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 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 which gives a very low collision rate.

**Assumptions**

* Zero length files can be ignored

**Solution Implementation**

The solution is implemented using **Java 15** and is composed of 3 main classes:

1. **FileSystemWalker** – These group of classes are concerned with the file system traversal, i.e scanning the file system for files that are potential duplicates based on file size attribute. There is a ***single-threaded*** and a ***multi-threaded*** variant for this class.
2. **FileComparator** – This utility class is used for calculating CRC32 checksums for files and can be used to compare any files for similarity based on their checksums
3. **DuplicateScanner** – This group of classes driver the scanning of duplicates in the file system. They take a list of directories (defaults to all root drives on the machine) and check for duplicates based on their checksums. There are 3 variants of this class:  
   1. **Single-Threaded** – This version uses the main thread of execution to traverse the file system for potential duplicates and then calculates checksums and returns all duplicates.
   2. **Multi-Threaded** – This variant uses multiple threads to traverse and calculate the checksums. It employs the Producer-Consumer design pattern where some threads scan the file system for potential duplicates and as soon as they find any, they put them to a shared queue for checksum threads to start calculating the checksums.
   3. **Hybrid** – This last variant is the middle ground between the discussed models. Upon testing, I found that traversing the file system was consistently faster for the single-threaded version compared to the multi-threaded version (more on this later). In this model, the traversing is done sequentially while the checksum is done in a parallel fashion.

**Data Structures and Performance Considerations**

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

Hash Maps were used for parts of the system where we needed to keep a mapping of file names to some attribute such as checksum or file size. This was chosen because of the constant time **O(1)** when adding, checking for existence or removing items from the data structure, which enhances performance.

Linked lists were used to collect dynamic lists of items which were basically processed in a FIFO manner. New items were always added to the end of the queue and those to be processed next, were taken from the front of the queue. Once an item was done, removal is a matter of de-linking the node. All these operations take **O(1)** time which is great for performance.

As for space considerations, I tried to use the most efficient algorithm I could think of. For example, instead of having a Map keyed on file size with collections of file paths as the value, I had a Map of **filesize** -> **Map.Entry<Path, Boolean>** such that when a new file size is encountered, it added to the map with its Boolean (processed) flag set to **false** (indicating that it hasn’t been added to the checksum queue). When a new file with the same size is encountered again, the new item is added to the checksum queue, then the existing item is retrieved, and its processed flag is checked. If the flag is **false**, it’s added to the checksum queue and its flag is set to **true**. This means that the Map is kept lean and there is only one checksum queue holding qualifying items for further processing.

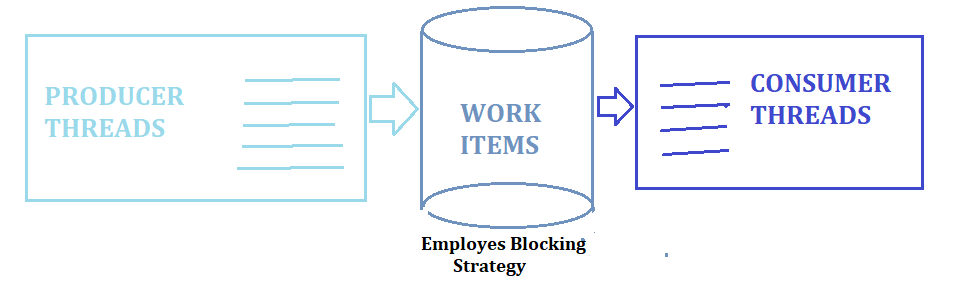
For the multi-threaded parts of the system that implemented the **Producer-Consumer** pattern, care was taken not to have threads spin doing nothing when there is no work, thereby wasting precious CPU time. To solve for this, I employed the blocking versions of the Queues. This meant that when a thread asks for work and there is none, it is blocked or suspended until there is work to be done. This enhances overall system resource usage.

**Multi-Threaded Model**

As mentioned earlier, parts 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 optimal threads to use, managing the threads’ lifecycle, synchronization, among others.

In the pure multi-threaded scanner class, I manually created 2 threads one for traversing the file system and the other for calculating the checksums. Each thread further employed more threads via the Parallel Streams API mentioned before. Below is a pictorial view of how the multi-threaded solution works:



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 by querying: **Runtime.getRuntime().availableProcessors().**

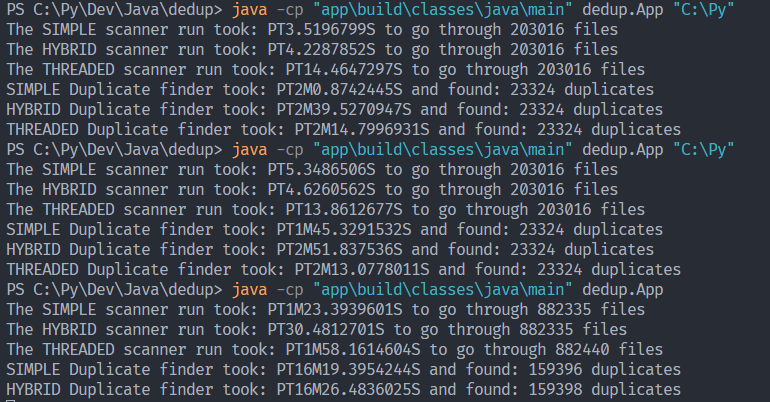
**Error Handling**

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**

As mentioned earlier, the single-threaded file traversal performed significantly better than the multi-threaded version which on first thought was surprising (see figure below, for an example of test runs). I expected it to be better since things were being done in parallel.

After thinking deeper about what could be happening, I realized that this is quite plausible since the greater benefit of multi-threading comes when the work is CPU bound than IO bound. Also, for NTFS file system, there is only one structure that maintains a listing of all the files on file system called the **master file table**, it made sense that accessing this structure from multiple threads only worsened performance.



(***Sample performance metrics of the various models***)

The performance of the multi-threaded version was not significantly better than the single-threaded version as expected. One reason could be that multiple threads requesting IO from the same IO subsystem only serves to have the subsystem work at maximum capacity and no real performance benefit. However, this model could be better on systems with multiple IO subsystems such as multiple drives or RAID enabled systems.