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

The design of HDFS

Filesystems that **manage the storage across a network of machines** are called **distributed filesystems**. One of the biggest challenges is making the distributed filesystem **tolerate node failure without suffering data loss**. The hadoop distributed filesystem is called **HDFS**, which stands for Hadoop Distributed Filesystem. HDFS is a filesystem designed for **storing very large files** with **streaming data access patterns**, running on **clusters of commodity hardware**.
HDFS is built around the idea that the most efficient data processing parttern is **write-once, read-many-times pattern**. 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.
Files in HDFS may be written to by a single writer. Writers are always made **at the end of the file, in append-only fashion**.

HDFS Concepts

Blocks

A disk has a **block size**, which is **the minimum amount of data that it can read or write**. Filesystems for a single disk build on this by dealing with data in blocks, which are **an integral multiple of the disk block size**. Filesystem blocks are typically a few **kilobytes in size**, whereas disk blocks are **normally 512 bytes**.
Like in a filesystem for a single disk, **files in HDFS are broken into block-sized chunks**, which are **stored as independent units**. **Unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage**.
HDFS has a much larger unit - **128 MB** by default. Why is a block in HDFS so large? The reason is to **minimize the cost of seeks**. If the block is large enough, **the time it takes to transfer the data from the disk can be significantly longer than the time to seek to the start of the block**. This argument shouldn't be taken too far, however. Map tasks in MapReduce normally **operate on one block at a time**, so if you

have too few tasks(fewer than nodes in the cluster), your jobs will run slower than they could oterwise.

There's nothing that requires the blocks from a file to be stored on the same disk, so they can take advantage of any of the disks in the cluster. In fact, it would be possible, if unusual, to store a single file on an HDFS cluster whose blocks filled all the disks in the cluster.

Having a block abstraction for a distributed filesystem brings several benefits:

- A file can be larger than any single disk in the network.
- Making the unit of abstraction a block rather than a file simplifies the storage subsystem. So the storage subsystem only deals with blocks, simplifying storage management: blocks are a fixed size.
- Furthermore, blocks fit well with replication for proiding fault tolerance and availability.

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,** which is callled **Data Integrity** on guarding against corrupt data.

Some applications can **choose to set a high replication factor for the blocks in a popular file to spread the read load on the cluster**.

Command to list the blocks that make up each file in HDFS:

```
$ hdfs fsck <path> -files -
blocks
```

Namenodes and Datanodes

An HDFS cluster has two types of nodes: **a namenode and a number of datanodes**.

The namenode **manages the filesystem namespace**. It maintains the **filesystem tree and the metadata** for all the files and directories. 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, which will be reconstructed from datanodes when the system starts. The block mappings are stored in a namenode's memory, and not on disk.

Datanodes are the workhorses of the filesystem. They will **report back to the namenode periodically with lists of blocks that they are storing**.
If the namenode failed, **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.

Hadoop provides two mechanisms to make the namenode resilient to failure.:

- **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** (local disk or remote NFS mount). These writes are synchronous and atomic.
- Run a **secondary namenode** which **does not act as a namdenode**. **Its main role is to periodically merge the namespace image with 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 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**. The usual course of action in this case is to copy the namenode's metadata files that are on NFS to the secondary and run it as the new primary.
- For **High Availability**, it is possible to run a **hot standby namenode** instead of a secondary.

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**.

The Command-Line Interface

By defalut, HDFS will replicate each filesystem block into **3 replications**. When running with a single datanode, HDFS can't replicate blocks to three datanodes, so it would **perpetually warn about blocks being under-replicated**.
Let's create a directory first just to see how it is displayed in the listing:

```
$ hdfs dfs -mkdir books
$ hdfs dfs -ls .

```

```
output
(file mode) (replication)
drwxr-xr-x   - root supergroup    0 2016-03-16 13:22
 books
 -rw-r--r--   1 root supergroup  119 2016-03-16 13:21
 test.txt
```

The entry in replication column is empty for directories because the concept of replication does not apply to them — **directories are treated as metadata and stored by the namenode, not the datanodes**.

File Permissions in HDFS
There are tree types of permission in HDFS:

- The read permission (r) is required to read files or list the contents of a directory.
- The write permission (w) is required to write a file or, for a directory, to create or delete files or directories in it.
- The execute permission (x) is ignored for a file because you can't execute a file on HDFS (unlike POSIX), and for a directory this permission is required to access its children.

Each file and directory has **an owner, a group, and a mode**. The mode (e.g. `drwxr-xr-x`) is made up of

- `d` for dir or `-` for files
- the permissions for the user who is the owner
- the permissions for the users who are members of the group
- the permissions for users who are neither the owners nor members of the group.

By default, Hadoop **runs with security disabled**, which means that a client's identity is not authenticated. There is a concept of a superuser, which is the identity of the namenode process. Permissions checks are not performed for the superuser.

Hadoop Filesystems

Hadoop has an **abstract notion of filesystems**, of which HDFS is just one implementation. The Java abstract class **org.apache.hadoop.fs.FileSystem** represents the client interface to a filesystem in Hadoop, and there are several concrete implementations:

Filesystem	URI scheme	Java implementation (all under org.apache.hadoop)	Description
Local	`file`	`fs.LocalFileSystem`	A filesystem for a lo disk with client-side `RawLocalFileSyste` filesystem with no c LocalFileSystem.
HDFS	`hdfs`	`hdfs.DistributedFileSystem`	Hadoop's distributed HDFS is designed to in conjunction with
WebHDFS	`webhdfs`	`hdfs.web.WebHdfsFileSystem`	A filesystem providi read/write access to See HTTP.
Secure WebHDFS	`swebhdfs`	`hdfs.web.SWebHdfsFileSystem`	The HTTPS version

Hadoop provides many interfaces to its filesystems, and it generally uses the **URI scheme to pick the correct filesystem instance** to communicate with

```
$ hadoop fs -ls file:///
$ hadoop fs -ls
hdfs://localhost:9000/
```

Interfaces

Hadoop is written in Java, so most Hadoop filesystem interactions are mediated through the Java API. The **filesystem shell**, for example, is a Java application that uses the Java `FileSystem` class to provide filesystem operations. Here are two commonly used other filesystem interfaces with HDFS:

- NFS
  It is possible to **mount HDFS on a local client's filesystem** using Hadoop's **NFSv3 gateway**. You can then **use Unix utilities** (such as `ls` and `cat`) to **interact with the filesystem**, upload files, and in general use POSIX libraries to access the filesystem from any programming language. **Appending to a file works, but random modifications of a file do not, since HDFS can only write to the end of a file**.

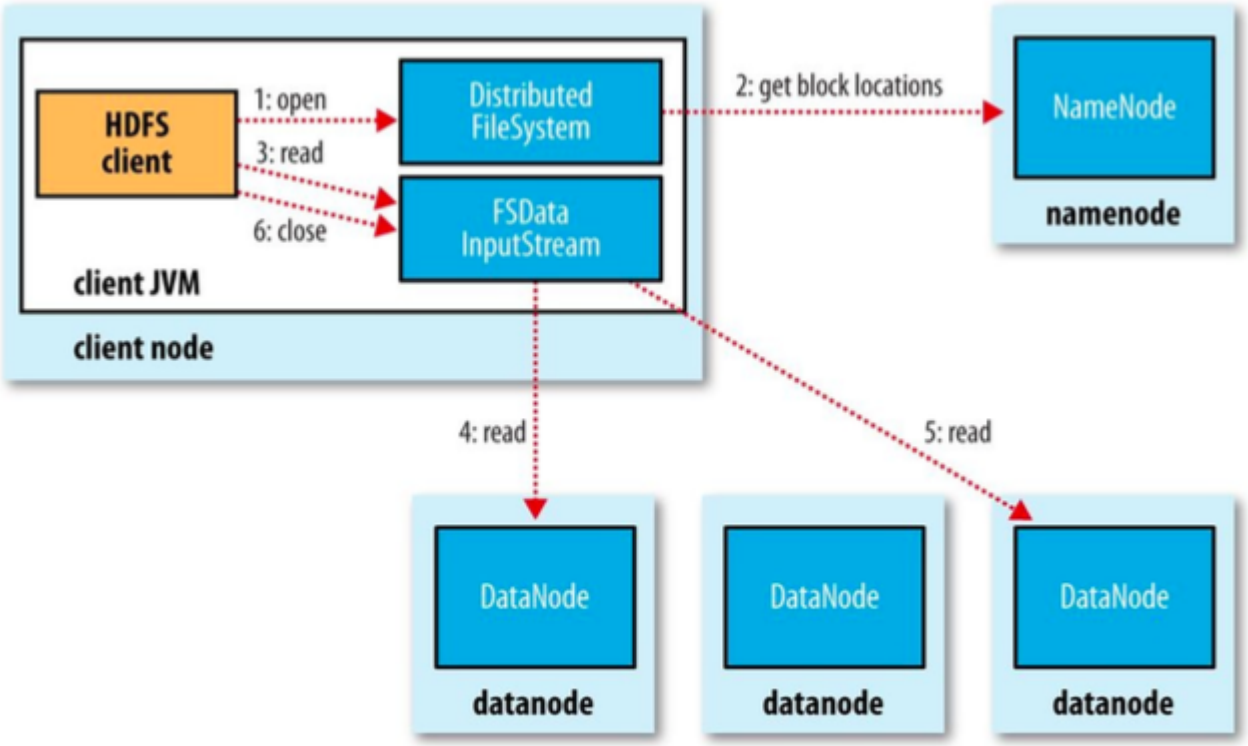
- FUSE
  Filesystem in Userspace (FUSE) allows filesystems that are implemented in user space to be integrated as Unix filesystems. Hadoop's Fuse-DFS contrib

module allows HDFS (or any Hadoop filesystem) to be mounted as a standard local filesystem. Fuse-DFS is implemented in C using libhdfs as the interface to HDFS. At the time of writing, the Hadoop NFS gateway is the more robust solution to mounting HDFS, so should be preferred over Fuse-DFS.

Data Flow

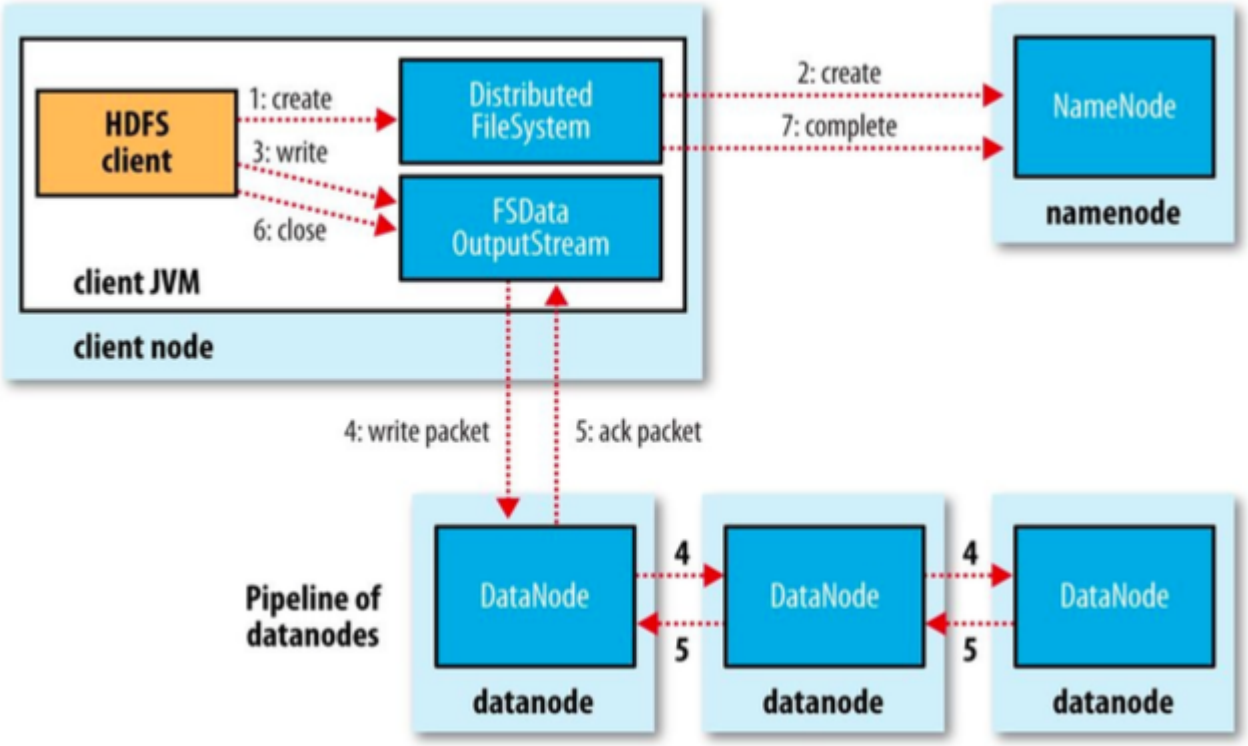
File Read

The image shows the main sequence of events when reading a file from HDFS cluster.



File Write

The image shows the main sequence of events when writing a file to HDFS cluster.



Parallel copying across clusters with distcp

The HDFS access patterns that we have seen so far focus on **single-threaded access**. It's possible to act on a collection of files. Hadoop comes with a useful program called `distcp` for copying data to and from Hadoop filesystems **in parallel**:

```
$ hadoop distcp file1 file2      # same to hadoop fs -cp file1 file2
$ hadoop distcp dir1 dir2        # If dir2 exists, new structure will be
dir2/dir1
$ hadoop distcp -overwrite dir1 dir2    # dir2 will be overwritten
$ hadoop distcp -update dir1 dir2       # synchronize the change with
 dir2
```

`distcp` is implemented **as a MapReduce job** where the work of copying is done **by the maps that run in parallel across the cluster**. Each file is copied by a single map, and distcp tries to give each map approximately the same amount of data by bucketing files into roughly equal allocations. By default, up to **20 maps are used**, but this can be changed by specifying the `-m` argument to distcp.

A very common use case for distcp is for **transferring data between two HDFS clusters**:

```
# -delete: delete any files or directories from the destination
#          that are not present in the source

```

```
# -p:      file status attributes like permissions, block size,
#          and replication are preserved
 $ hadoop distcp -update -delete -p hdfs://namenode1/foo
 hdfs://namenode2/foo
```

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 results that the first replica of each block would reside on the node running the map (until the disk filled up). So it's best to start by running distcp with the default of 20 maps per node. However, you can also use the **balancer tool** (known as **Balancer**) to subsequently even out the block distribution across the cluster.