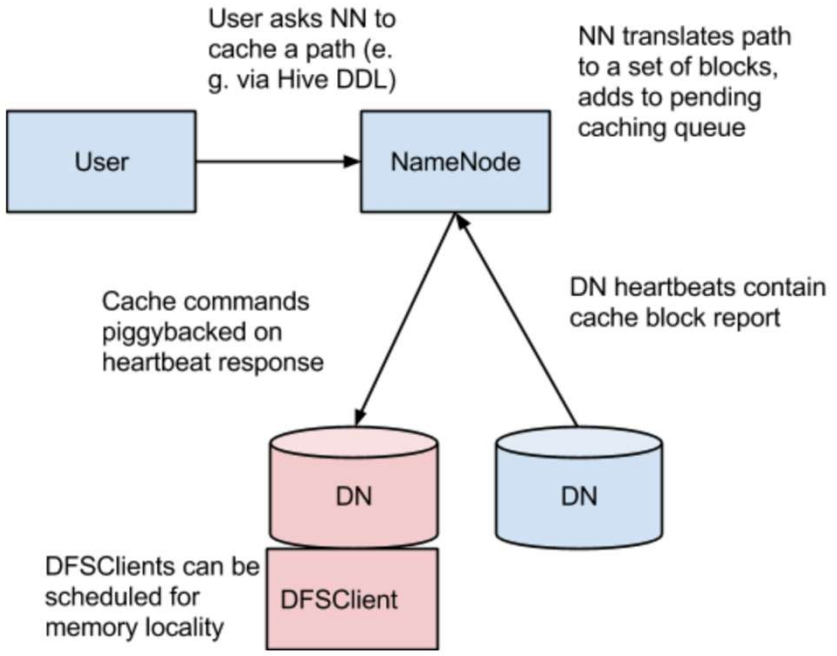
HDFS Read - Call sequence in code

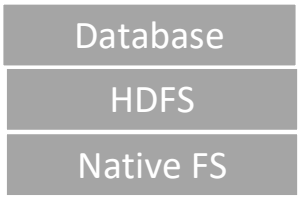
1) Client opens the file that it wishes to read from by calling open() on FileSystem
    1) FileSystem communicates with NameNode to get location of data blocks.
    2) NameNode returns the addresses of DataNodes on which blocks are stored.
    3) FileSystem returns FSDataInputStream to client to read 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

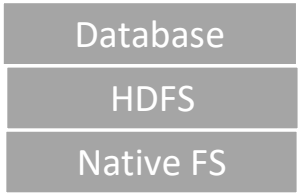
- Short-circuit reads can also be made by client to local file bypassing DataNode using /dev/shm for better performance.
- Client can ask certain HDFS paths to be cached
  - ✓ NameNode asks DNs to cache corresponding blocks and send cache block report along with heartbeat
- Clients can be scheduled for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





Security

- POSIX style file permissions
  - ✓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>**
- 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



Handson with HDFS

1. CheckExists /hdfstest
2. CreateHDFSdirectory /hdfstest
3. WriteFileToHDFS /hdfstest/read_write_hdfs_example.txt
4. ReadFileFromHDFS /hdfstest/read_write_hdfs_example.txt
5. AppendToHDFSFile /hdfstest/read_write_hdfs_example.txt
6. ReadFileFromHDFS /hdfstest/read_write_hdfs_example.txt

Topics for Today

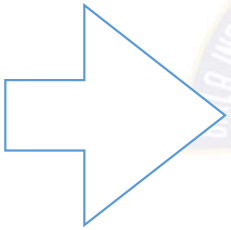
- Hadoop architecture overview
  - ✓ Components
  - ✓ Hadoop 2
- HDFS
  - ✓ Architecture
  - ✓ Robustness
  - ✓ Blocks and replication strategy
  - ✓ Read and write operations
  - ✓ **Big Data File formats**
  - ✓ Commands

Why Data Format on HDFS is important

- In big data world, we **denormalize** the data
  - ✓ Because joins are expensive
  - ✓ Duplication is not a problem – storage is cheap
  - ✓ Modern file formats/encodings can avoid reading unnecessary data anyway thanks to physical data layout, partitioning and statistics
- Benefits from choosing the right format
  - ✓ Decrease data volume
  - ✓ Increase data access speed
  - ✓ Reduce the amount of resources needed for data processing
- How to achieve this?
  - ✓ Encoding and compression
  - ✓ Partitioning
- Typical HDFS data formats
  - ✓ Natural: Text-log, CSV, JSON
  - ✓ Specialized: **Apache Avro, Apache Parquet**
  - ✓ Hadoop NoSQL database: Apache HBase, Apache Kudu
- Typical compression
  - ✓ Snappy
  - ✓ Bzip2/Gzip

Big Data File formats

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



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

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

Sequence File Writer in HDFS

```
public class SequenceFileWriter {
    private static final String[] text = { "aa","ccc", "eee", "fff" };
    public static void main(String[] args) {
        String uri = "hdfs://localhost:9000/user/seqdemo";
        Configuration conf = new Configuration();
        SequenceFile.Writer writer = null;
        try {
            FileSystem fs = FileSystem.get(URI.create(uri), conf);
            Path path = new Path(uri);
            IntWritable key = new IntWritable();
            Text value = new Text();
            writer = SequenceFile.createWriter(fs, conf, path,
                    key.getClass(), value.getClass());
            for (int i = 0; i < 100; i++) {
                key.set(100 - i);
                value.set(text[i % text.length]);
                writer.append(key, value);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            IOUtils.closeStream(writer);
        }
    }
}
```

- When we want to store binary(images)/test as key-value pairs, maybe attach some meta-data as well
- Contents 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

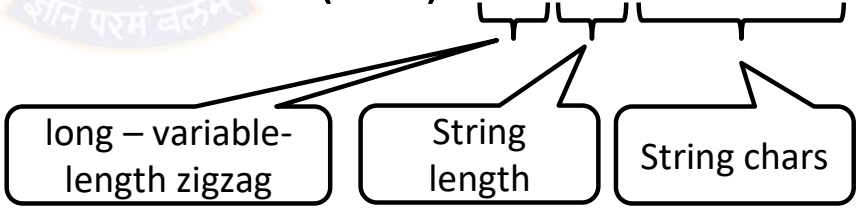
Apache Avro data file

- Fast, binary serialization format for structured data
- Internal schema with multiple data types including nested ones
  - ✓ scalars, arrays, maps, structs, etc
- Schema in JSON

```
{
        "type": "record",
        "name": "test",
        "fields" : [
        {"name": "a", "type": "long"},
        {"name": "b", "type": "string"}
        ]
}
```

Record {a=27, b='foo'}

Encoded (hex): 36 06 66 6f 6f



File formats - Optimized Row Columnar (ORC) *

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

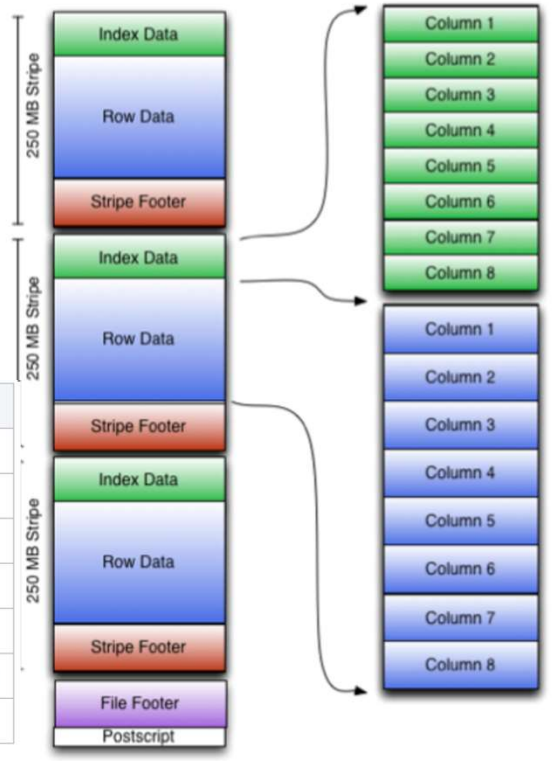
```
create table Addresses (
  name string,
  street string,
  city string,
  state string,
  zip int
) stored as orc tblprop
```

Index data used to skip rows

Includes min/max for each column and their row positions

Row data used for table scans

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)



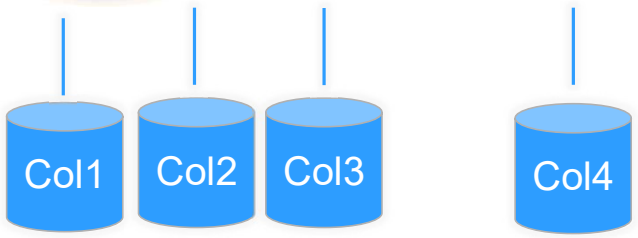
* more in Hive session

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

Columnar store for analytics

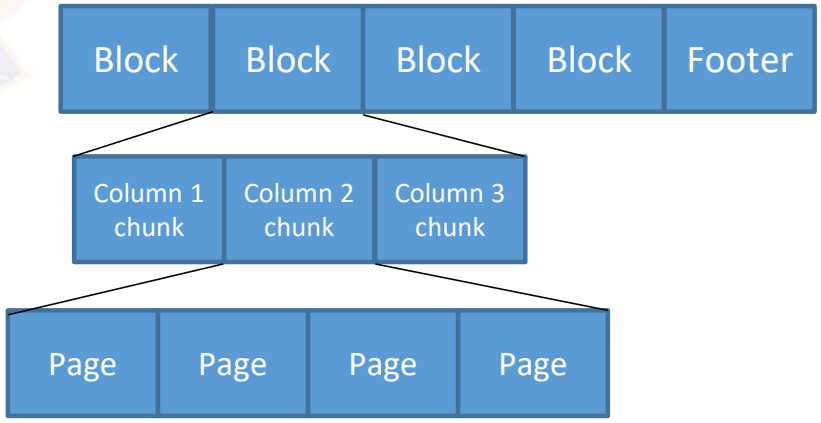
- We are interested in reading certain columns
- But in traditional row-stores we are always reading entire row data
- Columnar store

Day	Month	Year	No of customers
10	Aug	2013	17
11	Aug	2013	15
12	Aug	2013	21
2	Dec	2014	30
3	Dec	2014	34
4	Dec	2014	31
17	Feb	2015	12
18	Feb	2015	16



Apache Parquet – a columnar storage for HDFS

- Based on Google “Dremel”
- Internal schema with multiple data types including nested ones
- Multiple encoding applicable on per column-bases
  - ✓ RLE, Dictionary, Delta, Bit packing
  - ✓ Chosen automatically
- Column-level statistics per each block/rowgroup
- Compressions supported
  - ✓ Snappy, gzip, LZO



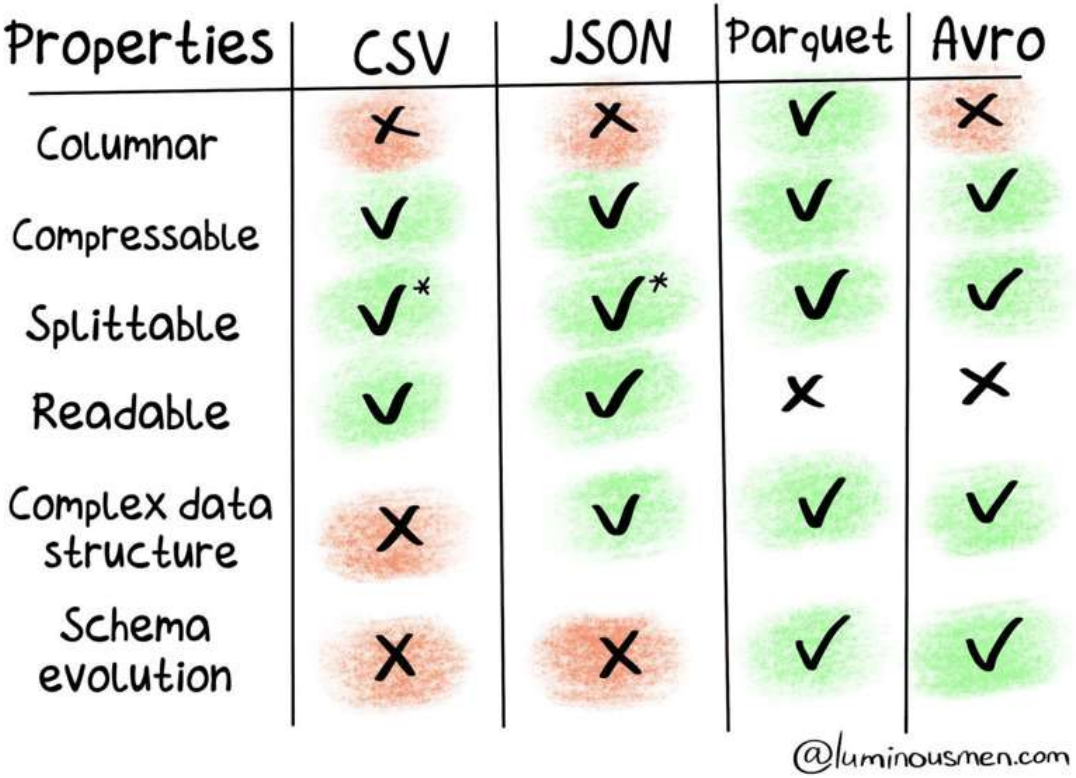
File formats - Parquet

- 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

Writing Parquet , orc files from Spark

```
# Read from a .csv file and write to HDFS in parquet format
#Read from local file
df =spark.read.format("csv").option("header","true").load("file:///home/jps/Handson/bigdata/SalesJan2009.csv")
#OR  Read from an HDFS file
df = spark.read.format("csv").option("header","true").load("/SalesJan2009.csv")
df.printSchema()
# Write the contents of the dataframe to HDFS in parquet format
df.write.parquet("/SalesJan20029.parquet", compression='lz4')

# Write the contents of dataframe to HDFS in orc format
df.write.orc("/SalesJan2009.orc")
# Read the ORC file, infer the schema, and ignore leading/trailing whitespace
orc_df = spark.read.format("orc") \
            .option("inferSchema", "true") \
            .option("ignoreLeadingWhiteSpace", "true") \
            .option("ignoreTrailingWhiteSpace", "true") \
            .load("/SalesJan2009.orc")
```

Big Data File formats -Comparison



Source: https://luminousmen.com/post/big-data-file-formats

Topics for today

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

https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoophdfs/HDFSCommands.html

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 -D dfs.block.size=134217728 -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 -secondarynamenodes
- hdfs getconf -confkey <a.value>
  - return the value of a particular setting (e.g. dfs.replication)
  - hdfs getconf -confkey dfs.replication
- hdfs dfsadmin -report
  - find out how much disk space us used, free, under-replicated, etc.
- hadoop fs -setrep 2 /jps/wc/sample01.txt
  - Set replication factor of 2 to sample01.txt file in HDFS

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:
Distributed Programming