Welcome to Big Data Analytics Module

Instructor : Tushar Kakaiya

HDFS – Hadoop Distributed File System

HDFS Basics

- When a dataset outgrows the storage capacity of a single physical machine, it becomes necessary to partition it across a number of separate machine
- Filesystems that manage the storage across a network of machines are called **distributed filesystems**.
- Since they are network based, all the complications of network programming kick in, thus making distributed filesystems more x than
  filesystems.

  For example, one of the biggest challenges is making the filesystem tolerate node failure without suffering dat oss.
- Hadoop comes with a distributed filesystem called HDFS, which stands for **Hadoop Distributed Filesystem**.
- HDFS is Hadoop's flagship filesystem. Other examples are local File System or Amazon S3.

References to "DFS"
older docume
configurations— is the

Design of HDFS

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

Very large files

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

Streaming data access

- HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. A dataset is typically generated or copied from source, and then various analyses are performed on that dataset over time.
- Each analysis will involve a large proportion, if not all, of the dataset, so the time to read the whole dataset is more important than the latency in reading the first record.

Commodity hardware

- Hadoop doesn't require expensive, highly reliable hardware. It's designed to run on clusters of commodity hardware (commonly available hardware that can be obtained from multiple vendors) for which the chance of node failure across the cluster is high, at least for large clusters.

- HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure.

HDFS is not a good fit for...

Low-latency data access

Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS. Remember, HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency. HBase is currently a better choice for low-latency access.

Lots of small files

Because the namenode holds filesystem metadata in memory, the limit to the number of files in a filesystem is governed by the amount of memory on the namenode.

As a rule of thumb, each file, directory, and block takes about 150 bytes. So, for example, if you had one million files, each taking one block, you would need at least 300 MB of memory. Although storing millions of files is feasible, billions is beyond the capability of current hardware.

Latency is the time required to perform some action or to produce some result. **Latency** is measured in units of time -- hours, minutes, seconds, nanoseconds or clock periods. **Throughput** is the number of such actions executed or results produced per unit of time.

Real Time Water Pipe Example
When you go to buy a water pipe, there are two completely independent parameters that you look at: the diameter of the pipe and its length. The diameter determines the throughput of the pipe and the length determines the latency, i.e., the time it will take for a water droplet to travel across the pipe. Key point to note is that the length and diameter are independent, thus, so are are latency and throughput of a communication channel.
More formally, Throughput is defined as the amount of water entering or leaving the pipe every second and latency is the average time required to for a droplet to travel from one end of the pipe to the other.

HDFS is not a good fit for…

Multiple writers, arbitrary file modifications

Files in HDFS may be written to by a single writer. Writes are always made at the end of the file, in append-only fashion. There is no support for multiple writers or for modifications at arbitrary offsets in the file.

HDFS Concepts – Blocks

- Block size, is the minimum amount of data that it can read or write.
- HDFS has the concept of a block:

128 MB by default in Hadoop 2.x versions

64 MB in Hadoop 1.x and 0.x versions

- Files in HDFS are broken into block-sized chunks, which are stored as independent units.
- A file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage.

  For example, a 1 MB file stored with a block size of 128 MB uses 1 MB of disk space, not 128 MB.

- The default is actually 128 MB, although many HDFS installations use larger block sizes. This figure will continue to be revised upward as transfer speeds grow with new generations of disk drives.

HDFS Concepts – Replication Factor

- To insure against corrupted blocks and disk and machine failure, each block is replicated to a small number of physically separate machines (typically three).
- If a block becomes unavailable, a copy can be read from another location in a way that is transparent to the client. A block that is no longer available due to corruption or machine failure can be replicated from its alternative locations to other live machines to bring the replication factor back to the normal level.
- Similarly, some applications may choose to set a high replication factor for the blocks in a popular file to spread the read load on the cluster.

HDFS Daemons – NameNodes and DataNodes

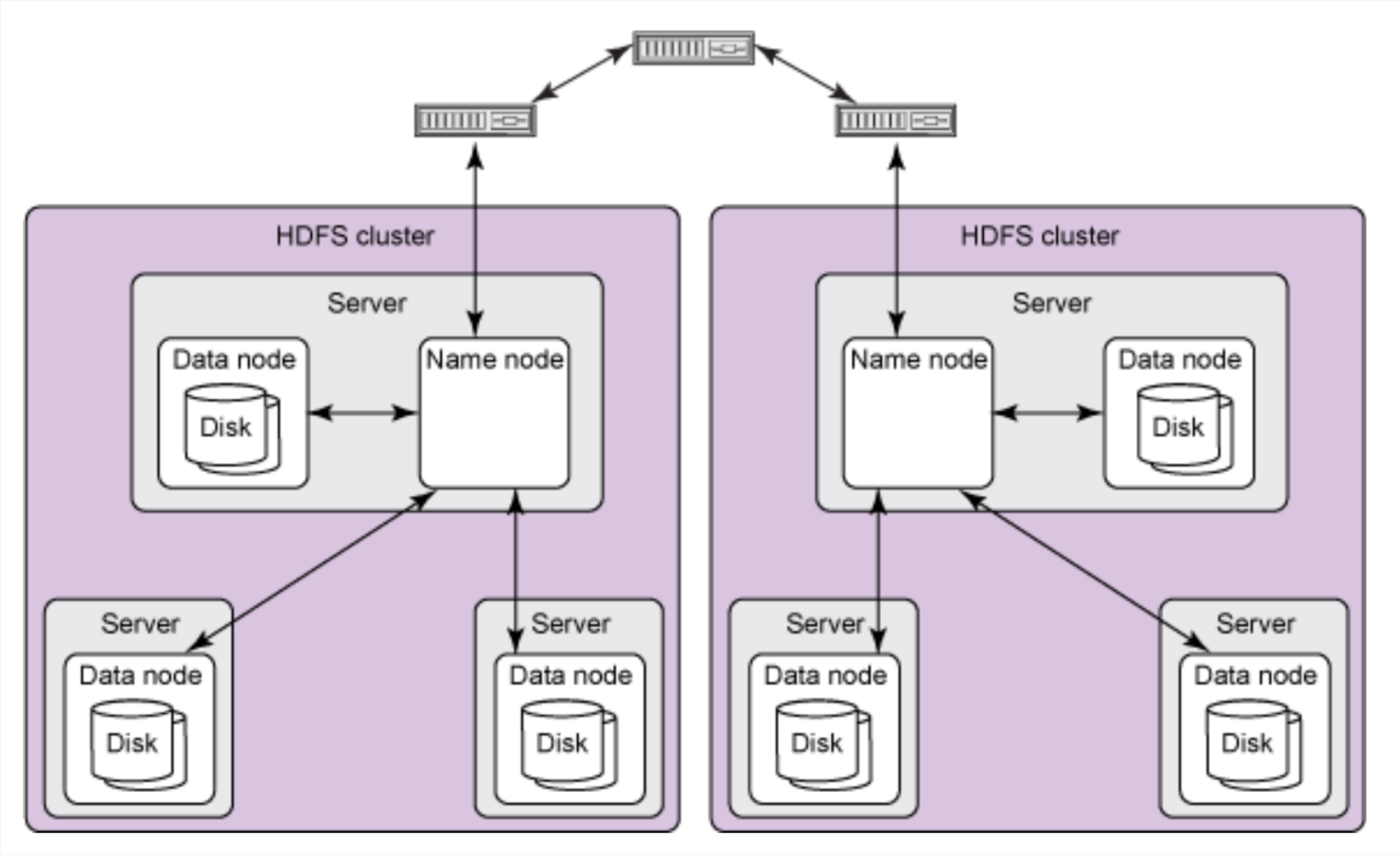
-An HDFS cluster has two types of nodes operating in a master–worker pattern:

- Namenode (the master)
- Datanodes (workers).

-The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log.

-The namenode also knows the datanodes on which all the blocks for a given file are located; however, it does not store block locations persistently, because this information is reconstructed from datanodes when the system starts.

HDFS Daemons – NameNodes and DataNodes



HDFS Daemons – NameNodes and DataNodes

-A client accesses the filesystem on behalf of the user by communicating with the namenode and datanodes. The client presents a filesystem interface similar to a Portable Operating System Interface (POSIX), so the user code does not need to know about the namenode and datanodes to function.

-Datanodes are the workhorses of the filesystem. They store and retrieve blocks when they are told to (by clients or the namenode), and they report back to the namenode periodically with lists of blocks that they are storing.

Making NameNode Resilient to Failure

Without the namenode, the filesystem cannot be used. In fact, if the machine running the namenode were obliterated, all the files on the filesystem would be lost since there would be no way of knowing how to reconstruct the files from the blocks on the datanodes.

For this reason, it is important to make the namenode resilient to failure, and Hadoop provides two mechanisms for this.

- The first way is to back up the files that make up the persistent state of the filesystem metadata.
    - Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems. These writes are synchronous and atomic. The usual configuration choice is to write to local disk as well as a remote NFS mount.
- It is also possible to run a *secondary namenode*, which despite its name does not act as a namenode.
    - Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. The secondary namenode usually runs on a separate physical machine because it requires plenty of CPU and as much memory as the namenode to perform the merge. It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing.
    - However, the state of the secondary namenode lags that of the primary, so in the event of total failure of the primary, data loss is almost certain.

Making NameNode Resilient to Failure

HDFS High Availability

- The combination of replicating namenode metadata on multiple filesystems and using the secondary namenode to create checkpoints protects against data loss, but it does not provide high availability of the filesystem. The namenode is still a *single point of failure* (SPOF).
- If it did fail, all clients—including MapReduce jobs—would be unable to read, write, or list files, because the namenode is the sole repository of the metadata and the file-to-block mapping. In such an event, the whole Hadoop system would effectively be out of service until a new namenode could be brought online.
- Hadoop 2 remedied this situation by adding support for HDFS high availability (HA).
- In this implementation, there are a pair of namenodes in an active-standby configuration.
- In the event of the failure of the active namenode, the standby takes over its duties to continue servicing client requests without a significant interruption.

Making NameNode Resilient to Failure

HDFS High Availability

A few architectural changes are needed to allow this to happen:

- The namenodes must use highly available shared storage to share the edit log.
- When a standby namenode comes up, it reads up to the end of the shared edit log to synchronize its state with the active namenode, and then continues to read new entries as they are written by the active namenode.
- Datanodes must send block reports to both namenodes because the block mappings are stored in a namenode's memory, and not on disk.
- Clients must be configured to handle namenode failover, using a mechanism that is transparent to users.
- The secondary namenode's role is subsumed by the standby, which takes periodic checkpoints of the active namenode's namespace.
- That means the Standby namenode should also do Secondary namenode's task of checkpointing periodic backup of active's namespace images.

Some Architectural Terms...

- The HDFS namespace is a hierarchy of files and directories.
- Files and directories are represented on the NameNode by inodes.
- Inodes record attributes like permissions, modification and access times, namespace and disk space quotas.
- The file content is split into large blocks (typically 128 megabytes, but user selectable file-by-file), and each block of the file is independently replicated at multiple DataNodes (typically three, but user selectable file-by-file).
- The NameNode maintains the namespace tree and the mapping of blocks to DataNodes. The current design has a single NameNode for each cluster. The cluster can have thousands of DataNodes and tens of thousands of HDFS clients per cluster, as each DataNode may execute multiple application tasks concurrently.

Image and Journal

- The inodes and the list of blocks that define the metadata of the name system are called the *image*. NameNode keeps the entire namespace image in RAM.
- The persistent record of the image stored in the NameNode's local native filesystem is called a checkpoint. The NameNode records changes to HDFS in a write-ahead log called the journal in its local native filesystem. The location of block replicas are not part of the persistent checkpoint.

HDFS High Availability

- If the active namenode fails, the standby can take over very quickly (in a few tens of seconds) because it has the latest state available in memory: both the latest edit log entries and an up-to-date block mapping.

- The actual observed failover time will be longer in practice (around a minute or so), because the system needs to be conservative in deciding that the active namenode has failed.

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

Block Caching

- Normally a datanode reads blocks from disk, but for frequently accessed files the blocks may be explicitly cached in the datanode's memory, in an off-heap *block cache*.
- By default, a block is cached in only one datanode's memory, although the number is configurable on a per-file basis. Job schedulers (for MapReduce, Spark, and other frameworks) can take advantage of cached blocks by running tasks on the datanode where a block is cached, for increased read performance.
- A small lookup table used in a join is a good candidate for caching.
- Users or applications instruct the namenode which files to cache (and for how long) by adding a *cache directive* to a *cache pool*. Cache pools are an administrative grouping for managing cache permissions and resource usage.

HDFS Federation

- The namenode keeps a reference to every file and block in the filesystem in memory, which means that on very large clusters with many files, memory becomes the limiting factor for scaling.
- HDFS federation, introduced in the 2.x release series, allows a cluster to scale by adding namenodes, each of which manages a portion of the filesystem namespace. For example, one namenode might manage all the files rooted under */user*, say, and a second namenode might handle files under */share*.
- Under federation, each namenode manages a *namespace volume*, which is made up of the metadata for the namespace, and a *block pool* containing all the blocks for the files in the namespace.
- Namespace volumes are independent of each other, which means namenodes do not communicate with one another, and furthermore the failure of one namenode does not affect the availability of the namespaces managed by other namenodes.

Failover and Fencing

- The transition from the active namenode to the standby is managed by a new entity in the system called **the *failover controller***. There are various failover controllers, but the default implementation uses **ZooKeeper** to ensure that only one namenode is active.
- Each namenode runs a lightweight failover controller process whose job it is to monitor its namenode for failures (using a simple heart beating mechanism) and trigger a failover should a namenode fail.
- Failover may also be initiated manually by an administrator, for example, in the case of routine maintenance. This is known as a ***graceful failover***, since the failover controller arranges an orderly transition for both namenodes to switch roles.
- In the case of an ungraceful failover, however, it is impossible to be sure that the failed namenode has stopped running. For example, a slow network or a network partition can trigger a failover transition, even though the previously active namenode is still running and thinks it is still the active namenode. The HA implementation goes to great lengths to ensure that the previously active namenode is prevented from doing any damage and causing corruption—a method known as ***fencing***.

Failover and Fencing

- The System only allows one namenode to write to the edit log at one time; however, it is still possible for the previously active namenode to serve stale read requests to clients, so setting up an SSH fencing command that will kill the namenode's process is a good idea.
- Stronger fencing methods are required when using an NFS filer for the shared edit log, since it is not possible to only allow one namenode to write at a time. The range of fencing mechanisms includes revoking the namenode's access to the shared storage directory, and disabling its network port via a remote management command.
- As a last resort, the previously active namenode can be fenced with a technique rather graphically known as *STONITH*, or "shoot the other node in the head," which uses a specialized power distribution unit to forcibly power down the host machine.
- Client failover is handled transparently by the client library. The simplest implementation uses client-side configuration to control failover. The HDFS URI uses a logical hostname that is mapped to a pair of namenode addresses (in the configuration file), and the client library tries each namenode address until the operation succeeds.

Network Topology and Hadoop

- What does it mean for two nodes in a local network to be “close” to each other? In the context of high-volume data processing, the limiting factor is the rate at which we can transfer data between nodes—bandwidth is a scarce commodity. The idea is to use the bandwidth between two nodes as a measure of distance.
- Measuring bandwidth between nodes, can be difficult as it requires a quiet cluster, and the number of pairs of nodes in a cluster grows as the square of the number of nodes,
- Hadoop takes a simple approach in which the network is represented as a tree and the distance between two nodes is the sum of their distances to their closest common ancestor. Levels in the tree are not predefined, but it is common to have levels that correspond to the data center, the rack, and the node that a process is running on.
- The idea is that the bandwidth available for each of the following scenarios becomes progressively less:

-Processes on the same node

-Different nodes on the same rack

-Nodes on different racks in the same data center

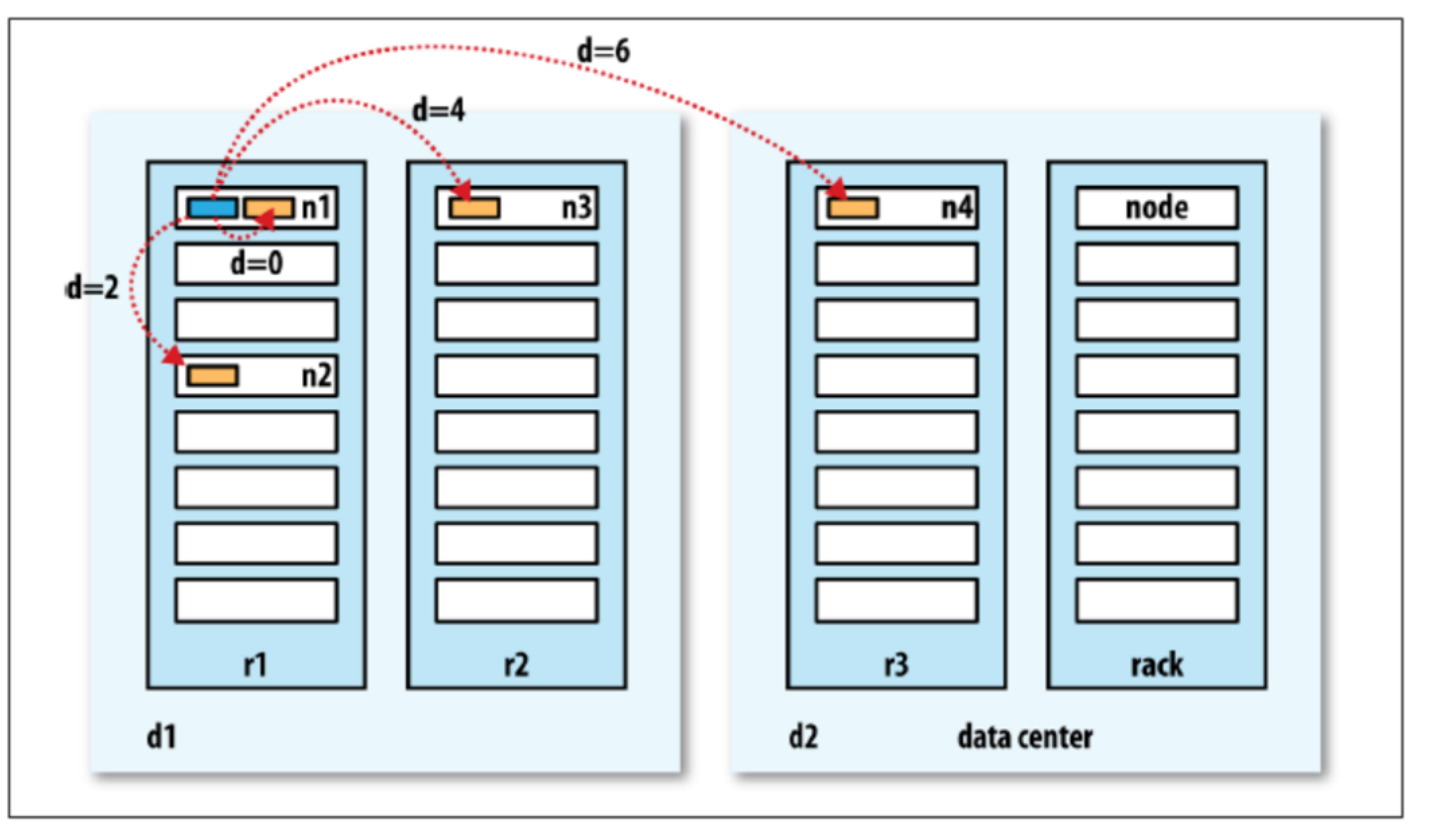
-Nodes in different data centers

Network Topology and Hadoop

For example, imagine a node *n1* on rack *r1* in data center *d1*. This can be represented as */d1/r1/n1*. Using this notation, here are the distances for the four scenarios:

- *distance(/d1/r1/n1, /d1/r1/n1)* = 0 (processes on the same node)
- *distance(/d1/r1/n1, /d1/r1/n2)* = 2 (different nodes on the same rack)
- *distance(/d1/r1/n1, /d1/r2/n3)* = 4 (nodes on different racks in the same data center)
- *distance(/d1/r1/n1, /d2/r3/n4)* = 6 (nodes in different data centers)

Network Topology and Hadoop



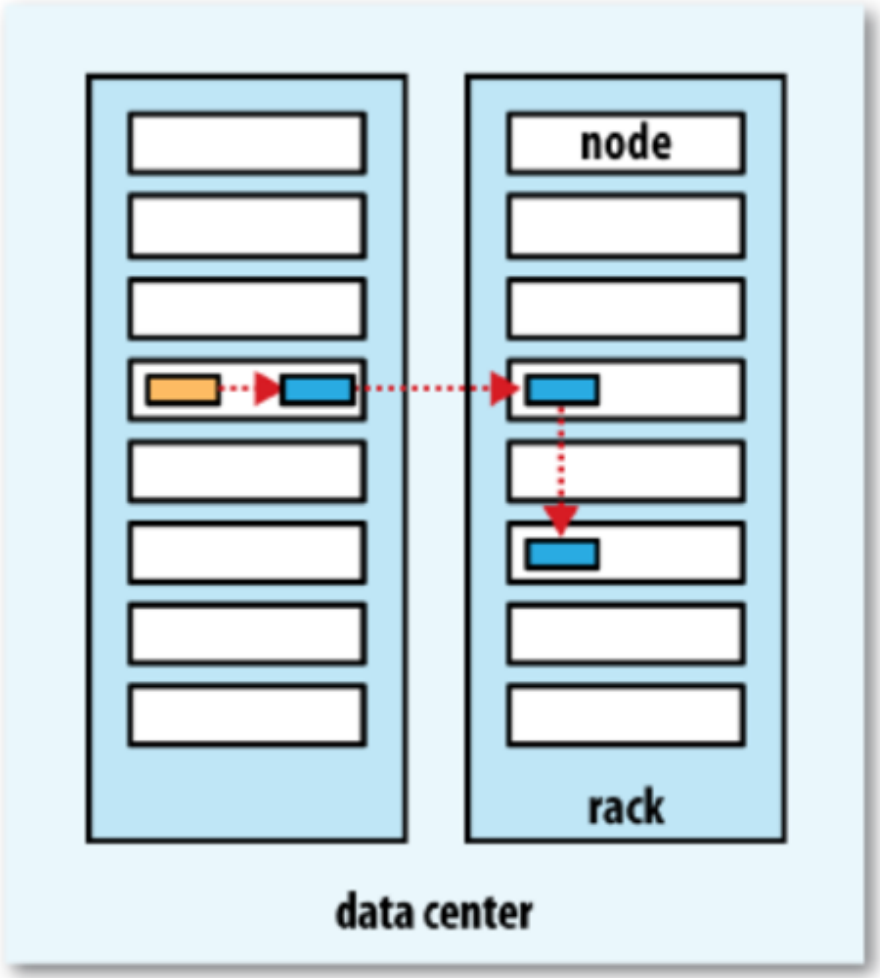
Mathematically inclined readers will notice that this is an example of a distance metric.

Replica Placement

- How does the namenode choose which datanodes to store replicas on? There's a tradeoff between reliability and write bandwidth and read bandwidth here.
- For example, placing all replicas on a single node incurs the lowest write bandwidth penalty (since the replication pipeline runs on a single node), but this offers no real redundancy (if the node fails, the data for that block is lost).
- Also, the read bandwidth is high for off-rack reads. At the other extreme, placing replicas in different data centers may maximize redundancy, but at the cost of bandwidth. Even in the same data center (which is what all Hadoop clusters to date have run in), there are a variety of possible placement strategies.

Replica Placement

- Hadoop's default strategy is to place the first replica on the same node as the client (for clients running outside the cluster, a node is chosen at random, although the system tries not to pick nodes that are too full or too busy).
- The second replica is placed on a different rack from the first (*off-rack*), chosen at random.
- The third replica is placed on the same rack as the second, but on a different node chosen at random.
- Further replicas are placed on random nodes in the cluster, although the system tries to avoid placing too many replicas on the same rack.
- Once the replica locations have been chosen, a pipeline is built, taking network topology into account. For a replication factor of 3, the pipeline might look like Figure:
- Overall, this strategy gives a good balance among reliability (blocks are stored on two racks), write bandwidth (writes only have to traverse a single network switch), read performance (there's a choice of two racks to read from), and block distribution across the cluster (clients only write a single block on the local rack).



Keeping an HDFS Cluster Balanced

- When copying data into HDFS, it's important to consider cluster balance. HDFS works best when the file blocks are evenly spread across the cluster, so you want to ensure that *distcp* doesn't disrupt this. For example, if you specified -m 1, a single map would do the copy, which—apart from being slow and not using the cluster resources efficiently — would mean that the first replica of each block would reside on the node running the map (until the disk filled up).
- The second and third replicas would be spread across the cluster, but this one node would be unbalanced. By having more maps than nodes in the cluster, this problem is avoided. For this reason, it's best to start by running *distcp* with the default of 20 maps per node.
- However, it's not always possible to prevent a cluster from becoming unbalanced. Perhaps you want to limit the number of maps so that some of the nodes can be used by other jobs. In this case, you can use the *balancer* tool to subsequently even out the block distribution across the cluster.

HDFS Practical Demo – Command Line Interface

- HDFS replicates file blocks for fault tolerance. An application can specify the number of replicas of a file at the time it is created, and this number can be changed any time after that. The name node makes all decisions concerning block replication.
- HDFS uses an intelligent replica placement model for reliability and performance. Optimizing replica placement makes HDFS unique from most other distributed file systems, and is facilitated by a rack-aware replica placement policy that uses network bandwidth efficiently.

Rack awareness

- Typically, large HDFS clusters are arranged across multiple installations (racks). Network traffic between different nodes within the same installation is more efficient than network traffic across installations. A name node tries to place replicas of a block on multiple installations for improved fault tolerance. However, HDFS allows administrators to decide on which installation a node belongs. Therefore, each node knows its rack ID, making it *rack aware*.
- When running with a single datanode, HDFS can't replicate blocks to three datanodes, so it would perpetually warn about blocks being under-replicated. This setting solves that problem.
- File where this property is set: /usr/lib/hadopp-0.20-mapreduce/conf/hdfs-site.xml

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
```

HDFS PRACTICAL DEMO – COMMAND LINE INTERFACE

- There are many other interfaces to HDFS, but the command line is one of the simplest and, to many developers, the most familiar.
- There are two properties that we set in the pseudo distributed configuration that deserve further explanation.
- The first is **fs.defaultFS**, set to hdfs://localhost/, which is used to set a default filesystem for Hadoop. Filesystems are specified by a URI, and hdfs URI is used to configure Hadoop to use HDFS by default.
- The HDFS daemons will use this property to determine the host and port for the HDFS namenode.
- Run it on localhost, on the default **HDFS port**, **8020**. And HDFS
- clients will use this property to work out where the namenode is running so they can connect to it.
- We can check this from Cloudera Machine at below mentioned File path : /usr/lib/hadoop-0.20-mapreduce/conf/core-site.xml

```
<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://quickstart.cloudera:8020</value>
  </property>
```

host name

HDFS PRACTICAL DEMO – BASIC FILE SYSTEM OPERATIONS

- Fs is a Hadoop file system shell.
- Most of Linux commands can be used to work with HDFS with prefix **hadoop dfs/ hadoop fs** command.
- The filesystem is ready to be used, and we can do all of the usual filesystem operations, such as reading files, creating directories, moving files, deleting data, and listing directories.
- You can type **hadoop fs -help** to get detailed help on every command.

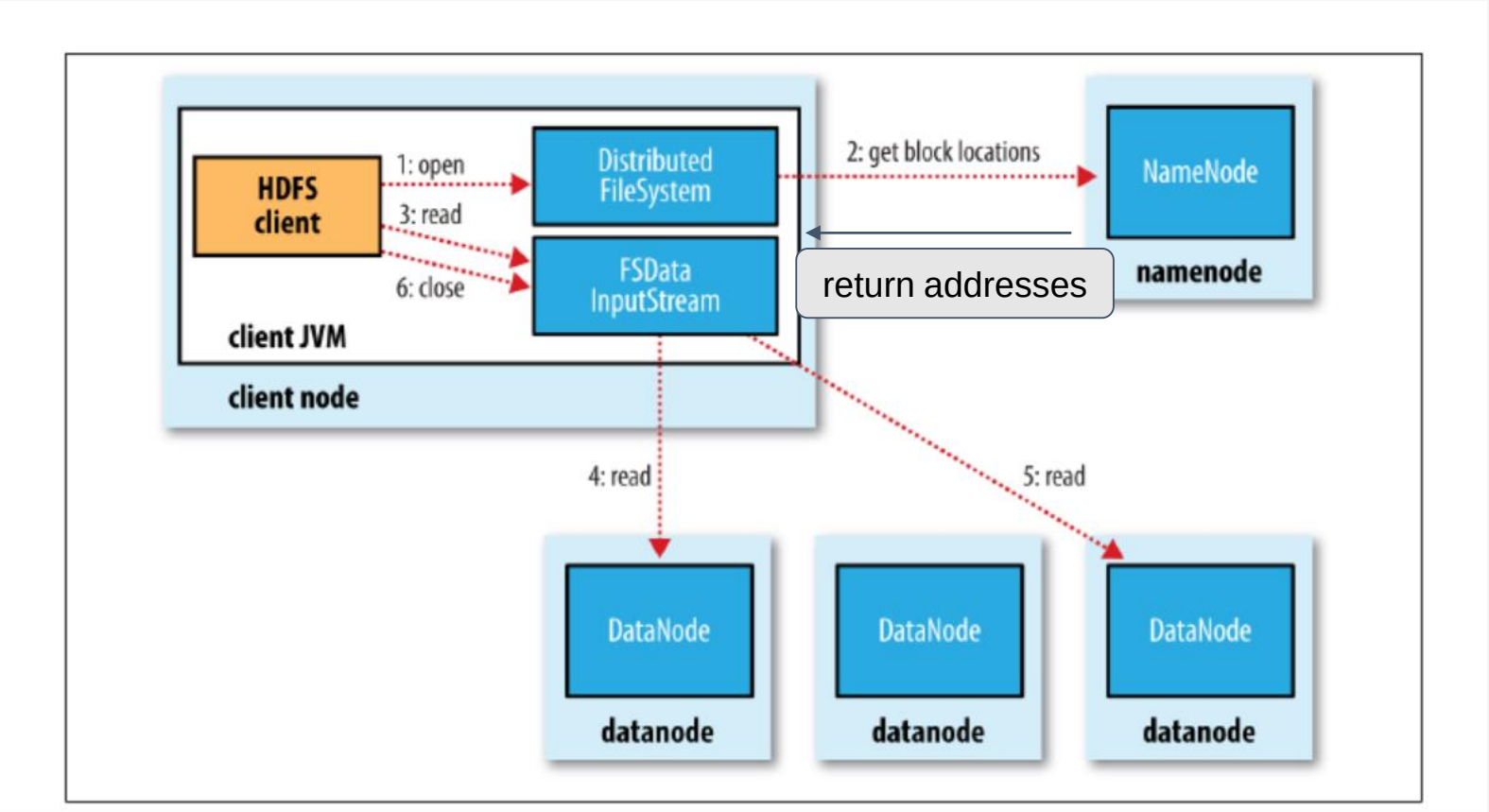
Let's try Practical on Cloudera QuickStart 5.12 Machine.

HDFS PRACTICAL DEMO – BASIC FILE SYSTEM OPERATIONS

Command	Usage
hadoop fs –ls	To list the files in HDFS file system
hadoop fs -lsr	To list all the files including its subdirectories and file lengths
hadoop fs -cat Type_hdfs_file_ path	To display content of a file in HDFS
hadoop fs –mkdir TestDir	To create a directory in Hdfs
hadoop fs –copyFromLocal Type_LocalMachine_file_ path Type_hdfs_Path_WhereToCopy Or hadoop fs –put Type_LocalMachine_file_ path Type_hdfs_Path_WhereToCopy Note – to check use cat	To copy a file from local file system to Hdfs file system
hadoop fs –copyToLocal Type_hdfs_Path_FromWhereToCopy Type_LocalMachine_file_ path Or hadoop fs –get Type_hdfs_Path_FromWhereToCopy Type_LocalMachine_file_ path	To copy a file from Hdfs to Local file system

DATA FLOW – ANATOMY OF A FILE READ

To get an idea of how data flows between the client interacting with HDFS, the namenode, and the datanodes, consider Figure shown below, which shows the main sequence of events when reading a file.

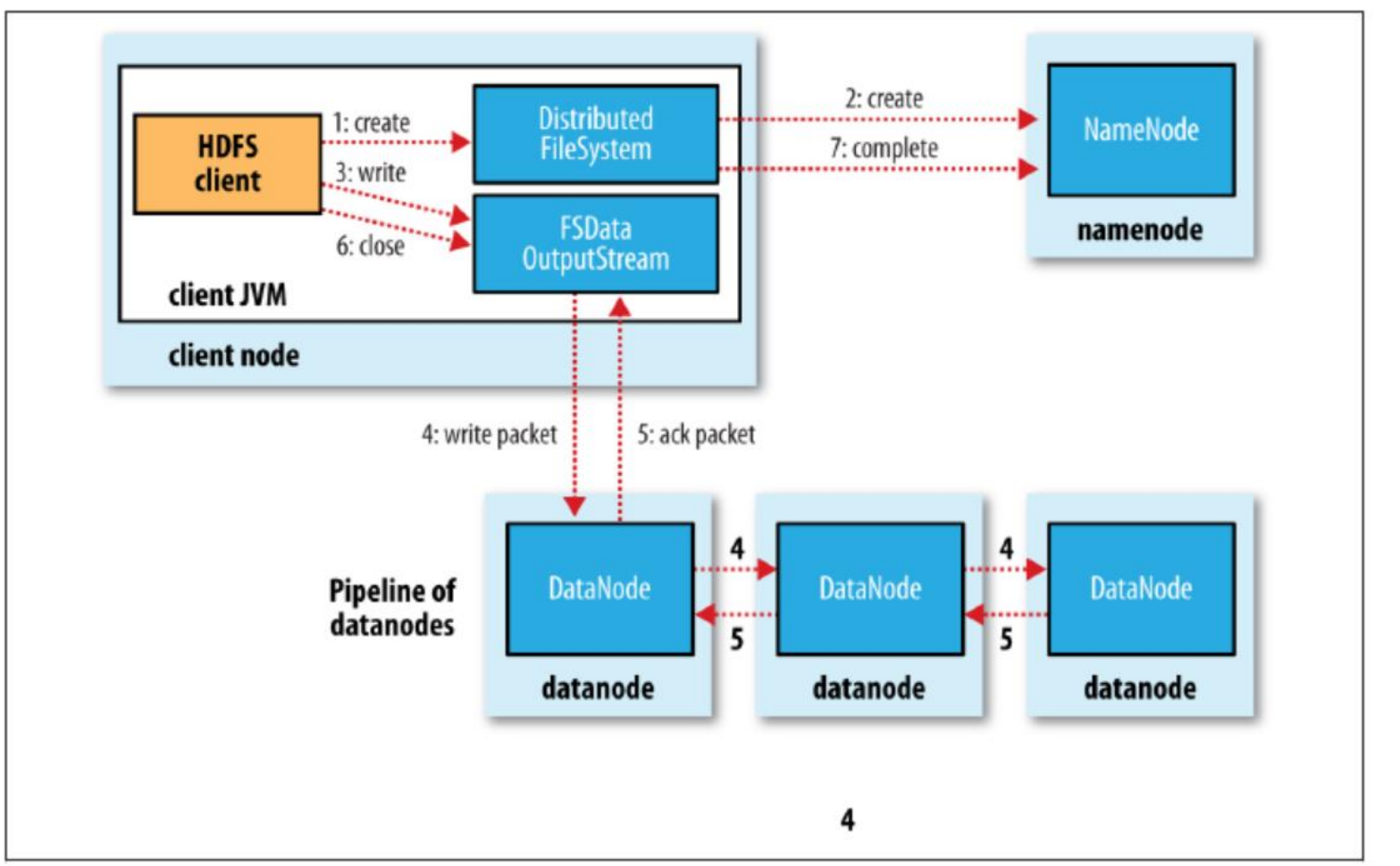


DATA FLOW – ANATOMY OF A FILE READ

- The client opens the file it wishes to read by calling open() on the FileSystem object, which for HDFS is an instance of DistributedFileSystem (step 1 in Figure).
- DistributedFileSystem calls the namenode, using remote procedure calls (RPCs), to determine the locations of the first few blocks in the file (step 2). 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 (step 3). 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 (step 4). 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 (step 5). 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 (step 6).
- During reading, if the DFSInputStream encounters an error while communicating with a datanode, it will try the next closest one for that block. It will also remember datanodes that have failed so that it doesn't needlessly retry them for later blocks. The DFSInputStream also verifies checksums for the data transferred to it from the datanode. If a corrupted block is found, the DFSInputStream attempts to read a replica of the block from another datanode; it also reports the corrupted block to the namenode.

DATA FLOW – ANATOMY OF A FILE WRITE

- Let's consider how files are written to HDFS. We're going to consider the case of creating a new file, writing data to it, then closing the file.



Data Flow – Anatomy of a File Write

- The client creates the file by calling create() on DistributedFileSystem (step 1 in Diagram).
- DistributedFileSystem makes an RPC call to the namenode to create a new file in the filesystem's namespace, with no blocks associated with it (step 2).
- The namenode performs various checks to make sure the file doesn't already exist and that the client has the right permissions to create the file.
- If these checks pass, the namenode makes a record of the new file; otherwise, file creation fails and the client is thrown an IOException.
- The DistributedFileSystem returns an FSDataOutputStream for the client to start writing data to. Just as in the read case, FSDataOutputStream wraps a DFSOutputStream, which handles communication with the datanodes and namenode.
- As the client writes data (step 3), 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.
- The list of datanodes forms a pipeline, and here we'll assume the replication level is three, so there are three nodes in the pipeline.
- 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 (step 4).
- 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 (step 5).

DATA FLOW – ANATOMY OF A FILE WRITE

- If any datanode fails while data is being written to it, then the following actions are taken, which are transparent to the client writing the data. First, the pipeline is closed, and any packets in the ack queue are added to the front of the data queue so that datanodes that are downstream from the failed node will not miss any packets.
- The current block on the good datanodes is given a new identity, which is communicated to the namenode, so that the partial block on the failed datanode will be deleted if the failed datanode recovers later on.
- The failed datanode is removed from the pipeline, and a new pipeline is constructed from the two good datanodes. The remainder of the block's data is written to the good datanodes in the pipeline.
- The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node. Subsequent blocks are then treated as normal.
- It's possible, but unlikely, for multiple datanodes to fail while a block is being written.
- As long as dfs.namenode.replication.min replicas (which defaults to 1) are written, the write will succeed, and the block will be asynchronously replicated across the cluster until its target replication factor is reached (dfs.replication, which defaults to 3).
- When the client has finished writing data, it calls close() on the stream (step 6). 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 (step 7).
- The namenode already knows which blocks the file is made up of (because Data Streamer asks for block allocations), so it only has to wait for blocks to be minimally replicated before returning successfully.