1. Distributed File Systems

This chapter describes how big-data systems store large files in a distributed file system. The goal is to educate data scientists how to manage big-data efficiently and the implications of some design or implementation decisions. The articles are largely based on the Hadoop Distributed File System (HDFS) but most of the concepts apply to other distributed file systems. It is based on the public documentation of Hadoop, the personal experience from working with Hadoop and Spark, and the source code of HDFS. It is not meant to be extensive or to cover all aspects of HDFS. It just describes the basic features and how they work in a distributed cluster.

1.1 What is HDFS?

HDFS is an open-source implementation of a distributed file system. The idea is based on the architecture of the Google File System (GFS). In its basic setup, HDFS consists of one namenode and one or more datanodes[1]. The architecture is shown in Figure 1.1. The name node keeps metadata about the files and directories in the file system, e.g., file names, owner, access rights, and replication factor. The actual file contents are stored as fixed-size blocks, typically 128MB, in data nodes. Metadata is kept in main-memory for efficiency while being logged to disk for persistence. Data blocks are kept as regular files.

Similar to a regular file system, HDFS does not know, or care, about the contents of the stored files. It just treats a file as sequences of bytes and provides an API to write these bytes and read them back. Parsing and processing the file is the job of someone else, e.g., Hadoop MapReduce or Spark RDD.

1.2 HDFS API

Before digging into the internal details of HDFS, we first describe the basic API that it provides. After that, we will explain how these functions are supported by the design.

[1]Multi namenode architecture is not covered in this book.

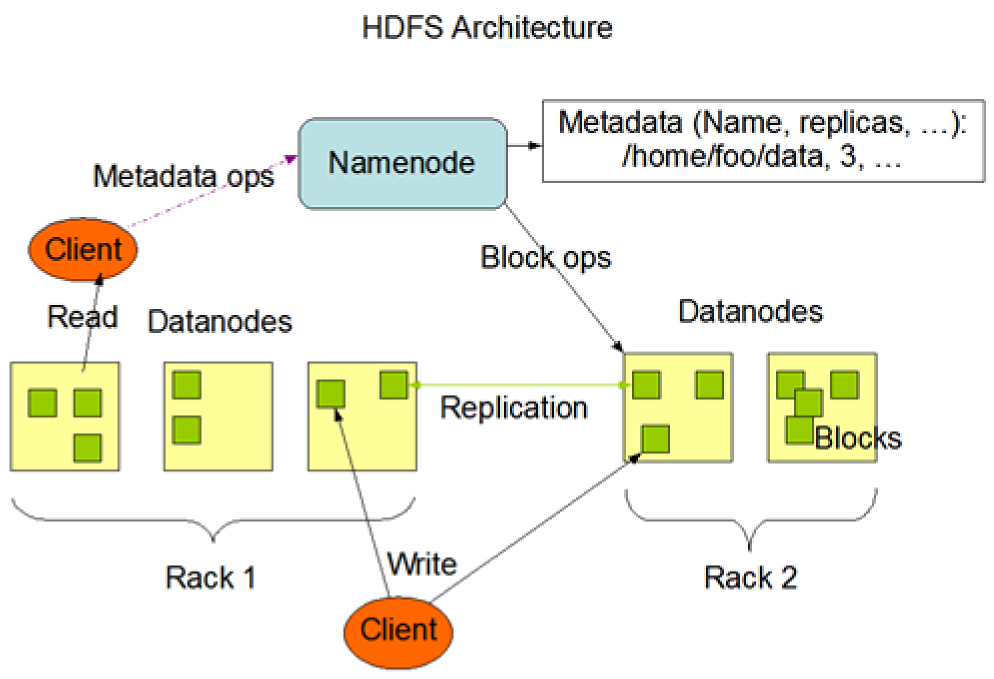

Figure 1.1: HDFS Architecture `https://hadoop.apache.org/docs/r3.2.2/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html`

1.2.1 Basic Classes

HDFS defines three new classes to access, Path, Configuration, and FileSystem.

Path

Path is essentially a String that stores a path to a file or directory in HDFS. The path object itself does not contain any information on whether it points to a file, a directory, a symbolic link, ... etc. It provides convenient functions to manipulate the path such as reaching the parent directory or converting a relative path to an absolute path. In general, a path is written in the following form.

hdfs://namenode:9000/absolute/path/filename

hdfs	is called the scheme of the file system. For example, this could be http, ftp, or file.
namenode	indicates the name or the IP address that hosts the file system. This can be the namenode for HDFS or the server address for http and ftp file systems.
9000	the port on which the server is listening.
/absolute/path/	the absolute path where the file or directory is located. It always starts with a leading '/'.
filename	is the name of the entity (file or directory) pointed at by the path.

Among all previous parts, only the filename is required. All other parts will be automatically replaced with their default values if not provided. The default values are defined as follows.

- Scheme: The default file system is defined by the Hadoop configuration "fs.defaultFS".
- namenode: If HDFS is used, the default namenode will be retrieved from Hadoop configuration "fs.defaultFS". For some file systems, this value is ignored, e.g., `file`, and for others it is required and there is no default, e.g., `ftp` and `http`.
- port: The default port for HDFS is again retrieved from Hadoop configuration. For other file systems, a corresponding default port is used, e.g., 80 for `http` and
- If the absolute path is not provided, the current working directory is used instead. The default working directory in the local file system is typically the directory in which you run the command. In HDFS, the working directory is "/user/$USER" where $USER is the system

username.

Notice that you cannot override the file system without specifying an absolute path. Examples of possible paths are provided below.

■ **Example 1.1 — Paths.** The following are examples of paths.

- `new Path("filename.txt")` – points to an entity named “filename.txt" in the working directory of the default file system.
- `new Path("/path/to/a/file")` – points to an entity named “file" at the given absolute path starting at the root and in the default file system.
- `new Path("hdfs:///path/to/a/file")` – points to a file at the absolute path “/path/-to/a/file" in HDFS with the default HDFS configuration.
- `new Path("file:///home/user/input")` – points to an entity at the absolute path “/home-/user/input" in the local file system, i.e., not in HDFS. This is useful to access the local file system when the default file system configures is HDFS.
- `new Path("http://example.com/path/to/file.txt")` - points to an entity named in a remote HTTP server named ‘example.com’ with the default HTTP port (80) and at the absolute path ‘/path/to/file.txt’ at the server.

■

The following code snippet shows a few examples of how to use the Path class in Java.

Listing 1.1: HDFS Path creation

```
// Initialization from a string
Path p = new Path("relativepath");
// Create a path that points to file in a directory
Path directory = new Path("directoryname")
Path fileInDirectory = new Path(directory, "filename")
// Retrieve the parent of a path, i.e., the containing directory
Path fullpath = new Path("/user/student/dir/name")
// The next path will point to "/user/student/dir"
Path parent = fullPath.getParent();
```

Configuration

Configuration is a key-value map that stores all the configuration that HDFS needs to work. One example just mentioned above is “fs.defaultFS" which stores the default file system. Other configuration parameters include the default replication factor and block size. For an exhaustive list of parameters related to HDFS and their default values, check the following link.

`https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml`

When you create a Configuration object using the default construction “new Configuration()" it will load the default values configured by the cluster. You can always override specific parameters using an appropriate “set" method or retrieve the current value using an appropriate “get” method. Notice that while some set and get methods seem to deal with non-string value, e.g., setInt and getBoolean, all values are internally stored as strings. This makes it easier to load the data from XML and serialize it over network.

FileSystem

This abstract class defines a general interface with dealing with any file system. Using this class in your code allows your application to seamlessly work on both the local file system and HDFS. For example, when you are developing and testing your code, you can run it with the local file system. Once you move it to a real distributed cluster, it will work fine with HDFS.

There are three ways to obtain a file system using the HDFS API.

1. From a Path instance. This is the most common way to get a file system. Given a Path, the method Path#getFileSystem(Configuration) returns the appropriate file system for the given path as follows.
   (a) If the Path contains a file system scheme, the corresponding file system instance is created.
   (b) If the path does not contain a file system scheme, the default file system is returned.
2. Default file system. To obtain the default file system, using the method FileSystem.get(Configuration).
3. Local fie system. If you want to access the local file system, using the method FileSystem.getLocal(Configuration).

Notice how all the above methods require a Configuration instance so that Hadoop can determine what the file system is and how to access the master node

1.3 File Writing

The following code example shows the standard way of writing a file to HDFS.

Listing 1.2: Write a file to HDFS

```
Path p = new Path("filename");
FileSystem fs = p.getFileSystem(new Configuration());
try (FSDataOutputStream out = fs.create(p)) {
  out.write(...);
}
```

The process is to first get the appropriate file system for the given path. This ensures that your code will work fine when deployed on a real cluster with HDFS. The try block ensures that the output stream will be closed automatically after you finish writing. FSDataOutputStream inherits from Java OutputStream and DataOutput which allows you to write to the file in the same way you do in a regular Java program. The only caveat with writing is that you can only write a file sequentially and once the file is closed, you can no longer modify it. This constraint simplifies the design and allows it run in a distributed cluster efficiently.

1.3.1 File Reading

The reading process is generally similar to the writing process. A code example is given below.

Listing 1.3: Read a file from HDFS

```
Path p = new Path("filename");
FileSystem fs = p.getFileSystem(new Configuration());
try (FSDataInputStream in = fs.open(p)) {
 in.read(...);
}
```

Similar to writing, you should generally obtain the correct file system using Path#getFileSystem method. Unlike writing, HDFS supports random reads. You can do that using two additional method, FSDataInputStreamseek(long) and FSDataInputStreampos(). The seek method will cause subsequent read commands to start reading from that position. The pos method returns the current position in the file. You should always close the file after finishing reading to free up the resources.

1.3.2 Other APIs

HDFS supports other common file system functionality. The following lists mentions the common ones.

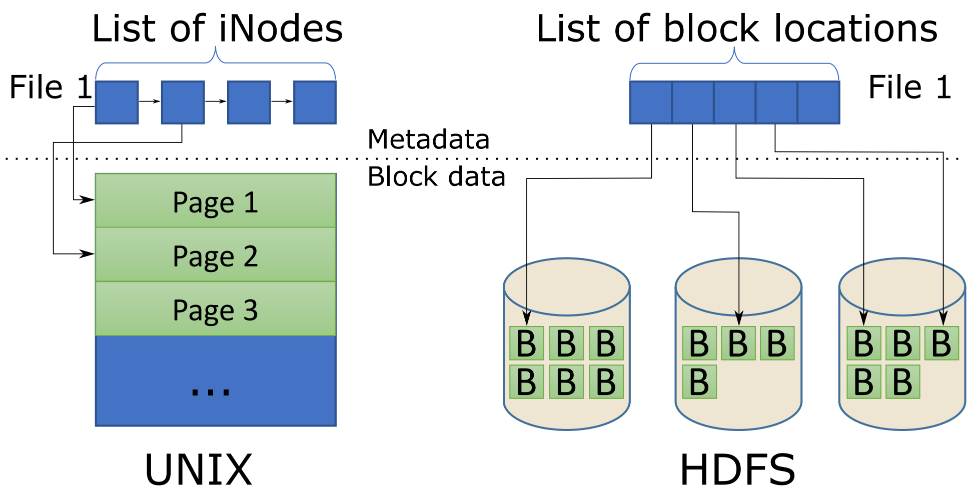

Figure 1.2: Analogy between Unix file system and HDFS

- listStatus: Lists all entities in a given path.
- rename: Changes the file name or moves it another location.
- delete: Deletes a file or a directory.
- getFileStatus: Gets additional information about a file or a directory such as the file size.

1.4 Internal Structure

This part opens the hood of the FileSystem API and describes in some detail how HDFS internally handles some of the common operations.

1.4.1 Data Storage

The data in HDFS can be split into two types of data, metadata and block data. Metadata includes everything but file contents. For example, metadata includes file name, location, block size, replication factor, owner, and access right. Block data includes the contents of the file, e.g., text data in a CSV file. In analogy to the Unix file system, the metadata corresponds to the iNodes whereas the block data corresponds to the disk pages. Figure 1.2 illustrates this analogy.

In HDFS, all metadata is kept in the main-memory of the namenode. To ensure that data is not lost on a system crash or restart, all operations are logged to disk on the namenode. When the namenode starts, it repeats all operations in the log to recover. The namenode also supports checkpointing to avoid repeating the entire log. Keeping all metadata in main-memory increases the efficiency of many filesystem operations that do not touch the actual data such as moving or listing the files. It also means that the namenode requires generally more memory than datanodes but does not require much of disk storage.

Block data is stored in datanodes as regular files on disk. Each block gets a unique ID that allows a reader to retrieve it. Only the namenode knows which block corresponds to which file. This allows datanodes to focus on just storing and retrieving individual blocks without worrying about linking blocks to files. In a distributed environment node failures become more often than a single machine. To ensure that block data is highly available, each block is stored on three different data nodes by default. Therefore, if one or two nodes become temporarily unavailable, all block data stays available from a third node. If one or two nodes become permanently unavailable, the lost block replicas are recreated from the available replica.

Problem 1.1 Given an HDFS cluster of one namenode and 10 datanodes. Each datanode has a 10TB disk that is allocated entirely for HDFS. What is the total capacity of this cluster assuming a default replication factor of three?

Answer: Total disk capacity allocated to HDFS = $10 \times 10 = 100TB$ Available HDFS capacity = $\frac{100}{3} = 33.3TB$

1.4.2 Writing Process

Since only the namenode keeps information about files, the file creation process must start through the namenode. During creation, there are three types of processes involved, writer, namenode, and datanodes. The writer is the process that has the data to be written to HDFS. The namenode and datanodes are described earlier. Notice that the machines that run these processes are not necessarily different. For example, the writer can be on one of the datanodes, the namenode, or outside the cluster. The process is generally the same with one exception when the writer process runs on one of the datanodes which is covered later.

Writing a file starts with the FileSystem#create(Path) method. It contacts the namenode which makes sure that a file can be created in the given path. It updates the metdata in memory, and logs the operation to disk. It also chooses three datanodes to store the three replicas of the first block as follows.

1. The first replica is assigned to a random datanode.
2. The second replica is assigned to a different node on a different rack than the first replica.
3. The third replica is assigned to a different node is the same rack as the second replica.

The first replica is assigned to a random node to balance the load. The second replica is assigned to a different rack for fault tolerance so that the block is not lost if the first rack becomes temporarily unavailable. The third replica is assigned to the same rack as the second one for efficiency to reduce inter-rack network communication. If all datanodes are in one rack, then all three replicas are assigned to three random datanodes in that rack.

When the writer starts writing to the FSDataOutputStream returned by the create method, the data will be sent over network to the datanode assigned to the first replica. The datanode will do two things. First, it will write the data to disk in the file associated with the block. Second, it will forward the data over network to the datanode of the second replica. The second datanode will do the same thing and forward the data to the third replica. This way, the writer needs to send the data to only one datanode. Since datanodes are usually within the same datacenter, forwarding the data to the remaining datanodes is usually done faster. When the first block is completely filled, the FSDataOutputStream contacts the namenode which creates three block replicas for the second block and the process continues. Notice that at any point of time, the writer is writing to only one datanode. Also, each datanode does not know which file it is currently writing which simplifies the design.

Self Writing

One special case in writing is when the writer process is running on one of the datanodes. In this case, the process works exactly the same except for one change. The namenode will detect that the writer is one of the datanodes and it will always assign the first replica to that node. The second and third replicas are assigned as before. This ensures that no network communication is needed to write the first block which reduces the network overhead. While this case might look like a special case, it is actually a very common case in distributed data processing. Typically, datanodes are compute nodes run on the same physical machines as discussed later. In this case, when the final output is being written to HDFS, every node is writing a small chunk of the output. It makes more sense in this case that each machine writes the first replica to its own disk while the second and third replicas are created on other datanodes. If all machines participate in the output writing, the load will still be roughly balanced.

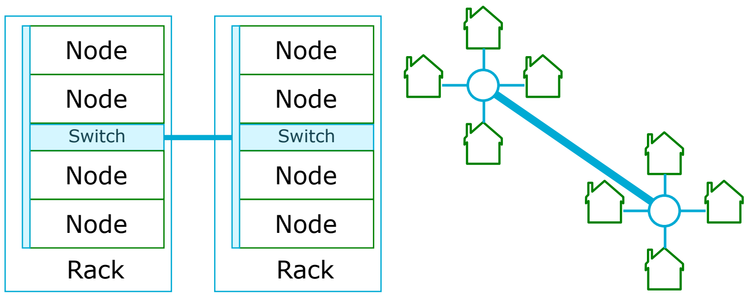

Figure 1.3: Analogy between cluster organization in racks and houses in cities

R [A note about racks] In data centers, nodes are organized physically in racks. Each rack contains about 16-32 nodes each. Nodes in one rack are connected with an on-rack switch that connects all nodes in the rack. Rack switches are connected together to allows nodes from different racks to communicate. In this setting, if two nodes in the same switch are communicating, the on-rack switch is busy but other switches are not involved. However, when two nodes in two racks are connected, both their on-rack switches and the medium connecting them are busy which reduces the network efficiency. That is why HDFS tries to reduce inter-rack communication by assigning two replicas to the same rack.

To understand this setting, think of each node as a house and each rack as a city. When cars (network packets) move between two houses in the same city (rack), the roads in the city become busy but other cities are not affected. However, when a car needs to move from one city to another city, then it has to use the roads inside the two cities as well as the highway that connects all cities. To optimize traffic, you would want more cars moving inside each city and fewer cars moving on the highway between two cities.

Problem 1.2 Consider an HDFS cluster with ten datanodes (DN1...DN10). DN1 is writing a file of 9GB and the block size is 128MB and the replication factor is three. Assume that all datanodes are in one rack.

1. How many blocks are created for this file? Answer: (9×1,024)/128=72 blocks
2. How many block replicas are created in total? Answer: 72×3=216 block replicas
3. How much data is written to the local disk of DN1? Answer: 9 GB (The entire files)
4. What is the average data size written to other datanodes? Answer: Total amount of data written to other machines = 9×2=18 GB. Average data per node = 18/9=2 GB

Problem 1.3 Repeat the previous question when the writer process is not one of the datanodes.

1.5 Reading Process

The reading process is easier than writing since no blocks are created. The process starts with the FileSystemopen method which calls the namenode to determine where the first block is. While the blockdata is not stored on the namenode, it is the only node that knows which blocks belong to the requested file. The namenode identifies the file from the metadata which is stored in memory and locates the three replicas that belong to the first block. The namenode will randomly choose one of the block replicas to read. If the reader process is a datanode, it will choose one of the replicas in the following order:

1. Choose a replica that is stored on the requester.
2. Choose a replica on another datanode in the same rack as the requester.
3. Choose any replica at random.

When the datanodes are also the compute nodes, option 1 is likely to be selected most of the time which reduce network access during the processing phase.

Once a block replica is selected, subsequent FSDataInputStream#read operations will request the data from the datanode. The datanode will locate the block and starts streaming the data to the reader. Once the end of the block is reached, the reading process will seamlessly move to the next block and the reading process continues. The seek operation will locate the block that contains the requested position and starts reading from that position.

Problem 1.4 Consider an HDFS cluster with one namenode and 8 datanodes. A file of size 3GB and a block size of 64MB is stored in HDFS. Assume the file is perfectly balanced across nodes. Now, consider a reader process running on one of the datanodes is reading the entire file. How much of the file will be read locally from that datanode and how much will be read remotely from other datanodes?

Total number of blocks in the file is = (3×1,024)/64=48 blocks

Total number of block replicas = 48×3=144 block replicas

Assuming perfect load balancing, we expect the number of block replicas on the reader datanode to be 144/8=18 block replicas.

The reading process will read these 18 blocks from the local disk while it will read all remaining blocks from remote nodes.

Amount of local reading = 18×64=1,152 MB

Amount of remote reading = (48-18)×64=1,920 MB

1.6 Advanced HDFS Operations

The operations described above cover the basic file access API, read, write, rename, and delete. These operations work for any supported file system, e.g., local file system and HDFS. However, HDFS provides some additional advanced features that are not available in all file systems. These are not common to use on a regular application but they can be useful in some cases.

1.6.1 Cheap Concatenation

HDFS provides a specialized API for concatenating a set of files. Recall that any file consists of a sequence of blocks and only the namenode knows which block belongs to which file. Therefore, to concatenate a set of files, the operation can be performed entirely in the main memory of the namenode. The namenode will simply change the in-memory block structure to assign all blocks to one file without even telling the datanodes. This operation requires all concatenated files to be deleted and replaced with one file since a block cannot belong to two files.

1.6.2 Retrieve Block Locations

Even though HDFS does a good job at hiding the distribution of the file, it still allows the program to determine where the blocks are stored. This is mainly helpful for distributed compute engines, e.g., Hadoop MapReudce and Spark, to assign tasks to compute nodes based on which machines have a block replica. The goal is to maximize the change of a compute node collocated with its data to increase local read and reduce remote read.

1.6.3 Load Balancing

HDFS provides a system admin interface that allows a sysadmin to balance the load across datanodes if the load becomes highly unbalanced. The load balancing process will move blocks among datanodes to reduce the load imbalance.

1.6.4 HDFS Shell

This is not really an advanced feature but we do not cover it in detail in this chapter so we mention it briefly here. HDFS provides a command-line shell interface that is very similar to the Unix shell.

If HDFS is properly configures, you can access shell commands similar to the following example.

```
hdfs dfs -ls
```

Simply, you precede your command with hdfs dfs –. Most commands work similar to Unix commands but there could be some changes. For a list of all commands and how to use them, check the following link.

`https://hadoop.apache.org/docs/r3.2.2/hadoop-project-dist/hadoop-common/FileSystemShell html`

1.7 Exercises

Exercise 1.1 In the following questions, assume we have a cluster of one namenode and 10 datanodes all in one rack. Each datanode has a disk of 10 terabytes which is all available to HDFS. HDFS is configured with a default replication factor of 3 and a default block size of 128 MB.

1. According to this configuration, what is the capacity of HDFS? In other words, how much data can we store in HDFS.
2. A driver node that is not one of the data nodes creates a file of size 2GB. How much is the total network IO (incurred on all the machines) required to upload the data file?
3. One of the data nodes is writing a 2GB file. How much is the total network IO incurred on all data nodes while the 2GB file is being written? Explain your answer.
4. How much network IO is required to download the file back from HDFS to the master node?
5. How much is the expected network IO to download the file back from HDFS to one of the data nodes? (Hint: Calculate the probability of a block being remote to one data node. You can assume that all nodes are in one rack.)

■