HDFS

Introduction

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. HDFS relaxes a few POSIX requirements to enable streaming access to file system data. HDFS was originally built as infrastructure for the Apache Nutch web search engine project. HDFS is now an Apache Hadoop subproject. The project URL is https://hadoop.apache.org/hdfs/.

Assumptions and Goals

- **Hardware Failure**

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

- **Streaming Data Access**

Applications that run on HDFS need streaming access to their data sets. They are not general purpose applications that typically run on general purpose file systems. HDFS is designed more for batch processing rather than interactive use by users. The emphasis is on high throughput of data access rather than low latency of data access. POSIX imposes many hard requirements that are not needed for applications that are targeted for HDFS. POSIX semantics in a few key areas has been traded to increase data throughput rates.

- **Large Data Sets**

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

- **Simple Coherency Model**

HDFS applications need a write-once-read-many access model for files. A file once created, written, and closed need not be changed. This assumption simplifies data coherency issues and enables high throughput data access. A MapReduce application or a web crawler application fits perfectly with this model. There is a plan to support appending-writes to files in the future.

- **"Moving Computation is Cheaper than Moving Data"**

A computation requested by an application is much more efficient if it is executed near the data it operates on. This is especially true when the size of the data set is huge. This minimizes network congestion and increases the overall throughput of the system. The assumption is that it is often better to migrate the computation closer to where the data is located rather than moving the data to where the application is running. HDFS provides interfaces for applications to move themselves closer to where the data is located.

- **Portability Across Heterogeneous Hardware and Software Platforms**

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

HDFS Tutorial: Features of HDFS

We will understand these features in detail when we will explore the HDFS Architecture in our next HDFS tutorial blog. But, for now, let's have an overview on the features of HDFS:

- ***Cost:*** The HDFS, in general, is deployed on a commodity hardware like your desktop/laptop which you use every day. So, it is very economical in terms of the cost of ownership of the project. Since, we are using low cost commodity hardware, you don't need to spend huge amount of money for scaling out your Hadoop cluster. In other words, adding more nodes to your HDFS is cost effective.

- ***Variety and Volume of Data:*** When we talk about HDFS then we talk about storing huge data i.e. Terabytes & petabytes of data and different kinds of data. So, you can store any type of data into HDFS, be it structured, unstructured or semi structured.

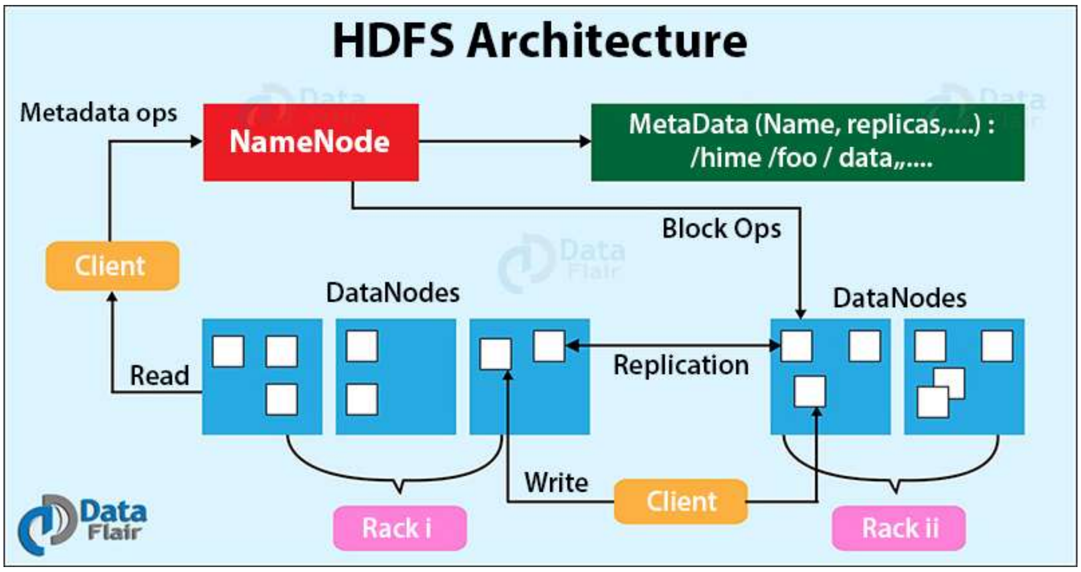
- ***Reliability and Fault Tolerance:*** When you store data on HDFS, it internally divides the given data into data blocks and stores it in a distributed fashion across your Hadoop cluster. The information regarding which data block is located on which of the data nodes is recorded in the metadata. **NameNode** manages the meta data and the **DataNodes** are responsible for storing the data.
Name node also replicates the data i.e. maintains multiple copies of the data. This replication of the data makes HDFS very reliable and fault tolerant. So, even if any of the nodes fails, we can retrieve the data from the replicas residing on other data nodes. By default, the replication factor is 3. Therefore, if you store 1 GB of file in HDFS, it will finally occupy 3 GB of space. The name node periodically updates the metadata and maintains the replication factor consistent.

- ***Data Integrity:*** Data Integrity talks about whether the data stored in my HDFS are correct or not. HDFS constantly checks the integrity of data stored against its checksum. If it finds any fault, it reports to the name node about it. Then, the name node creates additional new replicas and therefore deletes the corrupted copies.

- ***High Throughput:*** Throughput is the amount of work done in a unit time. It talks about how fast you can access the data from the file system. Basically, it gives you an insight about the system performance. As you have seen in the above example where we used ten machines collectively to enhance computation. There we were able to reduce the processing time from **43 minutes** to a mere **4.3 minutes** as all the machines were working in parallel. Therefore, by processing data in parallel, we decreased the processing time tremendously and thus, achieved high throughput.

- ***Data Locality:*** Data locality talks about moving processing unit to data rather than the data to processing unit. In our traditional system, we used to bring the data to the application layer and then process it. But now, because of the architecture and huge volume of the data, bringing the data to the application layer will reduce the network performance to a noticeable extent. So, in HDFS, we bring the computation part to the data nodes where the data is residing. Hence, you are not moving the data, you are bringing the program or processing part to the data.

HDFS Architecture



Hadoop Distributed File System follows the **master-slave architecture**. Each cluster comprises a **single master node** and **multiple slave nodes**. Internally the files get divided into one or more **blocks**, and each block is stored on different slave machines depending on the **replication factor** (which you will see later in this article).
The master node stores and manages the file system namespace, that is information about blocks of files like block locations, permissions, etc. The slave nodes store data blocks of files.

The Master node is the NameNode and DataNodes are the slave nodes.

Let's discuss each of the nodes in the Hadoop HDFS Architecture in detail.

What is HDFS NameNode?

NameNode is the centerpiece of the Hadoop Distributed File System. It maintains and manages the **file system namespace** and provides the right access permission to the clients.
The NameNode stores information about blocks locations, permissions, etc. on the local disk in the form of two files:

- **Fsimage:** Fsimage stands for File System image. It contains the complete namespace of the Hadoop file system since the NameNode creation.
- **Edit log:** It contains all the recent changes performed to the file system namespace to the most recent Fsimage.

Functions of HDFS NameNode

1. It executes the file system namespace operations like opening, renaming, and closing files and directories.
2. NameNode manages and maintains the DataNodes.
3. It determines the mapping of blocks of a file to DataNodes.
4. NameNode records each change made to the file system namespace.
5. It keeps the locations of each block of a file.
6. NameNode takes care of the replication factor of all the blocks.
7. NameNode receives heartbeat and block reports from all DataNodes that ensure DataNode is alive.
8. If the DataNode fails, the NameNode chooses new DataNodes for new replicas.

Before Hadoop2, NameNode was the single point of failure. The **High Availability** Hadoop cluster architecture introduced in Hadoop 2, allows for two or more NameNodes running in the cluster in a hot standby configuration.

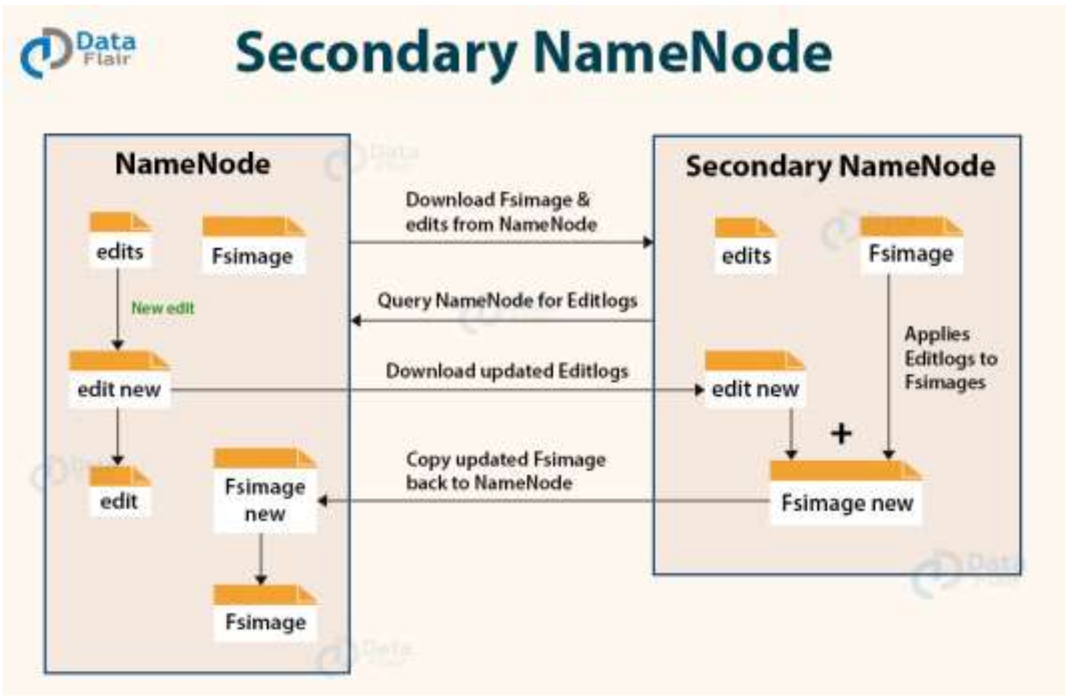
What is HDFS DataNode?

DataNodes are the slave nodes in Hadoop HDFS. DataNodes are **inexpensive commodity hardware**. They store blocks of a file.

Functions of DataNode

1. DataNode is responsible for serving the client read/write requests.
2. Based on the instruction from the NameNode, DataNodes performs block creation, replication, and deletion.
3. DataNodes send a heartbeat to NameNode to report the health of HDFS.
4. DataNodes also sends block reports to NameNode to report the list of blocks it contains.

What is Secondary NameNode?


Data Flair
Secondary NameNode
NameNode
edits
Fsimage
New edit
edit new
edit
Fsimage new
Fsimage
Download Fsimage & edits from NameNode
Query NameNode for Editlogs
Download updated Editlogs
Copy updated Fsimage back to NameNode
Secondary NameNode
edits
Fsimage
Applies Editlogs to Fsimages
edit new
+
Fsimage new

Apart from DataNode and NameNode, there is another daemon called the **secondary NameNode**. Secondary NameNode works as a helper node to primary NameNode but doesn't replace primary NameNode.
When the NameNode starts, the NameNode merges the Fsimage and edit logs file to restore the current file system namespace. Since the NameNode runs continuously for a long time without any restart, the size of edit logs becomes too large. This will result in a long restart time for NameNode.

Secondary NameNode solves this issue.

Secondary NameNode downloads the Fsimage file and edit logs file from NameNode.

It periodically applies edit logs to Fsimage and refreshes the edit logs. The updated Fsimage is then sent to the NameNode so that NameNode doesn't have to re-apply the edit log records during its restart. This keeps the edit log size small and reduces the NameNode restart time.

If the NameNode fails, the last save Fsimage on the secondary NameNode can be used to recover file system metadata. The secondary NameNode performs regular checkpoints in HDFS.

What is Checkpoint Node?

The Checkpoint node is a node that periodically creates checkpoints of the namespace.

Checkpoint Node in Hadoop first downloads Fsimage and edits from the Active Namenode. Then it merges them (Fsimage and edits) locally, and at last, it uploads the new image back to the active NameNode.

It stores the latest checkpoint in a directory that has the same structure as the Namenode's directory. This permits the checkpointed image to be always available for reading by the NameNode if necessary.

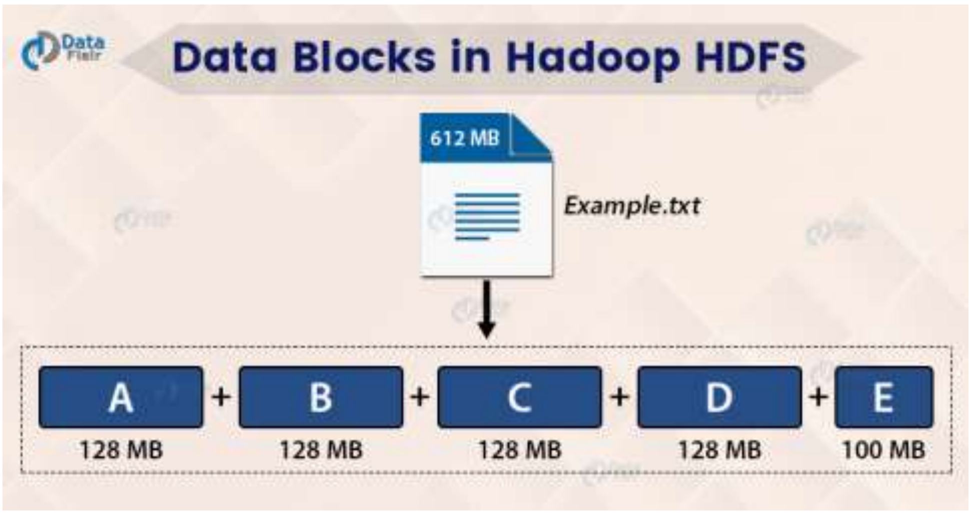
What is Backup Node?

A Backup node provides the same checkpointing functionality as the Checkpoint node.

In Hadoop, Backup node keeps an **in-memory, up-to-date copy** of the file system namespace. It is always synchronized with the active NameNode state.
It is not required for the backup node in HDFS architecture to download **Fsimage** and **edits files** from the active NameNode to create a checkpoint. It already has an up-to-date state of the namespace state in memory.
The Backup node checkpoint process is more efficient as it only needs to save the namespace into the local Fsimage file and reset edits. **NameNode supports one Backup node at a time**.
This was about the different types of nodes in HDFS Architecture. Further in this HDFS Architecture tutorial, we will learn about the Blocks in HDFS, Replication Management, Rack awareness and read/write operations.

Let us now study the block in HDFS.

What are Blocks in HDFS Architecture?



Internally, HDFS split the file into block-sized chunks called a block. The size of the block is **128 Mb** by default. One can configure the block size as per the requirement.

For example, if there is a file of size 612 Mb, then HDFS will create four blocks of size 128 Mb and one block of size 100 Mb.

The file of a smaller size does not occupy the full block size space in the disk.

For example, the file of size 2 Mb will occupy only 2 Mb space in the disk.

The user doesn't have any control over the location of the blocks.

Read the **HDFS Block** article to explore in detail.
HDFS is highly fault-tolerant. **Now, look at what makes HDFS fault-tolerant.**

What is Replication Management?

For a distributed system, the data must be redundant to multiple places so that if one machine fails, the data is accessible from other machines.

In Hadoop, HDFS stores replicas of a block on multiple DataNodes based on the replication factor.

The replication factor is the number of copies to be created for blocks of a file in HDFS architecture.
If the replication factor is 3, then three copies of a block get stored on different DataNodes. So if one DataNode containing the data block fails, then the block is accessible from the other DataNode containing a replica of the block.

If we are storing a file of 128 Mb and the replication factor is 3, then (3*128=384) 384 Mb of disk space is occupied for a file as three copies of a block get stored.

This replication mechanism makes HDFS fault-tolerant.

Read the **Fault tolerance** article to learn in detail.

What is Rack Awareness in HDFS Architecture?

Let us now talk about how HDFS store replicas on the DataNodes? What is a rack? What is rack awareness?

Rack is the collection of around 40-50 machines (DataNodes) connected using the same network switch. If the network goes down, the whole rack will be unavailable.
Rack Awareness is the concept of choosing the closest node based on the rack information.

To ensure that all the replicas of a block are not stored on the same rack or a single rack, NameNode follows a rack awareness algorithm to store replicas and provide latency and fault tolerance.

Suppose if the replication factor is 3, then according to the rack awareness algorithm:

- The first replica will get stored on the local rack.
- The second replica will get stored on the other DataNode in the same rack.
- The third replica will get stored on a different rack.

The File System Namespace

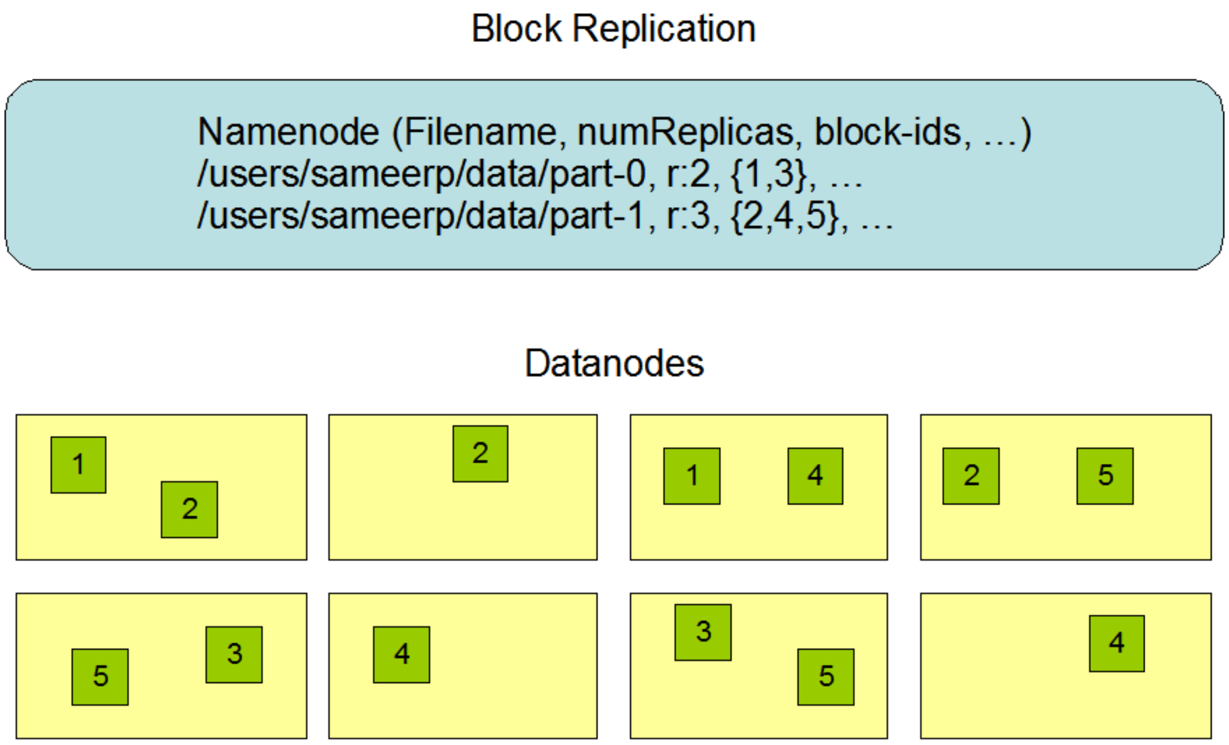
HDFS supports a traditional hierarchical file organization. A user or an application can create directories and store files inside these directories. The file system namespace hierarchy is similar to most other existing file systems; one can create and remove files, move a file from one directory to another, or rename a file. HDFS does not yet implement user quotas. HDFS does not support hard links or soft links. However, the HDFS architecture does not preclude implementing these features.

The NameNode maintains the file system namespace. Any change to the file system namespace or its properties is recorded by the NameNode. An application can specify the number of replicas of a file that should be maintained by HDFS. The number of copies of a file is called the replication factor of that file. This information is stored by the NameNode.

Data Replication

HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file. An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later. Files in HDFS are write-once and have strictly one writer at any time.

The NameNode makes all decisions regarding replication of blocks. It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly. A Blockreport contains a list of all blocks on a DataNode.


Block Replication
Namenode (Filename, numReplicas, block-ids, …)
/users/sameerp/data/part-0, r:2, {1,3}, …
/users/sameerp/data/part-1, r:3, {2,4,5}, …
Datanodes
1
2
2
1
4
2
5
5
3
4
3
5
4

Replica Placement: The First Baby Steps

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

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

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

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

The current, default replica placement policy described here is a work in progress.

Replica Selection

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

Safemode

On startup, the NameNode enters a special state called Safemode. Replication of data blocks does not occur when the NameNode is in the Safemode state. The NameNode receives Heartbeat and Blockreport messages from the DataNodes. A Blockreport contains the list of data blocks that a DataNode is hosting. Each block has a specified minimum number of replicas. A block is considered safely replicated when the minimum number of replicas of that data block has checked in with the NameNode. After a configurable percentage of safely replicated data blocks checks in with the NameNode (plus an additional 30 seconds), the NameNode exits the Safemode state. It then determines the list of data blocks (if any) that still have fewer than the specified number of replicas. The NameNode then replicates these blocks to other DataNodes.

The Persistence of File System Metadata

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

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

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

The Communication Protocols

All HDFS communication protocols are layered on top of the TCP/IP protocol. A client establishes a connection to a configurable TCP port on the NameNode machine. It talks the

ClientProtocol with the NameNode. The DataNodes talk to the NameNode using the DataNode Protocol. A Remote Procedure Call (RPC) abstraction wraps both the Client Protocol and the DataNode Protocol. By design, the NameNode never initiates any RPCs. Instead, it only responds to RPC requests issued by DataNodes or clients.

Robustness

The primary objective of HDFS is to store data reliably even in the presence of failures. The three common types of failures are NameNode failures, DataNode failures and network partitions.

Data Disk Failure, Heartbeats and Re-Replication

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

Cluster Rebalancing

The HDFS architecture is compatible with data rebalancing schemes. A scheme might automatically move data from one DataNode to another if the free space on a DataNode falls below a certain threshold. In the event of a sudden high demand for a particular file, a scheme might dynamically create additional replicas and rebalance other data in the cluster. These types of data rebalancing schemes are not yet implemented.

Data Integrity

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

Metadata Disk Failure

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

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

Snapshots

Snapshots support storing a copy of data at a particular instant of time. One usage of the snapshot feature may be to roll back a corrupted HDFS instance to a previously known good point in time. HDFS does not currently support snapshots but will in a future release.

Data Organization

Data Blocks

HDFS is designed to support very large files. Applications that are compatible with HDFS are those that deal with large data sets. These applications write their data only once but they read it one or more times and require these reads to be satisfied at streaming speeds. HDFS supports write-once-read-many semantics on files. A typical block size used by HDFS is 64 MB. Thus, an HDFS file is chopped up into 64 MB chunks, and if possible, each chunk will reside on a different DataNode.

Staging

A client request to create a file does not reach the NameNode immediately. In fact, initially the HDFS client caches the file data into a temporary local file. Application writes are transparently redirected to this temporary local file. When the local file accumulates data worth over one HDFS block size, the client contacts the NameNode. The NameNode inserts the file name into the file system hierarchy and allocates a data block for it. The NameNode responds to the client request with the identity of the DataNode and the destination data block. Then the client flushes the block of data from the local temporary file to the specified DataNode. When a file is closed, the remaining un-flushed data in the temporary local file is transferred to the DataNode. The client then tells the NameNode that the file is closed. At this point, the NameNode commits the file creation operation into a persistent store. If the NameNode dies before the file is closed, the file is lost.

The above approach has been adopted after careful consideration of target applications that run on HDFS. These applications need streaming writes to files. If a client writes to a remote file directly without any client side buffering, the network speed and the congestion in the network impacts throughput considerably. This approach is not without precedent. Earlier distributed file systems, e.g. AFS, have used client side caching to improve performance. A POSIX requirement has been relaxed to achieve higher performance of data uploads.

Replication Pipelining

When a client is writing data to an HDFS file, its data is first written to a local file as explained in the previous section. Suppose the HDFS file has a replication factor of three. When the local file accumulates a full block of user data, the client retrieves a list of DataNodes from the NameNode. This list contains the DataNodes that will host a replica of that block. The client then flushes the data block to the first DataNode. The first DataNode starts receiving the data in small portions (4 KB), writes each portion to its local repository and transfers that portion to the second DataNode in the list. The second DataNode, in turn starts receiving each portion of the data block, writes that portion to its repository and then flushes that portion to the third DataNode. Finally, the third DataNode writes the data to its local repository. Thus, a DataNode can be receiving data from the previous one in the pipeline and at the same time forwarding data to the next one in the pipeline. Thus, the data is pipelined from one DataNode to the next.

Accessibility

HDFS can be accessed from applications in many different ways. Natively, HDFS provides a Java API for applications to use. A C language wrapper for this Java API is also available. In addition, an HTTP browser can also be used to browse the files of an HDFS instance. Work is in progress to expose HDFS through the WebDAV protocol.

FS Shell

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

Action	
Create a directory named `/foodir`	`bin/hadoop dfs -mkdir /foodir`
Remove a directory named `/foodir`	`bin/hadoop dfs -rmr /foodir`
View the contents of a file named `/foodir/myfile.txt`	`bin/hadoop dfs -cat /foodir/myfile.txt`

FS shell is targeted for applications that need a scripting language to interact with the stored data.

DFSAdmin

The DFSAdmin command set is used for administering an HDFS cluster. These are commands that are used only by an HDFS administrator. Here are some sample action/command pairs:

Action	
Put the cluster in Safemode	`bin/hadoop dfsadmin -safemode enter`
Generate a list of DataNodes	`bin/hadoop dfsadmin -report`
Recommission or decommission DataNode(s)	`bin/hadoop dfsadmin -refreshNodes`

Browser Interface

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

Space Reclamation

File Deletes and Undeletes

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

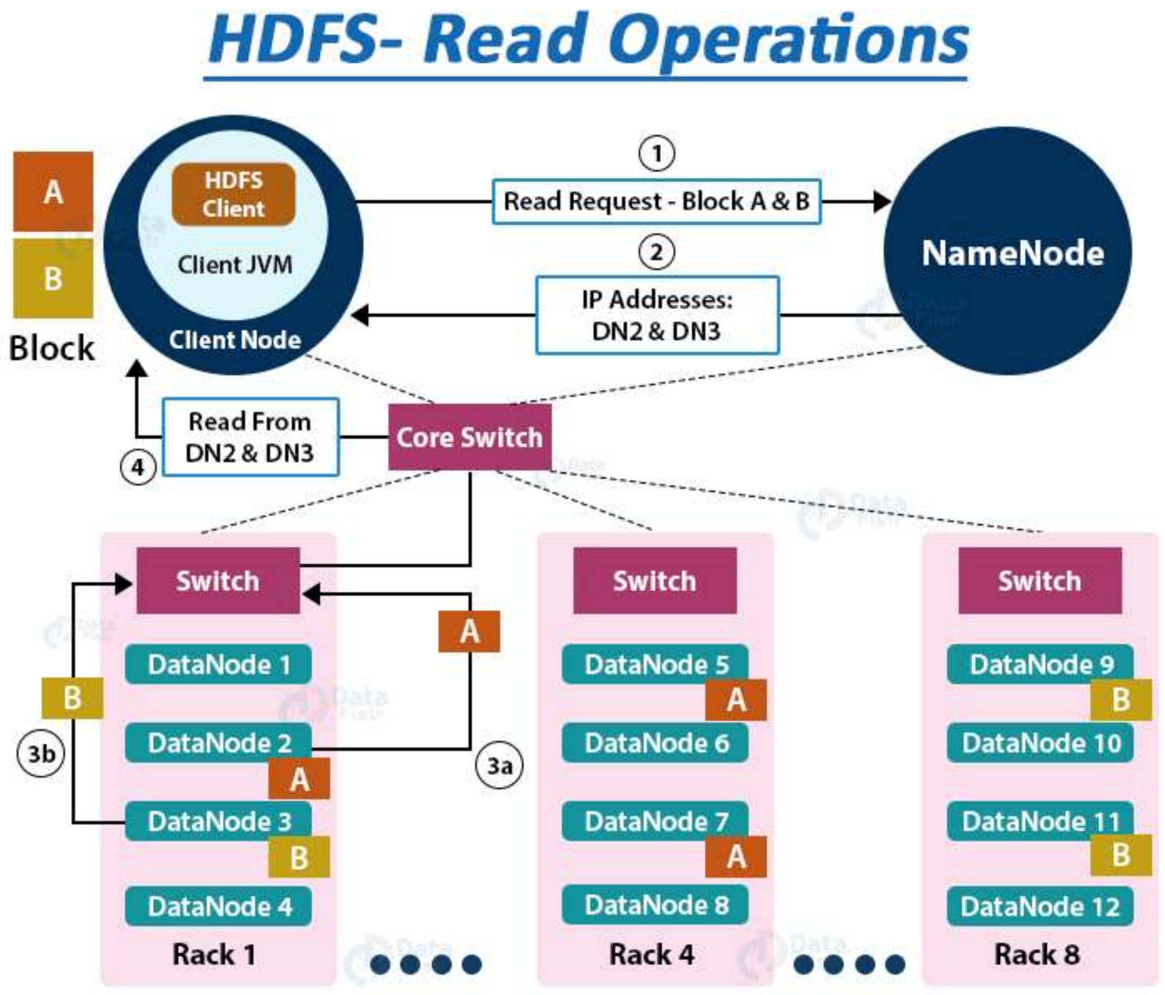
A user can Undelete a file after deleting it as long as it remains in the `/trash` directory. If a user wants to undelete a file that he/she has deleted, he/she can navigate the /trash directory and retrieve the file. The /trash directory contains only the latest copy of the file that was deleted. The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

Decrease Replication Factor

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

HDFS read operation

Suppose the HDFS client wants to read a file "File.txt". Let the file be divided into two blocks say, A and B. The following steps will take place during the file read:



1. The Client interacts with HDFS NameNode

- As the NameNode stores the block's metadata for the file "File.txt', the client will reach out to NameNode asking locations of DataNodes containing data blocks.
- The NameNode first checks for required privileges, and if the client has sufficient privileges, the NameNode sends the locations of DataNodes containing blocks (A and B). NameNode also gives a **security token** to the client, which they need to show to the DataNodes for authentication. Let the NameNode provide the following list of IPs for block A and B – for block A, location of DataNodes D2, D5, D7, and for block B, location of DataNodes D3, D9, D11.

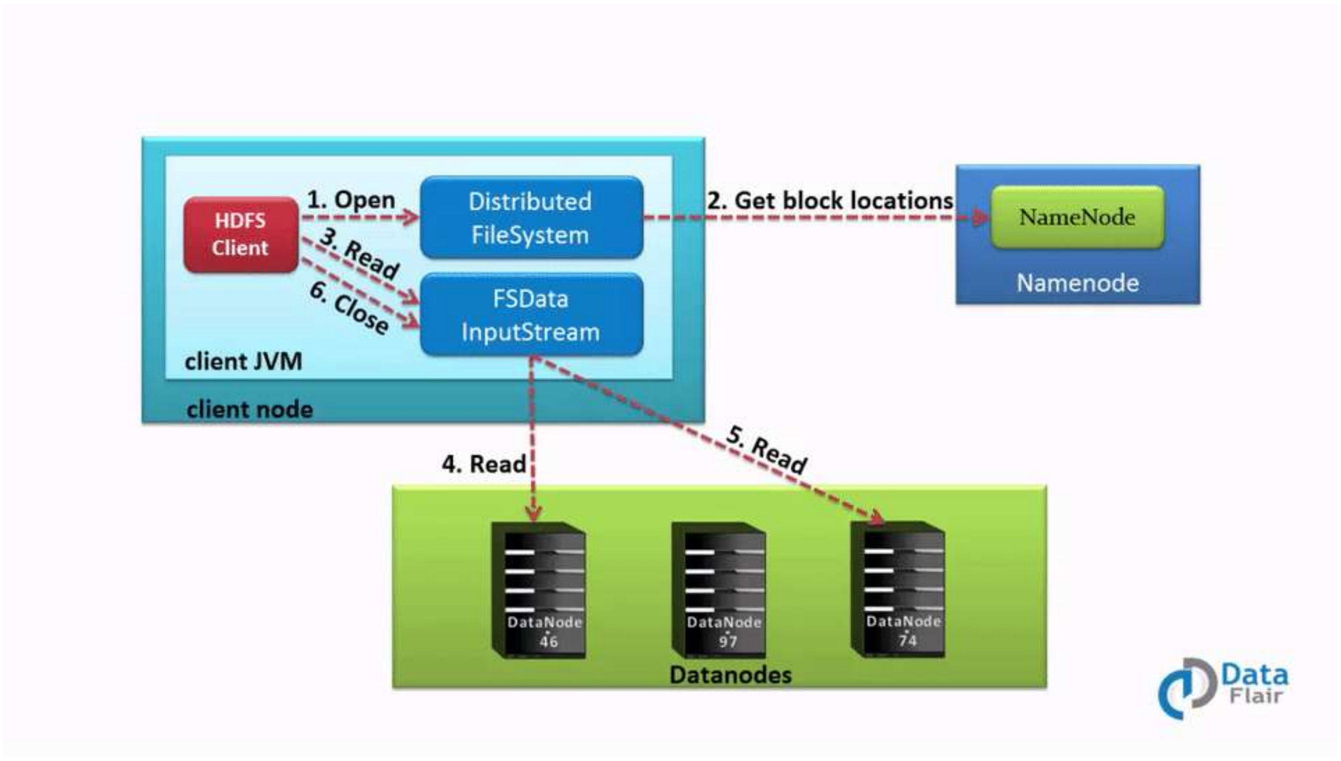
To perform various **HDFS operations** (read, write, copy, move, change permission, etc.) follow **HDFS command list**.

2. The client interacts with HDFS DataNode

- After receiving the addresses of the DataNodes, the client directly interacts with the DataNodes. The client will send a request to the closest DataNodes (D2 for block A and D3 for block B) through the **FSDataInputstream** object. The **DFSInputstream** manages the interaction between client and DataNode.
- The client will show the security tokens provided by NameNode to the DataNodes and start reading data from the DataNode. The data will flow directly from the DataNode to the client.
- After reading all the required file blocks, the client calls close() method on the FSDataInputStream object.

Now let us see how internally read operation is carried out in Hadoop HDFS, how data flows between the client, the NameNode, and DataNodes during file read.

Internals of file read in HDFS



1. In order to open the required file, the client calls the **open()** method on the **FileSystem object**, which for HDFS is an instance of DistributedFilesystem.
2. DistributedFileSystem then calls the NameNode using RPC to get the locations of the first few blocks of a file. For each **data block**, NameNode returns the addresses of Datanodes that contain a copy of that block. Furthermore, the DataNodes are sorted based on their proximity to the client.
3. The DistributedFileSystem returns an **FSDataInputStream** to the client from where the client can read the data. FSDataInputStream in succession wraps a DFSInputStream. **DFSInputStream** manages the I/O of DataNode and NameNode.
4. Then the client calls the **read()** method on the **FSDataInputStream** object.
5. The DFSInputStream, which contains the addresses for the first few blocks in the file, connects to the closest DataNode to read the first block in the file. Then, the data flows from DataNode to the client, which calls read() repeatedly on the FSDataInputStream.
6. Upon reaching the end of the file, DFSInputStream closes the connection with that DataNode and finds the best suited DataNode for the next block.
7. If the DFSInputStream during reading, faces an error while communicating with a DataNode, it will try the other closest DataNode for that block. DFSInputStream will also remember DataNodes that have failed so that it doesn't needlessly retry them for later blocks. Also, the DFSInputStream verifies checksums for the data transferred to it from the DataNode. If it finds any

corrupt block, it reports this to the NameNode and reads a copy of the block from another DataNode.

8. When the client has finished reading the data, it calls **close()** on the **FSDataInputStream**.

How to Read a file from HDFS – Java Program

A sample code to read a file from HDFS is as follows (To perform HDFS read and write operations:

```
FileSystem fileSystem = FileSystem.get(conf);
Path path = new Path("/path/to/file.ext");
if (!fileSystem.exists(path)) {
System.out.println("File does not exists");
return;
}
FSDataInputStream in = fileSystem.open(path);
int numBytes = 0;
while ((numBytes = in.read(b))> 0) {
System.out.prinln((char)numBytes));// code to manipulate the data which is read
}
in.close();
out.close();
fileSystem.close();
```

Summary

So in this article, we have studied the data flow between client, DataNode, and NameNode during a client read request.

Now you have a pretty good idea about the HDFS file read operation and how the client interacts with DataNode and NameNode.

HDFS write operation

To write data in HDFS, the client first interacts with the **NameNode** to get permission to write data and to get IPs of **DataNodes** where the client writes the data. The client then directly interacts with the DataNodes for writing data. The DataNode then creates a replica of the data block to other DataNodes in the pipeline based on the replication factor.

DFSOutputStream in HDFS maintains two queues (data queue and ack queue) during the write operation.

1. The client interacts with HDFS NameNode

- To write a file inside the HDFS, the client first interacts with the NameNode. NameNode first checks for the client privileges to write a file. If the client has sufficient privilege and there is no file existing with the same name, NameNode then creates a record of a new file.
- NameNode then provides the address of all DataNodes, where the client can write its data. It also provides a security token to the client, which they need to present to the DataNodes before writing the block.
- If the file already exists in the HDFS, then file creation fails, and the client receives an **IO Exception**.

2. The client interacts with HDFS DataNode

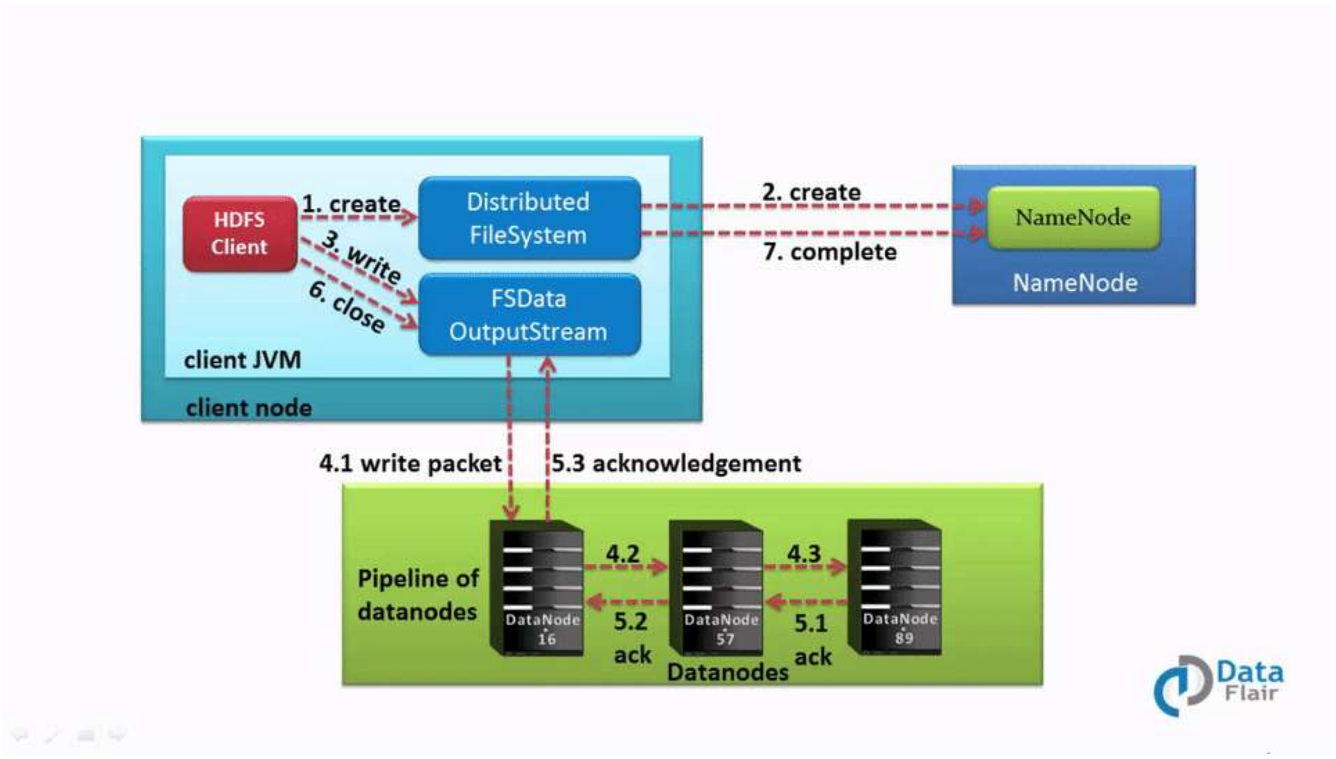
After receiving the list of the DataNodes and file write permission, the client starts writing data directly to the first DataNode in the list. As the client finishes writing data to the first DataNode, the DataNode starts making replicas of a block to other DataNodes depending on the replication factor.

If the replication factor is 3, then there will be a minimum of 3 copies of blocks created in different DataNodes, and after creating required replicas, it sends an acknowledgment to the client.

Thus it leads to the creation of a pipeline, and data replication to the desired value, in the cluster.

Internals of file write in Hadoop HDFS

Let us understand the HDFS write operation in detail. The following steps will take place while writing a file to the HDFS:



1. The client calls the **create()** method on **DistributedFileSystem** to create a file.

2. DistributedFileSystem interacts with NameNode through the RPC call to create a new file in the filesystem namespace with no blocks associated with it.

3. The NameNode checks for the client privileges and makes sure that the file doesn't already exist. If the client has sufficient privileges and no file with the same name exists, the NameNode makes a record of the new file. Otherwise, the client receives an I/O exception, and file creation fails. The DistributedFileSystem then returns an FSDataOutputStream for the client where the client starts writing data. FSDataOutputstream, in turn, wraps a DFSOutputStream, which handles communication with the DataNodes and NameNode.

4. As the client starts writing data, the **DFSOutputStream** splits the client's data into packets and writes it to an internal queue called the **data queue**. **DataStreamer**, which is responsible for telling the NameNode to allocate new blocks by choosing the list of suitable DataNode to store the replicas, uses this data queue.

The list of DataNode forms a pipeline. The number of DataNodes in the pipeline depends on the replication factor.

Suppose the replication factor is 3, so there are three nodes in the pipeline.

The DataStreamer streams the packet to the first DataNode in the pipeline, which stores each packet and forwards it to the second node in the pipeline. Similarly, the second DataNode stores the packet and transfers it to the next node in the pipeline (last node).

5. The **DFSOutputStream** also maintains another queue of packets, called **ack queue,** which is waiting for the acknowledgment from DataNodes.

Packet in the ack queue gets remove only when it receives an acknowledgment from all the DataNodes in the pipeline.

6. The client calls the **close()** method on the stream when he/she finishes writing data. Thus, before communicating the NameNode to signal about the file complete, the client close() method's action pushes the remaining packets to the DataNode pipeline and waits for the acknowledgment.

7. As the Namenode already knows about the blocks (the file made of), so the NameNode only waits for blocks to be minimally replicated before returning successfully.

What happens if DataNode fails while writing a file in the HDFS?

While writing data to the DataNode, if DataNode fails, then the following actions take place, which is transparent to the client writing the data.

1. The pipeline gets closed, packets in the ack queue are then added to the front of the data queue making DataNodes downstream from the failed node to not miss any packet.

2. Then the current block on the alive DataNode gets a new identity. This id is then communicated to the NameNode so that, later on, if the failed DataNode recovers, the partial block on the failed DataNode will be deleted.

3. The failed DataNode gets removed from the pipeline, and a new pipeline gets constructed from the two alive DataNodes. The remaining of the block's data is then written to the alive DataNodes, added in the pipeline.

4. The NameNode observes that the block is **under-replicated**, and it arranges for creating further copy on another DataNode. Other coming blocks are then treated as normal.

How to Write a file in HDFS – Java Program

A sample code to write a file to HDFS in Java is as follows:

```
FileSystem fileSystem = FileSystem.get(conf);
// Check if the file already exists
Path path = new Path("/path/to/file.ext");
if (fileSystem.exists(path)) {
System.out.println("File " + dest + " already exists");
return;
}
// Create a new file and write data to it.
FSDataOutputStream out = fileSystem.create(path);
```

```
InputStream in = new BufferedInputStream(new FileInputStream(
new File(source)));
byte[] b = new byte[1024];
int numBytes = 0;
while ((numBytes = in.read(b)) > 0) {
out.write(b, 0, numBytes);
}
// Close all the file descripters
in.close();
out.close();
fileSystem.close();
```

Summary

After reading this article, you have a good idea about the HDFS file write operation. From this article, we clearly understand the anatomy of file write in Hadoop.

The article has described the file write in detail along with the explanation of replicas creation during file write. We have also seen what happens if the DataNode fails while writing the file.

Rack Awareness in Hadoop HDFS – An Introductory Guide

Ever thought how NameNode choose the Datanode for storing the data blocks and their replicas?

HDFS stores files across multiple nodes (DataNodes) in a cluster. To get the maximum performance from Hadoop and to improve the network traffic during file read/write, NameNode chooses the DataNodes on the same rack or nearby racks for data read/write. Rack awareness is the concept of choosing the closer DataNode based on rack information.

In this article, we will study the rack awareness concept in detail.

We will first see what is the rack, what is rack awareness, the reason for using rack awareness, block replication policies, and benefits of Rack Awareness.

Let’s start with the introduction of the rack.

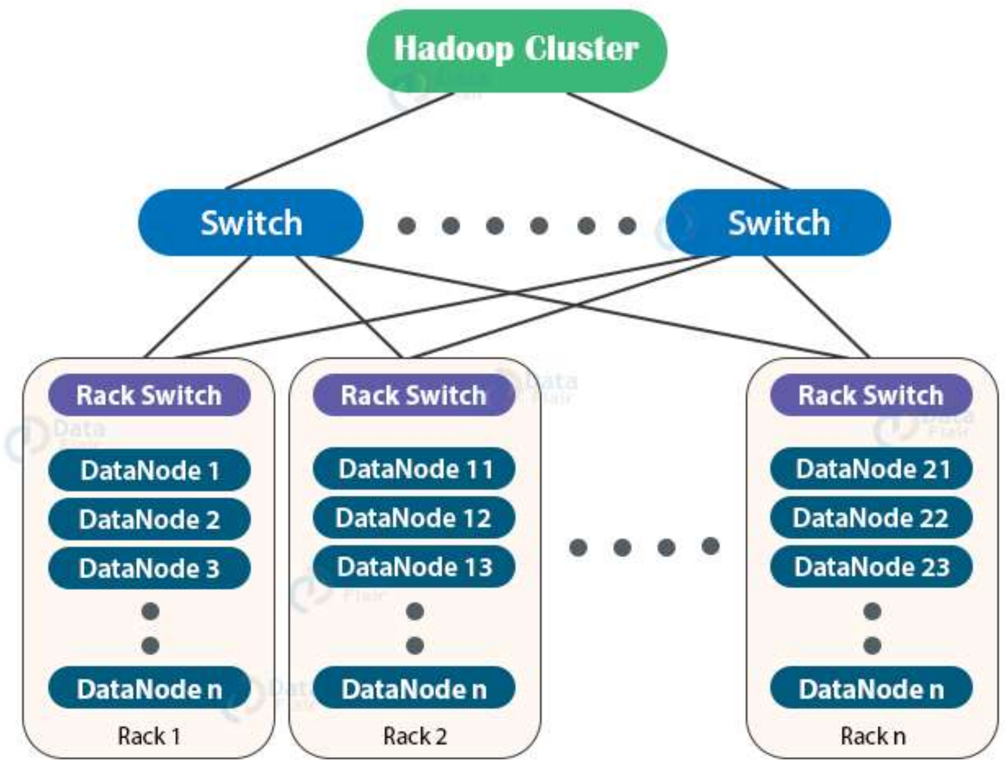
What is a rack?

The **Rack** is the collection of around 40-50 DataNodes connected using the same network switch. If the network goes down, the whole rack will be unavailable. A large Hadoop cluster is deployed in multiple racks.

What is Rack Awareness in Hadoop HDFS?

In a large Hadoop cluster, there are multiple racks. Each rack consists of DataNodes. Communication between the DataNodes on the same rack is more efficient as compared to the communication between DataNodes residing on different racks.

To reduce the network traffic during file **read/write**, NameNode chooses the closest DataNode for serving the client read/write request. NameNode maintains **rack ids** of each DataNode to achieve this rack information. This concept of choosing the closest DataNode based on the rack information is known as **Rack Awareness**.



A default Hadoop installation assumes that all the DataNodes reside on the same rack.

Why Rack Awareness?

The reasons for the Rack Awareness in Hadoop are:

1. To reduce the network traffic while file read/write, which improves the cluster performance.
2. To achieve **fault tolerance**, even when the rack goes down (discussed later in this article).
3. Achieve high availability of data so that data is available even in unfavorable conditions.
4. To reduce the latency, that is, to make the file read/write operations done with lower delay.

NameNode uses a rack awareness algorithm while placing the replicas in HDFS.

Let us now study the replica placement via Rack Awareness in Hadoop.

Replica placement via Rack awareness in Hadoop

We know HDFS stores replicas of data blocks of a file to provide fault tolerance and high availability. Also, the network bandwidth between nodes within the rack is higher than the network bandwidth between nodes on a different rack.
If we store replicas on different nodes on the same rack, then it improves the network bandwidth, but if the rack fails (rarely happens), then there will be no copy of data on another rack.

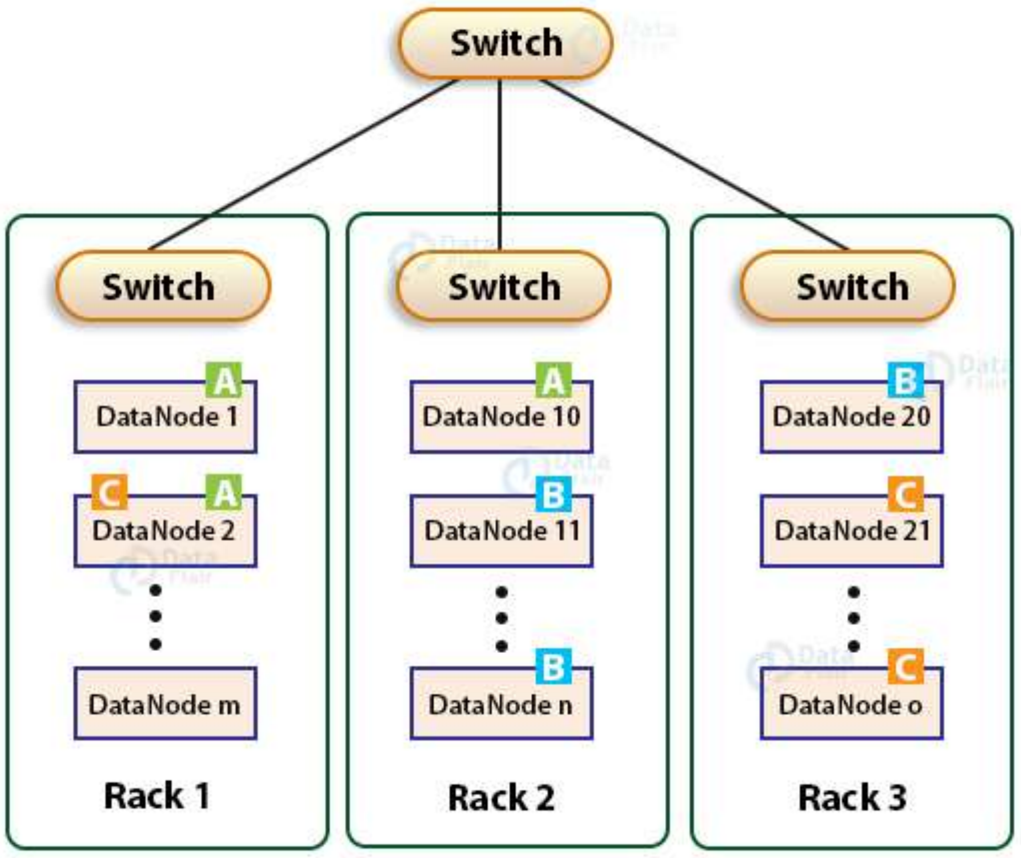
Again, if we store replicas on unique racks, then due to the transfer of blocks to multiple racks while writes increase the cost of writes.

Therefore, NameNode on multiple rack cluster maintains block replication by using inbuilt **Rack awareness policies** which says:

- Not more than one replica be placed on one node.
- Not more than two replicas are placed on the same rack.
- Also, the number of racks used for block replication should always be smaller than the number of replicas.

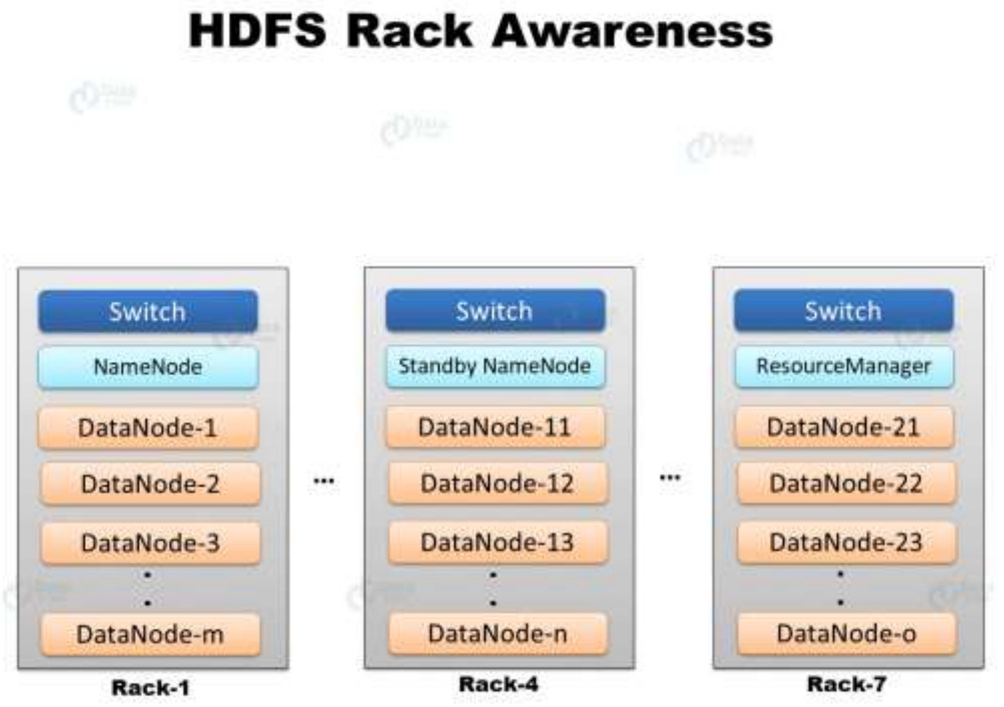
For the common case where the replication factor is three, the block replication policy put the first replica on the local rack, a second replica on the different DataNode on the same rack, and a third replica on the different rack.

Also, while re-replicating a block, if the existing replica is one, place the second replica on a different rack. If the existing replicas are two and are on the same rack, then place the third replica on a different rack.



This policy improves write performance and network traffic without compromising fault tolerance.

Rack Awareness Example



In the above GIF, we are having a file "File.txt" divided into three blocks A, B, and C. To provide fault tolerance, HDFS creates replicas of blocks. NameNode places the first copy of each block on the closest DataNode, the second replica of each block on different DataNode on the same rack, and the third replica on different DataNode on a different rack.

What about performance?

- Faster replication operation: Since the replicas are placed within the same rack it would use higher bandwidth and lower latency hence making it faster.
- If YARN is unable to create a container in the same data node where the queried data is located it would try to create the container in a data node within the same rack. This would be more performant because of the higher bandwidth and lower latency of the data nodes inside the same rack

Advantages of Implementing Rack Awareness

Some of the main advantages of Rack Awareness are:

1. Preventing data loss against rack failure

Rack Awareness policy puts replicas at different rack as well, thus ensures no data loss even if the rack fails.

2. Minimize the cost of write and maximize the read speed

Rack awareness reduces write traffic in between different racks by placing write requests to replicas on the same rack or nearby rack, thus reducing the cost of write. Also, using the bandwidth of multiple racks increases the read performance.

3. Maximize network bandwidth and low latency

Rack Awareness enables Hadoop to maximize network bandwidth by favoring the transfer of blocks within racks over transfer between racks. Especially with rack awareness, the YARN is able to optimize **MapReduce** job performance. It assigns tasks to nodes that are 'closer' to their data in terms of network topology. This is particularly beneficial in cases where tasks cannot be assigned to nodes where their data is stored locally.

Summary

In this article, you have studied the rack awareness concept, which is the selection of the closest node based on the rack information.

We have seen the reasons for introducing rack awareness in Hadoop like network bandwidth, high availability, etc.

We have also discussed the Rack awareness policy used by the NameNode to maintain block replication. The article also enlisted the advantages of Rack Awareness.