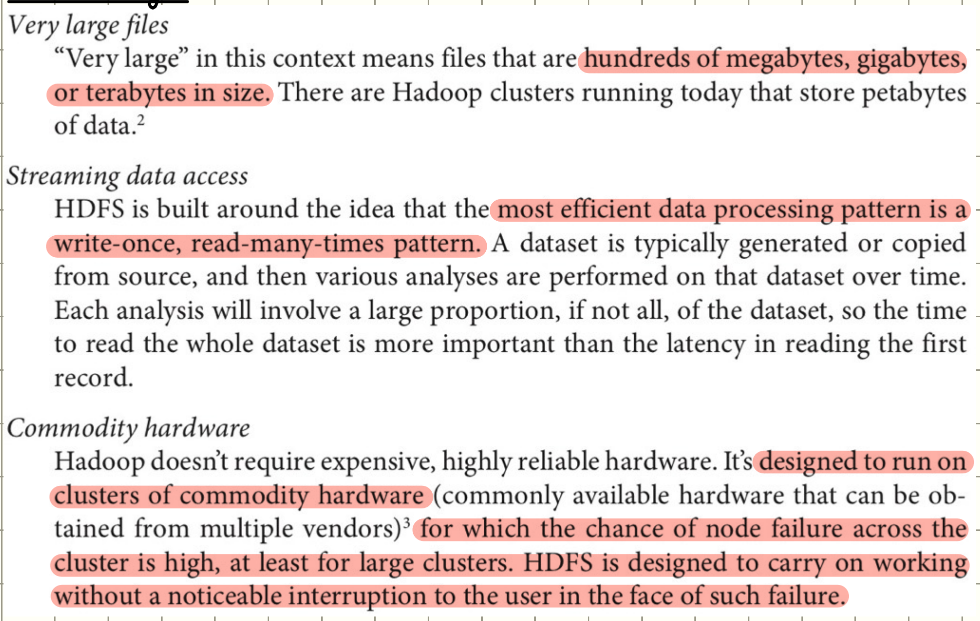
HDFS : hadoop distributed FS : the distributed FS that comes with hadoop

Filesystems that manage the storage across a network of machines are called distributed filesystems.

HDFS design


Very large files

"Very large" in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size. There are Hadoop clusters running today that store petabytes of data.[2]

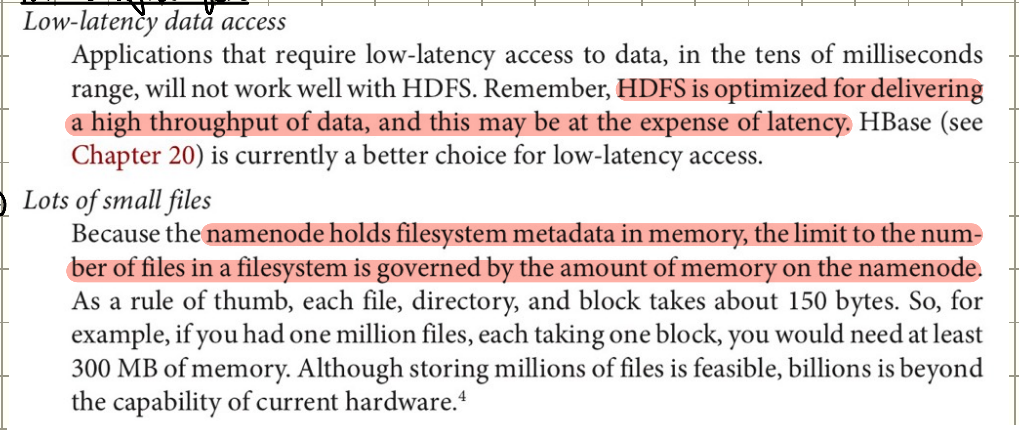
Streaming data access

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)[3] 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.

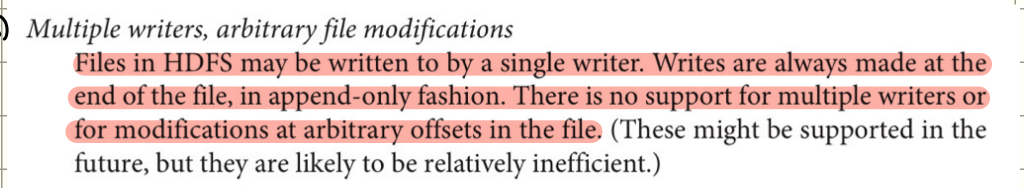
Not designed for


1) *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 (see Chapter 20) is currently a better choice for low-latency access.

2) *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.[4]


3) *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. (These might be supported in the future, but they are likely to be relatively inefficient.)

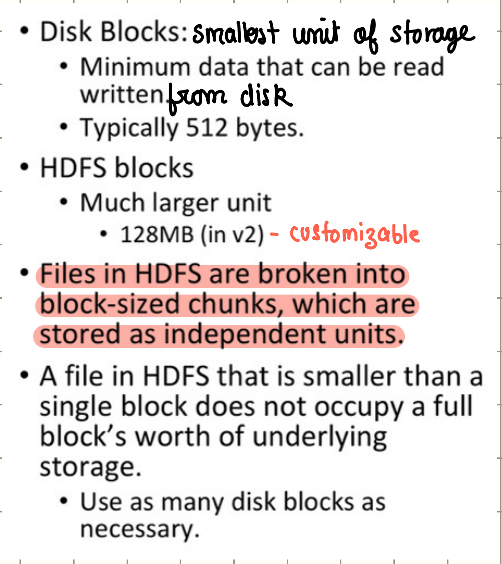
GFS supported this but were scarcely used & inefficient
Also can lead to undeterminism when multiple writers write.

4) Not POSIX compliant

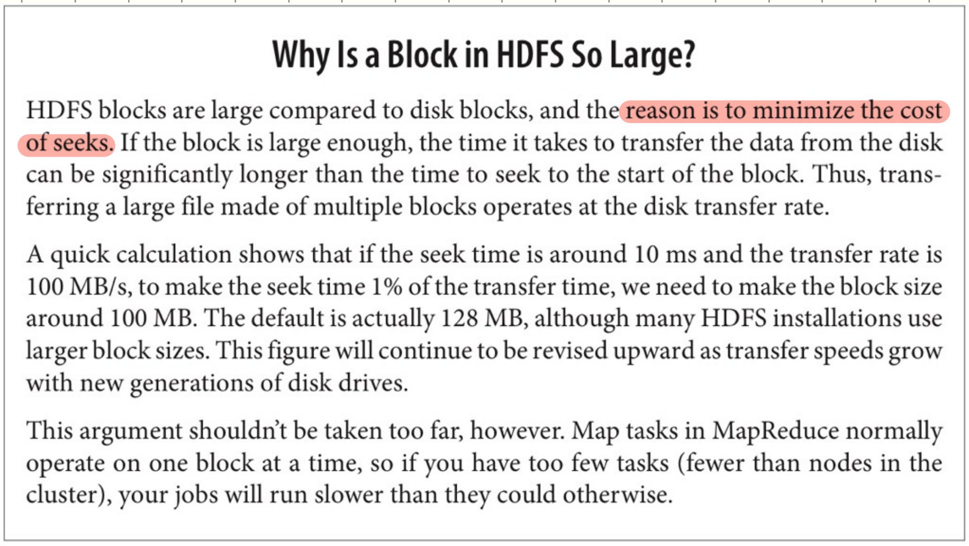
cant mount & use standard FS commands
Needs a separate client

HDFS concepts

1] Blocks



- Disk Blocks: smallest unit of storage
  - Minimum data that can be read written from disk
  - Typically 512 bytes.
- HDFS blocks
  - Much larger unit
    - 128MB (in v2) - customizable
- 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.
  - Use as many disk blocks as necessary.



Why Is a Block in HDFS So Large?

HDFS blocks are large compared to disk blocks, and 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. Thus, transferring a large file made of multiple blocks operates at the disk transfer rate.

A quick calculation shows that if the seek time is around 10 ms and the transfer rate is 100 MB/s, to make the seek time 1% of the transfer time, we need to make the block size around 100 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.

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 otherwise.

TLDR: the % of time spent for disk arm to reach the block should be << the time spent actually reading / transferring

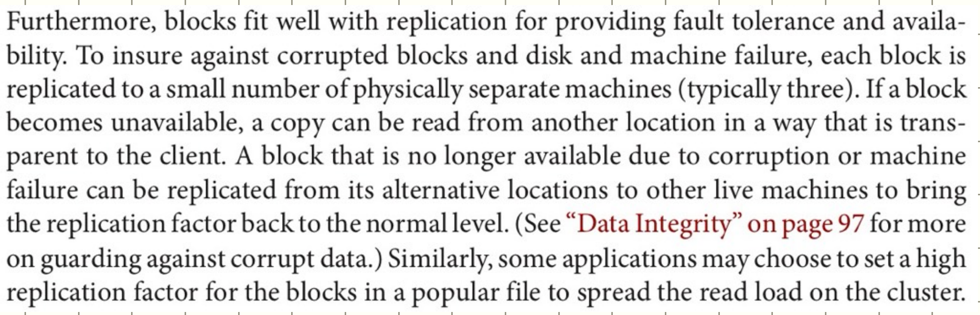
} block size shouldn't be too large also

Benifits of having blocks

1] Files physically larger than any disk can be split into blocks & stored in HDFS

2] Adds simplicity to storage subsystem by having fixed sized blocks. Eliminates a lot of metadata concerns.

3] Blocks work well with the idea of replication



Furthermore, blocks fit well with replication for providing fault tolerance and availability. 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. (See "Data Integrity" on page 97 for more on guarding against corrupt data.) 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.

2] Namenodes & Datanodes

2 types of nodes : one namenode (master)
multiple datanodes (slaves)

Namenode : maintains FS tree & metadata for all files & directories } the namespace
: 2 files managed : namespace image ★
: edit log
: does not store block locations persistantly, this info is reconstructed by communicating to all datanodes when system starts.

Datanode : all blocks located here
: Report to namenode periodically with list of blocks they store.

What if namenode fails?

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.

1] Back up namenode persistant state of the FS metadata
↳ only some info in namenode is persistant (Eg: block locations are not persistant while FS namespace is)

Usually hadoop backs it up to local disk as well as a remote NFS mount

2] We can also run a secondary namenode.

Main role : Merge namespace image with edit logs to prevent edit log from becoming very 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. 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.

3] Block Caching

- Frequently accessed files' blocks can be cached in datanode's memory in an off-heap block cache
- A block is by default cached in only one datanode (configurable on per file basis)

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.

4) HDFS Federation

- In namenode, the namespace is kept in memory.
  ⇒ For big clusters RAM in NN becomes limiting factor

The default of 1,000 MB of namenode memory is normally enough for a few million files, but as a rule of thumb for sizing purposes, you can conservatively allow 1,000 MB per million blocks of storage.

For example, a 200-node cluster with 24 TB of disk space per node, a block size of 128 MB, and a replication factor of 3 has room for about 2 million blocks (or more): 200 × 24,000,000 MB/(128 MB × 3). So in this case, setting the namenode memory to 12,000 MB would be a good starting point.

5] High Availability (HA) : HDFS should always be available

Before Hadoop 2 : NN = single point of failure
If NN down
: Need to start new NN
: Need to load namespace to RAM
: Replaying edit log
: Receive block reports from enough DN
⇒ large cold startup ⇒ large downtime

Solution ⇒ Hadoop 2 introduced HA
⇒ A standby NN, if primary NN fail, standby NN quickly picks up

Architectural changes required
: Shared edit log : active NN writes edit log to shared storage
: Block reports are sent to both NNs & both will have namespace in memory
⇒ Standby NN can take over seamlessly with these 2 information
: Clients must know both NN addresses to switch transparently
: No secondary NN now, standby NN takes up its role & does checkpointing

Failover process

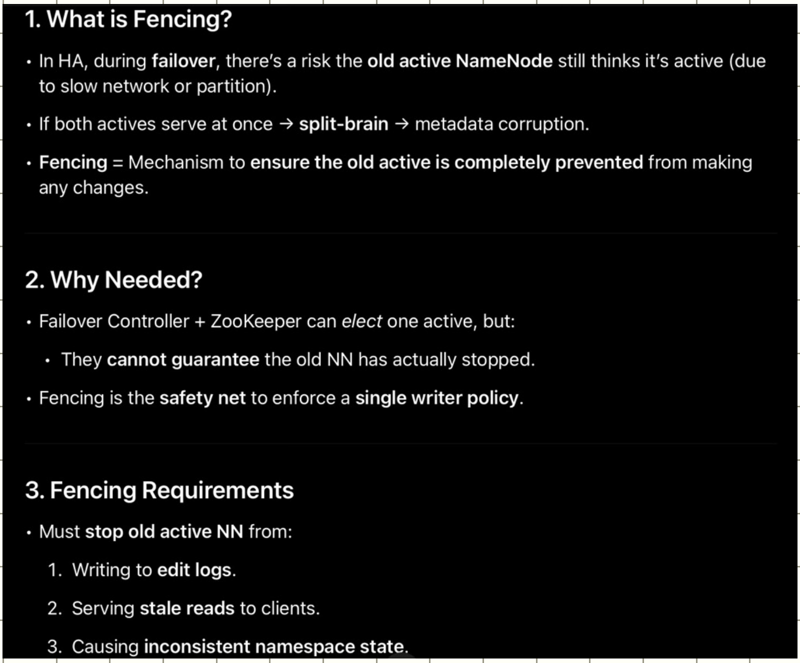
: each NN has a lightweight process monitoring it (via heartbeat)

: Uses Zookeeper failover controller by default , ensures only one NN active at a time, helps in case of network partitioning

2 types of failover 1] Graceful : manually triggered for maintanance
: easy transition from active NN → standby.

2] Ungraceful : automatically triggerd on NN fail or network partition

In case of network partitioning
Risk : standby NN becomes active while old NN thinks it is still active.
Solution : Fencing (prevents split brain)
↳ ensures old active NN cannot cause corruption



1. What is Fencing?

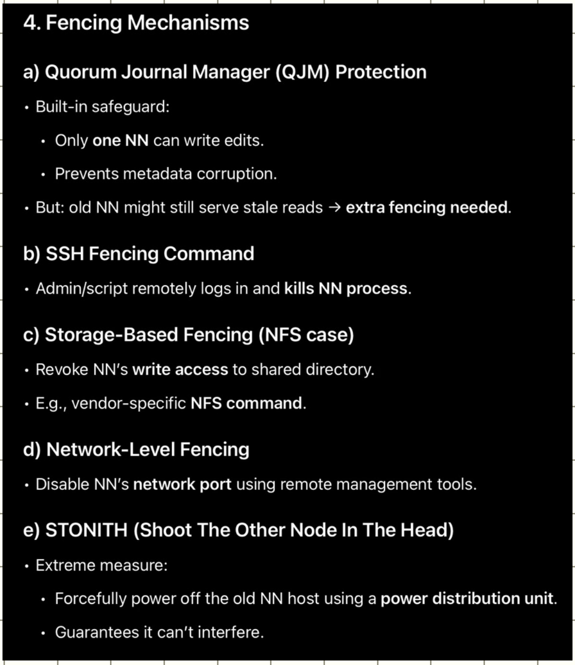
- In HA, during **failover**, there's a risk the **old active NameNode** still thinks it's active (due to slow network or partition).
- If both actives serve at once → **split-brain** → metadata corruption.
- **Fencing** = Mechanism to **ensure the old active is completely prevented** from making any changes.

2. Why Needed?

- Failover Controller + ZooKeeper can *elect* one active, but:
  - They **cannot guarantee** the old NN has actually stopped.
- Fencing is the **safety net** to enforce a **single writer policy**.

3. Fencing Requirements

- Must **stop old active NN** from:
  1. Writing to **edit logs**.
  2. Serving **stale reads** to clients.
  3. Causing **inconsistent namespace state**.



4. Fencing Mechanisms

a) Quorum Journal Manager (QJM) Protection

- Built-in safeguard:
  - Only **one NN** can write edits.
  - Prevents metadata corruption.
- But: old NN might still serve stale reads → **extra fencing needed**.

b) SSH Fencing Command

- Admin/script remotely logs in and **kills NN process**.

c) Storage-Based Fencing (NFS case)

- Revoke NN's **write access** to shared directory.
- E.g., vendor-specific **NFS command**.

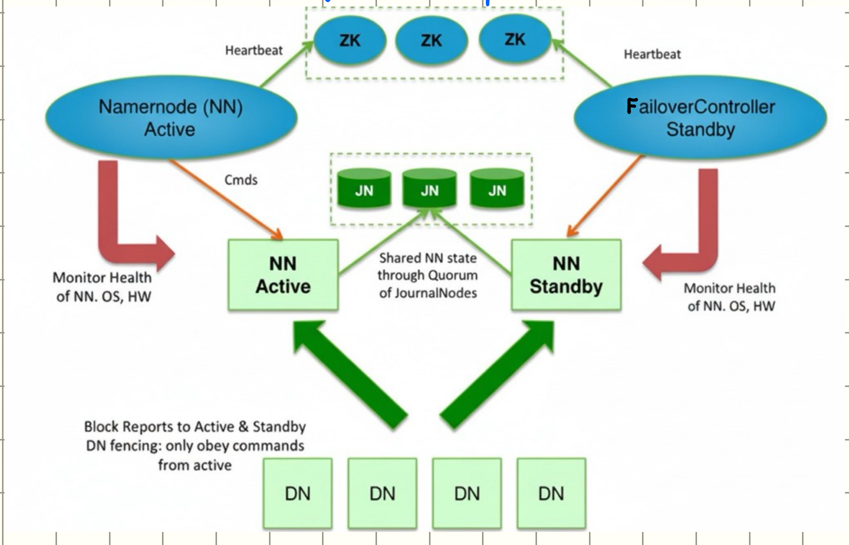
d) Network-Level Fencing

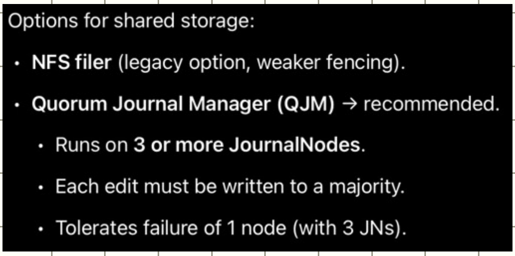
- Disable NN's **network port** using remote management tools.

e) STONITH (Shoot The Other Node In The Head)

- Extreme measure:
  - Forcefully power off the old NN host using a **power distribution unit**.
  - Guarantees it can't interfere.

HA with shared storage & Zookeeper


ZK
ZK
ZK
Heartbeat
Heartbeat
Namenode (NN) Active
FailoverController Standby
Cmds
JN
JN
JN
Monitor Health of NN. OS, HW
NN Active
Shared NN state through Quorum of JournalNodes
NN Standby
Monitor Health of NN. OS, HW
Block Reports to Active & Standby DN fencing: only obey commands from active
DN
DN
DN
DN



Options for shared storage:

- **NFS filer** (legacy option, weaker fencing).
- **Quorum Journal Manager (QJM)** → recommended.
  - Runs on **3 or more JournalNodes**.
  - Each edit must be written to a majority.
  - Tolerates failure of 1 node (with 3 JNs).

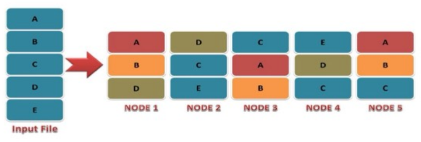
6] Fault Tolerence

HDFS Fault Tolerance

- Fault tolerance in Hadoop HDFS refers to the working strength of a system in unfavorable conditions and how that system can handle such a situation
- HDFS is highly fault tolerant and handled faults.
  - There are approaches used for handling faults of data nodes or disks holding data blocks.
    - These could be based on replication, to a replication factor till Hadoop 3. So whenever if any machine/disk in the cluster goes down, then data is accessible from other machines/disk in which the same copy of data was created
    - Using Erasure coding in Hadoop 3
  - There are also approaches to handle faults of Name Node and the availability of the metadata pointing to the data blocks

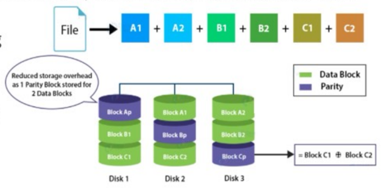
HDFS Fault Tolerance (Replication)

- Fault tolerance of data using Replicas is achieved by creating a replica of users' data based on the replication factor on different machines in the HDFS cluster.
- So if any machine in the cluster goes down, then data is accessible from other machines in which the same copy of data was created



HDFS Fault Tolerance (Erasure Coding)

- Fault tolerance of data could also be based on Erasure coding
- Erasure coding works similar to RAID by striping the file into small units of sequential blocks and storing them consecutively on various disks.
- For each strip of the original dataset, a certain number of parity cells are calculated using erasure coding algorithm called encoding and stored. If any of the machines fails, the block can be recovered from the parity cell.
- The error in any striping cell can be recovered from the calculation based on the remaining data and parity cells; the process known as decoding.



CLI interface

Start by copying a file from the local filesystem to HDFS:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt \
  hdfs://localhost/user/tom/quangle.txt
```

In fact, we could have omitted the scheme and host of the URI and picked up the default, `hdfs://localhost`, as specified in *core-site.xml*:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt /user/tom/quangle.txt
```

We also could have used a relative path and copied the file to our home directory in HDFS, which in this case is */user/tom*:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt quangle.txt
```

Let's copy the file back to the local filesystem and check whether it's the same:

```
% hadoop fs -copyToLocal quangle.txt quangle.copy.txt
% md5 input/docs/quangle.txt quangle.copy.txt
MD5 (input/docs/quangle.txt) = e7891a2627cf263a079fb0f18256ffb2
MD5 (quangle.copy.txt) = e7891a2627cf263a079fb0f18256ffb2
```

```
% hadoop fs -mkdir books
% hadoop fs -ls .
Found 2 items
drwxr-xr-x   - tom supergroup          0 2014-10-04 13:22 books
-rw-r--r--   1 tom supergroup        119 2014-10-04 13:21 quangle.txt
```

The information returned is very similar to that returned by the Unix command `ls`

file mode

replication factor

(directories don't have any concept of replication, only files)

file owner & group

size of file

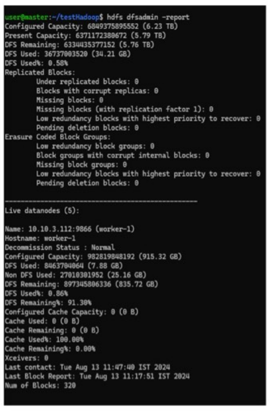
HDFS has a permissions model for files and directories that is much like the POSIX model. There are three types of permission: the read permission (`r`), the write permission (`w`), and the execute permission (`x`). The read permission is required to read files or list the contents of a directory. The write permission is required to write a file or, for a directory, to create or delete files or directories in it. The execute permission 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.

Some Basic Operations in HDFS

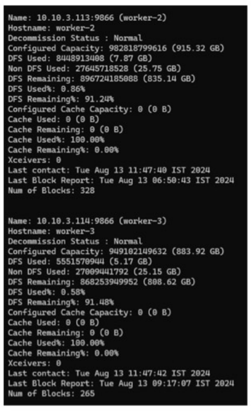
HDFS provides a command-line interface that mimics standard Unix file system commands, making it familiar to many users.

- `hdfs dfs -ls /`: List the files and directories in the root directory.
- `hdfs dfs -mkdir /mydir`: Create a new directory named `mydir`.
- `hdfs dfs -put localfile /mydir/`: Copy `localfile` from the local file system to the `/mydir/` directory in HDFS.
- `hdfs dfs -get /mydir/remotefile .`: Copy `remotefile` from HDFS to the current local directory.
- `hdfs dfs -rm /mydir/remotefile`: Delete a file from HDFS.

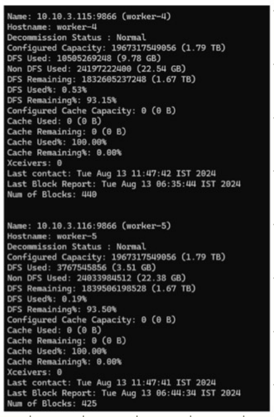
HDFS - DFS Admin Report



```
user@master:~/testHadoop$ hdfs dfsadmin -report
Configured Capacity: 6849375895552 (6.23 TB)
Present Capacity: 6371172380672 (5.79 TB)
DFS Remaining: 6334435377152 (5.76 TB)
DFS Used: 36737003520 (34.21 GB)
DFS Used%: 0.58%
Replicated Blocks:
        Under replicated blocks: 0
        Blocks with corrupt replicas: 0
        Missing blocks: 0
        Missing blocks (with replication factor 1): 0
        Low redundancy blocks with highest priority to recover: 0
        Pending deletion blocks: 0
Erasure Coded Block Groups:
        Low redundancy block groups: 0
        Block groups with corrupt internal blocks: 0
        Missing block groups: 0
        Low redundancy blocks with highest priority to recover: 0
        Pending deletion blocks: 0

-------------------------------------------------
Live datanodes (5):

Name: 10.10.3.112:9866 (worker-1)
Hostname: worker-1
Decommission Status : Normal
Configured Capacity: 982819848192 (915.32 GB)
DFS Used: 8463704064 (7.88 GB)
Non DFS Used: 27010301952 (25.16 GB)
DFS Remaining: 897345806336 (835.72 GB)
DFS Used%: 0.86%
DFS Remaining%: 91.30%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 0
Last contact: Tue Aug 13 11:47:40 IST 2024
Last Block Report: Tue Aug 13 11:17:51 IST 2024
Num of Blocks: 320
```

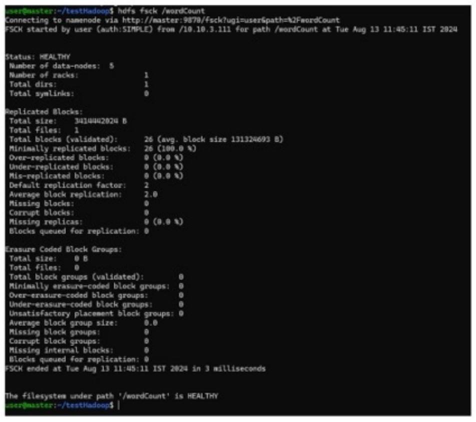


```
Name: 10.10.3.113:9866 (worker-2)
Hostname: worker-2
Decommission Status : Normal
Configured Capacity: 982818799616 (915.32 GB)
DFS Used: 8448913408 (7.87 GB)
Non DFS Used: 27645718528 (25.75 GB)
DFS Remaining: 896724185088 (835.14 GB)
DFS Used%: 0.86%
DFS Remaining%: 91.24%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 0
Last contact: Tue Aug 13 11:47:40 IST 2024
Last Block Report: Tue Aug 13 06:50:43 IST 2024
Num of Blocks: 328


Name: 10.10.3.114:9866 (worker-3)
Hostname: worker-3
Decommission Status : Normal
Configured Capacity: 949102149632 (883.92 GB)
DFS Used: 5551570944 (5.17 GB)
Non DFS Used: 27009441792 (25.15 GB)
DFS Remaining: 868253949952 (808.62 GB)
DFS Used%: 0.58%
DFS Remaining%: 91.48%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 0
Last contact: Tue Aug 13 11:47:42 IST 2024
Last Block Report: Tue Aug 13 09:17:07 IST 2024
Num of Blocks: 265
```



```
Name: 10.10.3.115:9866 (worker-4)
Hostname: worker-4
Decommission Status : Normal
Configured Capacity: 1967317549056 (1.79 TB)
DFS Used: 10505269248 (9.78 GB)
Non DFS Used: 24197222400 (22.54 GB)
DFS Remaining: 1832605237248 (1.67 TB)
DFS Used%: 0.53%
DFS Remaining%: 93.15%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 0
Last contact: Tue Aug 13 11:47:42 IST 2024
Last Block Report: Tue Aug 13 06:35:44 IST 2024
Num of Blocks: 440


Name: 10.10.3.116:9866 (worker-5)
Hostname: worker-5
Decommission Status : Normal
Configured Capacity: 1967317549056 (1.79 TB)
DFS Used: 3767545856 (3.51 GB)
Non DFS Used: 24033984512 (22.38 GB)
DFS Remaining: 1839506198528 (1.67 TB)
DFS Used%: 0.19%
DFS Remaining%: 93.50%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 0
Last contact: Tue Aug 13 11:47:41 IST 2024
Last Block Report: Tue Aug 13 06:44:34 IST 2024
Num of Blocks: 425
```

HDFS **fsck** is used to check the health of the file system, to find missing files, over replicated, under replicated and corrupted blocks.



```
user@master:~/testHadoop$ hdfs fsck /wordCount
Connecting to namenode via http://master:9870/fsck?ugi=user&path=%2FwordCount
FSCK started by user (auth:SIMPLE) from /10.10.3.111 for path /wordCount at Tue Aug 13 11:45:11 IST 2024


Status: HEALTHY
 Number of data-nodes:  5
 Number of racks:               1
 Total dirs:                    1
 Total symlinks:                0

Replicated Blocks:
 Total size:    3414442024 B
 Total files:   1
 Total blocks (validated):      26 (avg. block size 131324693 B)
 Minimally replicated blocks:   26 (100.0 %)
 Over-replicated blocks:        0 (0.0 %)
 Under-replicated blocks:       0 (0.0 %)
 Mis-replicated blocks:         0 (0.0 %)
 Default replication factor:    2
 Average block replication:     2.0
 Missing blocks:                0
 Corrupt blocks:                0
 Missing replicas:              0 (0.0 %)
 Blocks queued for replication: 0

Erasure Coded Block Groups:
 Total size:    0 B
 Total files:   0
 Total block groups (validated):        0
 Minimally erasure-coded block groups:  0
 Over-erasure-coded block groups:       0
 Under-erasure-coded block groups:      0
 Unsatisfactory placement block groups: 0
 Average block group size:      0.0
 Missing block groups:          0
 Corrupt block groups:          0
 Missing internal blocks:       0
 Blocks queued for replication: 0
FSCK ended at Tue Aug 13 11:45:11 IST 2024 in 3 milliseconds


The filesystem under path '/wordCount' is HEALTHY
user@master:~/testHadoop$ |
```

Data Flow chill & read :)

Anatomy of a File Read

To get an idea of how data flows between the client interacting with HDFS, the name-node, and the datanodes, consider Figure 3-2, which shows the main sequence of events when reading a file.

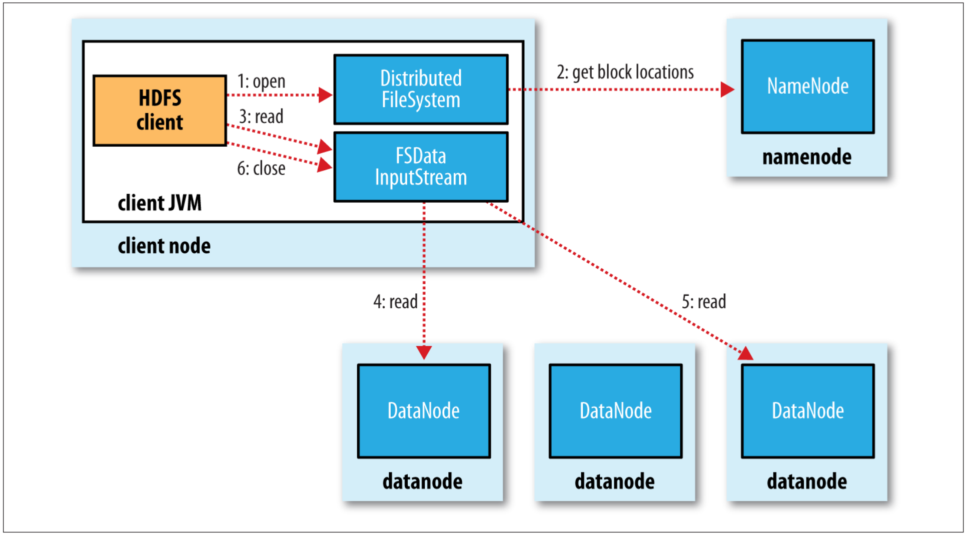


Figure 3-2. A client reading data from HDFS

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 3-2). `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; see "Network Topology and Hadoop" on page 70). 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 (see also Figure 2-2 and "Short-circuit local reads" on page 308).

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.

One important aspect of this design is that the client contacts datanodes directly to retrieve data and is guided by the namenode to the best datanode for each block. This design allows HDFS to scale to a large number of concurrent clients because the data traffic is spread across all the datanodes in the cluster. Meanwhile, the namenode merely has to service block location requests (which it stores in memory, making them very efficient) and does not, for example, serve data, which would quickly become a bottleneck as the number of clients grew.

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.

Rather than measuring bandwidth between nodes, which can be difficult to do in practice (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[8]

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)

This is illustrated schematically in Figure 3-3. (Mathematically inclined readers will notice that this is an example of a distance metric.)

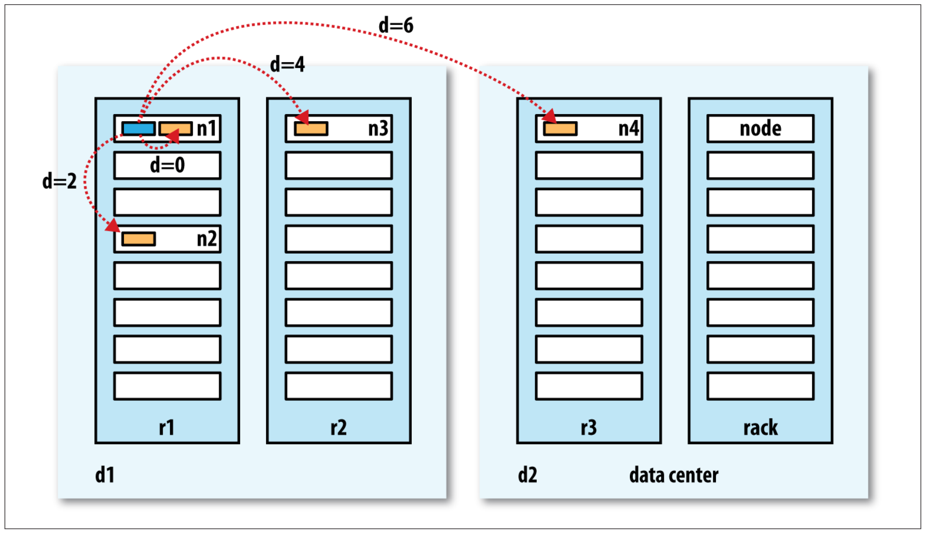


Figure 3-3. Network distance in Hadoop

Finally, it is important to realize that Hadoop cannot magically discover your network topology for you; it needs some help (we'll cover how to configure topology in "Network Topology" on page 286). By default, though, it assumes that the network is flat—a single-level hierarchy—or in other words, that all nodes are on a single rack in a single data center. For small clusters, this may actually be the case, and no further configuration is required.

8. At the time of this writing, Hadoop is not suited for running across data centers.

Anatomy of a File Write

Next we'll look at how files are written to HDFS. Although quite detailed, it is instructive to understand the data flow because it clarifies HDFS's coherency model.

We're going to consider the case of creating a new file, writing data to it, then closing the file. This is illustrated in Figure 3-4.

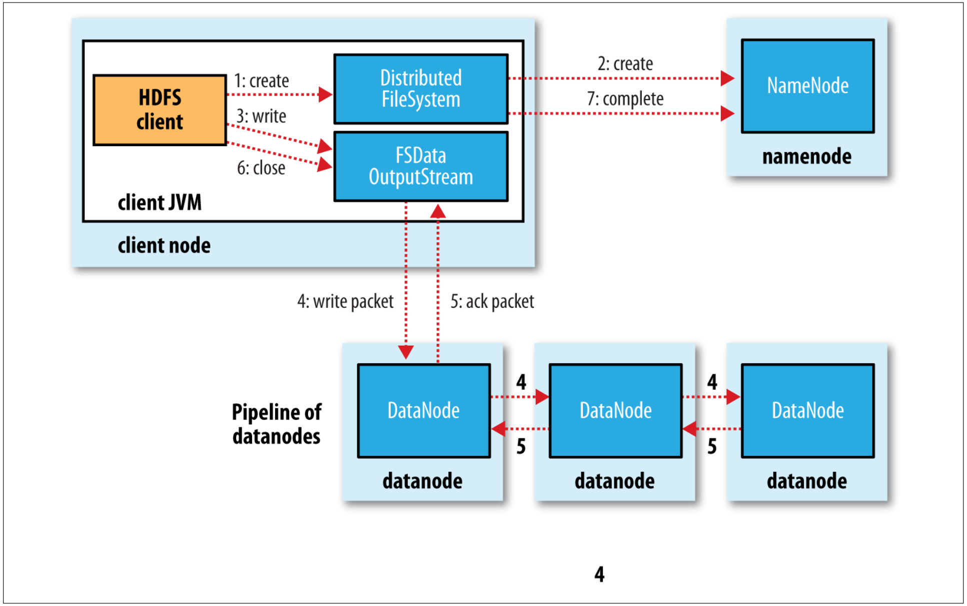


Figure 3-4. A client writing data to HDFS

The client creates the file by calling `create()` on `DistributedFileSystem` (step 1 in Figure 3-4). `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).

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 `DataStreamer` asks for block allocations), so it only has to wait for blocks to be minimally replicated before returning successfully.

Replica Placement

How does the namenode choose which datanodes to store replicas on? There's a trade-off 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.

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 3-5.

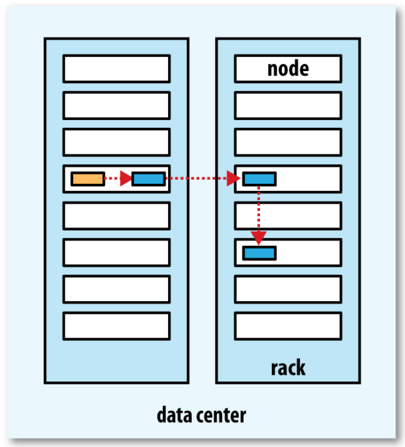


Figure 3-5. A typical replica pipeline

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).

Coherency Model

A coherency model for a filesystem describes the data visibility of reads and writes for a file. HDFS trades off some POSIX requirements for performance, so some operations may behave differently than you expect them to.

After creating a file, it is visible in the filesystem namespace, as expected:

```
Path p = new Path("p");
fs.create(p);
assertThat(fs.exists(p), is(true));
```