Duplicate Files -Algo Dev Assessment 2022

Given the **whole** file system (**not as input, you need to figure out how to iterate through the whole file system this is important**), find all the duplicate files in the whole file system in terms of their paths. You may return the answer in any order. A group of duplicate files consists of at least two files that have the same content. The output is a list of groups of duplicate file paths. For each group, it contains all the file paths of the files that have the same content.

Constraints & Tasks:

* The Problem needs to be solved strictly using Java
* Do not use any open-source software/jars
* **Bonus Points for using concurrency**
* Explain how you went about choosing the number of threads for each run
* Explain why you choose to solve the problem in that way and your choices of all data structures used
* Efficiency is key, apply efficiency mechanisms and explain why you choose those
* You should also write **junit test cases** for your solution
* You have 48 hrs to complete this 😊
* Add your code to Github and send a link through

**Example of a driver class is as bellow:**

public static void main(String [] arg) {

*//implement this method*  
 *searchFileSystem*();

*//print out the group of duplicates found in system* for (resultGroup : DuplicateGroups) {  
 print(resultGroup);  
 }  
 }  
}

**Solution Overview**

The solution was designed using the following algorithm:

1. Traverse the filesystem and get files with a size greater than 0
2. Keep a Map of files sizes to file paths, (using the file’s size attribute is cheaper than opening and calculating checksums of every file on the filesystem especially for bigger files and file systems).
3. For Map entries that have more than 2 files in the bucket, calculate the checksums of those files and store them in another Map that maps checksums to file paths.
4. Iterate through the checksum Map and select entries which have more than 1 file in them, those will be the true duplicates, as different files can have the same length but different checksums, I used the CRC32 checksum with results in very rare collisions.

**Assumptions**

* Zero length files can be ignored

**Solution Implementation**

The solution is implemented using Java 17 and is composed of 4 main classes that implement the requirements. 2 classes are used for traversing the file system, one using a single thread (**dedup.FileLister**) and another (**dedup.FileListerThreaded**) using multiple threads.

The other class, **dedup.FileChecksum** implements the checksum calculation logic. The same class can also be used to compare if 2 files are the same, content wise.

The other class, **dedup.DuplicateFinder** is the driver, it utilizes the functionalities provided by the other classes.

**Data Structures**

2 main data structures (HashMap and LinkedList) were employed in the solution, with 2 variants of each. The first variant is thread safe (supports concurrency) and the other not. The concurrent versions were used for the multi-threaded version of the solution.

HashMaps were used to keep the mapping of file sizes to a list of files that shared the same file size. Having the same file size is the first indication of a possible duplicate. I chose to use a Map because of the constant time when adding or removing items to the data structure, which enhances performance.

Linked list was used as the value component of the Map to hold file entries that shared the key. This was a good choice because they can grow dynamically depending on needs and that growth is less expensive as no data movement is required. In addition, adding and removing items has constant time. Since accessing of the list elements is sequential, this offers same performance as arrays when iterating through the list.

**Multi-Threaded Model**

As mentioned earlier, part of the solution leveraged multi-threading to enhance performance. Instead of working with naked threads and managing their lifecycle, which can easily get hairy and tricky, I leveraged the Parallel Streams API in Java.

Essentially, I would get a collection of items to process, stream them and process them in a parallel fashion. That way, I keep my code “simple and succinct” and let the standard library handle the gory details such as how many threads to use, managing the threads’ lifecycle, synchronization, among others.

If I were to do the thread handling manually, I would have used as many threads as the number of available CPU cores on the system using a call to: **Runtime.getRuntime().availableProcessors().**

**Error Handling**

For simplicity, the solution ignores any IO exceptions raised and continues processing. In a production ready system, such errors should be logged and monitored.

**Observations**

After implementation, I ran some tests and observed that when it comes to file traversal, the single threaded solution seemed to perform better than multi-threaded solution which on first thought, is quite surprising. This is quite plausible since the greater benefit of multi-threading comes when the work is CPU bound than IO bound. Given a system which has one File System and disk, having multiple threads requesting IO doesn’t make much of a positive impact. Same can be said when calculating the checksums, all threads will be issuing IO requests to the same IO subsystem, unless if some files are cached, only then could we observe significant benefits. The image below shows a sample run while testing:

