

Systems for Big Data Analytics

Team 4 - GROUP BY

Nested Loops

Nadir Abdou
Mickaël Corroyer
Ulysse Demay

Hashing

Yuhe Bai
Raphaël Boige
Joël Garde

Sorting

Taoufik Aghris
Aymane Berradi
Badr Laajaj

Introduction

The GROUP BY command is used in SQL to group the rows having the same value for one or several chosen features. It is often used with aggregate functions such as COUNT, MAX, MIN, SUM, or AVG to group the result-set by one or more columns.

Here is an example of the result of a GROUP BY request on a table:

sum(Annual Revenue) Group-by Departement

Employee	Department	Annual Revenue (K\$)		Department	Sum (Annual Revenue (K\$))
John	1	175	→	1	380
Alexis	2	80		2	170
Laurent	1	120		3	5
Nicolas	3	5			
Frederic	2	90			
Daniel	1	85			

In this project, we focused on the implementation of the GROUP BY statement in Java. We chose Java because it is overall faster than Python, it enables Multithreading and it supports Apache Spark. After a short literature survey we decided to implement three different GROUP BY techniques:

- GROUP BY with Nested Loops
- GROUP BY with Hashing
- GROUP BY with Sorting

For each technique we chose to implement three architectures:

- Single-threaded
- Multi-threaded with Java
- Multi-threaded with Apache Spark

In the first part of the report, you will find a complete literature survey for each of the three techniques, then in the second part, you will find descriptions of the implementations of these algorithms. Finally, the third part describes, interprets and compares the execution times of the different algorithms.

1. Literature Survey	5
1.1. GROUP BY with Nested Loops	5
1.1.1. Fundamental algorithm	5
1.1.2. Early aggregation	5
1.1.3. Multithreading	7
1.2. GROUP BY with Hashing	8
1.2.1. Fundamental Algorithm	8
1.2.2. Early Aggregation	8
1.2.3. Multithreading	9
1.3. GROUP BY with Sorting	12
1.3.1. Fundamental Algorithm	12
1.3.2. Comparisons	12
1.3.3. Sorting Algorithms	13
1.3.3.1. Selection Sort	13
1.3.3.2. Merge Sort	14
1.3.3.3. Heap Sort	14
1.3.4. Aggregation	15
2. Implementation	16
2.1. GROUP BY with Nested Loops	16
2.1.1. The 3 architectures	16
2.1.1.1. Single threading	16
2.1.1.2. Multithreading	17
2.1.1.3. Multithreading with Spark	17
2.1.2. Classical user and Test versions	17
2.1.2.1. The classical user version	18
2.1.2.2. The test version	18
2.2. GROUP BY with Hashing	19
2.2.1. Memory Assumptions	19
2.2.2. Single threaded application	19
2.2.3. Hashtable	20
2.2.4. Multi-threaded application	21
2.2.5. Concurrent hashtable	21
2.2.6. Parallel Hashtable	21
2.2.7. Partition hashtable	21
2.2.8 Spark Implementation	22
2.3. GROUP BY with Sorting	23
2.3.1. Java Methods	23
2.3.1.1. One hot encoding	23
2.3.1.2. Import and Export CSV functions	24
2.3.1.3. Selection Sort	24
2.3.1.4. Merge Sort	25
2.3.1.5. Heap Sort	25
2.3.1.6. Aggregation	26
2.3.2. Single thread	27

2.3.3. Multi-threads in JAVA:	27
2.3.4. Spark multi-threads in JAVA:	28
2.3.5. User interface	28
2.3.6. Results of tests on GROUP BY using sorting algorithms	29
2.4. A word on ORDER BY	32
3. Results and Comparisons	33
3.1. GROUP BY with Nested Loops	33
3.2. GROUP BY with Hashing	35
3.3. GROUP BY with Sorting	36
3.4. Comparison between algorithms	38
3.4.1 Our results	38
3.4.2 Theoretical and Real world results	38
References	39

1. Literature Survey

1.1. GROUP BY with Nested Loops

1.1.1. Fundamental algorithm

Among the techniques to perform a GROUP BY one can be interested in the nested-loops aggregation algorithm, which is the most simple-minded one. This algorithm is inspired by the nested-loops join and lies on successive loops over an inner and an outer relation. To sum up, for each tuple in the outer relation we make a loop over the entire inner relation.

The section “4.1 - Aggregation Algorithms Based on Nested Loops” of the article *Query Evaluation Techniques for Large Databases*^[1] written by Goetz Graefe focuses on this algorithm. It depicts a first version of it:

- Use a temporary file to accumulate the output.
- For each input item, loop over the output file accumulated so far and either aggregate the input item into the appropriate output item or create a new output and append it to the output file.

So the input file plays the role of the outer relation and the output file plays the role of the inner relation: for each input item we make a loop over the output file.

This article gives us some insights into how well this algorithm could perform. It seems that it is not very appropriate to large input relations because of the number of I/Os being too large also. The main advantage of this algorithm is that it doesn't really raise the issue of overflow. In fact, because everything is written in temporary files and every item is put in memory one by one we won't face memory overflow. One of its advantages is that it can support unusual aggregation where the input items are not divided into disjoint equivalence classes but where a single input item may contribute to multiple output items. It is not the case for the Grouping by Sorting and the Grouping by Hashing.

1.1.2. Early aggregation

Some improvements for this algorithm are given in *Query Evaluation Techniques for Large Databases*^[1]. We could loop over pages or clusters instead of over input items and output items. It is also possible to speed the inner loop with an index or to use bit vector filtering in order to determine if an item in the outer loop could have a match in the inner loop or not without inner loop or index.

The main enhancement of this algorithm to reduce its complexity is the “early aggregation”. In fact, to optimize such an algorithm, the aim is to mainly minimize the number of inputs and outputs (I/O), and these I/Os are numerous in the algorithm as presented above.

To achieve this optimization, the early aggregation will perform aggregations for as many groups as possible directly in memory before writing the result to the output file. The input data

which will not have its place in memory will be written in an overflow file, which will be used in the next step.

More specifically, here is a description of the steps of the nested-loops aggregation algorithm with early aggregation under the name “Repeated Scanning”, based on the section 4 of the article *Grouping and Duplicate Elimination: Benefits of Early Aggregation*^[2] written by Per-Ake Larson:

- Scan the input file (in a similar way to the initial algorithm) and maintain group records in memory.
- When the memory is complete, continue scanning the input, absorb all records that match a group already in memory and write all non-matching input records to an overflow file.
- The groups in memory are written in the output file when reaching the end of the scan of the input file.
- The overflow file is then used as the input file for the next pass.
- This process is repeated until a pass produces no overflow records.

We can notice that most of the work is now done in memory, which therefore minimizes the number of I/Os and theoretically improves the performance of the algorithm.

Indeed, as explained in the article, each pass outputs as many groups as there is room for in memory, which implies that the number of passes is equal to the total number of distinct groups in the input divided by the number of groups that fit in memory. But considering the complexity in terms of I/Os of the algorithm, what matters more is the amount of data written to and read from the overflow files and this amount depends on the distribution of group sizes. Indeed, large groups tend to be processed in the first pass, but if these large groups need to be stored in the overflow file, it will make the complexity increase.

After the description of the algorithm, the article^[2] written by Per-Ake Larson focuses on an analysis of the performances of such an algorithm. Calculating the volume of intermediate data generated by the algorithm involves probabilities over the distribution in groups of our relation. More precisely, the volume of intermediate data is the total of group records written to and read from overflow files that is to say the number of I/Os caused by our algorithm. When considering the volume intermediate data as a function of the input size for 10,000 groups and 10% of the computer’s memory used one can plot this graph (figure 3 of the article of Per-Ake Larson mentioned earlier):

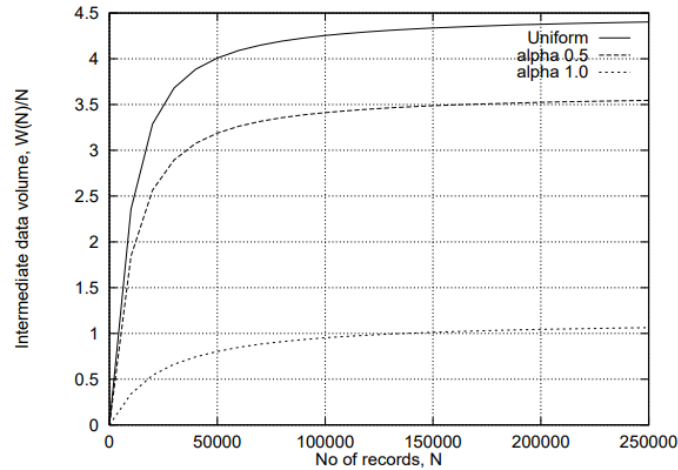


Figure 3: Intermediate data volume as a function of input size when using repeated scanning. 10,000 groups, 10% in memory.

Note that the amount of intermediate data is expressed as the fraction over the number of input records. In the first step, we will consider the curve for the uniform distribution that is to say there is a uniform distribution of the groups in our input relation. As you can see at the beginning the intermediate data volume increases with the number of records but it seems to converge. One will see if we also observe this steady state in our implementation. At such a step it is quite hard to express the complexity but since we loop over the input table and the memory containing the groups it may be of the form $O(\text{recordsNumber} * \text{groupsNumber} / \text{memorySize})$, one will see if our implementation verifies this complexity in the last part.

1.1.3. Multithreading

Now that we have seen how to reduce the number of I/Os of the nested-loops aggregation algorithm, let's take a look at how to increase its performance by implementing it in a multithreaded way.

The same article^[2] *Grouping and Duplicate Elimination: Benefits of Early Aggregation* of Per-Ake Larson briefly provides an intuitive technique for distributing the program across multiple nodes by partitioning the input data.

More precisely, here is a description of the multithreaded version of the algorithm:

- Divide the input file into several blocks.
- Each node performs grouping and aggregates on its block.
- After each node has finished its task, the result is shipped to one or more nodes which merge the partial results.

The article *Query Evaluation Techniques for Large Databases*^[1] deals with some issues this implementation of the nested-loops aggregation algorithm may raise. When we aggregate our data we have to choose an aggregation function like COUNT, SUM, AVG, etc. But to perform

a global count the local aggregation in each block counts while the global aggregation sums the different counts into a global count.

1.2. GROUP BY with Hashing

1.2.1. Fundamental Algorithm

For group by tasks, hashing is an alternative to sorting. In general, hashing has smaller complexity and should be considered as a good algorithm because the expected complexity of set algorithms based on hashing is $O(N)$ rather than $O(N \log N)$ as for sorting.

The main idea of this algorithm is to divide different groups into a hash table using a hash function, thus by iterating over the input once, we can get the result of a certain group by indexing its key in the hash table, therefore the complexity is $O(N)$.

The section “2.2 Hashing” and “4.3 Aggregation Algorithms Based on Hashing” of the article *Query Evaluation Techniques for Large Databases*^[1] written by Goetz Graefe focuses on this algorithm. The basic version (e.g. single-threaded) without considering overflows is like follows:

- Create a hash table in order to store the group information in memory
- For each input record, put its key and value into the hash table by a certain hash function
- Aggregate by groups then output

Unlike Nested loops where everything is written in temporary files and every item is put in memory one by one so we won't face memory overflow, in Group by hashing, we use a hash table in memory, so if the required hash table is larger than memory, hash table overflow occurs and must be dealt with. There are basically two methods for managing hash table overflow, namely avoidance and resolution.

- In avoidance, the input set is partitioned into several partition files before any in-memory hash table is built.
- In resolution, one starts with the assumption that overflow will not occur, but resorts to basically the same set of mechanisms as hash table overflow avoidance once it does occur.

In either case, the input is divided into multiple partition files such that partitions can be processed independently from one another, and the concatenation of the results of all partitions is the result of the entire operation. As we divide the input into multiple partitions, we can deal with it in multi threads.

1.2.2. Early Aggregation

The article *Grouping and Duplicate Elimination: Benefits of Early Aggregation*^[2] written by Per-Ake Larson provides a technique for speeding up the processing of GROUP BY queries by

reducing the amount of intermediate data transferred between main memory and disk called Early aggregation.

The basic idea is straightforward: when creating a run, maintain in memory a set of group records, one for each group seen so far; when an input record arrives, combine it with the matching group record if one exists, otherwise initialize a new group record.

In hashing algorithms, when an input record arrives, if the output hash table does not contain the group key, then we initialize a new group key with its value; if it contains the group key, then merge its value with the previous value. When memory becomes full, the group records are sorted and output as a run. The records carried into the merge phase then represent partially aggregated groups instead of individual input records. Early aggregation can also be applied during the merge phase by combining records from the same group whenever possible. Early aggregation always reduces the number of records processed during the merge phase. If the number of groups is small, all groups may fit in main memory and no merging is required.

1.2.3. Multithreading

There exists multiple architectures supporting the use of multiprocessors. In the context of in memory databases, *Adaptive Aggregation on Chip Multiprocessors*^[3] by Cieslewicz and A. Ross proposes three models that finally are augmented to five in *Scalable Aggregation on Multicore Processors*^[4] by Yang Ye, Kenneth A. Ross, Norases Vesdapunt. We implemented algorithms inspired by three of the five proposed methods: by “partitioning”, “sharing”, or “independence”.

An important concept in the context of hash-grouping is the one of partitions. This concept of partitions is central to approach multi-processors algorithms, as well as out-of-core algorithms. The insight is that for out-of-core algorithms it is essential to partition in order to divide the input into pieces that will fit in the main memory, and for multi-processors algorithms, it is still essential to partition in order to divide the input between the different processors. Indeed in *Adaptive parallel aggregation algorithms*^[5] they perform a cost comparison in the case of an out-of-core and multiprocessor scenario, and they use the same algorithms as the one seen in the previous cited papers.

The main idea of a multi-threaded method is to divide the input into multiple partitions, then one thread deals with one partition.

Partitioning should ensure that the partitioning files are of roughly even size and can be done using either hash partitioning (eg: by using a hash function to decide in which partition does a group belong. For instance, for b partitions, you can take the modulus of hashing, $\%b$). or range partitioning (e.g. half and other half). Usually, partition files can be processed using the original hash-based algorithm.

Unfortunately, it is possible that one or more partition files are larger than memory. In that case, partitioning is used recursively until the file sizes have shrunk to memory size. Figure 9 shows how a hash-based algorithm for a unary operation, such as aggregation or duplicate removal, partitions its input file over multiple recursion levels. The recursion terminates when the files fit into memory. In the deepest recursion level, hashing algorithms may be employed.

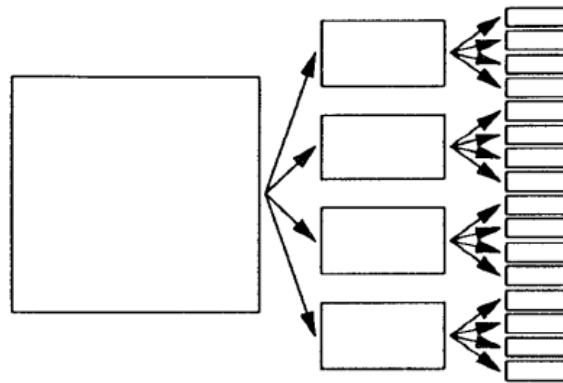


Figure 9. Recursive partitioning.

A major problem with hash-based algorithms is that their performance depends on the quality of the hash function. In many situations, fairly simple hash functions will perform reasonably well (A fair partitioning and the complexity is $O(N)$). We know that the purpose of using hash-based algorithms usually is to find database items with a specific key or to bring like items together, so the Hashing method is very suitable for group by tasks.

However, if the partitioning is skewed, the recursion depth may be unexpectedly high, making the algorithm rather slow. This is analogous to the worst-case performance of quicksort, $O(N^2)$ comparisons for an array of N items, if the partitioning pivots are chosen extremely poorly and do not divide arrays into nearly equal sub arrays (for example, we partition by $\%2$, but all of them are multiple of 2, then all the items will be put in one partition, while the other containing 0 item).

The two other main approaches to multi-processing are the “independent” and the “shared” approaches.

The “independent” approach to group-by operation is the easiest to implement. Each thread gets its part of the input and independently performs the grouping operation onto it. Once all threads are done, the algorithm enters a merging phase that merges the independent threads outputs to form the final output. This type of algorithm is really close to the map-reduce framework. In Java, it is implemented using the fork-join framework (<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html>). The fork-join framework makes it easy to express the recursive partitioning. However, there are two main downsides to this method. The first one is that each thread has to create its own table, meaning that the memory requirements are multiplied by the number of threads. The second problem is that the merging phase is still costly, and is fully avoided when using the partitioning scheme. Indeed, when partitioning, no merging whatsoever is required since a group can only be present in a unique partition. This is not the case for the independent approach.

The “shared” approach consists in using a unique global shared hash-table. Each thread then inserts directly onto this hash-table. One benefit is that this approach looks really similar to the single-threaded one. There is no need to partition the data, or to merge intermediary outputs, or create tables for each thread. However, having multiple threads mutating a single data-

structure means that one has to implement a concurrency control method. The above-mentioned papers propose two different concurrency controls. One is using atomic operations to update the values, and the other one is based on locks. Since insertion in a hash-table is cheap, the cost of acquiring locks can rapidly become an obstacle. for instance, the standard java implementation of such a map (<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html>), need to tune the granularity of the locks, to balance between the cost of thread waiting for a locked structure, and the cost of thread spending time acquiring and freeing locks.

The three main approaches described above were implemented in an “input fits in memory” setting.

An interesting remark is how the literature in grouping algorithms seems to keep turning around the same ideas. Indeed *Revisiting Aggregation for Data Intensive Applications: A Performance Study*^[6] claims a new algorithm they call “pre-partitioning”. And altho their contribution is non-negligible, with several optimizations, for instance using bloom filters and having a clear out-of-core methodology, the essence of their proposal is the same as what we described in the partition approach, from a paper that was published 20 years before. This can cause some confusion when approaching the field, as several similar algorithms are proposed under slightly different names. A personal opinion is that the main concerns for the group-by operations have stayed through the years, even when the cost model shifts from disk-main memory to main-memory-cache, or even for a distributed approach.

Finally *Cache-Efficient Aggregation: Hashing Is Sorting*^[7] gives a nice overview of hybrid approaches, using both hashing and sorting. They give valuable insight into the similarity of hashing and sorting based approaches in the main-memory to cache cost model. Sadly implementing data structures from this paper, that are optimized to fit in cache and minimize cache misses is outside the scope of our project.

1.3. GROUP BY with Sorting

1.3.1. Fundamental Algorithm

The general process of GROUP BY sorting consists in arranging the records in specific order of one column by grouping attributes and then performing aggregation on the sorted record stream. One of the efficient and fast sort algorithms used for processing large files is the Mergesort. The input data are written into initial sorted runs and then merged into larger and larger runs until only one run is left, the sorted output. Per-Åke Larson explains in the section “6.1 Run formation and 6.2 Run merging” of the article *Grouping and Duplicate Elimination: Benefits of Early Aggregation*^[2] how the Mergesort is implemented to group data. In this project we will focus on fixed length run formation algorithm that

- 1) reads an amount of records into memory,
- 2) sorts the data in memory using a sorting algorithm,
- 3) and then writes out the result as a run.

All sort algorithms are based either on partitioning, on merging, on selection or exchanging. In the article *Grouping Comparison Sort*^[8], Ibrahim Alturani and Khalid Alkharabsheh show the performance of Selection Sort and Quicksort in terms of time complexity and execution time and then propose a new approach of sorting an array.

The procedure of the algorithm:

- Inserting all elements of the array: it's the first step and the input of the algorithm.
- Blocks dividing and sorting: the array is split into groups of three elements, each block is sorting using the selection sort.
- Comparison: the biggest element in the first group BE_{ij} and the biggest element in the second group BE_{ik} are compared (i : number of elements in the group; j, k : number of group; $j < k$)
 - if BE_{ij} less than BE_{ik} then
 - We swap BE_{ij} and BE_{ik}
 - Sort the second group
 - Go to the next group, repeat the third point until access all groups and the biggest element of the array will be the first one.
- Move the second element of the first group
- Comparative between the second element in the first group and the biggest element in all groups as in step 3 and so on.
- Finished after Comparative between the third element of the penultimate group and the first element of the last group (since it will be the biggest element in this group).

1.3.2. Comparisons

Algorithm	Size of input					
	1000	10000	15000	20000	25000	30000
Selection sort	302	3580	5623	7947	10068	12785
Quick sort	269	3245	5045	6859	8483	10721
Grouping Comparison Sort	23	1721	3692	6337	9479	13314

Figure 1: Execution time for the three algorithms (ms)

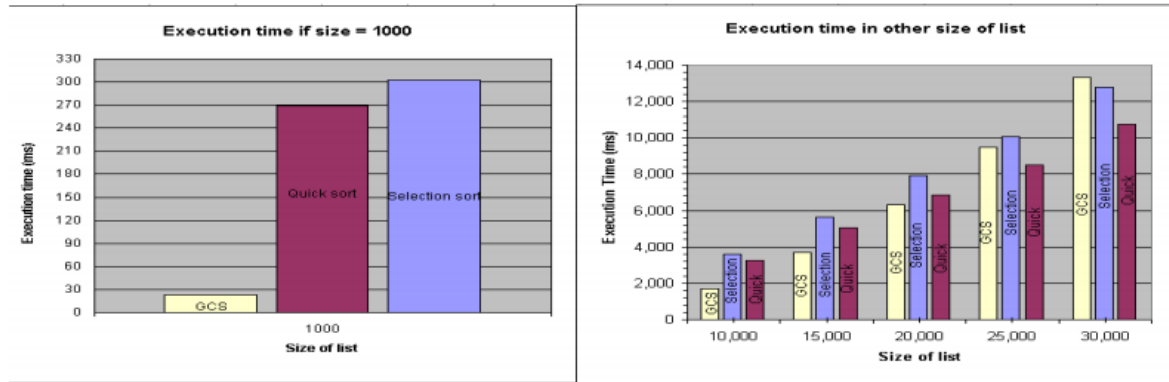


Figure 2: Execution time for the three algorithms (ms)

The proposed approach has less time execution when data size is below 20 000, otherwise the traditional sorting algorithms take the lead.

1.3.3. Sorting Algorithms

The first phase of the grouping by sorting algorithm is to sort data by a sorting algorithm before starting the grouping process. We used three different sorting algorithms: Selection Sort, Merge Sort and Heap Sort.

1.3.3.1. Selection Sort

The selection sort algorithm tries to find the minimum of an array and brings it at the beginning, considering ascending order, by swapping repeatedly the positions. In every iteration the algorithm picks the minimum from the unsorted part of the array and moves it to the sorted part.

Every algorithm is characterized by its complexity, and we hope that is not linked with data distribution, that's why we consider best, average and worst cases to analyze an algorithm. For the selection sort best, average and worst-case time complexity is n^2 which is independent of distribution of data.

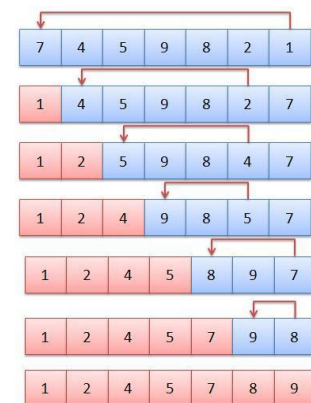


Figure 1: Selection Sort

1.3.3.2. Merge Sort

Merge sort algorithm is based on the divide-and-conquer approach to order elements in a certain data structure. The idea is to split recursively the collection into smaller groups by halving it until the groups only have one element or no elements, which are both entirely sorted groups. Then merge the groups back together so that their elements are in order. Mergesort runs in a guaranteed $O(n \log n)$ time, which is significantly faster than the average- and worst-case running times of several other sorting algorithms.

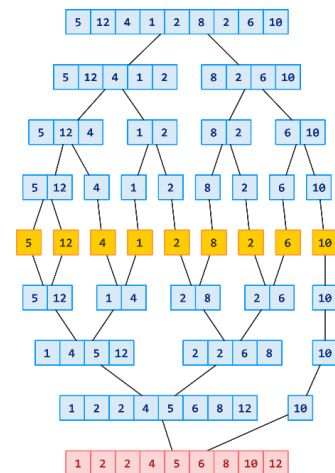


Figure: Merge Sort

1.3.3.3. Heap Sort

Heap sort is a comparison-based algorithm that uses two types of data structure which are trees and arrays. In fact, it visualizes the elements of an array as a special tree structure called binary heap tree. A binary heap data structure is first of all a complete binary tree, where the all elements of the tree are sorted, that is the root nodes should be greater than their children leaf (Max-heap) or the root nodes smaller than their children (Min-Heap). As shown in the figure below (figure 2), first thing to do is to build relation between array index and trees elements, where given an element i of the array, then the element $2*i+1$ will become the left child and the element $2*i+2$ will be the right child. And then we build a Max-heap (ordered tree), after that we extract the root of the tree, the first element as it is the maximum in the tree and put it in the end.

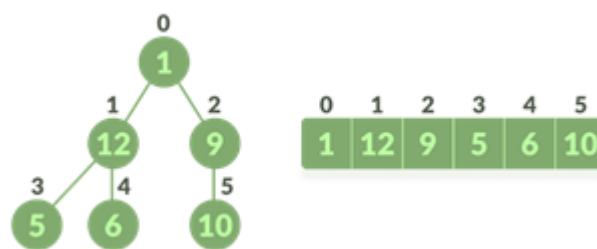


Figure 2 Heap tree elements and array indexes

Next step is to **heapify** the tree again, where we try to reorder the tree such that all root nodes are greater than their children (or opposite as explained before). As we can see in figure 3, the heapify function is recursive, so we apply it after each time we extract the greater element and put it at the end and consequently we apply it to smaller heap tree by one element (that is we discards the node we just moved to the bottom of the heap which was the largest element), and keep going until we reorder the tree again.

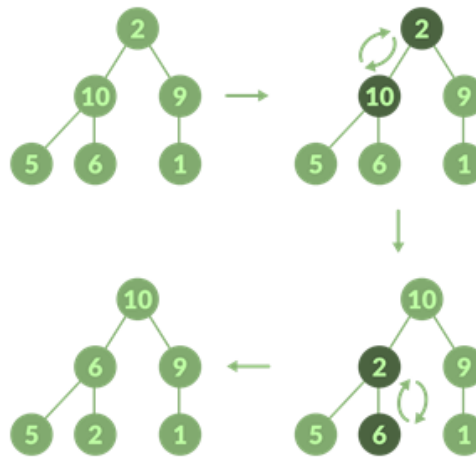


Figure 3 Heapify process example on root element

Concerning the complexity of Heap sort, it has a best, average and worst-case running time of $O(n \log n)$ like Merge sort, but it uses instead only $O(1)$ auxiliary space (since it is an in-place sort) while Merge sort takes up to $O(n)$ auxiliary space, which makes Heap sort a better choice if memory is an issue.

1.3.4. Aggregation

Now our data is sorted, the aggregation workflow comes into play. The idea of sorting before grouping is that all groups will be placed in the same block as it shown in the following image.

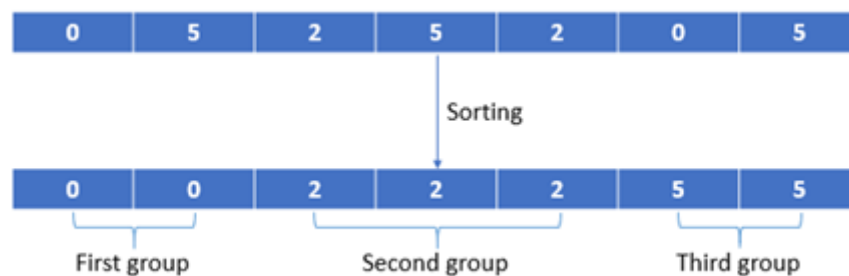


Figure: Benefit of Sorting before Grouping

There are many operations to aggregate data, the basic one is counting the group's members which require just one array. For other operations, an extra array is required to perform sum, average, minimum and maximum for the values that fall in the same group.

2. Implementation

2.1. GROUP BY with Nested Loops

We decided to implement the version of the algorithm with early aggregation to better visualize the impact of memory use on the performance of the algorithms, but also to obtain a program whose performance will be more comparable to those of the GROUP BY methods implemented by the other members of the group and deemed to be more effective. Note that we implemented the GROUP BY with COUNT as the aggregation function because the choice of this function has no real impact on the number of I/Os and as a result on the execution time.

We carried out this implementation using Java and in three successive steps: single-threading, multithreading with Java Threads tools, multithreading with Apache Spark tools. When the user runs the program, he is asked to select the file containing the input data, as well as the column on which he wishes to group by, and finally which method he wants to use (single threaded, multithreaded, or multithreaded with Spark). Now let's look at the implementation of each of these methods.

2.1.1. The 3 architectures

2.1.1.1. Single threading

As we have seen, the nested loops with early aggregation algorithm achieves a maximum of aggregations directly in memory and stores the other elements in an overflow file. Intuitively, the larger the size of available memory, the lower the number of I/Os, and the faster the program will perform. To analyze this phenomenon, we therefore “simulate” a memory of fixed size by a hash table whose size *MEMORY_SIZE* is a program variable. Thus, the program reads the records directly from the input file, performs aggregations on as many records as possible within the hash table, and stores the other records in an overflow file on disk. Once the program went through all the input dataset, the content of the hash table is written to an output file on the disk, then the process is repeated taking the overflow file as the input file. In this way, we can precisely control the maximum size of the memory used. Temporary files such as the overflow file are stored in a *tmp* folder created by the program and are deleted at the end of the program.

More precisely, we have implemented two classes *ReaderFile* and *WriterFile* which are used respectively to read data from a file on the disk, and to write in such a file. Another class *NestedLoopsSingleThread* contains the implementation of the algorithm. As we will see, this class implements the *Runnable* interface to be instanced as a *Thread* object, because it will be used for multithreading.

2.1.1.2. Multithreading

The multithreaded version of the nested loops algorithm is implemented within the *NestedLoopsMultiThread* class. It consists of dividing the input file into several blocks, each stored in a new temporary file on the disk, then performing a nested loops with early aggregation on each block in different threads, and finally merging the output files produced by each thread into a single output file. In this implementation, we kept our logic of memory simulated by a hash table. Indeed, each thread created by an instance of the *NestedLoopsMultiThread* class executes an instance of *NestedLoopsSingleThread* with its own hash table, and its own temporary files stored on the disk. Once these nested loops are executed, the *tmp* folder contains all the output files produced by each thread. These files are then concatenated in order to produce a new temporary input file, containing a much smaller number of records, on which a last iteration of the nested loops algorithm is performed, but this time with the sum aggregation function, to add the values of the aggregation functions of each group. This last iteration therefore produces the final output file stored in the *tmp* folder on the disk.

2.1.1.3. Multithreading with Spark

Using Spark for multithreading imposed a different method than what we had used before. In fact, in the previous multithreaded version, we imposed a maximum size for the memory used, and the program stored temporary files on the disk and read or wrote directly to them. With Spark, the input file must first be stored in a resilient distributed dataset (RDD), and the resulting output must then be written to a file. As a result, the execution time of this implementation is much greater than the execution time of the other nested loops algorithm.

More precisely, we first loaded the input CSV file in a *JavaRDD* object divided in the number of partitions chosen by the user. Then we map each partition with a function performing a nested loop algorithm directly in memory on each partition. After such a mapping our *JavaRDD* contains partitions corresponding to the results of the aggregation for each partition and we need to merge them to get the final output of the GROUP BY request. Instead of doing it with the *reduceByKey* function provided by Spark, we chose to merge them using the nested loops algorithm. To do so we mapped the entire *JavaRDD* object with a function merging the identical groups by adding their counts. Finally we get a *JavaRDD* containing the result of our request and after some transformations we write it in a CSV file as for the other implementations.

2.1.2. Classical user and Test versions

Our implementation of the Nested Loops algorithm with early aggregation contains two classes having a *main* function to provide two possibilities for the user.

2.1.2.1. The classical user version

The classical user version can be launched thanks to the *main* function of the *Main* class and provides a user friendly interface where the user can :

- Choose the CSV file he wants to GROUP BY in a windowed interface provided by a *JFileChooser*.
- Choose the feature he wants to GROUP BY.
- Choose if he wants to use Spark or classical Java threads.
- Choose the number of threads he wants to use.

After all these choices the algorithm runs and a CSV file containing the result of the request is created in a *tmp* directory in the parent directory of the project.

2.1.2.2. The test version

The test version is available by using the *main* function of the *MainForTest* class. This version is useful to execute our algorithm on every CSV file of a given directory. It executes the single-threaded, multi-threaded and multi-threaded with Spark implementations several times on all the files and builds a CSV file containing the mean of execution times for each architecture for each file. We compute the mean of all execution times for each file for each architecture because since our execution times are very small they can vary. The user has to provide two arguments to this function :

- First argument: the path toward the test directory.
- Second argument: the position of the feature on which the user wants to apply the GROUP BY request (starting at 0).
- Third argument: the number of repetitions of each algorithm for each architecture.

The times of execution are given in milliseconds in the file *result.CSV* which can be found in the *tmp* directory with one CSV file containing the output of the GROUP BY request for each file. Note that this version has been useful to provide data for the graphs found in the comparison part of the report.

2.2. GROUP BY with Hashing

2.2.1. Memory Assumptions

The GROUP-BY algorithms using hashing were implemented in single-threaded and multithreaded applications, as well as with spark.

The GROUP-BY hash makes the following important assumption:

- The output (the grouped records) fits in main memory.

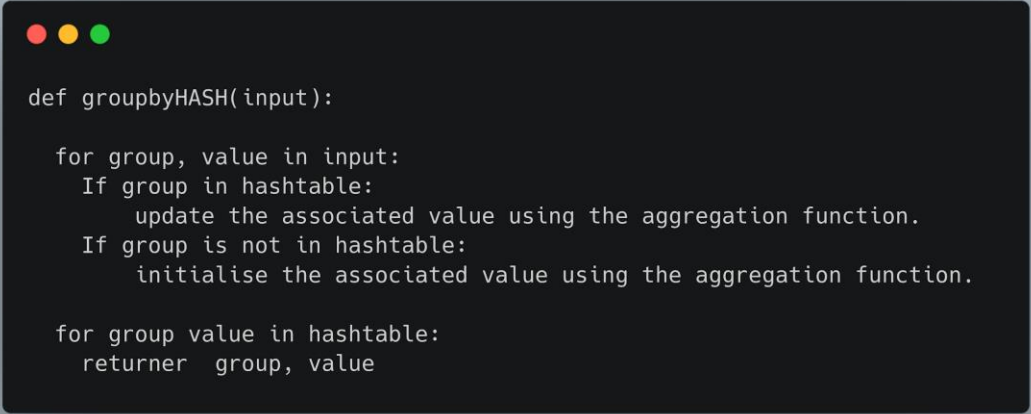
Please note that it does not mean that the full input has to fit in memory. Indeed, when aggregating early, you do not have to keep the input in memory and can simply stream it from the disk. Of course, for performance reasons, one still has to allocate a buffer to perform block-reading so as to avoid I/O costs.

This assumption is motivated by the time frame of our project and the literature survey. If you want to implement a fully online group-by hashing, you have to manage several partitions, each having to be spilled on disk independently. What is more, these partitions can't be implemented in java as simple arrays, since each partition will contain several buckets and a shared hash-table. All-in-all, having a fully online architecture means having a functional page and buffer system. We believe it is well outside the scope of our project.

What is more, to ease the implementation of the hash-table, we first limited records' fields to be integer-valued. This assumption was later reduced by using the built-in hash() function from Java. More about this subject in the hashing/hash table chapter.

2.2.2. Single threaded application

It is important to understand the concepts involved in the single-threaded version, because both multi-processors and spark implementation borrow from it. The algorithm works in two passes. One is to fill a hashtable with the aggregated groups values. The other one simply outputs the groups.



```
def groupbyHASH(input):
    for group, value in input:
        If group in hashtable:
            update the associated value using the aggregation function.
        If group is not in hashtable:
            initialise the associated value using the aggregation function.

    for group value in hashtable:
        returner group, value
```

From this, one can see that the most interesting part is the hashtable, and how to insert, retrieve, and update values.

2.2.3. Hashtable

A hashtable is an associative data-structure, that provides fast ($O(1)$, amortized) access. Indeed, for our application, we need a data structure that provides fast lookup to the group, fast update, for when we aggregate a new value and fast iteration over the results. Note that it is not necessarily important to have a data structure that provides fast insertion. Indeed, if you suppose that the number of groups M is small compared to the input size N , you will only have M insertions but N updates: the major driver of the cost is the insertion.

A hashtable is perfect for that. Lookup is $O(1)$, and insertion is also $O(1)$, amortized because you might have to rescale the hashtable.

The first step for a hash-table is to have a hashing-function. This function is used to get an index into the table, from the hashed value, here the group. This hash-function is actually the composition of two functions:

- one to go from the domain of the input to an integer
- one to go from an integer to a valid index in the hash-table.

For our implementation, at first we only supported integers, and used modulo hashing. So that the first function was the identity, and the second was a modulus operation to the size of the backing array.

More about the desired characteristics and possibilities for hash functions are here: <https://www.cs.cornell.edu/courses/cs312/2008sp/lectures/lec21.html>.

Our hashtable is an open-addressing hash table, with linear probing.

It means that the whole hash-table is a single array, and if there is a hashing conflict, we linearly iterate over the next buckets until we find one free. Linear-probing has the advantage of playing nice with the processor cache. Its main drawback is being prone to “ball of muds”, or hashing conflict concentrating on one point. A really good alternative is the roundrobin hashing. It still performs linear probing and has a good cache behavior, but it moves keys around to reduce the maximum probing length. Probing for one, or for three to five buckets has the same cost because the cache line will hold five buckets. So by making every probing to be length 3, it is as fast as if every probing was of length one.

An interesting fact about our hashtable is that it does not need to support removing operations: it is an append-only hashtable.

2.2.4. Multi-threaded application

We implemented three versions of the multiprocessor algorithm. We only kept the best performing option for the measures.

2.2.5. Concurrent hashtable

We implemented a concurrent hashtable to implement the “shared” version of the algorithm. In our implementation, each bucket is protected by a lock. We also use global locks to perform safe re-shaping for the table. Multiple threads are spawned to insert concurrently in this hashtable.

The interesting point about the concurrent hashtable is the necessity for an atomic update function. Indeed, what was previously implemented as a “check if the group is in the hashtable, retrieve the index, and modify the value at the corresponding index and to be implemented as a single atomic transaction.

The cost incurred by the locks were too high for this implementation to be worth it.

2.2.6. Parallel Hashtable

Parallel hash tables are implemented by making a new hash-table for each thread. We then implement an additional function, merge(), to merge() the outputs of the different hash-tables. The merging is done on the main thread. The merging operation is basically the same as the grouping operation, but this time with pre-formed groups. The cost incurred by the merge was too high to be worth it.

2.2.7. Partition hashtable

Our implementation of the partitioning hash table starts by creating a new hash-table for each thread. There is no need for a merge function. Instead, each thread will only input in its own hashtable the records that belong to its partition. The partition is determined by a modular

hashing on the group, by the number of threads. This way, no two records belonging to the same groups can be managed by different threads.

Because of this careful partitioning, there is no need to merge outputs. Each thread can start outputting groups as soon as it is finished.

2.2.8 Spark Implementation

The Spark implementation uses only the functions of the main Spark module and not the SparkSQL module. In fact, apart from the functions used to parse the input files we only used two functions :

- **mapPartitions** which splits the input for the different workers and performs an operation on each partition.
- **reduce** that take the results of **mapPartitions** and group 2 by 2 the output until we obtain a final result.

All of the other operations are handmade and are based on what was implemented for the single-threaded and multithreaded algorithms, though it has some small differences that are not supposed to influence performance (e.g. we hash on String rather than Integer).

2.3. GROUP BY with Sorting

We supposed at first that the groups of a given dataset are strings, in order to build the algorithm, we generate a dataset from an online website consisting on 100 rows and 3 columns, we will work mainly with 'Store region' (column that contains groups) and 'Revenues per year'.

id	Store region	Revenues per year
1	CH	25864
2	IL	26814
3	LA	59371
4	BR	68138
5	CH	18776
6	PA	9674
7	AQ	35308
8	IL	94518
9	PO	17337
10	IL	53454

Figure: Dataset for building the model

We will consider 3 implementations: single thread, multi-thread and Spark, before we get in depth in each one of them, we will present the common java methods that we used for the 3 modes.

2.3.1. Java Methods

2.3.1.1. One hot encoding

We choose to compress the strings by assigning numbers to them.

```
public static HashMap<String, Integer>, int []> one_hot_encoding(String tab [],String [] unique){
    HashMap<String, Integer> my_dict = new HashMap<String, Integer>();
    HashMap<String, Integer>, int []> hm=new HashMap<>();
    int[] index=new int[tab.length];
    for (int i=0;i<unique.length;i++)
    {
        for (int j=0;j<tab.length;j++)
        {
            if (tab[j] == null ? unique[i] == null : tab[j].equals(unique[i]))
            {
                index[j]=i;
                my_dict.put(tab[j], i);
            }
        }
    }
    hm.put(my_dict, index);
    return hm;
}
```

```
[CO = 13]
[HA = 12]
[LO = 10]
[IL = 0]
[CH = 3]
[AU = 7]
[LI = 16]
[CE = 14]
[AQ = 5]
[FC = 9]
[LA = 1]
```

Figure: One hot Encoding algorithm

Figure: One hot Encoding results

- The function has as parameters the String groups from the data, and the unique values of these Strings.
- We assign for every unique string in our array a number, we save the conversion on a hash table.

To have the same output with other hash and nested loops groups, we will consider that the desired column for grouping is integer and will consider only the count operation.

2.3.1.2. Import and Export CSV functions

The read CSV function takes the CSV file path as input and outputs the data as matrix, so we choose to store our records in a 2D array.

After implementing the process, we export the result in a CSV file, it consists of the group name and count of group members.

```
public class Read_CSV {
    public static String[][] read_csv(String path){
        List<String[]> rowList = new ArrayList<String[]>();
        try (BufferedReader br = new BufferedReader(new FileReader(path))) {
            String line;
            while ((line = br.readLine()) != null) {
                String[] lineItems = line.split(",");
                rowList.add(lineItems);
            }
            br.close();
        } catch (Exception e) {
            System.out.println("not found");
        }
        String[][] matrix = new String[rowList.size()][0];
        for (int i = 0; i < rowList.size(); i++) {
            String[] row = rowList.get(i);
            matrix[i] = row;
        }
        return matrix;
    }
}
```

Figure : Read CSV

```
public class Export_csv {
    public static String[] export_csv(Map<Integer, Integer> res,
        String csv_file) throws FileNotFoundException {
        try (PrintWriter writer = new PrintWriter(new File(csv_file))) {
            StringBuilder builder = new StringBuilder();
            builder.append("Group Name");
            builder.append(",");
            builder.append("Count");
            builder.append("\n");
            for (Map.Entry<Integer, Integer> kvp : res.entrySet()) {
                builder.append(kvp.getKey());
                builder.append(",");
                builder.append(kvp.getValue());
                builder.append("\n");
            }
            String content = builder.toString().trim();
            writer.write(builder.toString());
        }
        return null;
    }
}
```

Figure : Export CSV

2.3.1.3. Selection Sort

```
public static String [][] Selection_Sort_matrix(String matrix [][] , int col)
{
    int size, i, j;
    String[] temp=new String [matrix.length];
    size=matrix.length;
    System.out.println(Integer.parseInt(matrix[1][0]));
    for(i=1; i<size; i++)
    {
        for(j=i+1; j<size; j++)
        {
            if(Integer.parseInt(matrix[i][col]) > Integer.parseInt(matrix[j][col]))
            {
                temp = matrix[i];
                matrix[i] = matrix[j];
                matrix[j] = temp;
            }
        }
    }
    return matrix;
}
```

Figure: Selection Sort

We implemented the matrix version of the selection sort since we assume that a dataset can have multiple columns, so instead of swapping just two elements in the array, we swap 2 rows of matrix, so that all records positions are updated, the function has as arguments the original matrix and the column index (that contains groups).

2.3.1.4. Merge Sort

The algorithm is implemented in 2 step process:

Mergesort. This function takes in the input array and its length as the parameters. In this step, we divide the input array into 2 halves, the pivot being the midpoint of the array. This step is carried out recursively for all the half arrays until there are no more half arrays to divide.

Merge: this function compares the elements of both sub-arrays one by one and places the smaller element into the input array. When we reach the end of one of the sub-arrays, the rest of the elements from the other array are copied into the input array thereby giving us the final sorted array

```
public static String[][] MergeSort(String[][] matrix, int n, int col) {  
    if (n < 2) {  
        return matrix ;  
    }  
    int midpoint = (int) n / 2;  
    String[][] LEFT = new String[midpoint][];  
    String[][] RIGHT = new String[n - midpoint][];  
    for (int i = 0; i < midpoint; i++) {  
        LEFT[i] = matrix[i];  
    }  
    for (int i = midpoint; i < n; i++) {  
        RIGHT[i - midpoint] = matrix[i];  
    }  
    MergeSort(LEFT, midpoint, col);  
    MergeSort(RIGHT, n - midpoint, col);  
    return merge(matrix, LEFT, RIGHT, midpoint, n - midpoint, col);  
}
```

Figure: MergeSort algorithm

```
public static String[][] merge(  
    String[][] a, String[][] LEFT, String[][] RIGHT, int left, int right, int col) {  
    int i = 0, j = 0, k = 0;  
    while (i < left || j < right) {  
        if (Integer.parseInt(LEFT[i][col]) <= Integer.parseInt(RIGHT[j][col])) {  
            a[k++] = LEFT[i++];  
        }  
        else {  
            a[k++] = RIGHT[j++];  
        }  
    }  
    while (i < left) {  
        a[k++] = LEFT[i++];  
    }  
    while (j < right) {  
        a[k++] = RIGHT[j++];  
    }  
    return a;  
}
```

Figure: Merge algorithm

2.3.1.5. Heap Sort

In the Heap Sort implementation, we have two main functions: **Heap_sort_matrix** that has as input the **matrix** that we want to sort and a parameter integer **p** used for multi-threading manipulation ($p = 0$ if the sorting algorithm used for counting and aggregation, and $p = 1$ if used for merging the results of the threads), and **heapify** function that has as input the **array** we want to heapify, an integer **n** as the size of the heap and finally an integer **i** which represents the node index of the root the sub-tree that we want to heapify.

The **Heap_sort_matrix** ensures the first heapifying of the array, that is it creates in first place an array with a binary tree shape (index management) in an ordered way (Max-heap). And then it takes the max element (head of the tree) and put it at the end of the array (what we called it before extraction of the greatest element of the tree) after that it calls for the **heapify** function, that as we explained before compare between the root of the sub-tree and its children and reorder them if needs. And we keep doing this recursively until we get an ordered array.

```

public static String[][] Heap_sort_matrix(String [][] matrix,int p)
{
    int n = matrix.length;

    // Create the heap tree and reorder the matrix
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(matrix, n, i,p);

    // Extract greatest element (the root of the heap and put it at the end) that's why loop n-1 to 1
    for (int i=n-1; i>0; i--)
    {
        // Move current root to end
        String [] temp = matrix[0];
        matrix[0] = matrix[i];
        matrix[i] = temp;

        // Create max heap tree using the new sub-tree
        heapify(matrix, i, 0, p);
    }
    return matrix;
}

```

Figure Heap Sort Algorithm

2.3.1.6. Aggregation

After sorting our data, we need to aggregate the results. As explained before, we try to aggregate (using count or sum...) the results until we notice a variation of the group where we break the loop, stores the couple (group, result – count for example –) and the index where we break so we can start from there again until we aggregate all the groups results.

```

public static String[][] Aggregation(String[][] matrix, int p) {
    Map<String, String> map = Collections.synchronizedMap(new LinkedHashMap<String, String>());
    //String[][] map;
    int count = 0;
    int pos = 0;
    int i = 0;
    int k = 0;
    int m = 0;
    int sum = 0;

    do {
        for (m = k; m < matrix.length; m++) {
            if (p == 0 && Integer.parseInt(matrix[k][1]) == Integer.parseInt(matrix[m][1])) {
                count++;
            } else if (p == 1 && Integer.parseInt(matrix[k][0]) == Integer.parseInt(matrix[m][0])) {
                sum = sum + Integer.parseInt(matrix[m][1]);
                count++;
            } else {
                break;
            }
        }
        if (p == 0) {
            map.put(matrix[k][1], Integer.toString(count));
        } else if (p == 1) {
            map.put(matrix[k][0], Integer.toString(sum));
        }
        pos = pos + count;
        k = pos;
        count = 0;
        sum = 0;
    } while (k < matrix.length);
    String[][] output = new String[map.size()][2];
    int t = 0;
    for (Map.Entry<String, String> entry : map.entrySet()) { //We convert the map to matrix so it can be used in other processes
        output[t++] = new String[]{entry.getKey(), entry.getValue()};
    }
    return output;
}

```

Figure: Aggregation of sorted matrix using count

Group Name	Count	Average	Minimum	Maximum	Sum
CO	1	36712	36712	36712	36712
HA	4	50070	7103	89844	200280
LO	2	65706	36182	95230	131412
IL	23	56593.043	5101	94518	1301640
CH	3	41040.332	7119	97226	123121
AU	3	85386.664	81690	89214	256160
LI	1	95069	95069	95069	95069
CE	2	40520.5	32912	48129	81041
AQ	6	47476.832	18467	87497	284861

Figure: Example of results for different Aggregation functions we implemented

2.3.2. Single thread

The first implementation mode is the single thread, it assumes that the execution of instructions is done in a single sequence. In other words, processes all the data in just one thread.

A	B
Group Name	Count
328959	2450
337257	2502
429508	2491
462050	2547
480854	2533
508228	2497
606543	2533
610262	2513
703299	2487
967577	2543

Figure: Result of single thread implementation

2.3.3. Multi-threads in JAVA:

In multi-threads Java, we created **Groupby_Thread** class that implements the Callable class which is an improved version of Runnable class which allows the use of thread under JAVA with the possibility of returning values such as string matrix in our case. The idea behind multithreading in JAVA is to split tasks between threads that work on small partitions of data so in a faster way and output a sorted aggregated matrix of 2 columns: Id of the group and its counting. So, the next step is to merge the outputs by summing the counting of the groups that may exist in other partitions, so to do that we use **Merge_parts** function that concatenates the output of two threads and apply grouping by sorting again but using Sum as aggregation function (and here where we use the parameter p that we mentioned in heap sort class's description above). These steps are organized in the global function **GroupingMulti**.

```

public static String[][] GroupingMulti(String[][] matrix, int nb_threads) throws InterruptedException, ExecutionException {
    //Main class for Multithread Group By using Sorting algorithm, which executes the different steps
    ExecutorService executor = Executors.newFixedThreadPool(nb_threads);
    List<Future<String[][]>> Futures = new ArrayList<>();
    int pas_part = (int) matrix.length / nb_threads;
    int t = 0;
    for(int i = 0; i < matrix.length; i += pas_part) //We assign each thread a partition of the data
    {
        t++; //check in which thread we are so we can avoid small subset at the end
        if( t < nb_threads){
            Groupby_Thread task = new Groupby_Thread(Arrays.copyOfRange(matrix, i, i+pas_part)); // allocate thread for each partition of 1
            //System.out.println(i);
            Futures.add(executor.submit(task));
        }
        else{
            Groupby_Thread task = new Groupby_Thread(Arrays.copyOfRange(matrix, i, matrix.length)); // allocate thread for each partition 1
            //System.out.println(i);
            Futures.add(executor.submit(task));
            break;
        }
    }

    List<String[][]> outs = new ArrayList<>();
    for(int i = 0; i < Futures.size(); i++){ //We get the output of each task run by the threads
        outs.add(Futures.get(i).get());
    }
    String[][] sol = outs.get(0);
    for(int i = 1; i < outs.size(); i++){
        String[][] out = outs.get(i);
        sol = Merge_parts(sol, out); // Merge the outputs of the threads
    }
    executor.shutdown();
    return sol;
}

```

Figure: GroupingMulti function that implement the steps of multithreading in JAVA

2.3.4. Spark multi-threads in JAVA:

Spark is an open-source parallel data processing engine for performing large-scale analysis through clustered machines which allows users to take advantage of multicore systems without imposing the overhead of creating multiple processes and providing direct access to a common address space. The use of fast performing Resilient Distributed Datasets (RDDs) in Spark allows to store temporary the input file in a number of partitions specified by the user in a JavaRDD object, then perform the sorting algorithm to order the data and aggregate it in each partition using the mapping. Finally, the partitions in the JavaRDD are merged by summing the count of each identical group from all the partitions. The RDD structure makes the execution time of the grouping implementation very low than single and multi thread implementation. However, Transforming the stored data from JavaRDD to a CSV file takes more time.

2.3.5. User interface

In order to facilitate the utilization of the algorithm, we gather all classes in one interface that allows the user to select the desired CSV file.

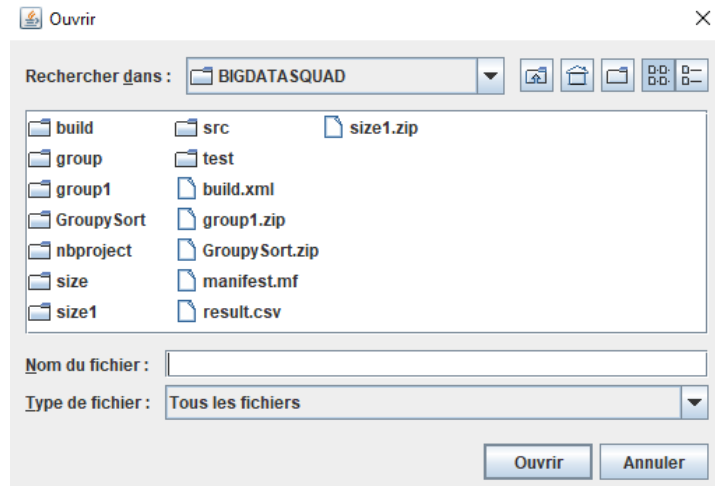


Figure: Choice of CSV file path

After getting the path, the user can choose the column name from the CSV file picked previously that contains groups and the implementation mode (single thread, multi-thread and Spark).

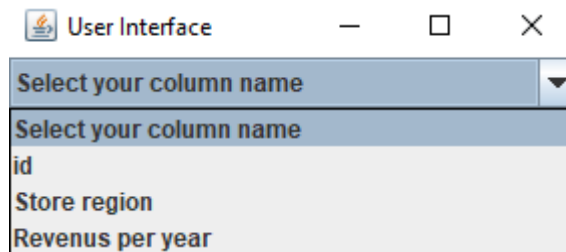
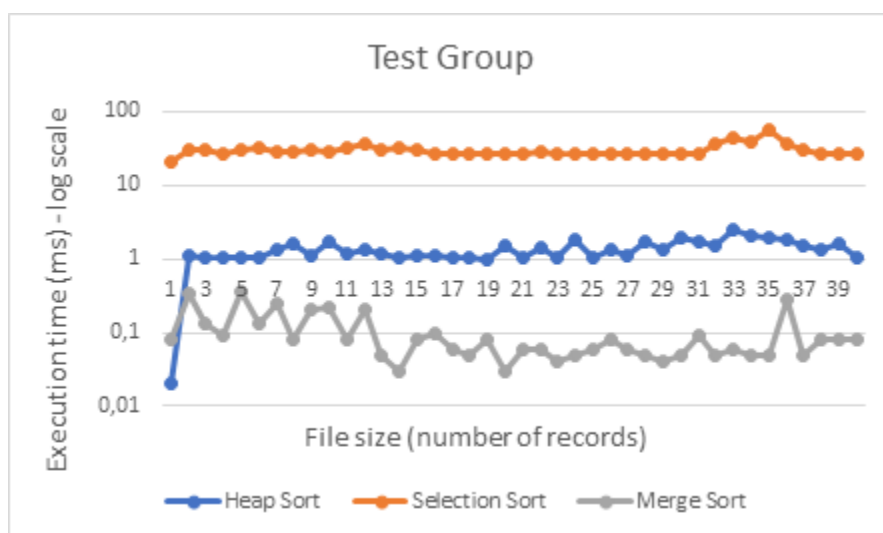
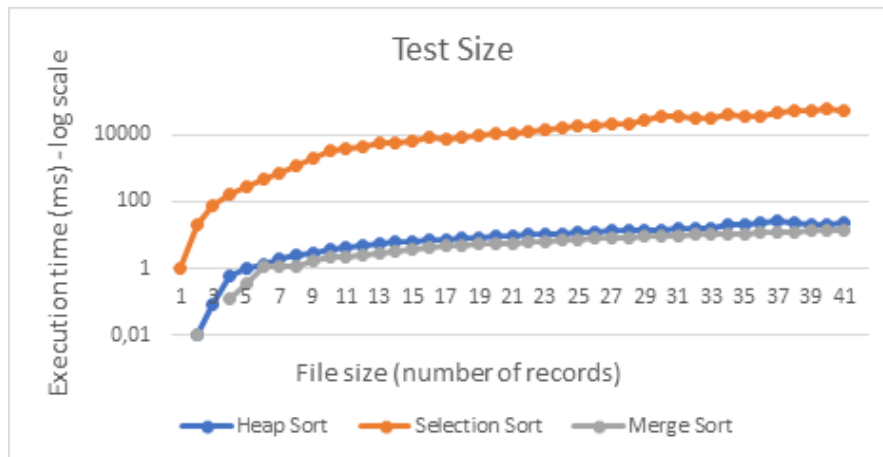


Figure: User interface

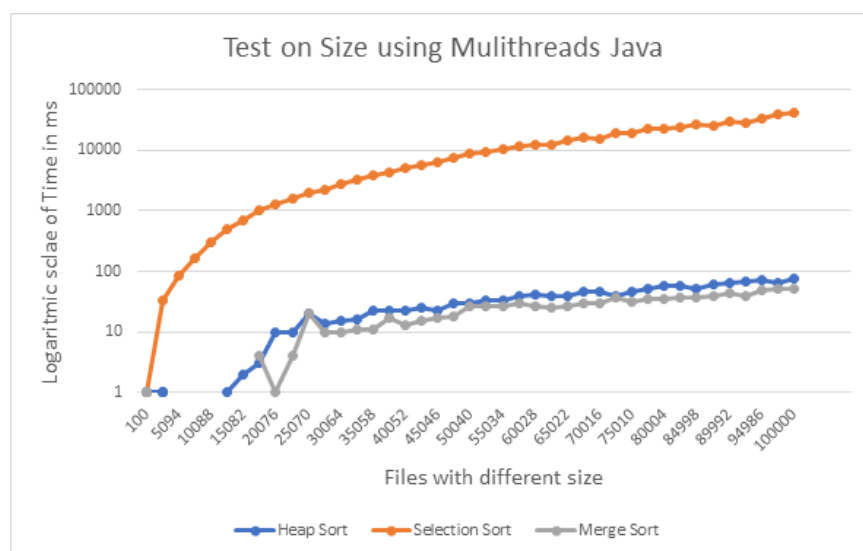
2.3.6. Results of tests on GROUP BY using sorting algorithms

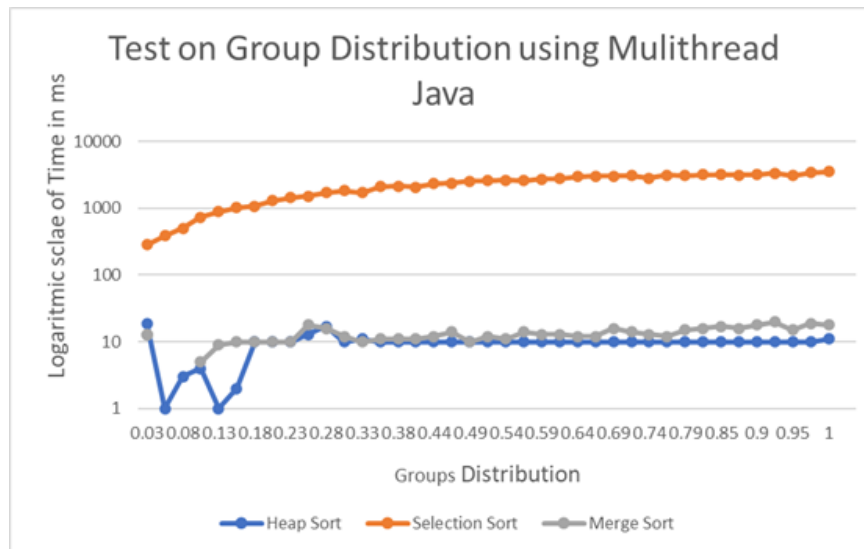
In this part, we want to compare our group by algorithm using the three sorting algorithms in terms of data size and groups distribution. Because Selection Sort is too slow, we create 40 CSV files for testing size from 100 to 100 000 rows and 40 CSV files to test the groups distributions with fixed row size (10 000), we run the program over 100 iterations for each CSV file and we draw the results in logarithmic scale.

Single Thread:

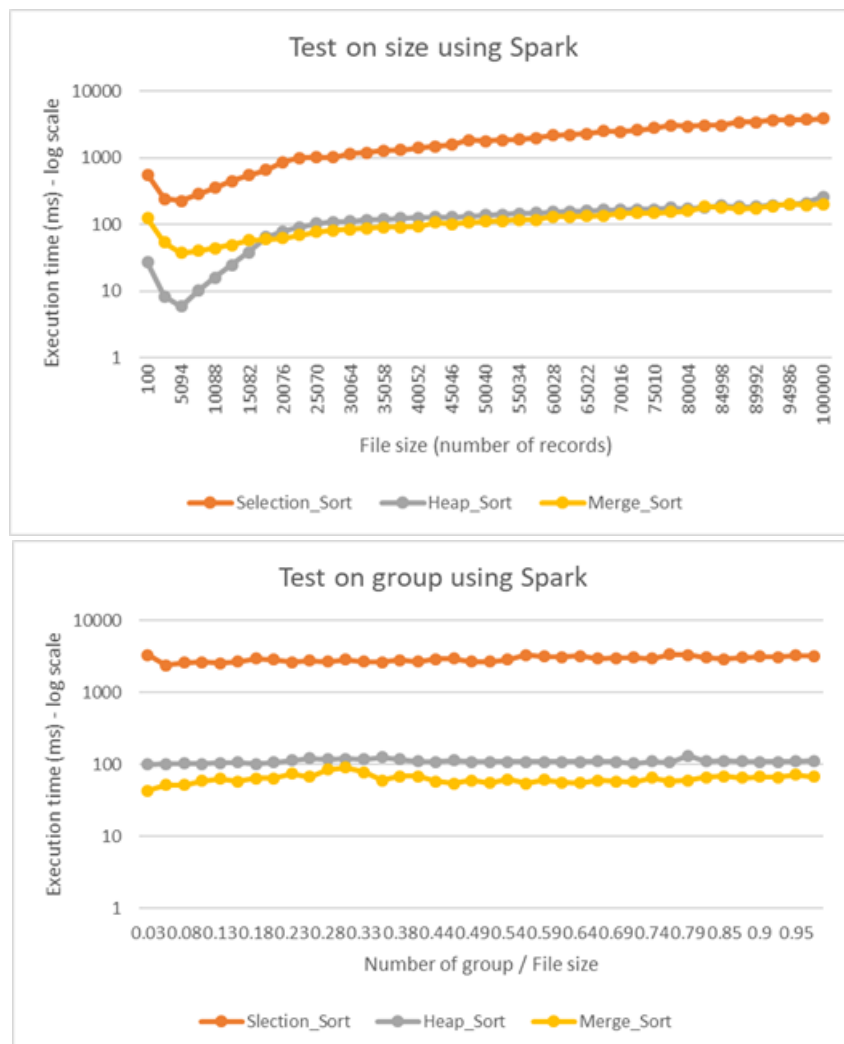


Multi-threads on JAVA:





Spark implementation:



- We observe that the Merge Sorting algorithm consumes less time execution over the three algorithms, the merge sort is slightly faster than the heap sort, but it requires

twice the memory of the heap sort because of the second array in the test on files size.

- For the group's distribution, we notice that heap sort is slightly better than the other two algorithms in multi-threads implementation. However, Merge sort remains better in single and spark implementations.

2.4. A word on ORDER BY

The ORDER BY operation on a single attribute is implemented as part of the GROUP BY sort routine.

One last interesting point is the queries that both GROUP BY and ORDER BY the same attribute. For those specific queries, you can either use the Sort group by, or use another group by routine, and then sort the results using the sort by routine.

The cost for using the sort by group is in $O(n \log(n))$ where n is the number of inputs. The cost for using the hash group by and then the order by routine is in $O(n + m \log(m))$. This shows that you might prefer to group first and then sort if the number of groups is little compared to the number of inputs.

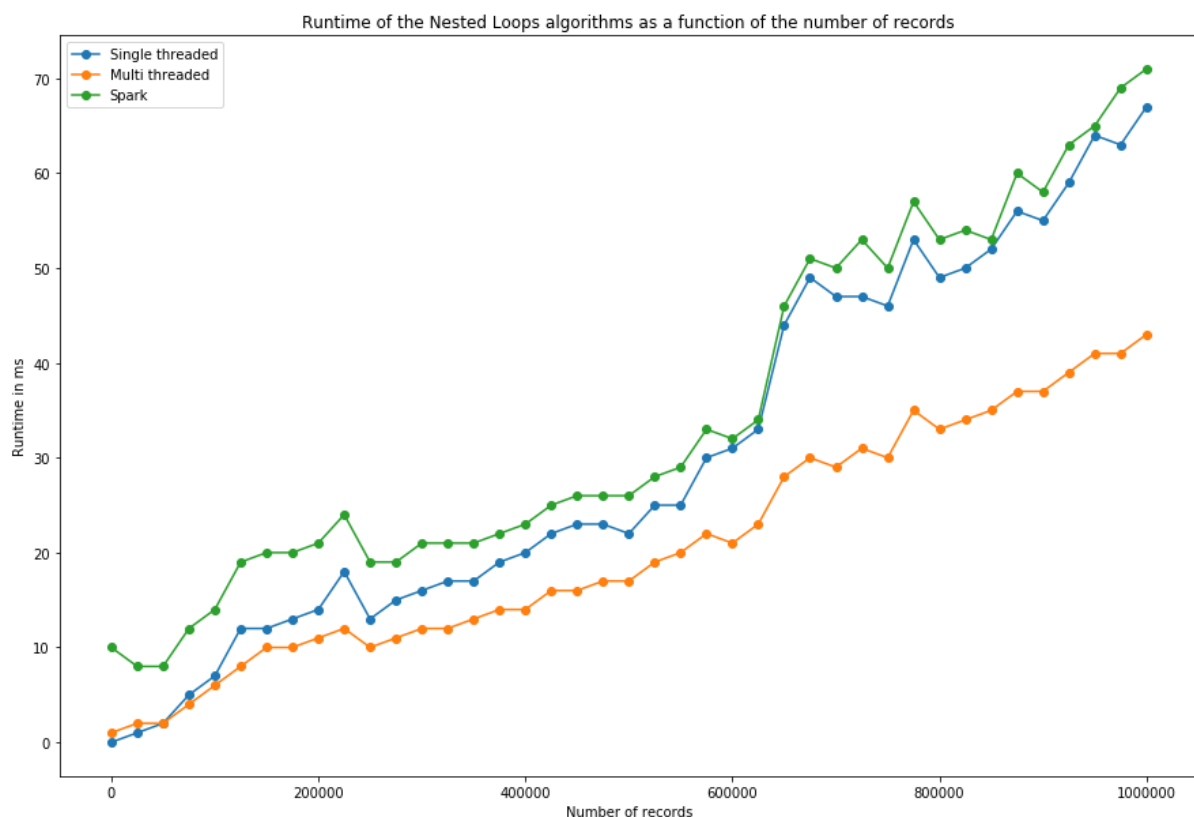
And as a last food for thoughts, one could imagine a heap-group by. This algorithm would be the same as the hash group by, but using a sorted array / heap structure instead of the hash table, providing lookup in $O(\log(m))$. The total cost for such an algorithm would be $O(n \log(m))$, and it could be competitive when m is smaller than n .

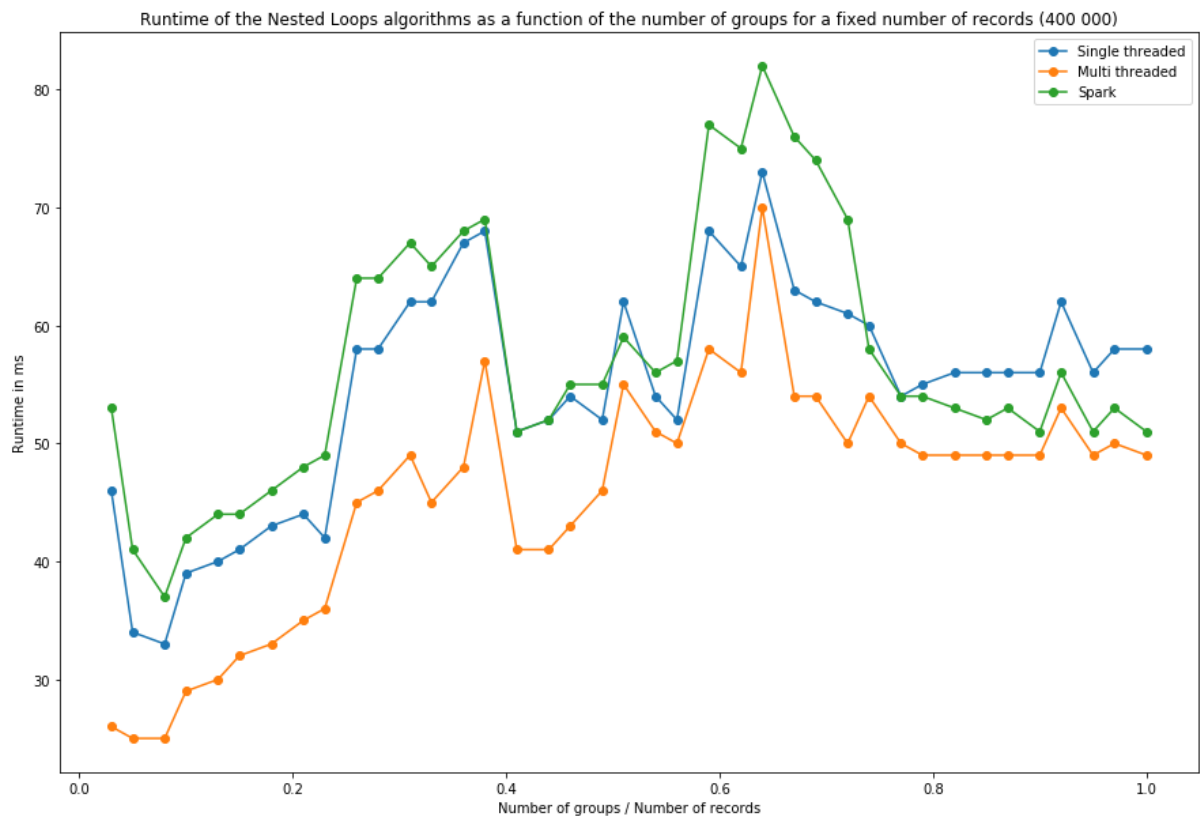
3. Results and Comparisons

Here you can find the results in terms of execution runtime for our three implementations of the GROUP BY statement associated with a COUNT. Each graph has been plotted for the three different architectures: Single-threaded, Multi-threaded and Multi-threaded with Spark.

Among the 2 graphs of each part the first one depicts the runtime as a function of the number of records for a fixed number of groups (10 in our case). The second one shows the runtime as a function of the number of groups for a fixed number of records (400 000 in our case). These tests have been realised on 2 directories containing files generated with the Python script *gen.py*. This script generates a list of files with a varying number of records of a varying number of groups, these files are CSV containing two columns of integers: the first represents the id of the record and the second is its group.

3.1. GROUP BY with Nested Loops





For the **size test**:

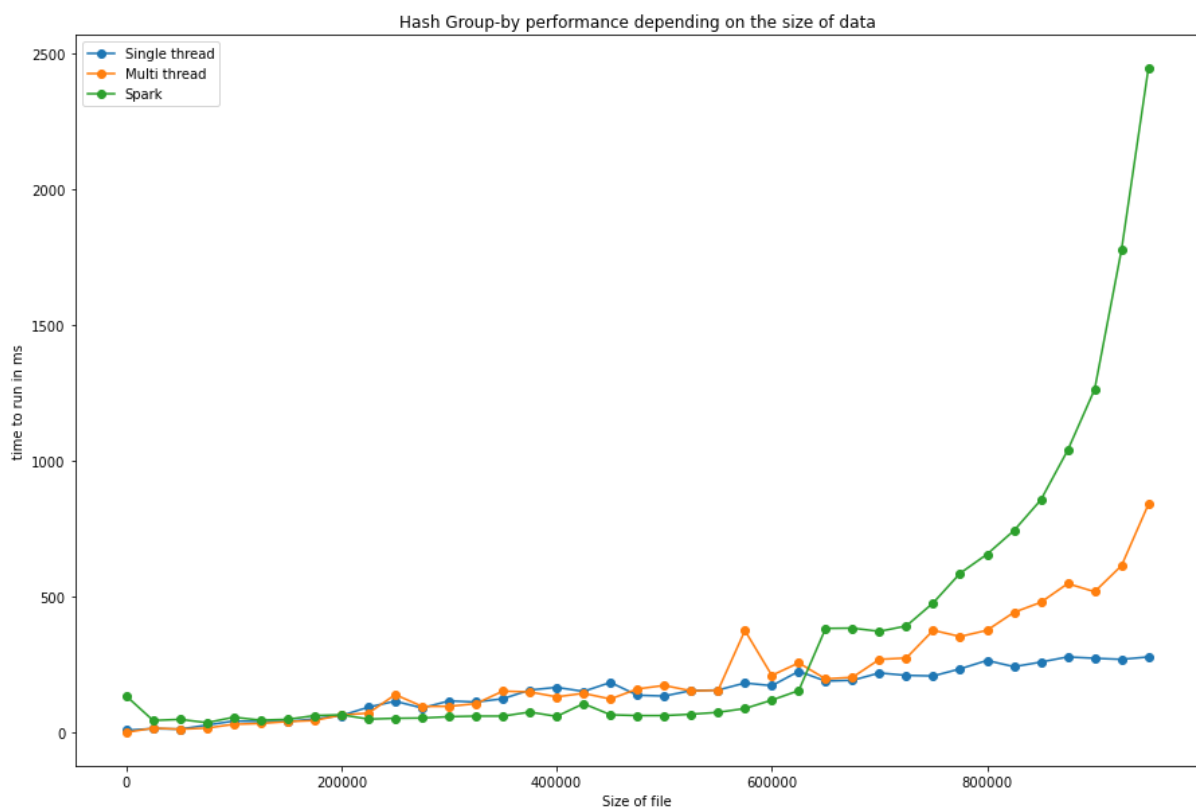
- We can see that the 3 curves are linear up to a number of records of around 650000, then there is a jump for the single-thread and Spark and then they increase faster. At the beginning, the multi-thread and single-thread curves are very close while Spark takes a little bit longer, but after around 100000 records the multi-threaded architecture becomes faster. Throughout the graph we notice that the spark is always the slowest, then comes single-threaded and finally multi-threaded which seems to be the best.
- As you can see the single-thread and Spark curves have pretty always the same behavior. The single-threaded curve is almost a copy of the Spark one but translated down. It may be linked with the fact that our multi-threaded implementation has been conceived as if it had four times more memory than the single one because of our memory size modelling the number of records that can be sent in the hashtable. Conversely, Spark always works in memory and the threads it creates have to share the same computer memory.
- The multi-threaded implementation seems to be the most efficient especially when the size gets bigger. We don't see the jump that occurs for the single-threaded architecture and Spark, it remains linear with the same coefficient for the entire test. In addition, it seems important to note that even if the multi-threaded implementation works with four times more memory than the single-threaded one the execution time is not divided by four. So one can wonder if the cost of this scale up is worth the decrease in execution time.

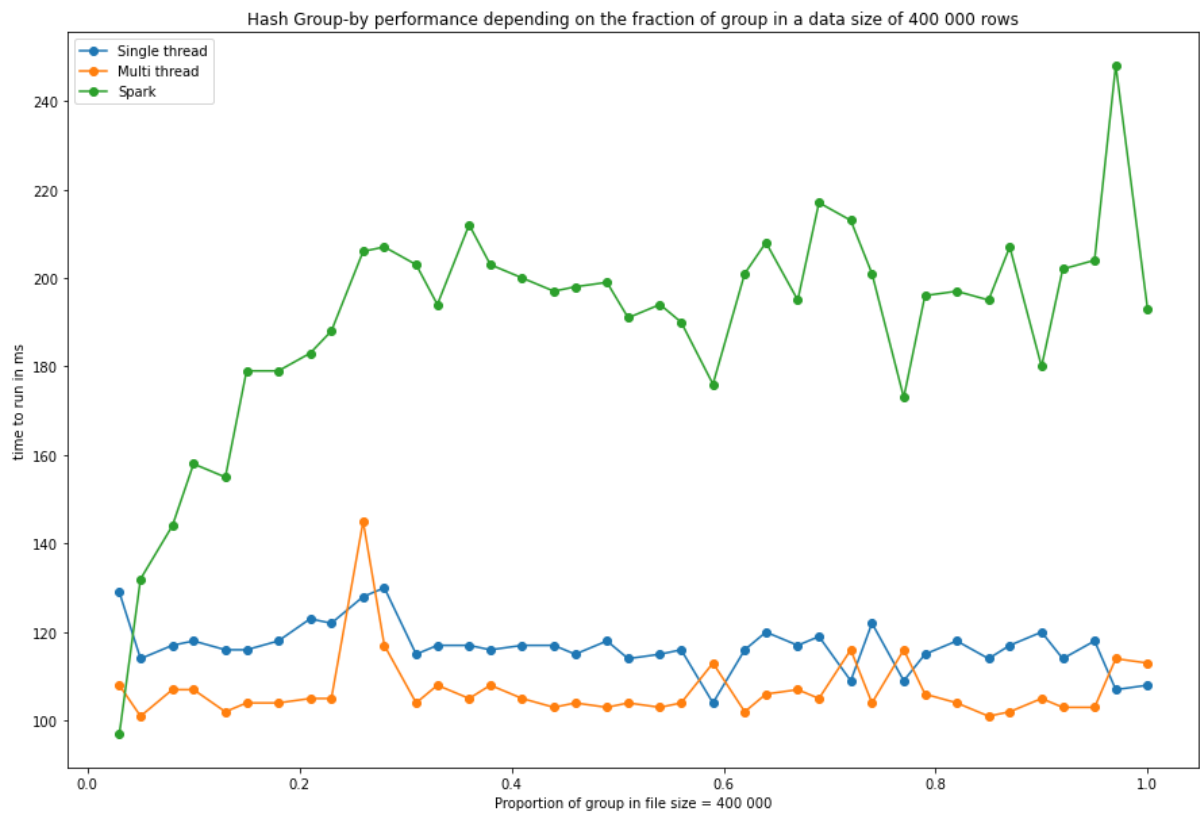
For the **group test** :

- Throughout the graph we notice that the spark always takes the longest time to execute, then comes single-threaded then finally multi-threaded which is the fastest. The execution time seems to increase quite linearly (if we don't think about the two bumps) before 0.8 groups for one record and then it remains quite stable.
- All implementations seem to behave similarly but with a different offset : their curves look the same but are translations. As for variation in the number of records, Spark and single-threaded implementations are close from each other, when the multi-threaded one is a bit faster.
- We can notice two bumps around 0.3 and 0.7 groups per record which are present for the three architectures. It is hard to know where they come from because they could be linked with the distribution of our groups but also the fact that for these numbers of groups our memory is overflowed and has to be put in CSV files.

3.2. GROUP BY with Hashing

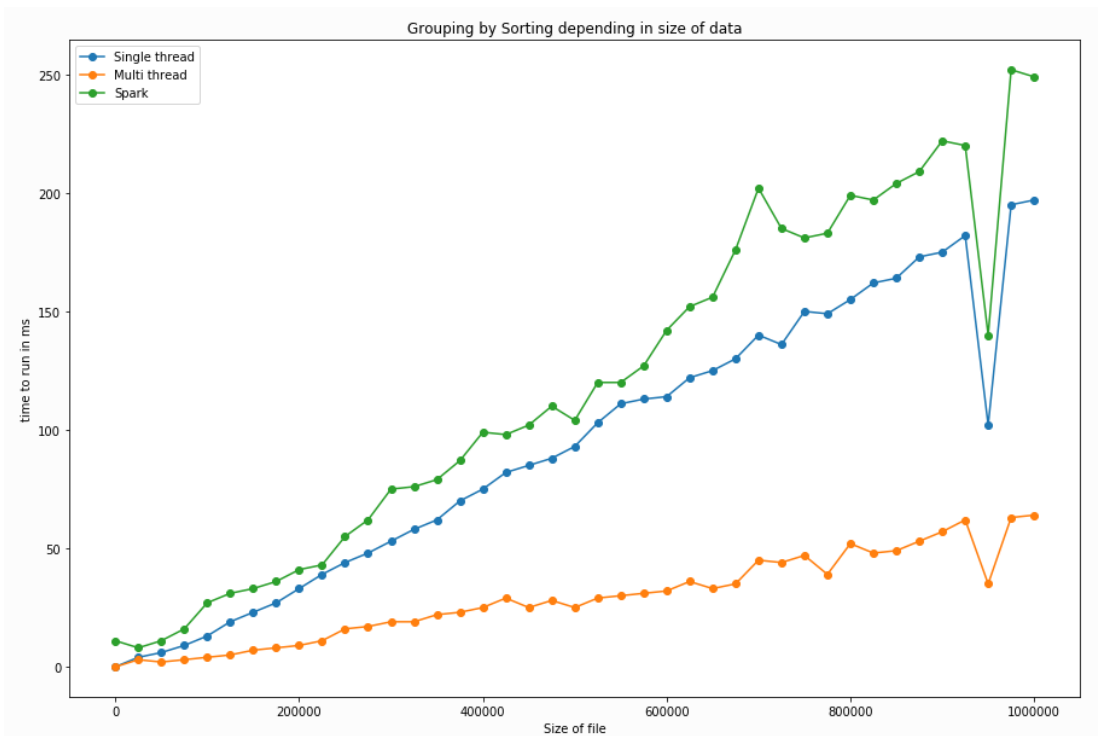
Below you can find the results of the tests for the two folders (**size** and **group**).

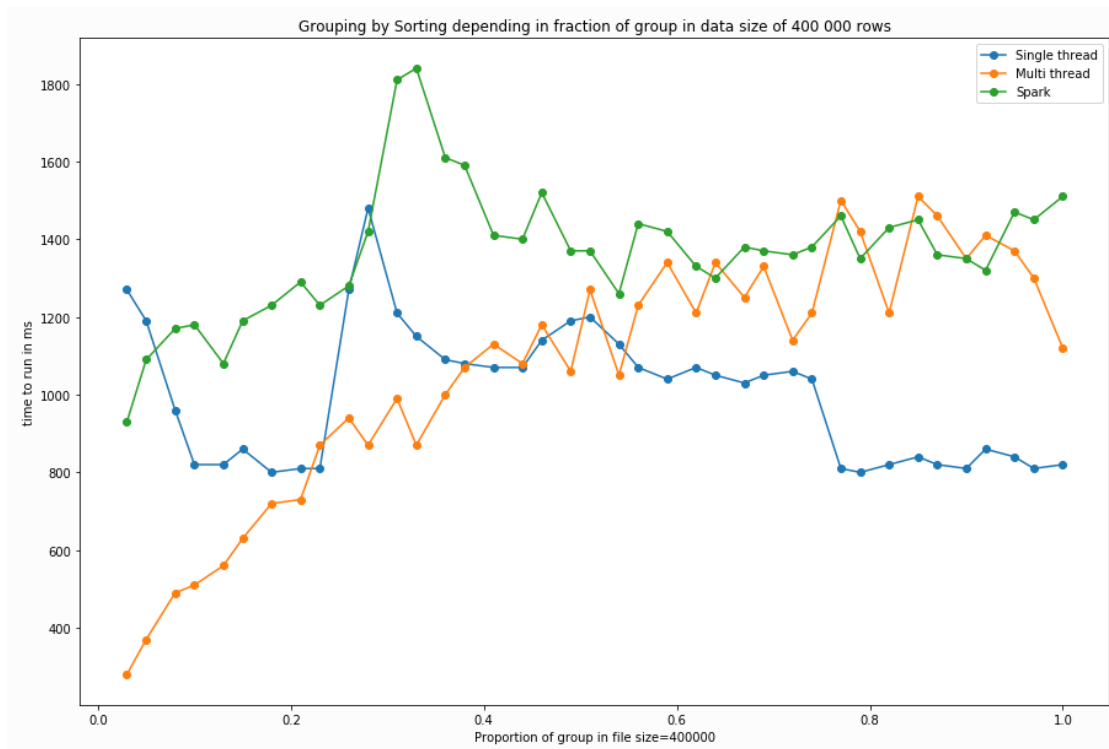




3.3. GROUP BY with Sorting

The following graphs show the group by sorting results on **size** and **group** folders):





For the **size test**:

- We notice that all three graphs have almost the same behavior, they are quite linear at the beginning, with a drop at the same CSV file of size 949 986, it's quite normal because we have just two groups, one record that represents one group and what left are in the second group, hence they are easily recognized by the three modes.
- At the beginning, the single threaded and Spark are quite near with more stability for the single mode, the multi-threaded outperforms others since it consists of processing data simultaneously and depends on hardware performance.
- Spark is surprisingly slow compared to single threaded, we can justify this by the nature of Spark workflow, it is meant to work on multiple computers, however in our case we run its tests just on one machine with 4 threads.

For the **group test**:

- We notice that in the beginning of the graphs the multi-thread in Java is faster than single. Which means that with low variation of groups (lower than 50% of the size of the file) multi-threads implementation is better and with more variation single becomes better. This can be explained by the fact that in each thread there is less aggregation, which means that at end we will have a table with many groups that need that may have a count of 1, in this case the data doesn't need to be split between threads
- We notice that the spark implementation is always slower that the two implementations, this can be due to the Spark process in itself.

3.4. Comparison between algorithms

3.4.1 Our results

It is complex to compare the three algorithms, as they do not exactly perform the same tasks. Most notably, the GROUP-BY with Nested Loops is designed to handle out-of-core execution for when the groups do not fit in memory. This is the only one to offer such an advantage, but the additional I/O to read and write the spill pages increases the time cost. The GROUP BY with sorting is designed for an input that fits in memory; the GROUP BY with hashing for groups that fits in memory.

Overall, regarding the graphs for the three techniques one can still deduce some informations :

- Generally the multi-threaded architecture seems to have better performance than the single-threaded and the Spark ones. It is quite logical to have a smaller execution time than single-threaded algorithms but it is more surprising for Spark. We can explain this by the way we have used Spark : Spark is supposed to be optimized to run on several computers with distributed calculus and for us it only runs on one computer with Spark initializations times.
- GROUP BY with Nested Loops is the only one that seems to have an overall linear behavior when we increase the number of groups for a fixed number of records. Indeed the GROUP BY with Hashing and GROUP BY with Sorting doesn't seem to have an increasing execution time when the number of groups increases. It may be linked with the fact that Nested Loops involves loops over the groups while the other algorithms don't.

3.4.2 Theoretical and Real world results

Real word databases, such as PostGreSQL mainly implement two variations of the group by routine. The first one is the loop with page spilling, same as our implementation, and the second one is the out-of-core sorting. We only have an in-memory version of the sorting routine. The choice between the two is made depending on whether the output groups will fit in memory or not. If the groups fit, use the loop version, if not use the sorting version. The loop version will have a theoretical complexity of $O(n)$ where n is the number of input pages, while the sort version is in $O(n(\log(n)))$.

References

- [1] Graefe G. (June 1993). Query evaluation techniques for large databases *ACM Computing Surveys*, Vol.25, No.2.
- [2] Larson P.A (December 1997). Grouping and Duplicate Elimination : Benefits of Early Aggregation, *Technical Report MSR-TR-97-36*.
- [3] Cieslewicz J. Ross K. (September 2007). Adaptive on Chip Multiprocessors. *VLDB Endowment ACM 978-1-59593-649-3/07/09*.
- [4] Ross K. Ye Y. (June 2011). Scalable aggregation on multicore processors. *DaMoN '11: Proceedings of the Seventh International Workshop on Data Management on New Hardware*
- [5] Shatdal A. Naughton J. (May 1995). Adaptive parallel aggregation algorithms. *ACM SIGMOD international conference on Management of data*.
- [6] Wen J. Borkar V. Carey M. Tsotras V. (October 2013). Data Intensive Applications: A Performance Study. *arXiv:1311.0059*
- [7] Muller I. Lacurie A. Sanders P. Lehner W. (June 2015). Cache-Efficient Aggregation: Hashing Is Sorting. *ACM SIGMOD international conference on Management of data*.
- [8] Alturani I. Alkharabsheh K. (2013). Grouping Comparison Sort. *Australian Journal of Basic and Applied Sciences*