Big Data Technologies

HDFS

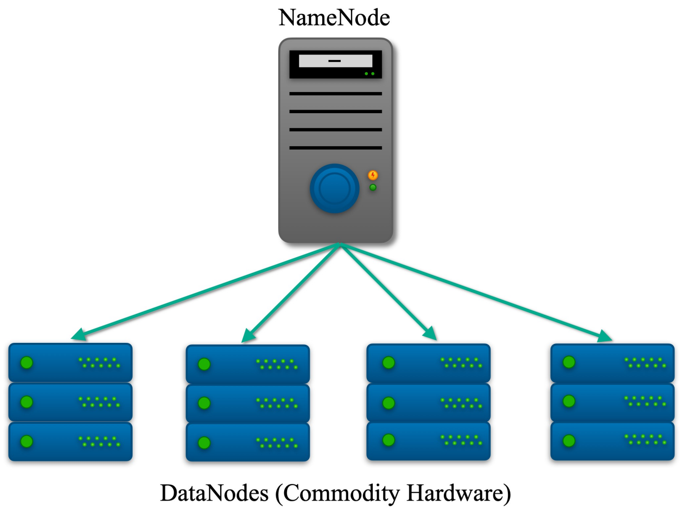
Lionel Fillatre
Polytech Nice Sophia
lionel.fillatre@univ-cotedazur.fr

Outlines

Introduction

HDFS (Hadoop Distributed File System)

- **To start building an application, you need a file system**
  - In Hadoop world that would be HDFS
- **Appears as a single disk**
- **Runs on top of a native filesystem**
- **Fault Tolerant**
  - Can handle disk crashes, machine crashes, etc...
- **Based on Google's Filesystem (GFS or GoogleFS)**
- **Not the only solution:** MapR distribution does not use HDFS but a tuned version of HBASE



HDFS is Good for...

- **Storing large files**
  - Terabytes, Petabytes, etc...
  - Millions rather than billions of files
  - 100MB or more per file

- **Streaming data**
  - Write once and read-many times patterns
  - Optimized for streaming reads rather than random reads
  - Append operation added to Hadoop 0.21

- **“Cheap” Commodity Hardware**
  - No need for super-computers, use less reliable commodity hardware

HDFS is not so good for...

- **Low-latency reads**
  - High-throughput rather than low latency for small chunks of data
  - HBase addresses this issue

- **Large amount of small files**
  - Better for millions of large files instead of billions of small files
    - For example each file can be 100MB or more

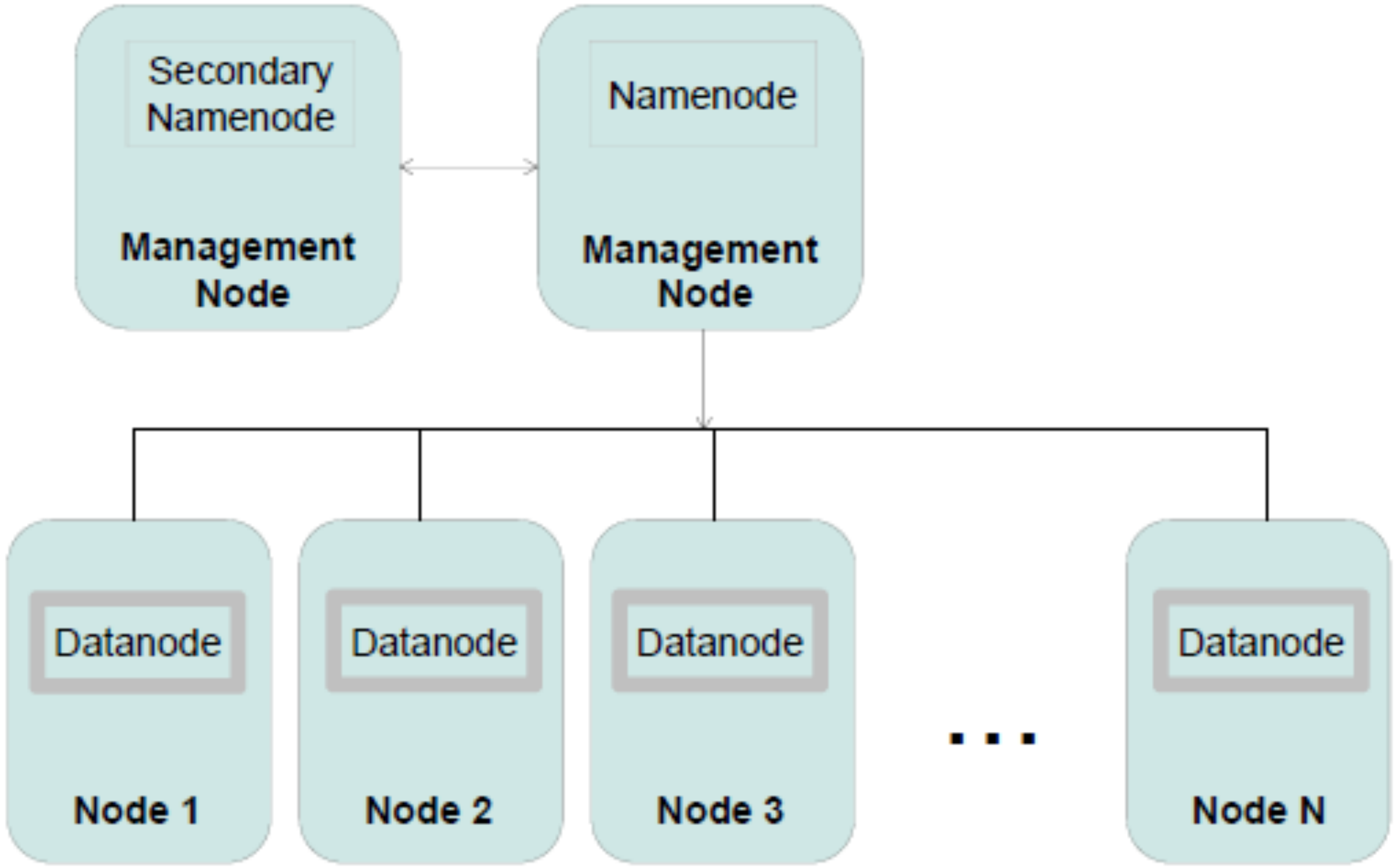
- **Multiple Writers**
  - Single writer per file
  - Writes only at the end of file, no-support for arbitrary offset

HDFS Daemons

HDFS Daemons

- **Filesystem cluster is managed by three types of processes**
  - Namenode
    - manages the File System's namespace/meta-data/file blocks
    - Runs on 1 machine to several machines
  - Datanode
    - Stores and retrieves data blocks
    - Reports to Namenode
    - Runs on many machines
  - Secondary Namenode
    - Performs house keeping work so Namenode doesn’t have to
    - Requires similar hardware as Namenode machine
    - Not used for high-availability – not a backup for Namenode

HDFS Daemons



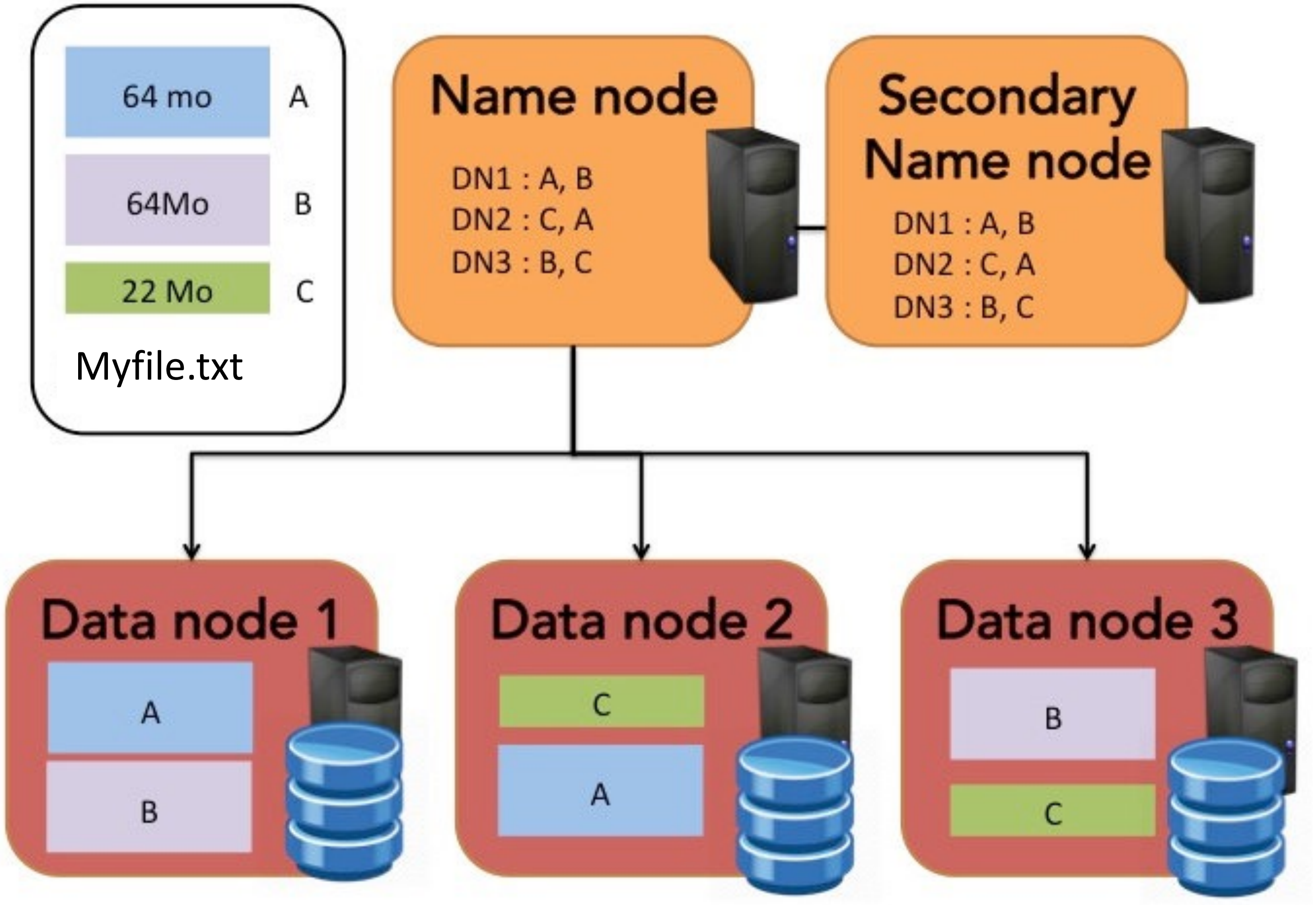
Files and Blocks

Files and Blocks

- **Files are split into blocks (single unit of storage)**
    - Managed by Namenode, stored by Datanode
    - Transparent to user

- **Replicated across machines at load time**
    - Same block is stored on multiple machines
    - Good for fault-tolerance and access
    - Default replication is 3

Files and Blocks



HDFS Blocks

- **Blocks are traditionally either 64MB or 128MB**
  - Default is 64MB
- **The motivation is to minimize the cost of seeks as compared to transfer rate**
  - 'Time to transfer' > 'Time to seek'
- **For example, lets say**
  - seek time = 10ms = 0.01s
  - Transfer rate = 100 MB/s
- **To achieve seek time of 1% transfer time**
  - Transfert time in second for x MB: $\frac{x}{100}$
  - Constraint: $0.01 = \frac{1}{100}(0.01 + \frac{x}{100})$
  - Block size will need to be 99 MB (about 100 MB)

Block Replication

- **Namenode determines replica placement**
- **Replica placements are rack aware**
    - Balance between reliability and performance
        - Attempts to reduce bandwidth
        - Attempts to improve reliability by putting replicas on multiple racks
    - Default replication is 3
        - 1st replica on the local rack
        - 2nd replica on the local rack but different machine
        - 3rd replica on the different rack
    - This policy may change/improve in the future

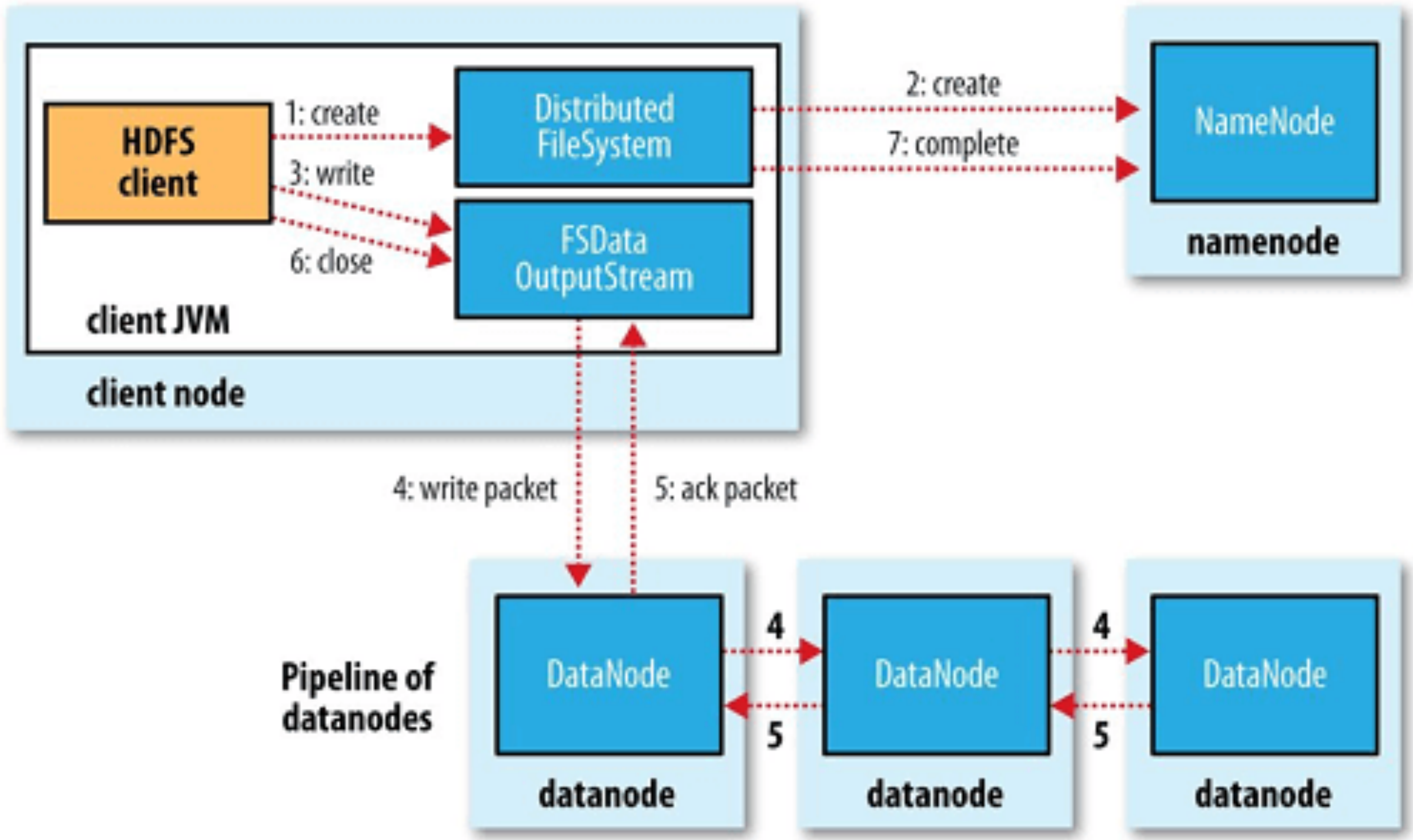
Client, Namenode, and Datanodes

- Namenode does NOT directly write or read data
  - One of the reasons for HDFS's Scalability

- Client interacts with Namenode to update Namenode's HDFS namespace and retrieve block locations for writing and reading

- Client interacts directly with Datanode to read/write data

HDFS File Write



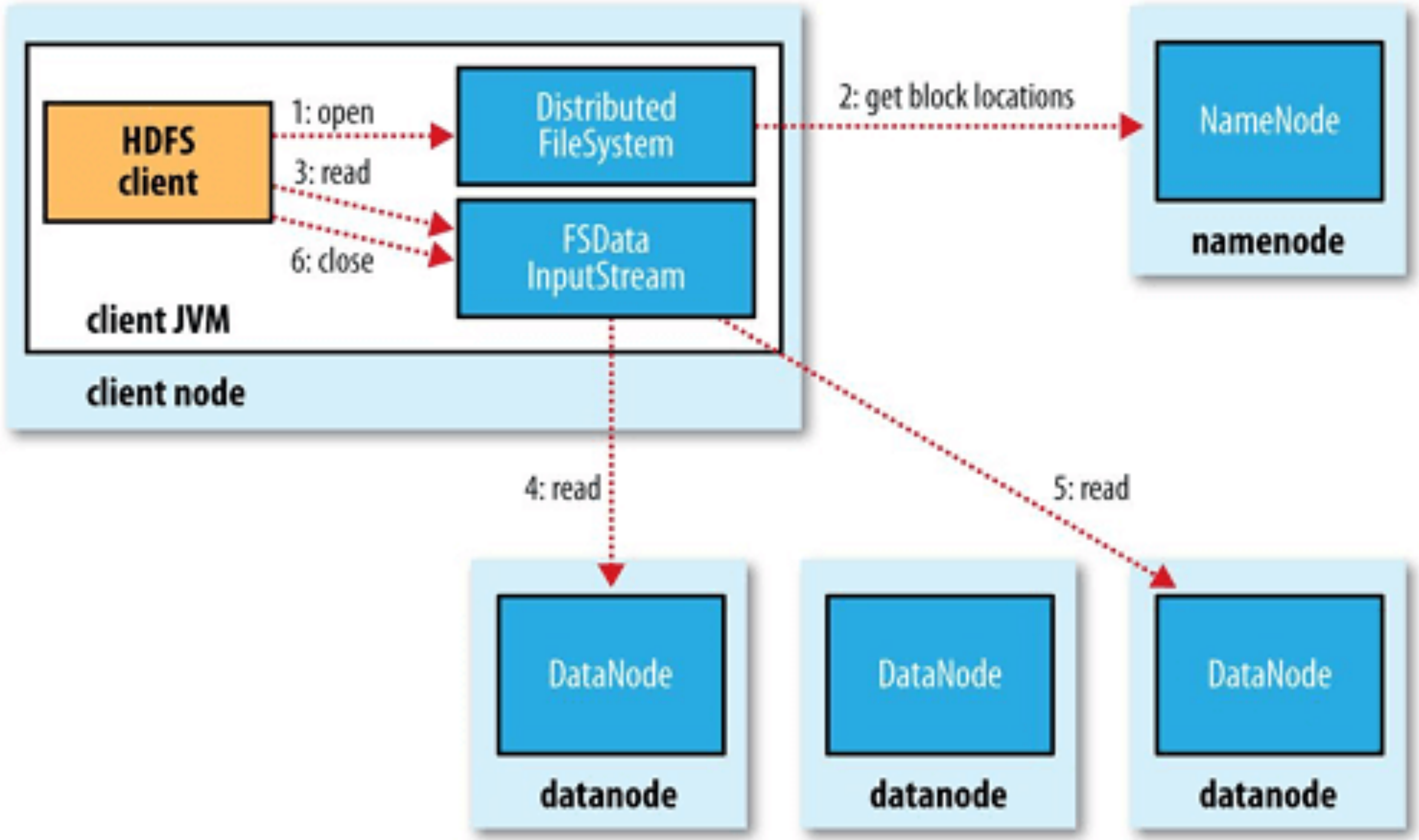
Details on HDFS File Write (1/2)

- The client creates the file by calling *create()* on DistributedFileSystem.
- DistributedFileSystem makes an RPC call to the namenode to create a new file in the filesystem’s namespace, with no blocks associated with it.
- The DistributedFileSystem returns an FSDataOutputStream for the client to start writing data to. Just as in the read case, FSDataOutputStream wraps a DFSOutputStream, which handles communication with the datanodes and namenode.
- As the client writes data, the DFSOutputStream splits it into packets, which it writes to an internal queue called the data queue.
- The data queue is consumed by the DataStreamer, which is responsible for asking the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas.

Details on HDFS File Write (1/2)

- The DataStreamer streams the packets to the first datanode in the pipeline, which stores each packet and forwards it to the second datanode in the pipeline. Similarly, the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline.
- The DFSOutputStream also maintains an internal queue of packets that are waiting to be acknowledged by datanodes, called the ack queue. A packet is removed from the ack queue only when it has been acknowledged by all the datanodes in the pipeline.
- When the client has finished writing data, it calls close() on the stream. This action flushes all the remaining packets to the datanode pipeline and waits for acknowledgments before contacting the namenode to signal that the file is complete.
- The namenode already knows which blocks the file is made up of (because DataStreamer asks for block allocations), so it only has to wait for blocks to be minimally replicated before returning successfully.

HDFS File Read



Details on HDFS File Read (1/2)

- The client opens the file it wishes to read by calling *open()* on the FileSystem object,which for HDFS is an instance of DistributedFileSystem.
- DistributedFileSystem calls the namenode, using remote procedure calls (RPCs), to determine the locations of the first few blocks in the file.
- For each block, the namenode returns the addresses of the datanodes that have a copy of that block. Furthermore, the datanodes are sorted according to their proximity to the client (according to the topology of the cluster’s network). If the client is itself a datanode (in the case of a MapReduce task, for instance), the client will read from the local datanode if that datanode hosts a copy of the block.
- The DistributedFileSystem returns an FSDataInputStream (an input stream that supports file seeks) to the client for it to read data from. FSDataInputStream in turn wraps a DFSInputStream, which manages the datanode and namenode I/O.
- The client then calls read() on the stream.

Details on HDFS File Read (2/2)

- DFSInputStream, which has stored the datanode addresses for the first few blocks in the file, then connects to the first (closest) datanode for the first block in the file. Data is streamed from the datanode back to the client, which calls read() repeatedly on the stream.
- When the end of the block is reached, DFSInputStream will close the connection to the datanode, then find the best datanode for the next block. This happens transparently to the client, which from its point of view is just reading a continuous stream.
- Blocks are read in order, with the DFSInputStream opening new connections to datanodes as the client reads through the stream. It will also call the namenode to retrieve the datanode locations for the next batch of blocks as needed.
- When the client has finished reading, it calls close() on the FSDataInputStream.

Architecture

Datanode

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

Namenode

- The namenode stores the entire file system metadata in memory

- The NameNode maintains the namespace tree and the mapping of blocks to Datanodes

- Namespace:
  - Consists of directories, files and blocks.
  - It supports all the namespace related file system operations such as create, delete, modify and list files and directories

- The bigger the cluster - the more RAM required
  - Best for millions of large files (100mb or more) rather than billions

Namenode Federation

- Namenode Federations (horizontally scale the Namenode)
  - Federation uses multiple independent namenodes/namespaces.
  - The namenodes are federated, that is, the namenodes are independent and don’t require coordination with each other.
  - The datanodes are used as common storage for blocks by all the namenodes.
  - Each Namenode will host part of the blocks

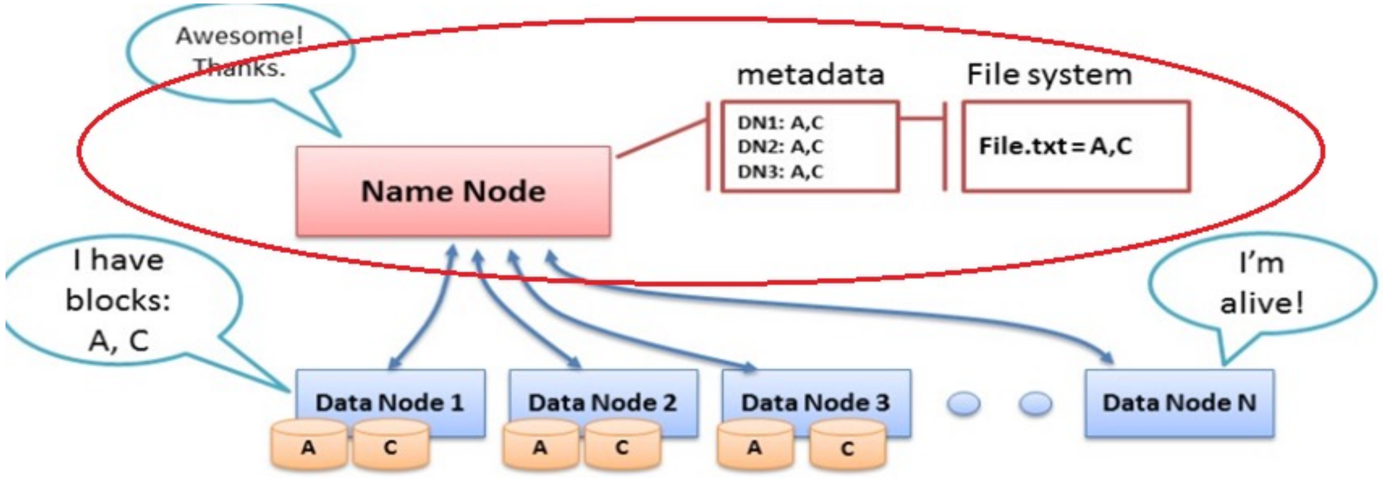
Namenode State

- Namenode stores its state on local/native file-system mainly in two files: edits and fsimage
    - Stored in a directory configured via dfs.name.dir property in hdfs-site.xml
    - edits : log file where all filesystem modifications are appended
    - fsimage: on start-up namenode reads hdfs state, then merges edits file into fsimage and starts normal operations with empty edits file

- Namenode start-up merges will become slower over time but Secondary Namenode to the rescue (discussed later)

Heartbeats

- Namenode and Datanode communication: Heartbeats.
- Datanodes send heartbeats to the Namenode to confirm that the Datanode is operating and the block replicas it hosts are available.



- Data Node sends Heartbeats
- Every 10th heartbeat is a Block report
- Name Node builds metadata from Block reports

Blockreports

- A Datanode identifies block replicas in its possession to the Namenode by sending a blockreport.
- A blockreport contains the block id, the generation stamp and the length for each block replica the server hosts.
- Blockreports provide the Namenode with an up-to-date view of where block replicas are located on the cluster
- Namenode constructs and maintains latest metadata from blockreports.

Points of Failure

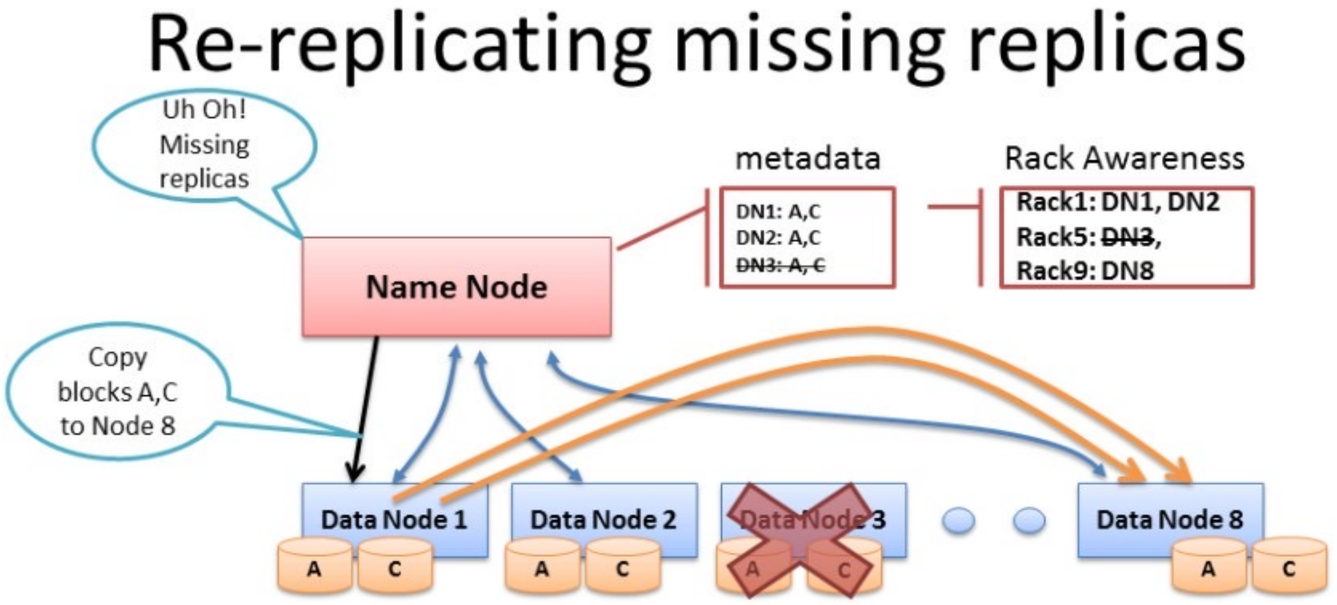
- Namenode daemon process must be running at all times
  - If process crashes then cluster is down

- The Namenode does not directly call Datanodes. It uses replies to heartbeats to send instructions to the Datanodes.

- The instructions include commands to:
  - Replicate blocks to other nodes (datanode died, etc.)
  - Remove local block replicas
  - Re-register or to shut down the node

- Namenode is a single point of failure
  - Host on a machine with reliable hardware (ex. sustain a diskfailure)
  - When datanode died, Namenode will notice and instruct other datanode to replicate data to new datanode. What if NameNode died?

Missing Replicas



- Missing Heartbeats signify lost Nodes
- Name Node consults metadata, finds affected data
- Name Node consults Rack Awareness script
- Name Node tells a Data Node to re-replicate

Failure Recovery

- CheckpointNode and BackupNode (two other roles of Namenode)
- CheckpointNode:
  - Keep journal (the modification log of metadata).
  - When journal becomes too long, checkpointNode combines the existing checkpoint and journal to create a new checkpoint and an empty journal.
- BackupNode: a read-only Namenode
  - It maintains an in-memory, up-to-date image of the file system namespace that is always synchronized with the state of the Namenode.
  - If the Namenode fails, the BackupNode's image in memory and the checkpoint on disk is a record of the latest namespace state.

Snapshot

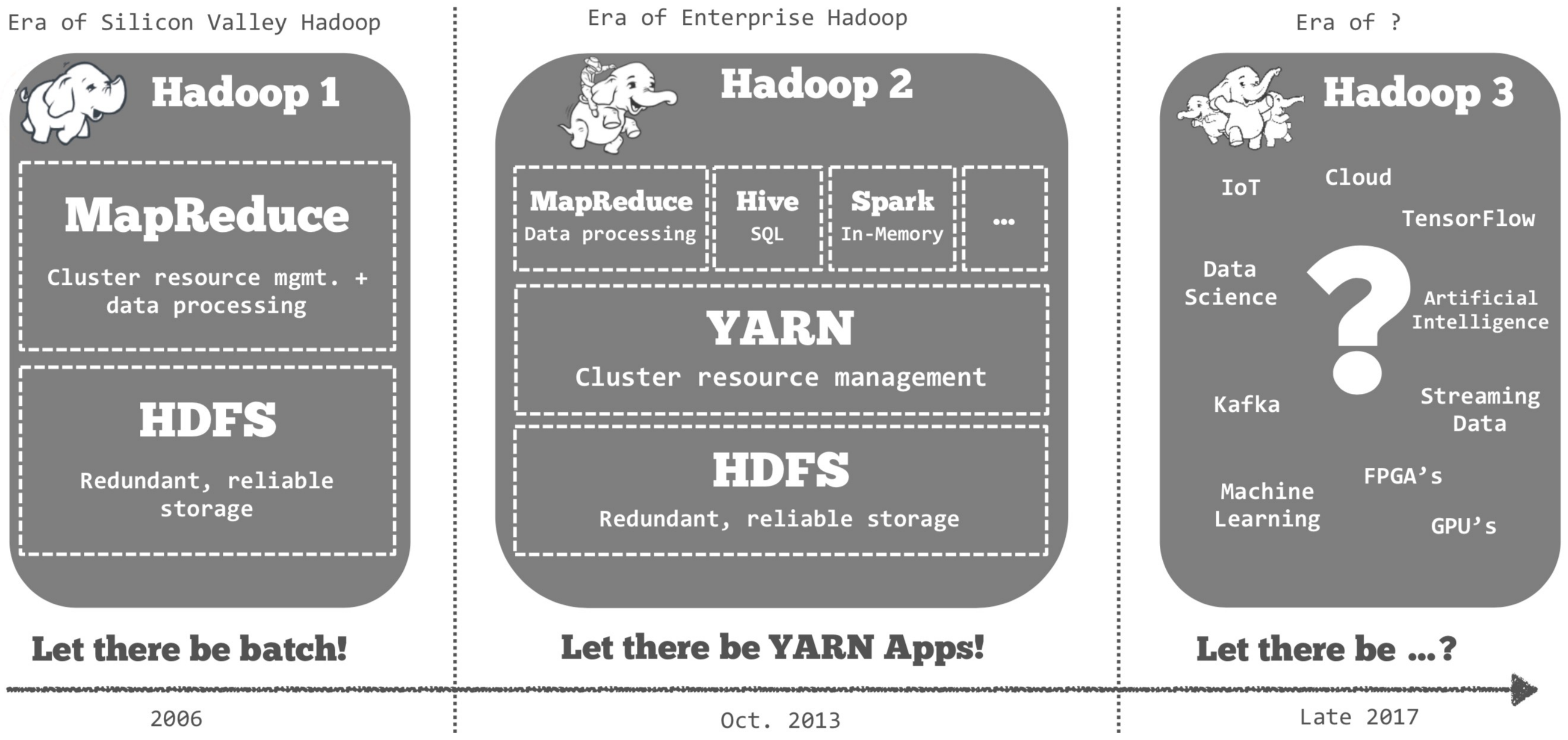
- The purpose of creating snapshots in HDFS is to minimize potential damage to the data stored in the system during upgrades.
- During software upgrades the possibility of corrupting the system due to software bugs or human mistakes increases.
- The snapshot mechanism lets administrators persistently save the current state of the file system (both data and metadata)
- If the upgrade results in data loss or corruption, it is possible to rollback the upgrade and return HDFS to the namespace and storage state as they were at the time of the snapshot.

Secondary Namenode

- **Secondary Namenode is a separate process**
  - Responsible for merging periodically edits and fsimage file to limit the size of edits file (checkpoint is not cheap)
  - Usually runs on a different machine than Namenode
  - Memory requirements are very similar to Namenode’s
- **Secondary Namenode uses the same directory structure as Namenode**
  - This checkpoint may be imported if Namenode's image is lost
- **Secondary Namenode is NOT**
  - Fail-over for Namenode
  - Doesn't provide high availability
  - Doesn't improve Namenode's performance

HDFS in Hadoop 3.0

Hadoop History



Erasure Coding

- Erasure coding stores the data and provide fault tolerance with less space overhead as compared to HDFS replication.

- Erasure Coding (EC) can be used in place of replication, which will provide the same level of fault-tolerance with less storage overhead.

- To understand Erasure Coding, let us introduce two terms
  - Durability: How many simultaneous failures can be tolerated?
  - Storage Efficiency: How much portion of storage is used for data?

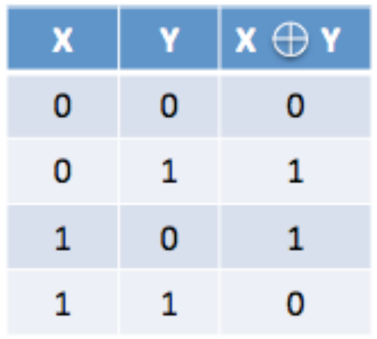
HDFS Erasure Coding

- There are two algorithms available for it:-
  - XOR algorithm(Simple EC Algo – single failure)
  - Reed-Solomon (RS) algorithm(Improved EC Algo – multiple failures)

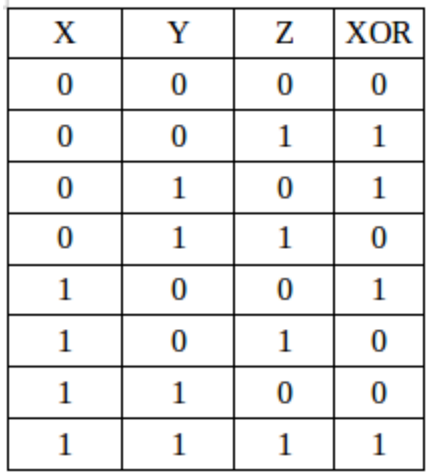
- Advantages of HDFS Erasure Coding in Hadoop
  - **Saving Storage –** Initially, blocks are triplicated when they are no longer changed by any additional data, after this, a background task encode it into codeword and delete its replicas.
  - **Two-way Recovery –** HDFS block errors are discovered and recovered not only during reading the path but also we can check it actively in the background.
  - **Low overhead –** Overhead is reduced from 200% to just 50% in RS encoding algorithm.

XOR (exclusive-or) algorithm

- It is also called parity bit erasure coding
- XOR operations are associative, meaning that $X \oplus Y \oplus Z = (X \oplus Y) \oplus Z$.
- This means that XOR can generate 1 parity bit from an arbitrary number of data bits
- For example, $1 \oplus 0 \oplus 1 \oplus 1 = 1$.

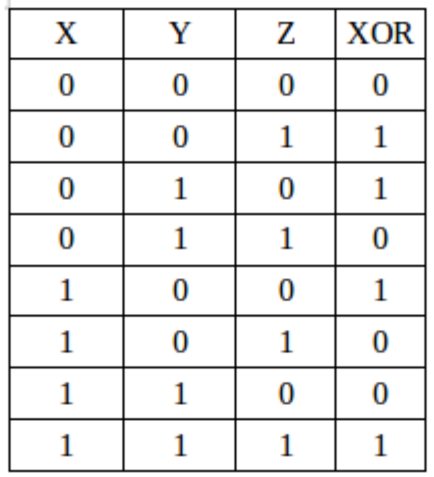


X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0



X	Y	Z	XOR
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

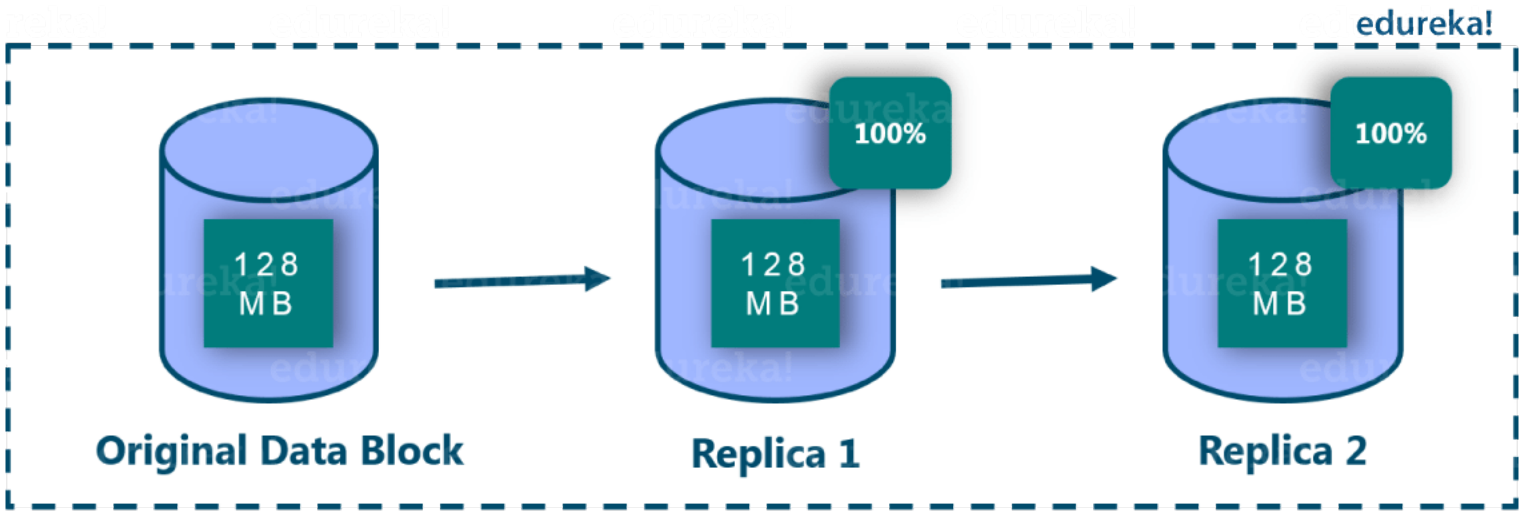
Parity bit - Illustration



X	Y	Z	XOR
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

- Assume we have 3 bits to store $(1,0,1)$
- The parity bit is $1 \oplus 0 \oplus 1 = 0$
- We store 4 bits $(1,0,1,0)$
- Scenario 1 :
  - We lost the third bit: we got $(1,0,?,0)$
  - Since the parity bit is 0, the number of 1 should be even, so the missing bit is 1
- Scenario 2 :
  - We lost the second bit ; we got $(1,?,1,0)$
  - Since the parity bit is 0, the number of 1 should be even, so the missing bit is 0

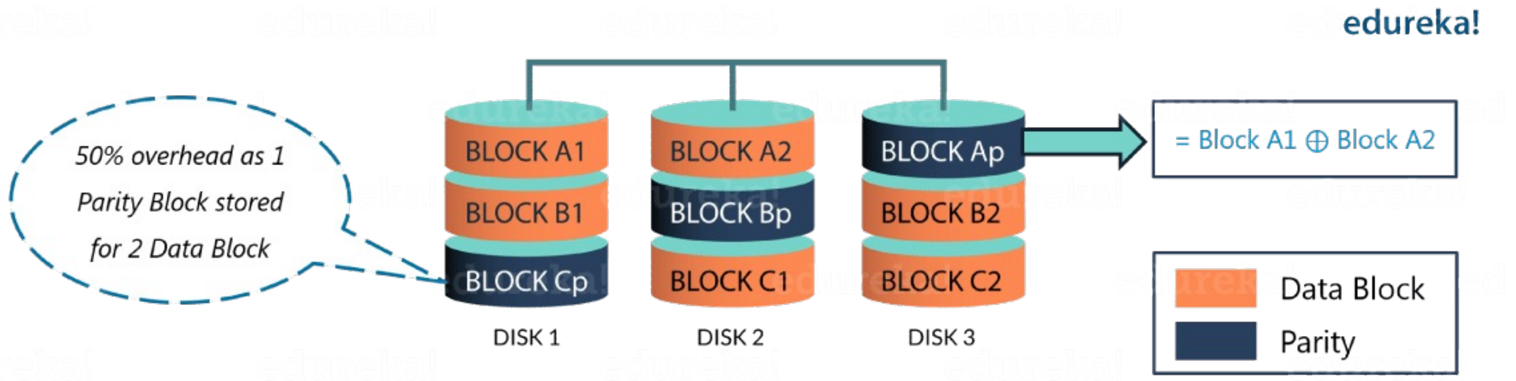
Support for Erasure Encoding in HDFS



- The default replication factor in HDFS is 3 in which one is the original data block and the other 2 are replicas which require 100% storage overhead each.
- So that makes 200% storage overhead and it consumes other resources like network bandwidth.

Storage Efficiency

- Integrating EC with HDFS can maintain the same fault-tolerance with improved storage efficiency.
- Example:
    - As an example, a 3x replicated file with 6 blocks will consume 6*3 = 18 blocks of disk space.
    - But with EC (6 data, 3 parity) deployment, it will only consume 9 blocks (6 data blocks + 3 parity blocks) of disk space. This only requires the storage overhead up to 50%.



Conclusion

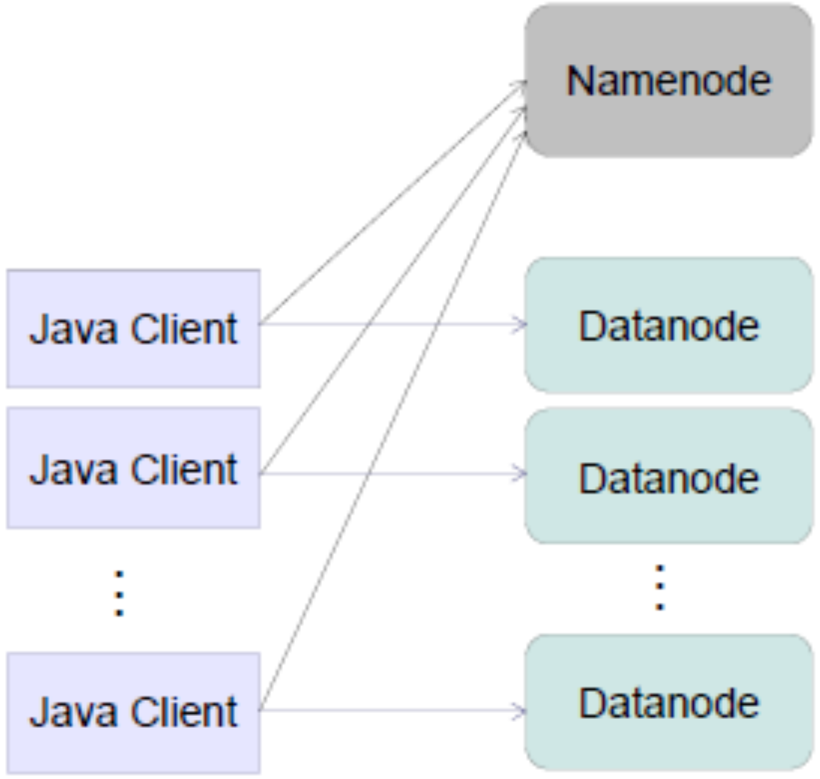
Conclusion

- Distributed file system is the heart of any distributed processing systems
- Not good for everything but very useful for very large files
- It is based on many processes called the HDFS Daemons (Namenode, etc.)
- Must be used with an appropriate processing framework (MapReduce, Spark, etc.)

HDFS Access

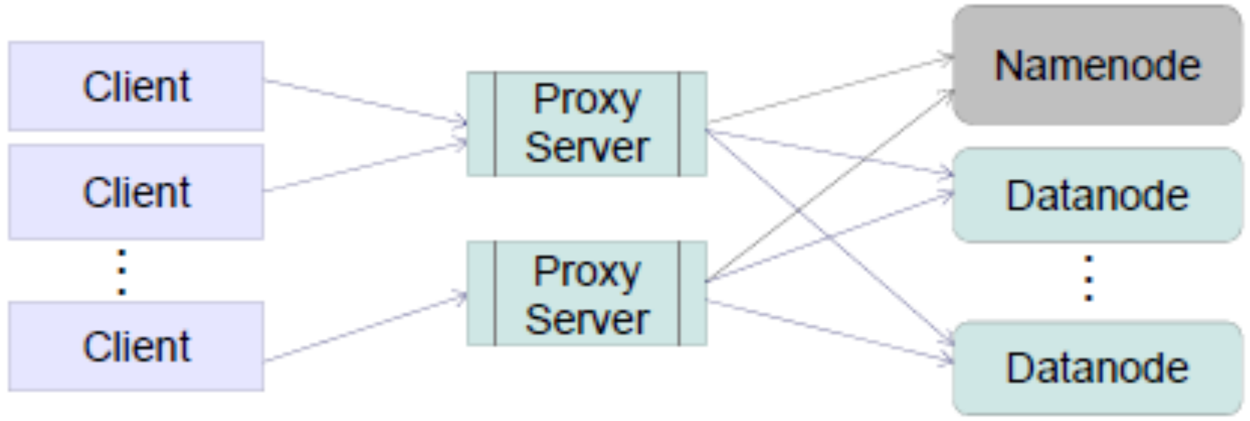
Direct Access

- Java, Scala and C++ APIs
- Clients retrieve metadata such as blocks' locations from Namenode
- Client directly access datanode(s)
- Used by MapReduce



Proxy Based Access

- Clients communicate through a proxy
  - Strives to be language independent
- Several Proxy Servers are packaged with Hadoop:
  - Thrift – interface definition language
  - WebHDFS REST – response formatted in JSON, XML or Protocol Buffers
  - Avro – Data Serialization mechanism



HDFS Configuration

Installation

- **Three options**
  - Local (Standalone) Mode
  - Pseudo-Distributed Mode
  - Fully-Distributed Mode (not presented here)

Installation: Local

- **Default configuration after the download**
- **Executes as a single Java process**
- **Works directly with local filesystem**
- **Useful for debugging**
- **Simple example, list all the files under /**

```
$ cd <hadoop_install>/bin
$ hdfs dfs -ls /
```

Installation: Pseudo-Distributed

- **Still runs on a single node**
- **Each daemon runs in it's own Java process**
  - Namenode
  - Secondary Namenode
  - Datanode
- **Location for configuration files is a dedicated directory**
  - For instance: /etc/hadoop/conf
  - Stored in the variable **$HADOOP_CONF_DIR** in the following
- **The main Hadoop configuration files are**
  - core-site.xml
  - hdfs-site.xml
  - hadoop-env.sh

Installation: Pseudo-Distributed

- **hadoop-env.sh**
  - Specify environment variables
    - Java and Log locations
  - Utilized by scripts that execute and manage hadoop
    - Required:

```
# The java implementation to use.
export JAVA_HOME=/usr/lib/jvm/java
```

    - Optionnal:

```
# Hadoop home directory
export  HADOOP_HOME=…
# Where log files are stored.
export HADOOP_LOG_DIR=/var/log/hadoop/$USER
```

Installation: Pseudo-Distributed

- **$HADOOP_CONF_DIR/core-site.xml**
- Configurations for core of Hadoop
- Specify location of Namenode
- Example:

```
<property>
    <name>fs.defaultFS</name>
    <value>hdfs://sandbox.hortonworks.com:8020</value>
    <final>true</final>
</property>
```

Installation: Pseudo-Distributed

- **$HADOOP_CONF_DIR/hdfs-site.xml**
  - Configurations for Namenode, Datanode and Secondary Namenode daemons

```
 <property>
    <name>dfs.namenode.name.dir</name>
    <value>/hadoop/hdfs/namenode</value>
    <final>true</final>
</property>

<property>
    <name>dfs.datanode.data.dir</name>
    <value>/hadoop/hdfs/data</value>
    <final>true</final>
</property>
```

Installation: Pseudo-Distributed

- Password-less SSH (Secure Shell) is required for Namenode to communicate with Datanodes
- In this case just to itself
- In the fully-distributed case, you need to copy your public key to all of your “slave” machine (don’t forget your secondary namenode).

Installation: Pseudo-Distributed

- **Prepare filesystem for use by formatting**

```
$ hdfs namenode -format
```

- **Start the distributed filesystem**

```
$ cd <hadoop_install>/sbin
$ start-dfs.sh
```

- **start-dfs.sh prints the location of the logs**

Management Web Interface

- **Namenode comes with web based management**

  http://localhost:50070

- **Features**
  - Cluster status
  - View Namenode and Datanode logs
  - Browse HDFS
- **Can be configured for SSL (https:) based access**
- **Secondary Namenode also has web UI**

  http://localhost:50090

Shell

Shell Commands

- **Interact with FileSystem by executing shell like commands**
- **Usage:**

  $hdfs dfs -<command> -<option> <URI>
  - Example $hdfs dfs -ls /
- **URI usage:**
  - HDFS: $hdfs dfs -ls hdfs://localhost/to/path/dir
  - Local: $hdfs dfs -ls file:///to/path/file3
  - Schema and namenode host is optional, default is used from the configuration
    - In core-site.xml - fs.default.name (or fs.defaultFS) property

Hadoop URI

- URI: Uniform Resource Identifier

- The URI generic syntax consists of a hierarchical sequence of five components:

  URI = scheme:[//authority]path[?query][#fragment]

- Example of Hadoop URI

$$\text{hdfs:}\underbrace{\text{//localhost:8020}}_{\text{Autority}}\underbrace{\text{/user/home}}_{\text{Path}}$$

  - The autority determines which file system implementation to use (in this example, it will be HDFS)
  - The autority (hostname - localhost - and the port - 8020 -) depend on the Hadoop cluster
  - The path « /user/home » determines the path on the file system.

https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

Shell Commands

- **Most commands behave like UNIX commands**
  - ls, cat, du, etc..
- **Supports HDFS specific operations**
  - Ex: changing replication
- **List supported commands**

```
$ hdfs dfs -help
```

- **Display detailed help for a command**

```
$ hdfs dfs -help <command_name>
```

Shell Commands

- **Relative Path**
- Is always relative to user's home directory
- Home directory is at /user/<username>
- **Shell commands follow the same format:**

```
$ hdfs dfs -<command> -<option> <path>
```

- **For example:**

```
$ hdfs dfs -rm -r /removeMe
```

Shell Basic Commands

- **cat – stream source to stdout**
  - entire file: $hdfs dfs -cat /dir/file.txt
  - Almost always a good idea to pipe to head, tail, more or less
  - Get the fist 25 lines of file.txt

```
$hdfs dfs -cat /dir/file.txt | head -n 25
```

- **cp – copy files from source to destination**

```
$hdfs dfs -cp /dir/file1 /otherDir/file2
```

- **ls – for a file displays stats, for a directory displays immediate children**

```
$hdfs dfs -ls /dir/
```

- **mkdir – create a directory**

```
$hdfs dfs -mkdir /brandNewDir
```

Moving Data with Shell

- **mv – move from source to destination**

```
$hdfs dfs -mv /dir/file1 /dir2/file2
```

- **put – copy file from local filesystem to hdfs**

```
$hdfs dfs -put localfile /dir/file1
```

  - Can also use copyFromLocal

```
$hdfs fs -copyFromLocal localfile /dir/file1
```

- **get – copy file to the local filesystem**

```
$hdfs dfs -get /dir/file localfile
```

  - Can also use copyToLocal

Deleting Data with Shell

- **rm – delete files**

```
$hdfs dfs -rm /dir/fileToDelete
```

- **•rm -r – delete directories recursively**

```
$hdfs dfs -rm -r /dirWithStuff
```

Filesystem Stats with Shell

- **du – displays length for each file/dir (in bytes)**

```
$hdfs dfs -du /someDir/
```

- **Add -h option to display in human-readable format instead of bytes**

```
$hdfs dfs -du -h /someDir
```

```
206.3k /someDir
```

Learn More About Shell

- **More commands**
  - tail, chmod, count, touchz, test, etc...
- **To learn more**

```
$hdfs dfs -help
$hdfs dfs -help <command>
```

- **For example:**

```
$ hdfs dfs -help rm
```

fsck Command

- **Check for inconsistencies**
- **Reports problems**
  - Missing blocks
  - Under-replicated blocks
- **Doesn't correct problems, just reports (unlike native fsck)**
  - Namenode attempts to automatically correct issues that fsck would report
- **$ hdfs fsck <path>**
  - Example : $ hdfs fsck /

HDFS Permissions

- **Limited to File permission**
  - Similar to POSIX model, each file/directory
  - has Read (r), Write (w) and Execute (x)
  - associated with owner, group or all others
- **Client's identity determined on host OS**
  - Username = \`whoami\`
  - Group = \`bash -c groups\`
- **Authentication and Authorization with Kerberos**
- Hadoop set-up with Kerberos is beyond the scope of this class

DFSAdmin Command

- **HDFS administrative operations**

```
$hdfs dfsadmin <command>
```

  - Example: $hdfs dfsadmin -report
- **-report : displays statistic about HDFS**
  - Some of these stats are available on Web Interface
- **-safemode : enter or leave safemode**
  - Maintenance, backups, upgrades, etc..
- **Safemode:**
  - HDFS cluster read-only mode
  - Modifications to filesystem and blocks are not allowed

Rebalancer

- **Data on HDFS Clusters may not be uniformly spread between available Datanodes.**
  - Ex: New nodes will have significantly less data for some time
  - The location for new incoming blocks will be chosen based on status of Datanode topology, but the cluster doesn't automatically rebalance
- **Rebalancer is an administrative tool that analyzes block placement on the HDFS cluster and re-balances**

```
$ hdfs balancer
```

Rebalancer

- **Data on HDFS Clusters may not be uniformly spread between available Datanodes.**
  - Ex: New nodes will have significantly less data for some time
  - The location for new incoming blocks will be chosen based on status of Datanode topology, but the cluster doesn't automatically rebalance
- **Rebalancer is an administrative tool that analyzes block placement on the HDFS cluster and re-balances**

```
$ hdfs balancer
```