Week 3

Monday, September 28, 2020 9:09 AM

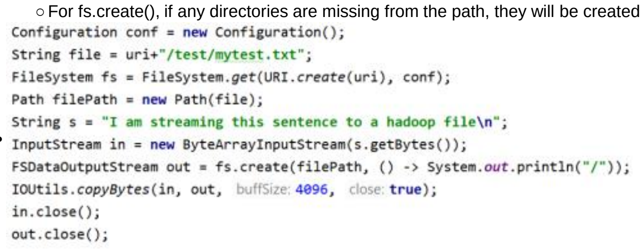
LECTURE NOTES

Hadoop CLI

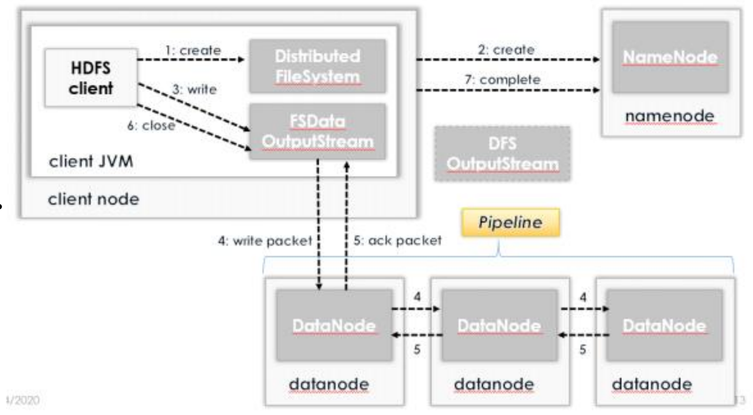
- Simple interface, similar to Linux (like hadoop fs or hdfs dfs)
  - hdfs dfs only works for HDFS
  - hadoop fs more generic, includes HDFS but also works with other file systems, like local FS, WebHDFS, View, FTP, Azure, etc
    - Hadoop's filesystem is an abstract notion of a filesystem, which is why it works on other systems
    - It will use a URI to pick the right filesystem, ie: hdfs://localhost/
    - If you want to use MapReduce, it's best to use HDFS as it has data locality optimization
  - Commands can get deprecated
  - hadoop fs -help will give more details

NAVIGATION	COMMANDS
hadoop fs -ls [-R] {path}	returns [recursive] listing of files & directories
hadoop fs -find {path} -name {file or dir name}	returns the complete path of a file or directory
hadoop fs -du -v - h {path}	returns the size of files/directories under the path (du stands for directory usage). Shows size of file plus disk space used for all replicas
hadoop fs -stat "type:%F perm:%a %u:%g size:%b mtime:%y atime:%x name:%n" {path}	shows/echos stats (type, perm, size, etc) on files/directory
hadoop fs -count -h -v {path}	counts files & directories
DIRECTORY	**COMMANDS**
hadoop fs -mkdir [-p]	make directories [and parents]
hadoop fs -rmdir	deletes directory
hadoop fs -rm -r	removes directories recursively
FILE	**COMMANDS**
hadoop fs -cp {origin} {destination}	copy files within the file system
hadoop fs -copyFromLocal {origin} {destination}	copy files from local file system. Local file system = client and local drive
hadoop fs -copyToLocal {origin} {destination}	copy files to local file system
hadoop fs -moveFromLocal {origin} {destination}	move files from a local system and delete the source
hadoop fs -moveToLocal {origin} {destination}	move files to the local system and delete the source
hadoop fs -appendToFile {origin file} {file to be appended to}	appends to existing file
hadoop fs -rm {path}	delete file
hadoop fs -cat {path}	copy file to standard output
hadoop fs -head {path}	top of file/first kb of file to stdout
headoop fs -tail {path}	bottom/last kb of file to stdout
hadoop fs -checksum {path}	returns checksum
ACCESS	**CONTROL**
hadoop fs -chmod {path}	change permissions
hadoop fs -chown {path}	change owner

Writing to Hadoop with Java:

- Need `java.net.URI` for HDFS root and `org.apache.hadoop.fs.FileSystem`
  - FileSystem is an abstract class, and HDFS is an implementation of it. Other examples are Local File System, Amazon S3, Azure, FTP, etc
  - Configuration stores the client or server's configuration (that is, values for Hadoop's configuration that we set)
  - For fs.create(), if any directories are missing from the path, they will be created (as opposed to throwing an error). You can alternatively use the mkdirs method
- 

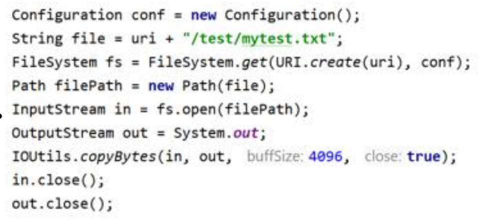
```java
Configuration conf = new Configuration();
String file = uri+"/test/mytest.txt";
FileSystem fs = FileSystem.get(URI.create(uri), conf);
Path filePath = new Path(file);
String s = "I am streaming this sentence to a hadoop file\n";
InputStream in = new ByteArrayInputStream(s.getBytes());
FSDataOutputStream out = fs.create(filePath, () -> System.out.println("/"));
IOUtils.copyBytes(in, out,  buffSize: 4096,  close: true);
in.close();
out.close();
```

- 

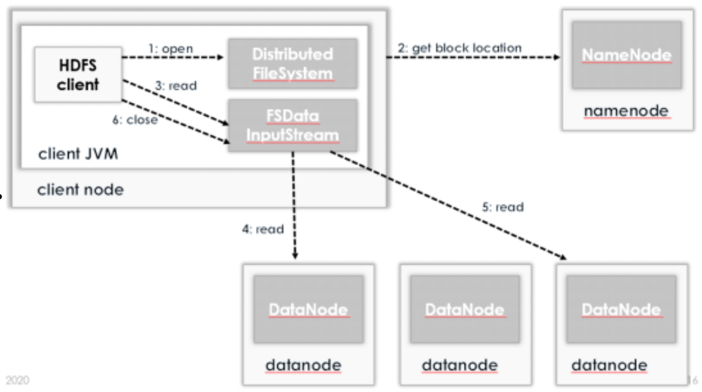
- To start: HDFS client has our code
  1. create a file, call Create() on the DFS
  2. DFS sends a call to create a new file with NO blocks associated to it yet. Namenode does checks (makes sure file doesn't exist already, client has permission to create, etc). If successful, the namenode makes a record of the new file (if fail, throws an IOException). DFS returns the output stream for client to write to

3. Client writes data to output stream
4. DFS splits the data into packages, which are written to the Data Queue. DataStreamer manages the queue which gets the namenode to allocate blocks for the file by picking datanodes to store replicas. Those datanodes create a pipleine. DataSteamer sends it to the first datanode, which writes then passes along the rest to the second (and so on). Datanodes return an acknowledgement to the DataStreamer when they finish writing (called ack queue)
   i. If there is failure, pipeline is closed and any packets that were written (in ack queue) are added back to the data queue. Failed datanode is removed and a new pipeline is made with working datanodes.
5. Acknowledgement package is returned
6. Stream is closed
7. Send confirmation to namenode, namenode now knows where the files are saved

Reading from Hadoop with Java:

- Again need `java.net.URI` for HDFS root and `org.apache.hadoop.fs.FileSystem`, but also need `org.apache.hadoop.fs.FSDataInputStream` and `org.apache.hadoop.io.IOUtils.copyBytes` (copies stream from one to another)
- 

```
Configuration conf = new Configuration();
String file = uri + "/test/mytest.txt";
FileSystem fs = FileSystem.get(URI.create(uri), conf);
Path filePath = new Path(file);
InputStream in = fs.open(filePath);
OutputStream out = System.out;
IOUtils.copyBytes(in, out, buffSize: 4096, close: true);
in.close();
out.close();
```

- 

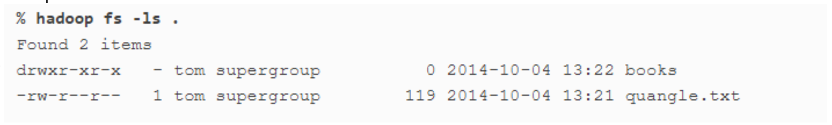
- HDFS has our code
  1. Open session with file system
  2. FileSystem contacts the namenode to find blocks, namenode returns with datanode locations of the blocks (sorted by proximity to client)
  3. Client calls read on the DFS InputStream object, which has those datanode addresses
  4. InputStream opens, block is read from the datanode, and then closed on the closest datanode
     i. If there is an error while reading, it will try to find whatever datanode that also has that info and is closest (see textbook notes). It will also remember which datanode failed so it doesn't retry it later
     ii. Checksum checking is also done to ensure no corruption during transfer
  5. Done again with the next node, each read is open and closed on the individual data node, but to the client looks like a continuous stream
  6. Read is returned & stream is closed

TEXTBOOK NOTES

Setting up HDFS

- Properties that "deserve further explanation":
  - fs.defaulFS set to hdfs://localhost/
    - Sets the default filesystem for Hadoop
    - Uses a hdfs URI to configure hadoop to use HDFS by default (as there are other options as described in lecture)
    - HDFS daemons will use this property to decide host and port for HDFS namenode (default is 8020)
  - dfs.replication set to 1
    - We didn't do this in class, but we discussed last week how the default is 3. In the textbook, they default it to 1 because they only have 1 datanode
    - If they left it at 3, they would get warnings about not being able to fully replicate

Commands:

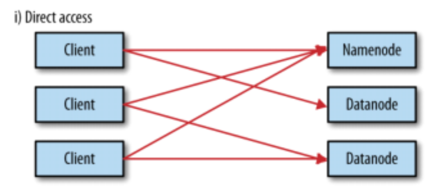
- md5 --> checks hashvalue of file (to compare that a copied file from or to local matches file in HDFS)
- ls output:
- 

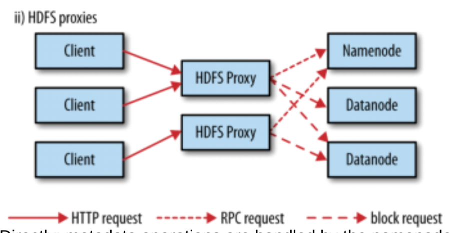
```
% hadoop fs -ls .
Found 2 items
drwxr-xr-x   - tom supergroup          0 2014-10-04 13:22 books
-rw-r--r--   1 tom supergroup        119 2014-10-04 13:21 quangle.txt
```

- the second column (- and 1) describes how replicated the file is. The first line is a directory, and those are not replicated as they are considered metadata
- More about file permissions…
  - In HDFS, the execute (x) permission is ignored because you cannot execute a file on HDFS.
  - By default, Hadoop runs with security disabled/clients are not authenticated
  - Can change this in `dfs.permissions.enabled` property
- Superuser is the identity of a namenode process, and permissions checks are not done on it

Hadoop Interfaces

- Hadoop is written in Java, but other languages can interact with it by using HTTP directly or indirectly



- 

- Directly: metadata operations are handled by the namenode. Read & write operations are sent to the namenode, which then does an HTTP redirect with the datanode
- Indirectly/proxies: Clients never interact with the namenode or datanode directly, which is more secure/allows more firewall setup

- For C, there is a library (libhdfs) that is similar to Java's FileSystem or libwebhdfs that uses WebHDFS. It's similar to Java's implementation but normally a bit behind
- NFS and FUSE are also supported

Querying Hadoop

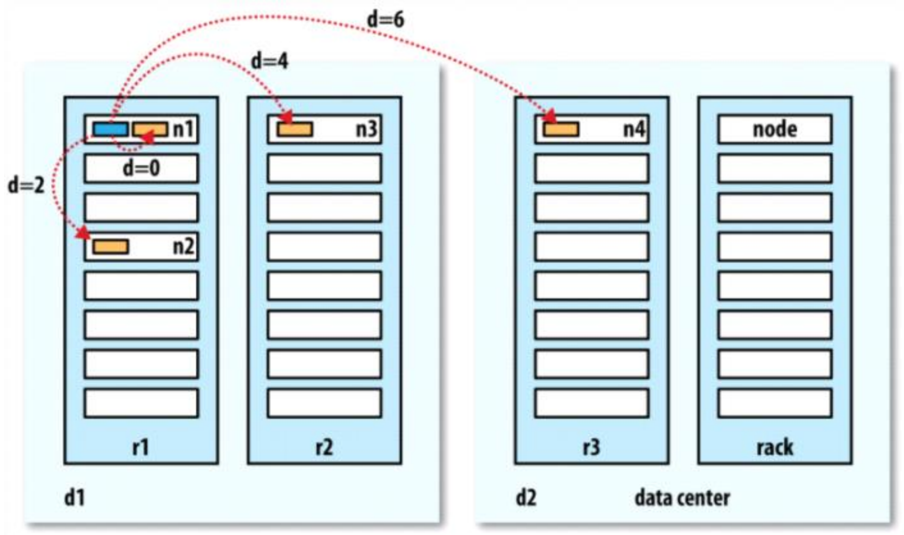
- To get file metadata: FileStatus class and the getFileStatus() method
- You can also do exists() to see if the file exists
- Can list files with FileSystem class and listStatus(), which returns an array of FileStatus objects
- File patterns: FileStatu's globStatus() returns an array of FileStatus objects who have a path that matches the given pattern, sorted by path
  - Globs supported:

*	zero or more
?	single char
[ab]	matches single char in set (ie, either a or b)
[^ab]	matches a single char that is not in the set
[a-b]	char range
[^a-b]	negated char range
{a,b}	alternation: matches either expression
\c	escaped char: matches char c when it is ametachar

- Globs might not work for what you're looking for, so you can use PathFilter to make your own matching
- You can use delete() from the FileSystem class to delete files or directories. It allows for recursive or not with a bool

Datanode Distances

- Instead of using bandwidth as a measure of distance, distance is determined by file system levels
  - Blue to N1: same data centre, same rack, same node, so 0 distance
  - Blue to N2: same data centre, same rack, but different nod; distance is 2
  - Blue to N3: same data centre, different rack; distance is 4
  - Blue to N4: different data centre; distance is 6



- Hadoop assumes your network is flat (all nodes are in one rack in one data centre) but this can be taught to Hadoop if that's not the case

Choosing Where to Store Replicas (discussed in Week 2)

- Default strategy is to place first replica on the same node as the client
- Second replica is chosen to be on a different rack (any node, it's random)
- Third is chosen to be on the same rack as the second, but a different node
- Any further replicas are stored at random, but it does try to evenly spread between racks

Coherency (Cohesion) Model

- Coherency model describes data visibility of reads and writes for a file
- After creating a file, it can be seen in the namespace, but the data written to it/content might not be visible until the next block is written (whatever block is currently being written on cannot be read)
- To force a way to flush/close all blocks, the FSDataOutputStream has a hflush() method to make it visible. This only guarantees that it's in the datanodes memory (not written to disk). For a more

durable option, you can use hsync()
- If you don’t use either of these methods, you ca lose blocks of data in the event of a system failure
- You should always do a flush (at least) after writing

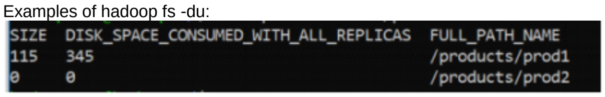
Parallel Copying
- distcp copies data to and from Hadoop in parallel
- Ie, instead of `hadoop fs -cp f1 f2` you could do `hadoop distcp f1 f2`
  - This also works on directories! Second directory will be created if not already
- distcp is a MapReduce job, using up to 20 maps
- Common to be used for transferring data between two HDFS clusters (ie, backups)
- Beware of copying, as it may cause your cluster to be imbalanced (that is, the redundancies are not properly shared across nodes & racks). There is a balancer tool to help with this

LAB NOTES

DEV
- hd-master
  - Port 2221
  - 192.168.56.5
- hd-data-01
  - 192.168.56.6
- hd-data-02
  - 192.168.56.7
- ROUTER
  - 10.31.152.103
  - Specific to us but others can login

PROD
- Has the same ports and IP addresses for the clients, but the router IP is different
  - 10.31.152.42

Examples of hadoop fs -du:



```
SIZE  DISK_SPACE_CONSUMED_WITH_ALL_REPLICAS  FULL_PATH_NAME
115   345                                    /products/prod1
0     0                                      /products/prod2
```