**Comparison of Map Reduce runtime above Object Storage between**

**Virtual Big Files and Small Files Container**

For this project we implemented two versions of Map Reduce above Object Storage:

* “Classic” Map Reduce with Virtual Big Files
* Map Reduce with Small Files Container

A source code of a generic version of Map Reduce is located at MapReduceEninge.py

The execution of Map Reduce is consists of three phases:

* The Map phase: Each Map thread collects the data from its portion of the input files, and aggregate it into its temporary CSV files, where each row is a pair of key\_value and a file name.
* The Shuffle single thread phase: Read all the temporary output of the Map phase, for each key\_value instance, collect all its file names values into one new row.
* The Reduce phase: Each Reduce thread reads its own portion of the output rows of the shuffle phase, removes the duplicate file names, and sorts the resulting row. Then it writes its output rows into the final CSV file. Each Reduce thread creates its own output CSV file.

The temporary output files from the Map phase and Shuffle phase and also the final output files of the Reduce phase, were too big to be considered a “small file”. Therefore we used the exact same mechanism for them in both of our comparison tests. The only difference is in the implementation of the input files for the data creation phase and Map phase.

We tested this program on a single Laptop with 8 CPUs, hence we used 8 threads for each Map Reduce phase. The data consists of 100K CSV files, with 10 rows and 3 columns each. The columns were: “first name”, “city”, “second name” and the values were created randomly from a predefined small set.

The classic implementation of MapReduce with the VirtualBigFile is in the Jupyter Notebook: **MapReduceBigFiles**. The creation phase of the input data took 121 seconds, and the Map phase took 110 seconds.

Our small files implementation of MapReduce is in the Jupyter Notebook: **MapReduceSmallFiles**. The creation phase of the input data took 61 seconds, and the Map phase took 51 seconds.

We made another implementation of MapReduce that uses Object Storage directly. It used only for the sake of comparison, because it does not have the full capability of VirtualBigFile which let’s the user the ability to append data to existing files. This implementation is in the Jupyter Notebook **MapReduceObjectStorage**. The process run times are: 92 seconds for the creation phase, and 81 seconds for the Map phase. This is faster than the VirtualBigFile solution because it uses a simpler API; however, this implementation is still slower than our proposed solution.

**We conclude that our implementation of the small files container reduced the process time of the Map stage by half, and also reduced the creation time by half.**

**Mocking Object Storage**

For this project we created a class that mimics the behavior of Object Storage. The source code for this class is at **MockObjectStorage.py**

The MockObjectStorage class has the following methods:

* Creating a new data object to the disk with a given file name and given data.
* Reading an existing data object.
* Deleting an existing data object.

The MockObjectStorage class contains also a LRU cache with the following properties:

* Keeping track over the last recently used data objects with a LRU cache mechanism. The size of this cache is a parameter of the constructor of this manager.
* The LRU cache is implemented with a dictionary to meet the best time complexity .
* The LRU cache is thread safe, which means every create/read/delete operation can be executed from any thread that runs in parallel.

You can use the Jupyter Notebook: **MockObjectStorage\_Test** for watching a simple use case for this class.

**Virtual Big File**

On top of the MockObjectStorage, we implemented a virtual big file class that gives the user the ability to append new data into an existing virtual file. This higher level ability is achieved by storing each additional new data partition into a different physical data object. For each virtual file we also create an index file named the same as the user’s virtual file with a .csv file name extension. In that index file we keep track over all the data partition names and sizes of the virtual file.

This implementation source code is in the file: **VirtualBigFile.py**

To speed up the writing process of the virtual file creation, we use an in-memory partition named “self.appendix”, and when this virtual partition reaches a predefined block size, the class writes it into a new physical data object. The predefined block size is a parameter for the VirtualBigFile constructor.

The format of the index CSV file is: partition\_name,size

Another important property of this class is to let the user read a chunk of data from a random location in the virtual file in two different ways:

* According to the partition’s index: This gives the user the ability to read an entire partition regardless of its size.
* According to its starting position and ending position within the virtual file regardless of its partition index.

The Jupyter Notebook **VirtualBigFile\_Test** demonstrates a simple use case of this class.

**Small Files Container**

The VirtualBigFile stores a single file in several partitions. However, for very small files that are smaller than a single block, a size that is determined by the hardware configuration, this solution is very slow.

We are introducing the Small Files Container which uses a single instance of VirtualBigFile to store many small files. Its source code is at **SmallFilesContainer.py**

The Small Files Container gives the user the following abilities:

* Create a new small file with a given name and data
* Append to an existing small file.
* Read an existing small file
* Delete an existing small file
* Retrieve a list of all available small files within the current container.
* Update its index into the disk only upon calling the flush() method.

Modifying the SmallFilesContainer:

* The VirtualBigFile gives the user only the ability to append new data chunks, therefore, in order to keep track of all the small files, an index is created in the **last partition** of the VirtualBigFile. Every event of saving a new revision of the index data, is translated into an additional new partition in the VirtualBigFile.
* Appending data to an existing small file, is translated into a relevant change in the index data, and to a new revision of the small file. The new revision of the small file tries to use the existing temporary partition named “appendix” of the attached VirtualBigFile object. If the appendix becomes bigger than the block size, then the process dumps the appendix into a new partition, and adds the new small file revision into the newest empty appendix.
* Deleting a small file translates only to a change in the index.
* The container updates the index only when asked specifically to do so upon calling the flush() method, or when the constructor is executed on exit. This way we minimize the redundant partitions used within the container on the disk.

There are several restrictions and limitations when using our proposed solution for Small Files Container:

* The container uses VirtualBigFile class with a given block size as a parameter which determines the maximum size of each partition of the VirtualBigFile.
* A small file must belong to a specific partition, which implies each small file must be smaller than the given block size.
* We didn’t implement a mechanism for defragmentation of the container.
* In order to give the user the ability to modify an existing small file, while reading another small file in a multithreaded program, we added an automatic locking for all the I/O functions, which is active by default and can be turned off upon request.

The index is in a CSV format, where each row corresponds to a single small file. The CSV index file consists of 5 columns as follows:

* A filename: small\_file\_name.extension
* Index of partition that contains the small file
* Starting offset within the partition
* Ending offset within the partition
* File type flag: 0 = buffer of bytes array, 1 = text strings

The Jupyter Notebook **SmallFilesContainer\_Test** demonstrates a simple use case for this class.

**Suggested Future Work**

In this project we have several parameters that can be investigated for fine tuning:

* Block Size: We used a default value of 1MB. A smaller block size will cause more partitions in the SmallFilesContainer which might improve the overall process time because there will be less collisions between threads when reading the same container’s partition.
* Maximal Number of Threads: We choose it to be 8 threads, the same as the number of cores in our computer. We could use more threads, because each thread spends 85% of its time waiting for the disk I/O. However, there is no clear understanding about what could be the effect with respect to the time difference between our two implementations.
* Number of input files: We choose it to be 100K in order to demonstrate the power of our solution. One could plot a graph of the time difference between the two implementations, and see if it converges to some degree.
* Number of rows in each input file: We chose it to be 10 in order to demonstrate the usage of very small files. A bigger number of rows should reduce the time difference between the two implementations of Map Reduce.
* Cache size in the Mock Object Storage class: We chose it to be 32. Other Map Reduce problems can be greatly influenced from this value. One can investigate the cache hits/miss ratio and find its relation to the time difference of the two Map Reduce implementations.
* Using pickle file types instead of CSV files. We assume that this change would make only a little difference.