Architecture of Enterprise Applications 28
HDFS

Haopeng Chen

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
http://reins.se.sjtu.edu.cn/~chenhp
e-mail: chen-hp@sjtu.edu.cn

Contents and Objectives

- HDFS
  - Basic Concepts
  - Namenodes and Datanodes
  - Other Issues

- Objectives
  - 能够根据大尺寸数据文件的存储需求，设计使用HDFS存储的方案，设计并实现对HDFS中存储文件的读写操作

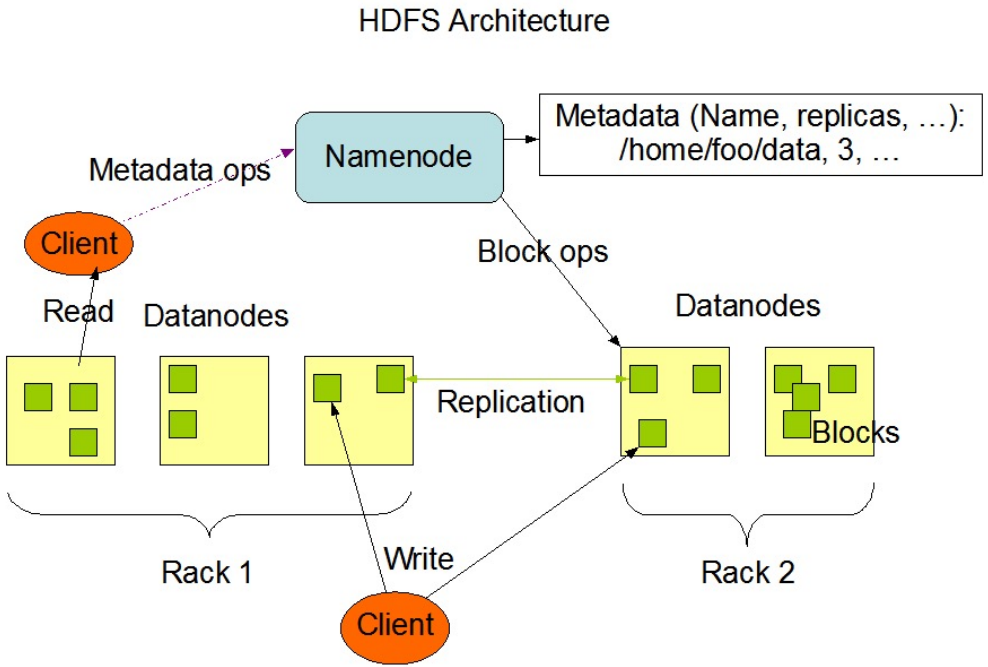
- HDFS
  - is a file system designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.

- Very large files
  - "Very large" in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size.
- Streaming data access
  - HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern.
- Commodity hardware
  - Hadoop doesn't require expensive, highly reliable hardware to run on.

- These are areas where HDFS is not a good fit today:
  - Low-latency data access
    - Remember, HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency.
  - Lots of small files
    - Since the namenode holds file system metadata in memory, the limit to the number of files in a file system is governed by the amount of memory on the namenode.
  - 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. There is no support for multiple writers, or for modifications at arbitrary offsets in the file.

NameNode and DataNodes

- HDFS has a master/slave architecture.



HDFS Concepts

- Blocks
  - A disk has a block size, which is the minimum amount of data that it can read or write.
  - It is a much larger unit—128 MB by default.

- HDFS has a master/slave architecture.
  - The master server that manages the file system namespace and regulates access to files by clients.
  - In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on.

Namenodes and Datanodes

- HDFS exposes a file system namespace and allows user data to be stored in files.
  - Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes.
  - The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes.
  - The DataNodes are responsible for serving read and write requests from the file system's clients.
  - The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.

- The existence of a single NameNode in a cluster greatly simplifies the architecture of the system.
  - The NameNode is the arbitrator and repository for all HDFS metadata.
  - The system is designed in such a way that user data never flows through the NameNode.

Namenodes and Datanodes

- The NameNode and DataNode are pieces of software designed to run on commodity machines.
  - These machines typically run a GNU/Linux operating system (OS).
  - HDFS is built using the Java language; any machine that supports Java can run the NameNode or the DataNode software. Usage of the highly portable Java language means that HDFS can be deployed on a wide range of machines.
  - A typical deployment has a dedicated machine that runs only the NameNode software. Each of the other machines in the cluster runs one instance of the DataNode software.
  - The architecture does not preclude running multiple DataNodes on the same machine but in a real deployment that is rarely the case.

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 supports user quotas and access permissions.
  - HDFS does not support hard links or soft links. However, the HDFS architecture does not preclude implementing these features.

The File System Namespace

- While HDFS follows naming convention of the FileSystem, some paths and names (e.g. /.reserved and .snapshot) are reserved.
  - Features such as transparent encryption and snapshot use reserved paths.

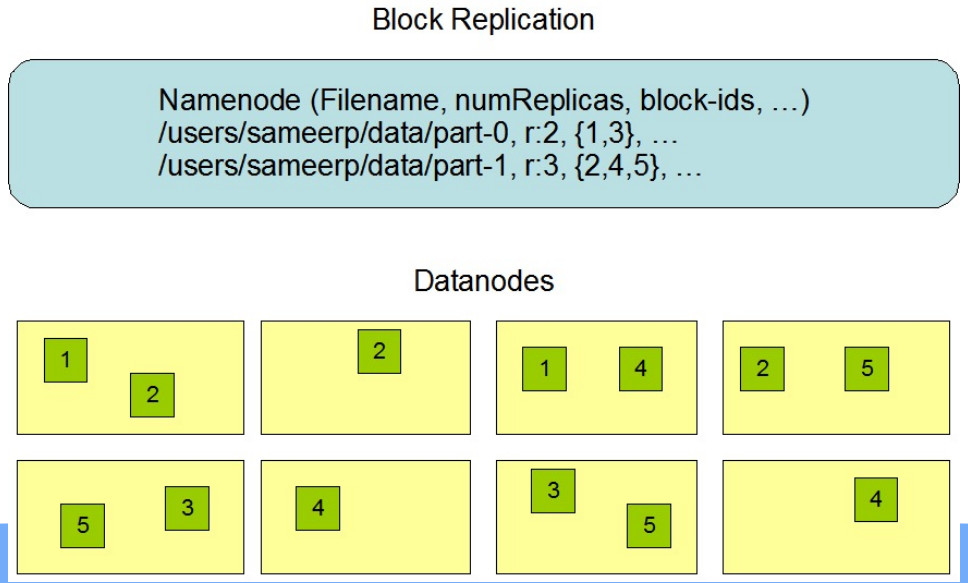
- The NameNode maintains the file system namespace.
  - Any change to the file system namespace or its properties is recorded by the NameNode.
  - An application can specify the number of replicas of a file that should be maintained by HDFS.
  - The number of copies of a file is called the replication 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.
  - The blocks of a file are replicated for fault tolerance.
  - The block size and replication factor are configurable per file.
- All blocks in a file except the last block are the same size,
  - while users can start a new block without filling out the last block to the configured block size after the support for variable length block was added to append and hsync.
- 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 (except for appends and truncates) and have strictly one writer at any time.

Data Replication

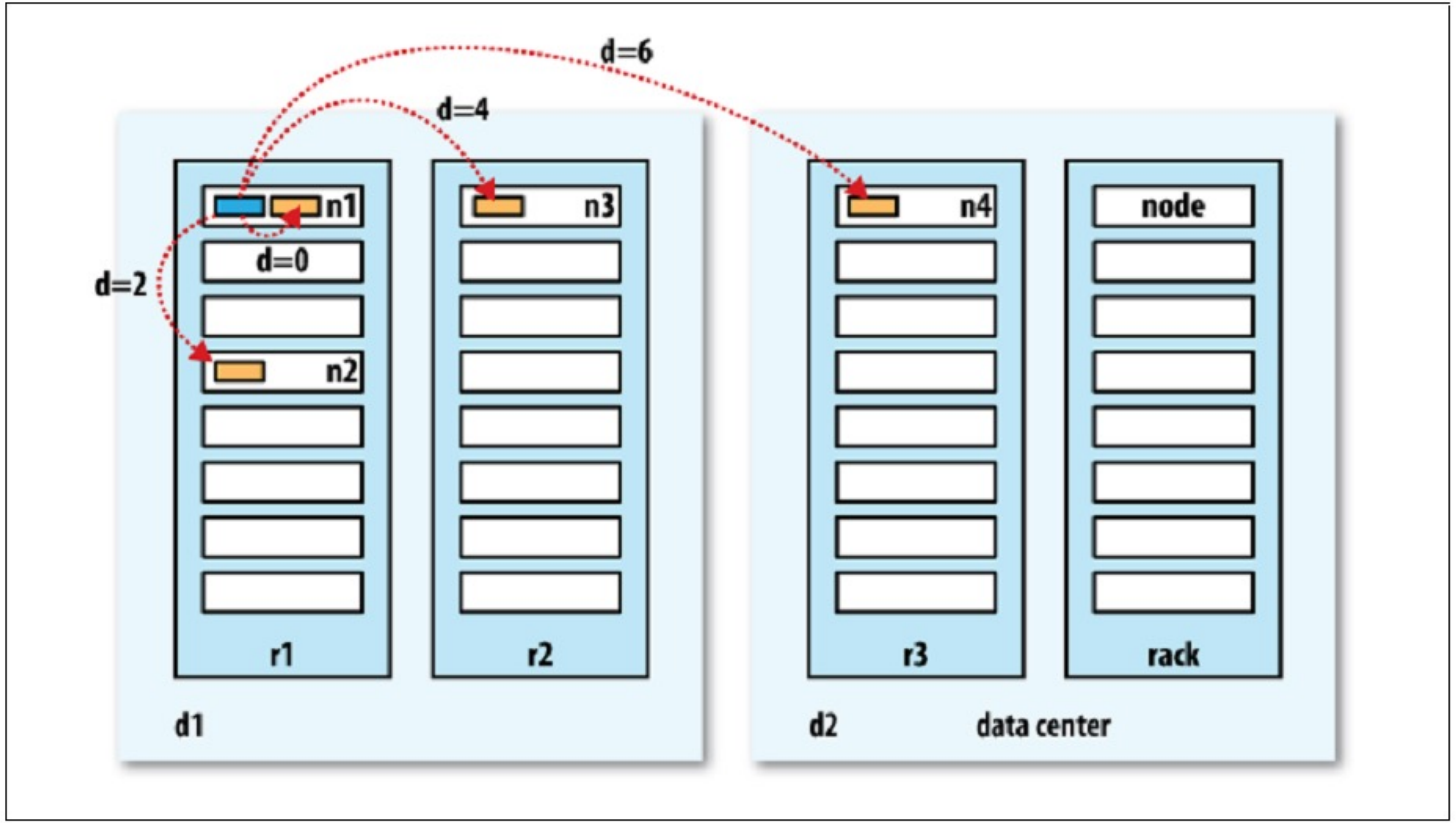
- The NameNode
  - makes all decisions regarding replication of blocks.
  - It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster.
  - Receipt of a Heartbeat implies that the DataNode is functioning properly.
  - A Blockreport contains a list of all blocks on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



Replica Placement: The First Baby Steps

- The placement of replicas is critical to HDFS reliability and performance.
  - The purpose of a rack-aware replica placement policy is to improve data reliability, availability, and network bandwidth utilization.
- 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 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.

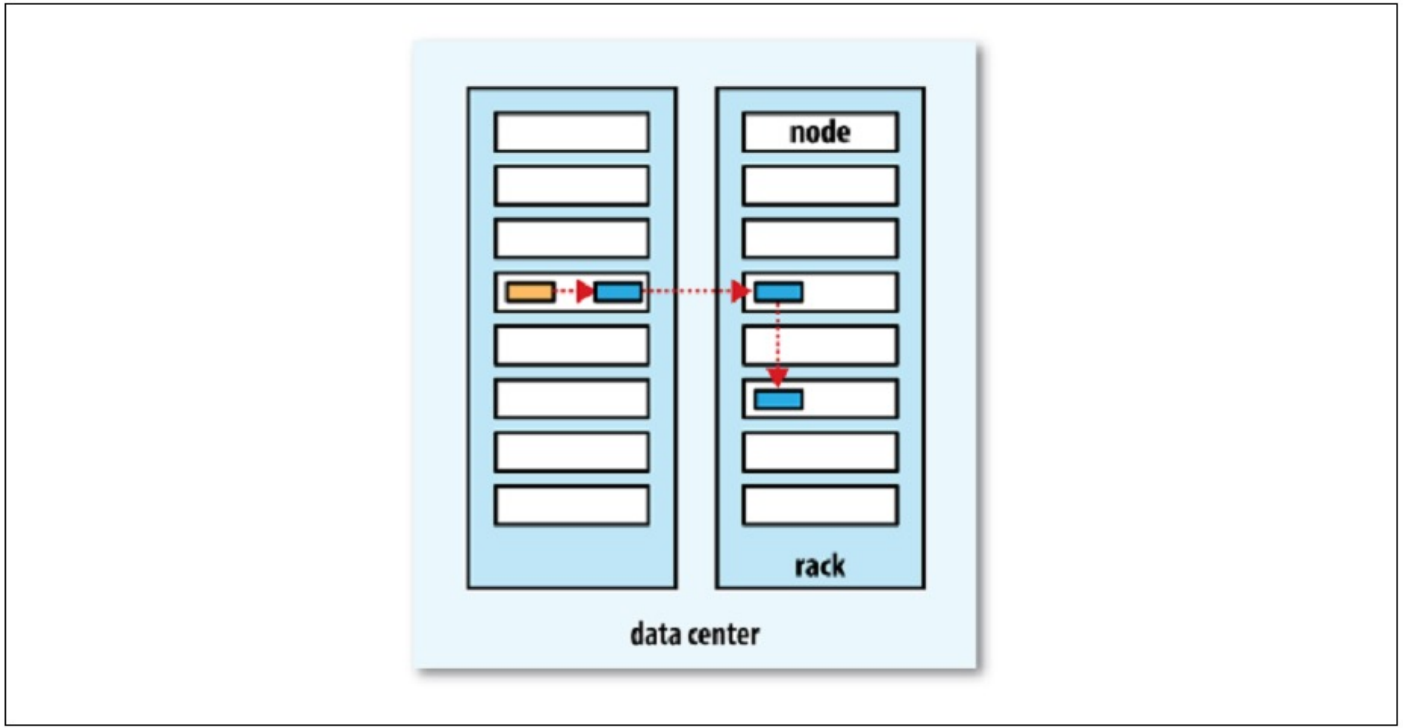
HDFS: network distance



Replica Placement: The First Baby Steps

- For the common case, when the replication factor is three,
  - HDFS's placement policy is to put one replica on the local machine if the writer is on a datanode, otherwise on a random datanode in the same rack as that of the writer,
  - another replica on a node in a different (remote) rack, and the last on a different node in the same remote rack.
- If the replication factor is greater than 3,
  - the placement of the 4th and following replicas are determined randomly while keeping the number of replicas per rack below the upper limit (which is basically (replicas - 1) / racks + 2).

HDFS: Replicas



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 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 Persistence of File System Metadata

- The NameNode keeps an image of the entire file system namespace and file Blockmap in memory.
  - When the NameNode starts up, or a checkpoint is triggered by a configurable threshold, 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.
  - The purpose of a checkpoint is to make sure that HDFS has a consistent view of the file system metadata by taking a snapshot of the file system metadata and saving it to FsImage.
  - A checkpoint can be triggered at a given time interval (dfs.namenode.checkpoint.period) expressed in seconds, or after a given number of filesystem transactions have accumulated (dfs.namenode.checkpoint.txns).
  - If both of these properties are set, the first threshold to be reached triggers a checkpoint.

The Persistence of File System Metadata

- 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.
  - The report is called 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 Client Protocol 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.

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**
  - 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.
  - The time-out to mark DataNodes dead is conservatively long (over 10 minutes by default) in order to avoid replication storm caused by state flapping of DataNodes.

- **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.
  - When a NameNode restarts, it selects the latest consistent FsImage and EditLog to use.
- **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.

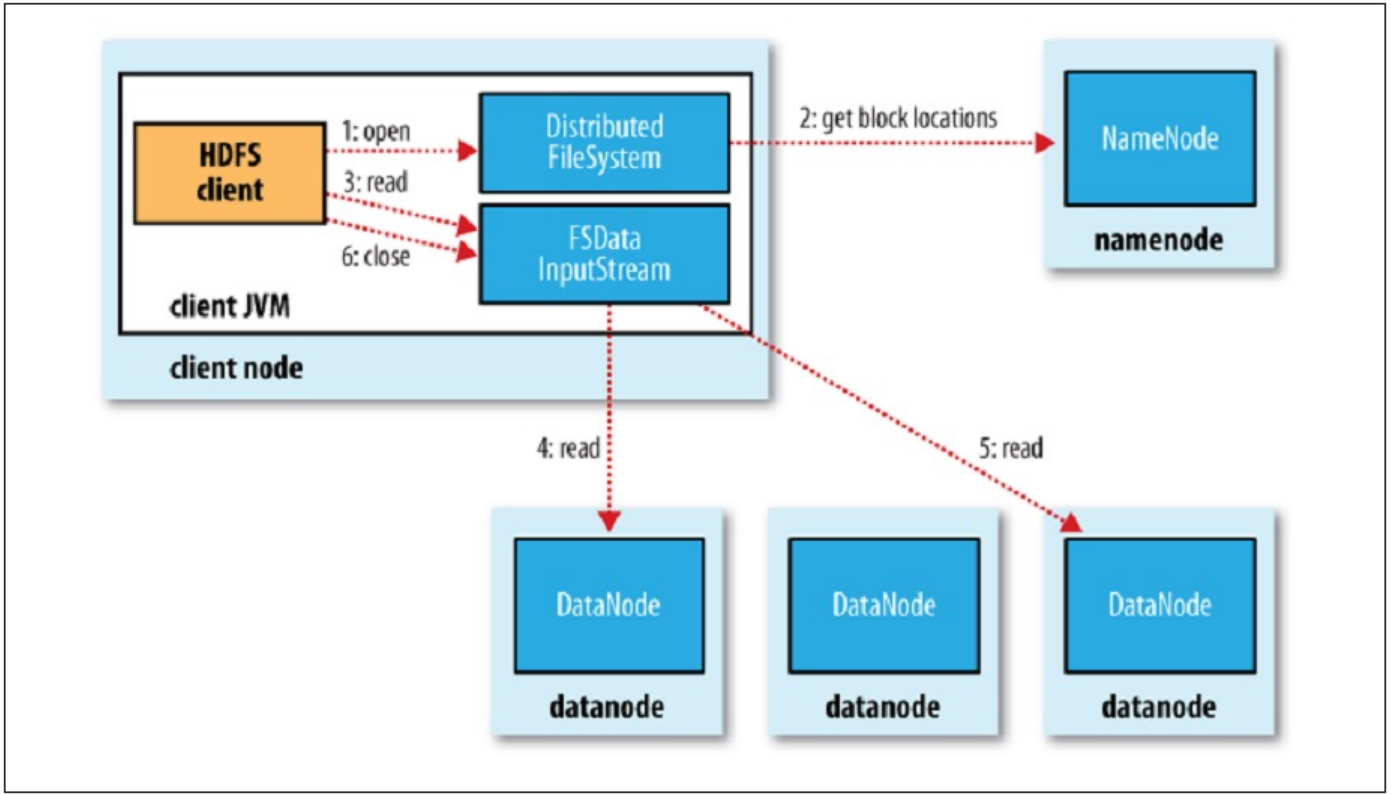
Data Organization

- **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 128 MB. Thus, an HDFS file is chopped up into 128 MB chunks, and if possible, each chunk will reside on a different DataNode.

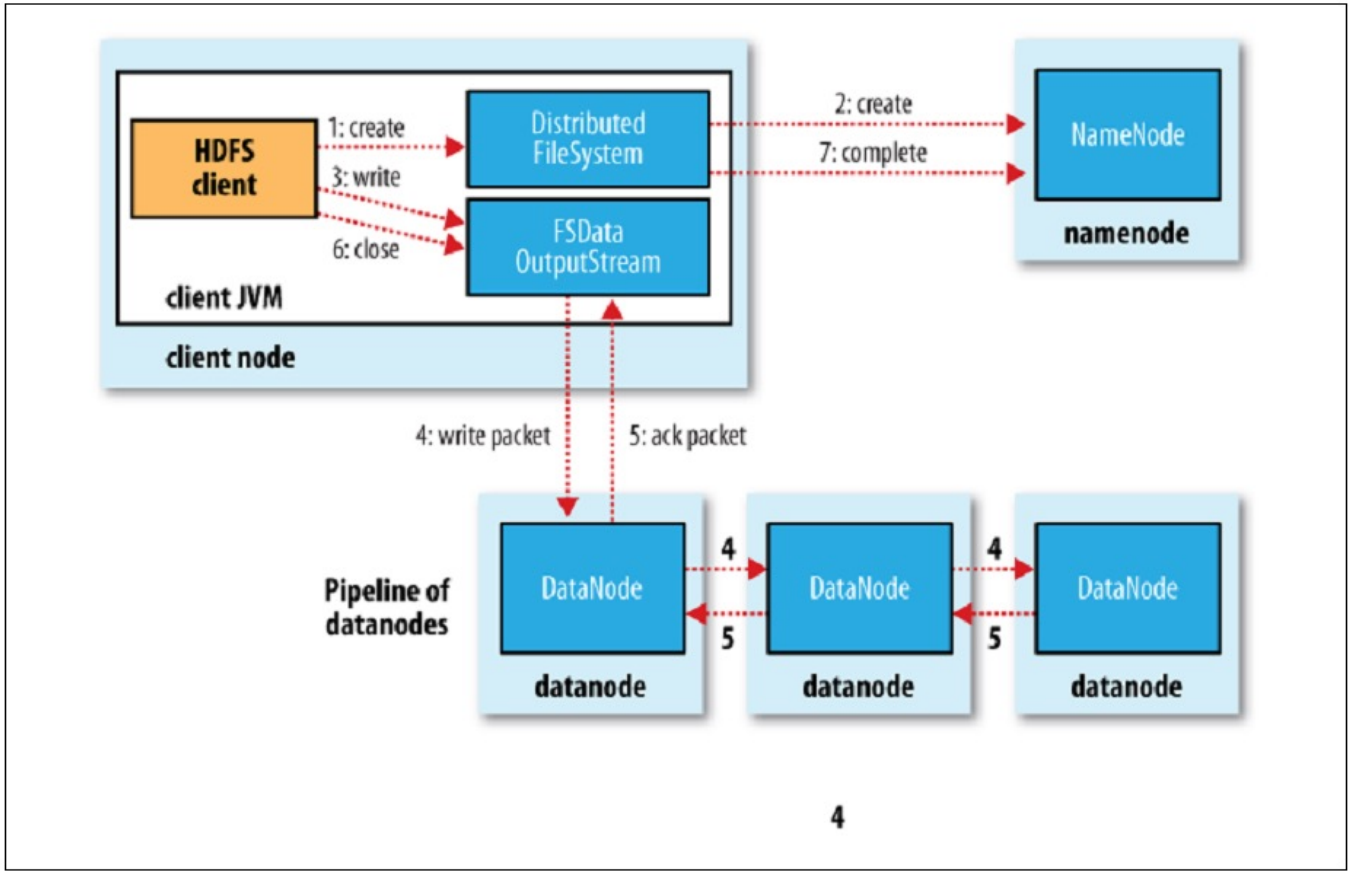
Data Organization

- **Replication Pipelining**
- When a client is writing data to an HDFS file with a replication factor of three,
  - the NameNode retrieves a list of DataNodes using a replication target choosing algorithm.
  - This list contains the DataNodes that will host a replica of that block.
  - The client then writes to the first DataNode.
  - The first DataNode starts receiving the data in portions, 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.

HDFS: Reading data



HDFS: Writing data



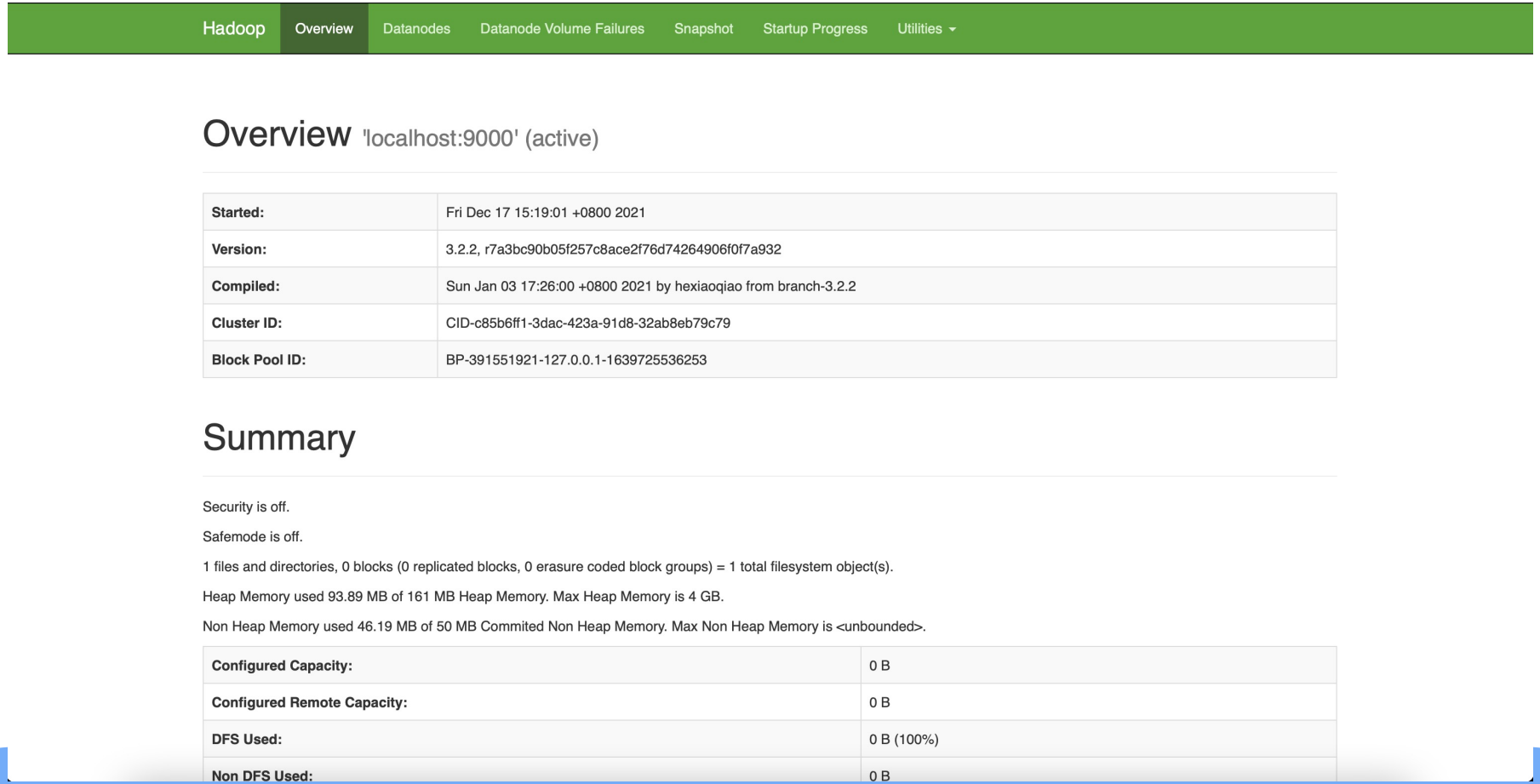
Accessibility

- HDFS can be accessed from applications in many different ways.
  - Natively, HDFS provides a FileSystem Java API for applications to use.
  - A C language wrapper for this Java API and REST API is also available.
  - In addition, an HTTP browser and can also be used to browse the files of an HDFS instance.
  - By using NFS gateway, HDFS can be mounted as part of the client's local file system.

- **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.

Accessibility

REliable, INtelligent & Scalable Systems



- **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	Command
Create a directory named /foodir	bin/hadoop dfs -mkdir /foodir
Remove a directory named /foodir	bin/hadoop fs -rm -R /foodir
View the contents of a file named /foodir/myfile.txt	bin/hadoop dfs -cat /foodir/myfile.txt

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

Space Reclamation

- **File Deletes and Undeletes**
  - If trash configuration is enabled, files removed by FS Shell is not immediately removed from HDFS.
  - Instead, HDFS moves it to a trash directory (each user has its own trash directory under /user/<username>/.Trash).
  - The file can be restored quickly as long as it remains in trash.
  - Most recent deleted files are moved to the current trash directory (/user/<username>/.Trash/Current), and in a configurable interval, HDFS creates checkpoints (under /user/<username>/.Trash/<date>) for files in current trash directory and deletes old checkpoints when they are expired.
  - 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.

Space Reclamation

- Following is an example which will show how the files are deleted from HDFS by FS Shell.
  - We created 2 files (test1 & test2) under the directory delete

```
$ hadoop fs -mkdir -p delete/test1
$ hadoop fs -mkdir -p delete/test2
$ hadoop fs -ls delete/
Found 2 items
drwxr-xr-x - hadoop hadoop 0 2015-05-08 12:39 delete/test1
drwxr-xr-x - hadoop hadoop 0 2015-05-08 12:40 delete/test2
```

Space Reclamation

- We are going to remove the file test1. The comment below shows that the file has been moved to Trash directory.

```
$ hadoop fs -rm -r delete/test1
Moved: hdfs://localhost:8020/user/hadoop/delete/test1 to trash at:
hdfs://localhost:8020/user/hadoop/.Trash/Current
```

- now we are going to remove the file with skipTrash option, which will not send the file to Trash. It will be completely removed from HDFS.

```
$ hadoop fs -rm -r -skipTrash delete/test2
Deleted delete/test2
```

Space Reclamation

- We can see now that the Trash directory contains only file test1.

```
$ hadoop fs -ls .Trash/Current/user/hadoop/delete/
Found 1 items\
drwxr-xr-x - hadoop hadoop 0 2015-05-08 12:39
.Trash/Current/user/hadoop/delete/test1
```

- So file test1 goes to Trash and file test2 is deleted permanently.

Space Reclamation

- **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.

Execution

- Format the filesystem:
- $ bin/hdfs namenode -format

- Start NameNode daemon and DataNode daemon:
- $ sbin/start-dfs.sh

- Browse the web interface for the NameNode; by default it is available at:
  - NameNode - http://localhost:9870/

Samples

- TestHDFS.java

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class TestHDFS {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();

        conf.set("dfs.defaultFS", "hdfs://hadoop:9000");
        FileSystem fs = FileSystem.get(conf);
        fs.create(new Path("./test/helloByJava"));
        fs.close();
    }
}
```

Samples

- HDFSApp.java

```
public class HDFSApp {
    public static final String HDFS_PATH = "hdfs://hadoop:9000";
    //文件系统
    FileSystem fileSystem = null;
    //配置类
    Configuration configuration = null;

    //Before适用于类加载之前
    @Test
    /**
     * 创建HDFS系统
     */
    public void mkdir() throws Exception {
        fileSystem.mkdirs(new Path("./output"));
    }
```

Samples

- HDFSApp.java

```
@Test
public void create() throws Exception {
    FSDataOutputStream output = fileSystem.create(new Path("./output/a.txt"));
    output.write("hello hadoop".getBytes());
    output.flush();
    output.close();
}

@Test
/**
 * 查看HDFS文件上的内容
 */
public void cat() throws Exception {
    FSDataInputStream in = fileSystem.open(new Path("./output/b.txt"));
    IOUtils.copyBytes(in, System.out, 1024);
    in.close();
}
```

Samples

- HDFSApp.java

```
@Before
public void setUp() throws Exception {
    System.out.printf("HDFSapp.setup");
    configuration = new Configuration();
    configuration.set("dfs.defaultFS", "hdfs://hadoop:9000");
    fileSystem = FileSystem.get(configuration);
}

//关闭资源用的这个
@After
public void tearDown() throws Exception {
    //释放资源
    configuration = null;
    fileSystem = null;
    System.out.printf("HDFSAPP.tearDown");
}
```

- HDFSApp.java

```
/**
 * 重命名文件
 */
@Test
public void rename() throws Exception {
    Path oldPath = new Path("./output/a.txt");
    Path newPath = new Path("./output/b.txt");
    fileSystem.rename(oldPath, newPath);
}
```

- HDFSApp.java

```
/**
 * 上传一个文件
 *
 * @throws Exception
 */
@Test
public void copyFromLocalFile() throws Exception {
    Path localPath = new Path("./test/helloByJava");
    Path hdfsPath = new Path("./output/helloByJava");
    fileSystem.copyFromLocalFile(localPath, hdfsPath);
}
```

- HDFSApp.java

```
/**
 * 上传一个大文件
 * @throws Exception
 */
@Test
public void copyFromLocalBigFile() throws Exception {
    InputStream in = new BufferedInputStream(
            new FileInputStream(
                    new File("./resources/hadoop-hdfs-3.2.1.jar")));
    FSDataOutputStream output = fileSystem.create(
                    new Path("./output/hadoop-hdfs-3.2.1.jar"),
            new Progressable() {
                public void progress() {
                    System.out.print(".");  //带进度提醒信息
                }
            });
    IOUtils.copyBytes(in, output, 4096);
}
```

Samples

- HDFSApp.java

```
/**
 * 下载HDFS文件
 */
@Test
public void copyTOLocalFile() throws Exception{
    Path localPath = new Path("./test/b.txt");
    Path hdfsPath = new Path("./output/b.txt");
    fileSystem.copyToLocalFile(false,hdfsPath,localPath,true);
}
```

Samples

- HDFSApp.java

```
@Test
public void listFiles() throws Exception {
    FileStatus[] fileStatuses = fileSystem.listStatus(new Path("./output"));

    for (FileStatus fileStatus : fileStatuses) {
        String isDir = fileStatus.isDirectory() ? "文件夹" : "文件";
        //副本
        short replication = fileStatus.getReplication();
        //大小
        long len = fileStatus.getLen();
        //路径
        String path = fileStatus.getPath().toString();

        System.out.println(isDir + "\t" + replication + "\t" + len + "\t" + path);
    }
}
```

Samples

- HDFSApp.java

```
    @Test
        public void delete() throws Exception{
            fileSystem.delete(new Path("./test/"),true);
        }

    }
```

Samples

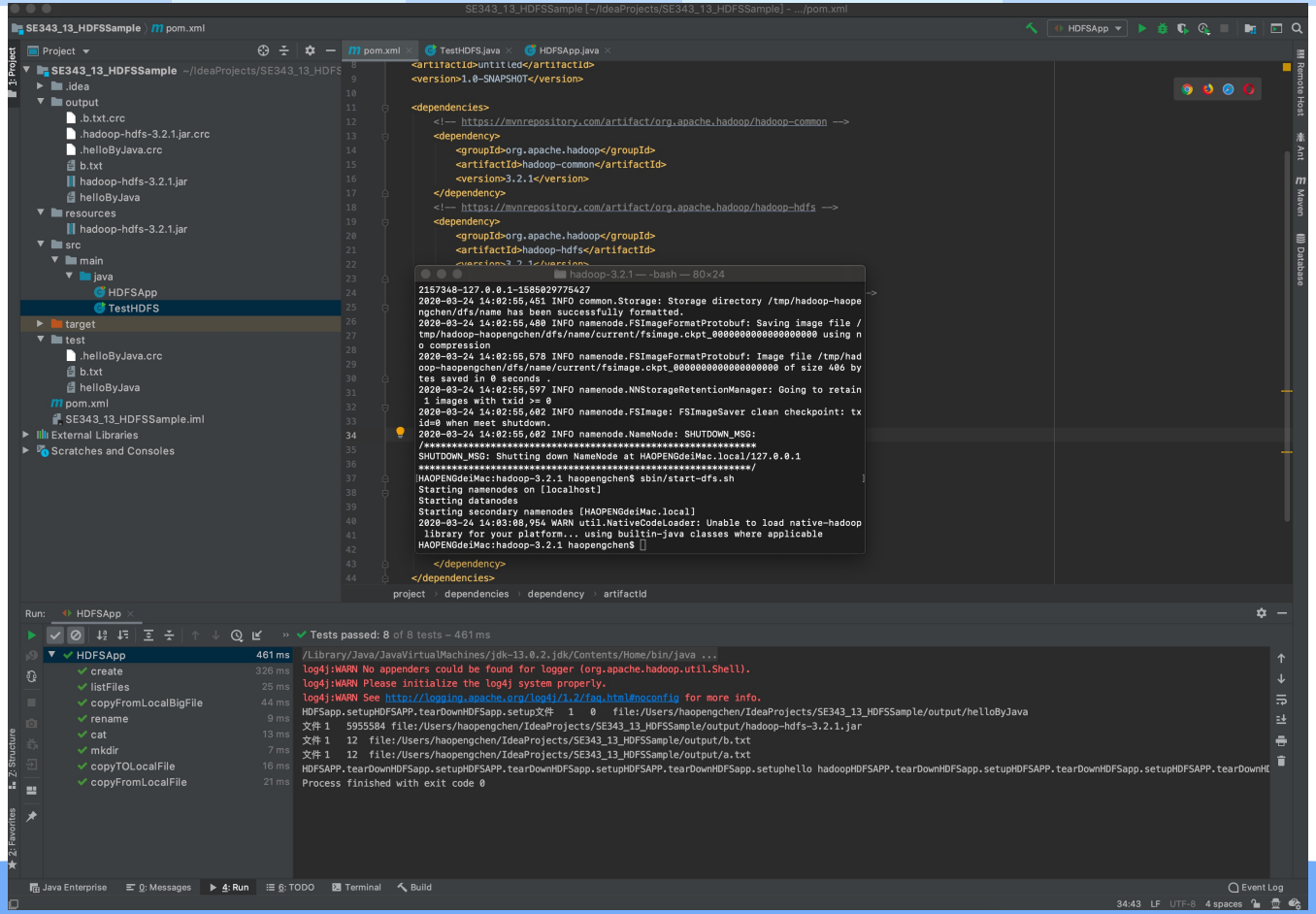
- pom.xml

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>3.2.1</version>
</dependency>
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-hdfs</artifactId>
    <version>3.2.1</version>
</dependency>
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-hdfs-client</artifactId>
    <version>3.2.1</version>
    <scope>provided</scope>
</dependency>
```

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13</version>
    <scope>compile</scope>
</dependency>
```

Samples

REin
REliable, INtelligent & Scalable Systems



References

- HDFS Architecture
  - https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html
- Hadoop: The Definitive Guide
  - By Tom White
  - O'Reilly Publishing
- Hadoop入门IntelliJ编写Maven创建HDFS文件
  - https://blog.csdn.net/qq_39622065/article/details/86600322
- IDEA 创建HDFS项目 JAVA api
  - https://www.cnblogs.com/chenligeng/p/9348120.html

Thank You!