Dig Data Management Assignment 1

Task3 - Comparison between bulk image file as big data and piles of ones as small peices 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('http://localhost:50070', user='jack')
    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.

```
# 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()
```

- 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` .

```
        file_index_list.append(str(file_index_count))
        file_index_count += len(img)
```

- Done with the reading task, produce index file `index.txt` to help our reading task on HDFS bulk file in the future.

```
# 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))
```

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:

```
0,12583052,25166104,37749156,50332208,62915260,75498312,88081364,100664416,113247468,125830520,138413572,150996
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).

```
# Upload the bulk image up to HDFS
s_upload = time.time()
client = InsecureClient('http://localhost:50070', user='jack')
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))
```

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])
        with open(os.path.join(img_dir, 'bulk_img.tiff'), 'rb') as f:
            img = f.read()[start_index: end_start]

        return img

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

- Read our index file `index.txt` and trasform 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` .

```
    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", "")
```

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

```
    try:
        start_index = int(file_index_list[filename_list.index(img_fn)])
        end_start = int(file_index_list[filename_list.index(img_fn) + 1])
        with open(os.path.join(img_dir, 'bulk_img.tiff'), 'rb') as f:
            img = f.read()[start_index: end_start]
```

Analysis : Advantages of Bulk

- 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 uplaoding

	Bulk file	**Original Files**
Creating bulk file (s)	28.60	-
Uploading file (s)	68.36	95.2
Total (s)	96.96	95.2

The totally 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.