Big Data Management Assignment 1

郑元嘉 **1800920541** 靳立晨 **1600012459**

Task 2 - Create many small files in HDFS, analyze block size

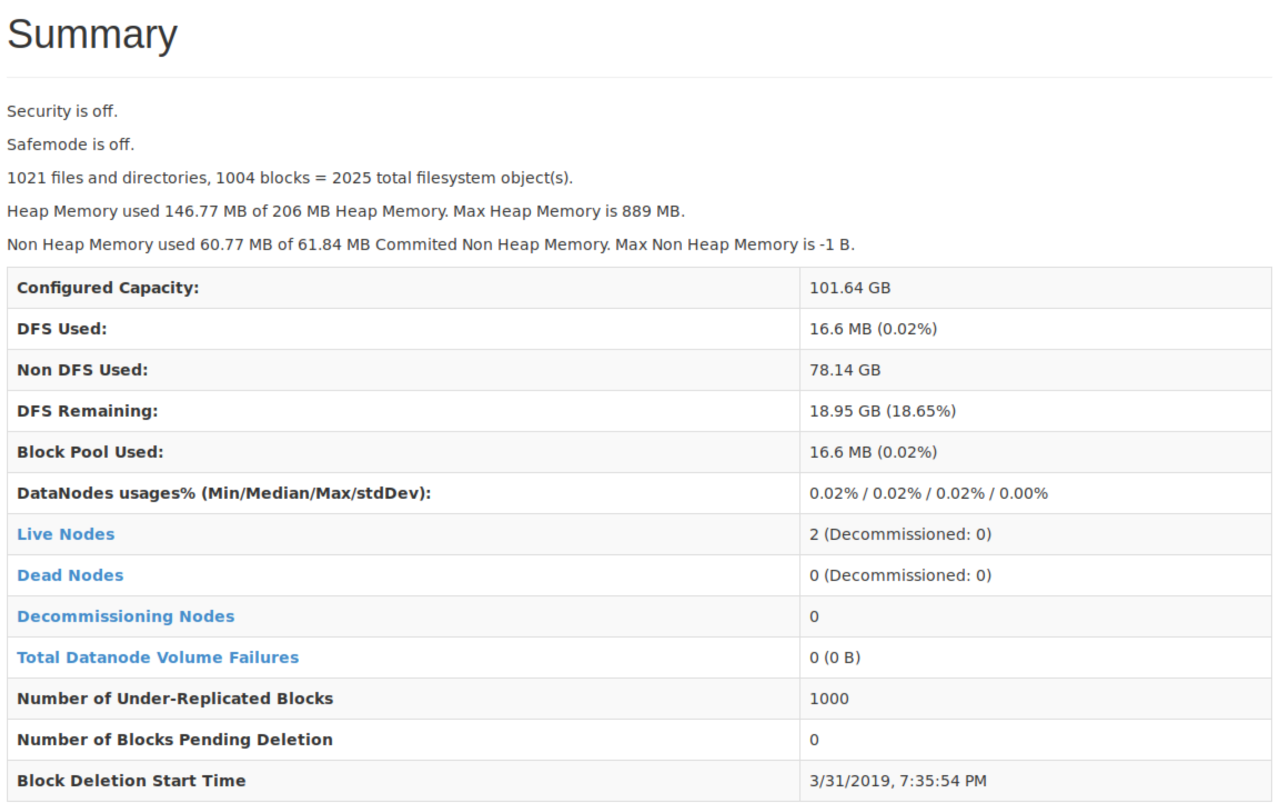
Start HDFS first. Then we create 1000 files with Java API.

The code is as follows:

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.fs.FSDataOutputStream;
import java.io.IOException;
public class thousandfiles {
    public static void main(String[] args){
        try{
            for(int i = 0;i < 1000;++i)
            {
            String fileName = String.valueOf(i);
            Configuration conf = new Configuration();
            conf.set("fs.defaultFS", "hdfs://Master:9000");
            conf.set("fs.hdfs.impl", "org.apache.hadoop.hdfs.DistributedFileSystem");
            FileSystem fs = FileSystem.get(conf);
            Path file = new Path("hw1_2" + fileName);
            if(fs.exists(file)){
                System.out.println("File exists");
            }else{
                createFile(fs,file);
            }
            fs.close();
            }

        }catch (Exception e){
            e.printStackTrace();
        }
    }
    public static void createFile(FileSystem fs, Path file) throws IOException{
        byte[] buff = "Death of trivials".getBytes();
        FSDataOutputStream os =fs.create(file);
        os.write(buff,0,buff.length);
        System.out.println("Create:"+file.getName());
        os.close();
    }
}
```

For each file, only a short String **"Death of trivials"** is written in each created file.

Then, we visit localhost:50070 on the Master node to check the state of HDFS.



Summary

Security is off.

Safemode is off.

1021 files and directories, 1004 blocks = 2025 total filesystem object(s).

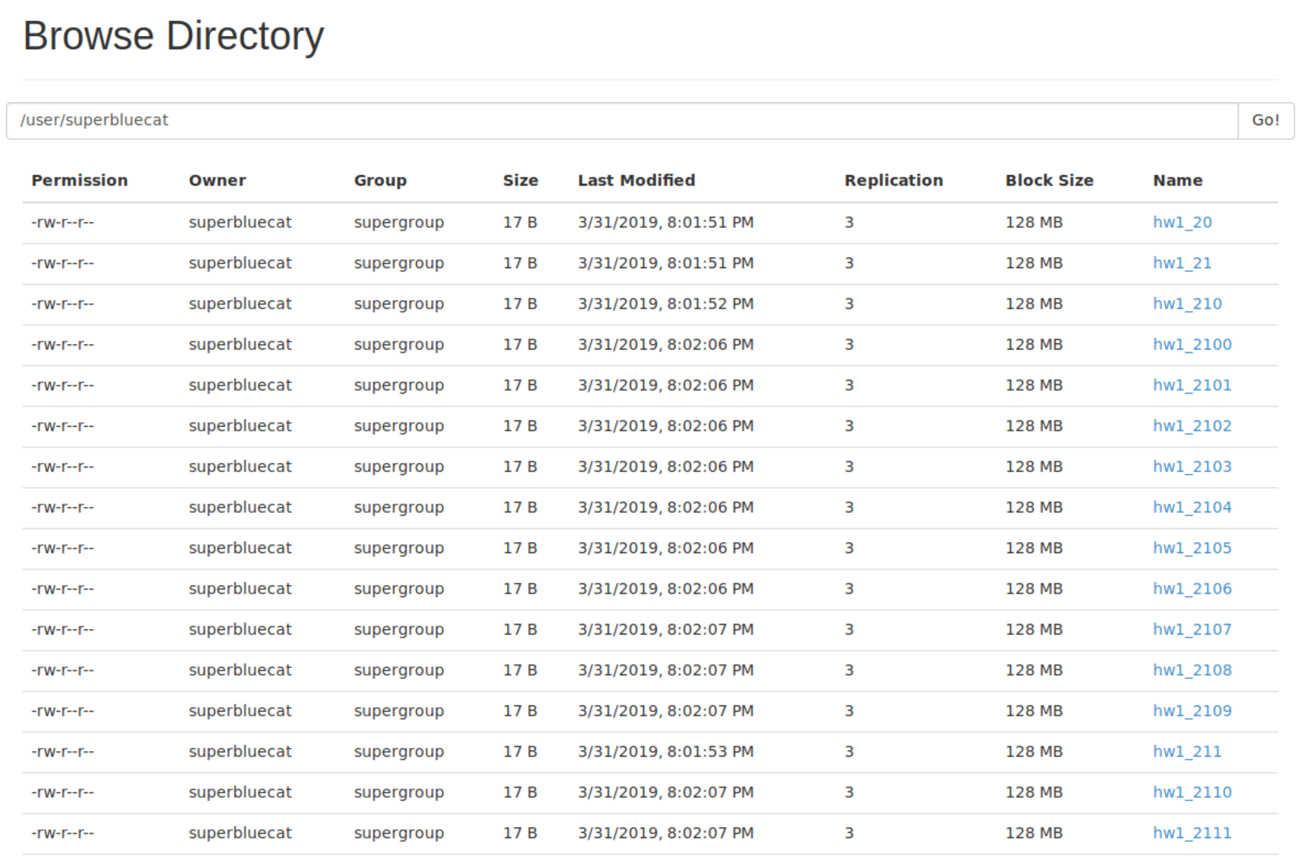
Heap Memory used 146.77 MB of 206 MB Heap Memory. Max Heap Memory is 889 MB.

Non Heap Memory used 60.77 MB of 61.84 MB Commited Non Heap Memory. Max Non Heap Memory is -1 B.

Configured Capacity:	101.64 GB
DFS Used:	16.6 MB (0.02%)
Non DFS Used:	78.14 GB
DFS Remaining:	18.95 GB (18.65%)
Block Pool Used:	16.6 MB (0.02%)
DataNodes usages% (Min/Median/Max/stdDev):	0.02% / 0.02% / 0.02% / 0.00%
Live Nodes	2 (Decommissioned: 0)
Dead Nodes	0 (Decommissioned: 0)
Decommissioning Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	1000
Number of Blocks Pending Deletion	0
Block Deletion Start Time	3/31/2019, 7:35:54 PM

1004 blocks show that seemingly each file is assigned a different block.

We continue to take a closer look. Browse the file system from utilities to see files' block sizes.



Browse Directory

/user/superbluecat Go!

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	superbluecat	supergroup	17 B	3/31/2019, 8:01:51 PM	3	128 MB	hw1_20
-rw-r--r--	superbluecat	supergroup	17 B	3/31/2019, 8:01:51 PM	3	128 MB	hw1_21
-rw-r--r--	superbluecat	supergroup	17 B	3/31/2019, 8:01:52 PM	3	128 MB	hw1_210
-rw-r--r--	superbluecat	supergroup	17 B	3/31/2019, 8:02:06 PM	3	128 MB	hw1_2100
-rw-r--r--	superbluecat	supergroup	17 B	3/31/2019, 8:02:06 PM	3	128 MB	hw1_2101
-rw-r--r--	superbluecat	supergroup	17 B	3/31/2019, 8:02:06 PM	3	128 MB	hw1_2102
-rw-r--r--	superbluecat	supergroup	17 B	3/31/2019, 8:02:06 PM	3	128 MB	hw1_2103
-rw-r--r--	superbluecat	supergroup	17 B	3/31/2019, 8:02:06 PM	3	128 MB	hw1_2104
-rw-r--r--	superbluecat	supergroup	17 B	3/31/2019, 8:02:06 PM	3	128 MB	hw1_2105
-rw-r--r--	superbluecat	supergroup	17 B	3/31/2019, 8:02:06 PM	3	128 MB	hw1_2106
-rw-r--r--	superbluecat	supergroup	17 B	3/31/2019, 8:02:07 PM	3	128 MB	hw1_2107
-rw-r--r--	superbluecat	supergroup	17 B	3/31/2019, 8:02:07 PM	3	128 MB	hw1_2108
-rw-r--r--	superbluecat	supergroup	17 B	3/31/2019, 8:02:07 PM	3	128 MB	hw1_2109
-rw-r--r--	superbluecat	supergroup	17 B	3/31/2019, 8:01:53 PM	3	128 MB	hw1_211
-rw-r--r--	superbluecat	supergroup	17 B	3/31/2019, 8:02:07 PM	3	128 MB	hw1_2110
-rw-r--r--	superbluecat	supergroup	17 B	3/31/2019, 8:02:07 PM	3	128 MB	hw1_2111

Each small file(about 17B) is put in a block of 128MB, and the block ids are distinct.

Although the bunch of small files don't occupy much space totally in HDFS, they possess these shortcomings:

- **Due to the large block, it's spatial-wasteful for HDFS to store many small files. Lots of unnecessary blocks are allocated, and files are really separated by blocks.**
- **It's not presented in the pictures, but each file has to store its index information in the namenode, which is also wasting the memory and availability of namenode.**

We will analyze time efficiency of reading bulk files and small files using a real data set in the next task.

Task3 - Comparison between bulk file as big data and piles of ones as small pieces of data

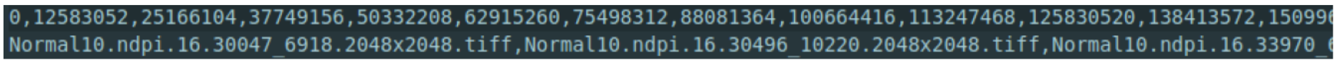
Here, we split whole task into two main functions. One *(Part I)* is implemented for uploading bulk file to HDFS where we fusion all images into one bulk file, while the other *(Part II)* is implemented for reading image files from HDFS with corresponding index file we create and maintain.

Part I : Uploading Bulk File

First of all, the main source code is shown below:

```
def save_img(img_dir):
    BULK_IMG_FN = "bulk_img.tiff"
    INDEX_FN = "index.txt"
    filename_list = []
    file_index_list = []
    file_index_count = 0
    bulk_img = b'';
    img_list = sorted(os.listdir(img_dir))

    if BULK_IMG_FN in img_list: img_list.remove(BULK_IMG_FN)
    if INDEX_FN in img_list: img_list.remove(INDEX_FN)

    # Compress images into one bulk image file
    output_file = open(os.path.join(img_dir, 'bulk_img.tiff'), 'wb')

    for fn in img_list:
        filename_list.append(fn)
        with open(os.path.join(img_dir, fn), 'rb') as f:
            img = f.read()
            output_file.write(img)
            file_index_list.append(str(file_index_count))
            file_index_count += len(img)

    output_file.close()

    # Build up index file
    with open(os.path.join(img_dir, 'index.txt'), 'w') as f:
        f.write(','.join(file_index_list))
        f.write('\n')
        f.write(','.join(filename_list))
    # Upload the bulk image up to HDFS
    s_upload = time.time()
    client = InsecureClient(HDFS_URL, user=HDFS_USERNAME)
    client.upload('./bulk_img.tiff', os.path.join(img_dir, 'bulk_img.tiff'),
overwrite=True)
    e_uplaod = time.time()
    print("Uploading images takes {} s".format(e_uplaod - s_upload))
```

- Read the local origin files (around 1.7 GB) and append the file bytes **output_file** together.
- In the meanwhile, record the offset each single image file holds in the bulk bytes variable **output_file** in variable **file_index_list** in favor of variable **file_index_count** .
- Done with the reading task, we produce index file **index.txt** to help our reading task on HDFS bulk file in the future. Note that the index is designed to contain two row with delimiter of , , one of which is for offsets, the other of which is for file names. **index.txt** is shown below:



```
0,12583052,25166104,37749156,50332208,62915260,75498312,88081364,100664416,113247468,125830520,138413572,15099
Normal10.ndpi.16.30047_6918.2048x2048.tiff,Normal10.ndpi.16.30496_10220.2048x2048.tiff,Normal10.ndpi.16.33970_
```

- At the end, upload the bulk file via HDFS API on Python (we use library hdfs here).

Part II : Reading Bulk File

First of all, the main source code is shown below:

```
def read_img(img_dir, img_fn):
    with open(os.path.join(img_dir, 'index.txt')) as f:
        file_index_list = f.readline().split(",")
        filename_list = f.readline().split(",")
        file_index_list[-1] = file_index_list[-1].replace("\n", "")

    try:
        start_index = int(file_index_list[filename_list.index(img_fn)])
        end_start = int(file_index_list[filename_list.index(img_fn) + 1])
        client = InsecureClient(HDFS_URL, user=HDFS_USERNAME)

        with client.read(hdfs_path='./bulk_img.tiff') as reader:
            bulk = reader.read()
        img = bulk[start_index: end_start]

        return img

    except Exception as e:
        print(e)
        print("No such a file name {}".format(img_fn))
```

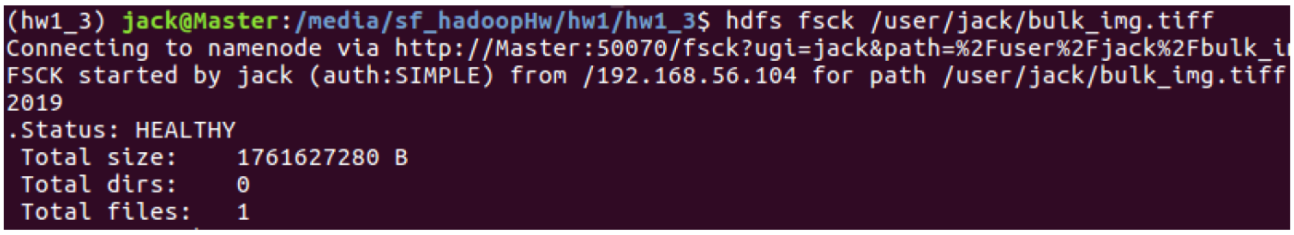
- Read our index file **index.txt** and transform it into two search lists. One is for file name, and the other is for corresponding offset. Read the whole bulk file into memory, then look up and extract the target image bytes based on its offset obtained by searching its filename in list **filename_list** .

- Do a try-catch in the case of absence of the target file name.

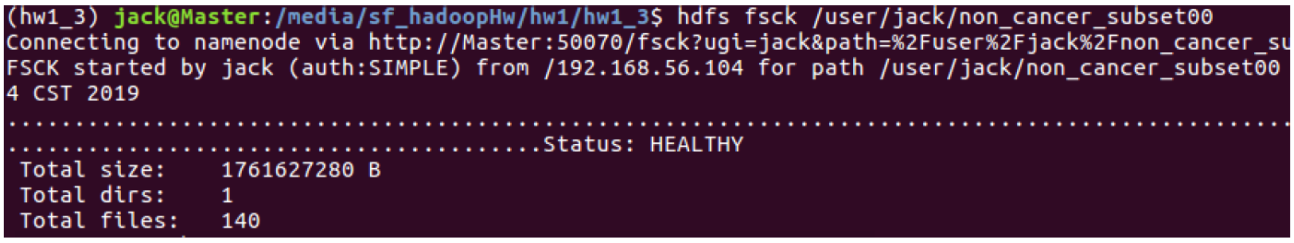
Analysis : Advantages of Bulk File

- Equal consumed logical memory

	Bulk file	Origin Files
Total size (B)	1761627280	1761627280



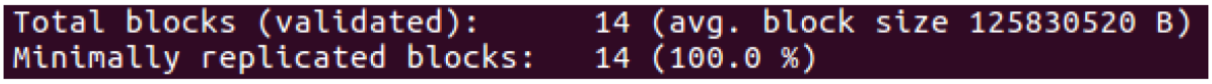
```
(hw1_3) jack@Master:/media/sf_hadoopHw/hw1/hw1_3$ hdfs fsck /user/jack/bulk_img.tiff
Connecting to namenode via http://Master:50070/fsck?ugi=jack&path=%2Fuser%2Fjack%2Fbulk_i
FSCK started by jack (auth:SIMPLE) from /192.168.56.104 for path /user/jack/bulk_img.tiff
2019
.Status: HEALTHY
 Total size:    1761627280 B
 Total dirs:    0
 Total files:   1
```



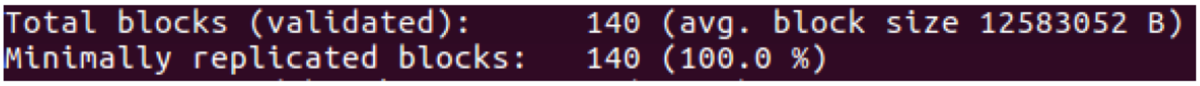
```
(hw1_3) jack@Master:/media/sf_hadoopHw/hw1/hw1_3$ hdfs fsck /user/jack/non_cancer_subset00
Connecting to namenode via http://Master:50070/fsck?ugi=jack&path=%2Fuser%2Fjack%2Fnon_cancer_su
FSCK started by jack (auth:SIMPLE) from /192.168.56.104 for path /user/jack/non_cancer_subset00
4 CST 2019
....................................................................................................
........................................Status: HEALTHY
 Total size:    1761627280 B
 Total dirs:    1
 Total files:   140
```

- Less number of occupied blocks

	Bulk file	Origin Files
Total blocks	14	140



```
Total blocks (validated):      14 (avg. block size 125830520 B)
Minimally replicated blocks:   14 (100.0 %)
```



```
Total blocks (validated):      140 (avg. block size 12583052 B)
Minimally replicated blocks:   140 (100.0 %)
```

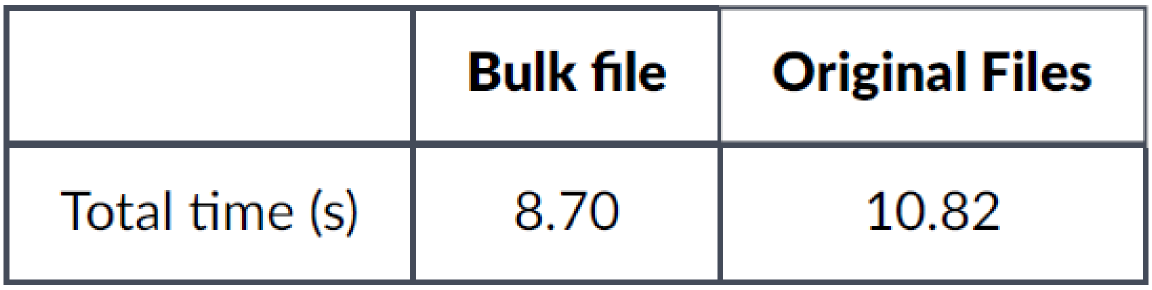
- Higher speed of saving and uploading

	Bulk file	Original Files
Creating bulk file (s)	11.21	-
Uploading file (s)	24.41	37.69
Total (s)	35.62	37.69

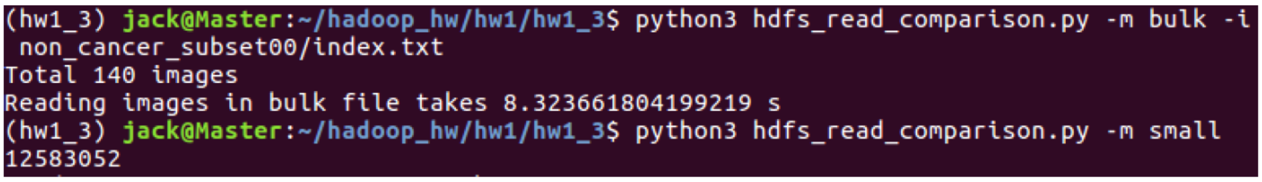
The total consumed time is quite the same, while uploading one big bulk file is faster than uploading piles of small files even if they're the same in total size. Note that we upload original small files via terminal which is based on Java while we upload the bulk file via Python, which indicates that the totally consumed time in the case of bulk file is faster than the one shown in the table.

- Higher speed when reading through all image files

  We read through all 140 image files in two cases. One case is that we fetch the bulk file back and cut them into original image files according to the index file. The other is that we iteratively call reading API to get all image files one by one.



	Bulk file	Original Files
Total time (s)	8.70	10.82



```
(hw1_3) jack@Master:~/hadoop_hw/hw1/hw1_3$ python3 hdfs_read_comparison.py -m bulk -i
 non_cancer_subset00/index.txt
Total 140 images
Reading images in bulk file takes 8.323661804199219 s
(hw1_3) jack@Master:~/hadoop_hw/hw1/hw1_3$ python3 hdfs_read_comparison.py -m small
12583052
```