1. Explanation of the Algorithm.

Two-Phase Merge Multiway Merge-Sort (TPPMS) is basically when the size of input data is larger than the memory size, it helps to sort without overflowing of the memory. In this project, we are given 2 data sets, assuming we call it input1 and input2. From this data, we need to sort all of them regardless of which data sets and make one output with all sorted lists. When we make an output, we also need to remove the duplicates if there are the same employee ID, and if it's the case, we only keep the most recent record.

For sorting algorithms, the basic concept is the same as TPMMS(two-phase Multiway Merge-sort). Firstly, we fill the blocks into memory blocks until there is a remaining space. If the job is done, we make a sorted list; otherwise, we store in a temporary space in the disk (and this is a 1 run), and keep going until all data has been read. From these sub-sorted lists, we merge them and make an output, which ideally chooses the last sorted list (means the smallest list). For this project, we also have one more task to do which is removing the duplicate.

Therefore, Phase 1 means sorting operation with given data, as explained. We read 2 files, read whole things and write in the sorted sublists. After that, in Phase 2, we read again, and merge all of them while removing the duplicates, and write back. The professor suggested another possible way, which we read 2 files separately, and at Phase 2, we merge them together; however, we do not see a lot of performance difference in this case.

2. Implementation of the Algorithm.

For our implementation of this project, we use arguments to identify input and output files. For instance, <<-i C:\dev\dat\sample_01.txt C:\dev\dat\sample_02.txt -o C:\dev\dat\sample_output.txt>> commands determines the 2 input files with argument -i followed, and -o for output file.

We also have a file generated function, which automatically generates arbitrary employee records, with -f for file generation, -r for the number of rows, and -x for a multiplier of these rows. This function is very helpful since we can create any kind of instances.

There is a special argument -m, which is deciding how many records per list. It means how much memory size we want to assign for the created sorted list. Because we cannot fully use this memory only for sorting operations since some memory is required to run the program. In this way, we avoid overflow issues from the memory, more specifically, garbage collector overhead limit exceeded error. So, users can assign % of memory size which will be used for the sorting operation and if users do not put anything, it gives 10% of memory by default.

There is an Employee class, which stores the object of Employee when we read from the file, and this only has 3 attributes, ID, date, and data. Because if we separate them all attributes such as address or department, the performance becomes too slow (around 2 times slower), so we decided to reduce to have only 3 attributes. That is enough for sorting algorithms since sorting requires the only ID, and when we make a result output, we need to remove duplicate but we only need to compare the recent records.

EmployeeFile java file is used for storing static values, such as the size of Date, size of ID, Block Size, … etc. EmployeeFileReader file helps to read the records from the given data sets and EmployeeFileWrite file helps to write back to make temporarily sorted sublists or to make a resulting output file. They both use BufferStream (BufferInputStream and BufferOutputStream) and read/write with given input or output. Both files keep track of how many records, Disk I/O, and Block I/O there are.

ExecTimer is used for recording the processing time and file generator is making sample data as explained above.

2.1 TPMMS-based duplicates elimination algorithm implementation

This section provides a detailed overview of the implementation of TPMMS algorithm applied to a task of duplicates elimination from a relation that cannot be fit into main memory.

An algorithm implementation is realized in Java programming language and is based on concepts of object-oriented programming (OOP). Main functionality of the algorithm is contained within class `Tpmms`. Additionally, there are hardware abstraction classes (HAL) `EmployeeFileReader`, `EmployeeFileWriter` and a few auxiliary classes. HAL classes are utilized to create abstraction of a physical hard disk with specified characteristics, such as a block size; at the same time HAL classes provide clear and concise interface to interact with relation stored on a disk while hiding all the device-related details.

Both phases of TPMMS algorithm are implemented in method `Process()` of class `Tpmms`. Below we provide detailed overview of this method.

Phase 1

During this phase algorithm iterates though all the files provided in "`-i`" command line parameter. During processing, each file is read one record at a time. Here, a record granularity is an abstraction provided by `EmployeeFileReader.ReadRecord()`; internally `EmployeeFileReader` reads data from disk in blocks - one block at a time. It is worth noting that there is no means to directly control hardware, particularly from within Java. Every record read on a previous step is placed into memory - into a Tree-like structure (`TreeMap`), where ID is used as a key. Tree assures that all records within it are in sorted order. As there can be more than one record with given ID records are placed within the array in a Tree. In essence, this is a MultiMap. A priority queue could also be used for this purpose. Whenever the number of records in the tree is about to exceed maximum memory size for the TPMMS list, the content of a tree is saved into a file and a tree memory is freed. Ultimately, all the input files are saved into temporary ordered list files of specified maximum size.

Phase 2

Initially, reading of the first record of each list file is performed. Here again, `EmployeeFileReader` invoked in read operation is the only abstraction that hides implementation in which reading is done at block granularity. Every read record is placed into a priority queue. Priority queue uses a customized comparator `EmployeeComparator` which ensures that records in the queue ordered by ID and

DATE so that the smallest ID will be at the head, if more than one record has the same ID then the one with the most recent DATE will be at the head. It is important to note that records saved in the queue are wrapped into `EmployeeQueueContext` structure which bedside record itself contains reference to a `EmployeeFileReader` associated with list-file generated in Phase 1.

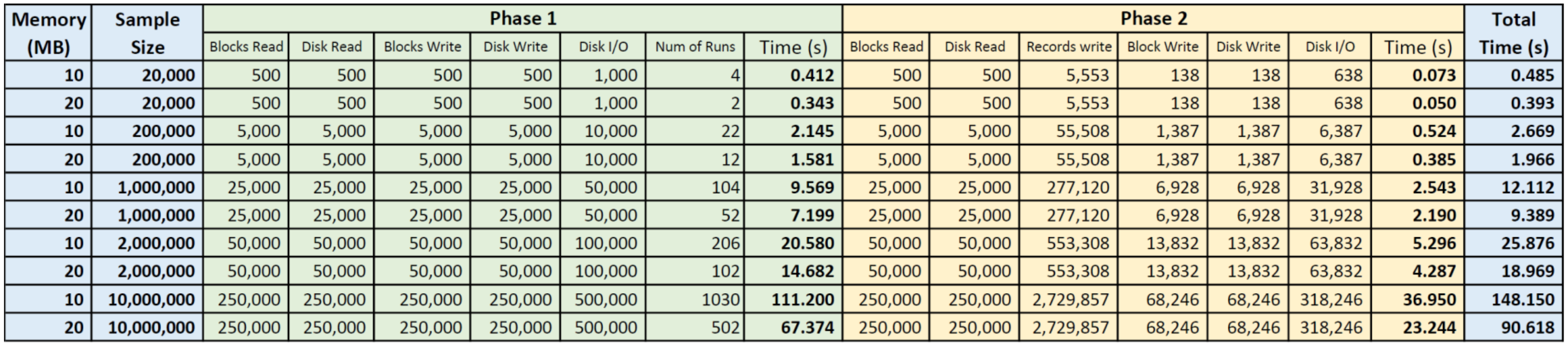
After priority queue has been initialized with the first record of every list, queue processing loop commences. On each iteration of the loop a head element of the priority queue is popped and saved into variable holding *current* record structure (`lTopValue`). Additionally, there is another variable that holds record structure which was read on a *previous* iteration (`lLastTopValue`) and initially initialized to NULL. Then, the current record is compared with a record read on the previous iteration - if $ID_{previous}<ID_{current}$ then the previous record is written to the result file; if $ID_{previous}==ID_{current}$ and the current record have more recent DATE then `lLastTopValue` is updated with `lTopValue`. In case if `lLastTopValue` is NULL it is simply initialized with `lTopValue`. At the end of the loop iteration a `EmployeeFileReader` reference from `lTopValue` is used to read another record from the corresponding list. This record which is wrapped into `EmployeeQueueContext` is then pushed into the priority queue. Loop runs till priority queue is not empty. Ultimately, the result file will contain sorted sequence of records from the input files in which all duplicates are eliminated.

3. Test results/Test cases.

In Figure 1, based on Memory (in MB) plus sample size, we have separated our observed result. The sample size means we have 20,000 tuples, 200,000 tuples, 1,000,000 tuples, 2,000,000 tuples, and 10,000,000 tuples as their record sizes. More specifically, for 20,000 tuples, it means we have a total sample size of 20,000, and it could be 10,000 + 10,000, or any other combinations. For our report, we divide them evenly. Mainly, as a result, we have the number of blocks, the total number of disk I/O’s to perform the task, and the processing time in seconds. To compare the performance easily, the internal main memory size is always fixed as default (for our implementation, it’s 10%). In other words, Figure 1 implicitly contains that internal memory size is 10%.

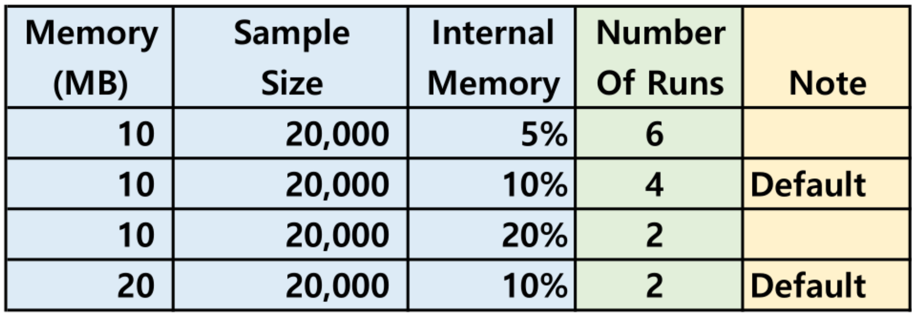
Figure 2 shows the difference between the number of runs by actual memory size with internal memory proportions. Also, in Figure 3, it shows the different processing time by different internal main memory size (= sorted sublist size). Depending on the percentage of the place it took, the performance is different as well.

The detailed explanations will be given in 5. Evaluation of the Performance.



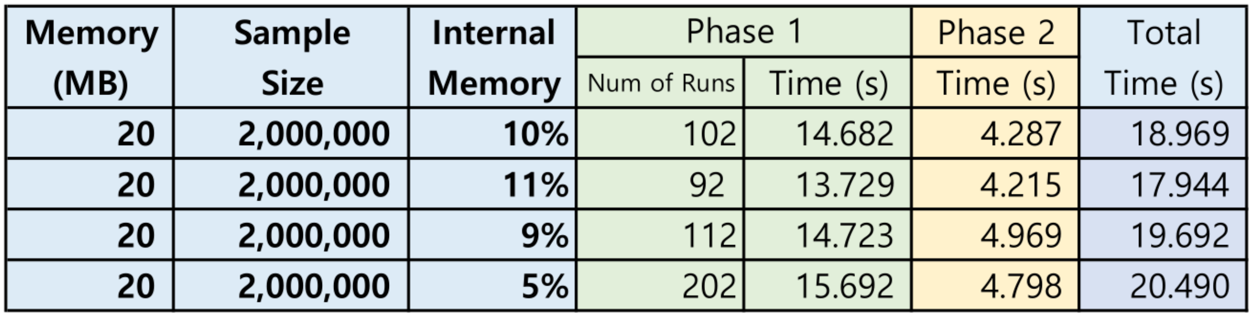
Memory (MB)	Sample Size	Phase 1							Phase 2							Total Time (s)
		Blocks Read	Disk Read	Blocks Write	Disk Write	Disk I/O	Num of Runs	Time (s)	Blocks Read	Disk Read	Records write	Block Write	Disk Write	Disk I/O	Time (s)	
10	**20,000**	500	500	500	500	1,000	4	**0.412**	500	500	5,553	138	138	638	**0.073**	**0.485**
20	**20,000**	500	500	500	500	1,000	2	**0.343**	500	500	5,553	138	138	638	**0.050**	**0.393**
10	**200,000**	5,000	5,000	5,000	5,000	10,000	22	**2.145**	5,000	5,000	55,508	1,387	1,387	6,387	**0.524**	**2.669**
20	**200,000**	5,000	5,000	5,000	5,000	10,000	12	**1.581**	5,000	5,000	55,508	1,387	1,387	6,387	**0.385**	**1.966**
10	**1,000,000**	25,000	25,000	25,000	25,000	50,000	104	**9.569**	25,000	25,000	277,120	6,928	6,928	31,928	**2.543**	**12.112**
20	**1,000,000**	25,000	25,000	25,000	25,000	50,000	52	**7.199**	25,000	25,000	277,120	6,928	6,928	31,928	**2.190**	**9.389**
10	**2,000,000**	50,000	50,000	50,000	50,000	100,000	206	**20.580**	50,000	50,000	553,308	13,832	13,832	63,832	**5.296**	**25.876**
20	**2,000,000**	50,000	50,000	50,000	50,000	100,000	102	**14.682**	50,000	50,000	553,308	13,832	13,832	63,832	**4.287**	**18.969**
10	**10,000,000**	250,000	250,000	250,000	250,000	500,000	1030	**111.200**	250,000	250,000	2,729,857	68,246	68,246	318,246	**36.950**	**148.150**
20	**10,000,000**	250,000	250,000	250,000	250,000	500,000	502	**67.374**	250,000	250,000	2,729,857	68,246	68,246	318,246	**23.244**	**90.618**

Figure 1. All test cases with information on Phase 1, Phase 2 with its processing time.



Memory (MB)	Sample Size	Internal Memory	Number Of Runs	Note
10	20,000	5%	6	
10	20,000	10%	4	Default
10	20,000	20%	2	
20	20,000	10%	2	Default

Figure 2. A different number of runs by Internal memory size.



Memory (MB)	Sample Size	Internal Memory	Phase 1		Phase 2	Total
			Num of Runs	Time (s)	Time (s)	Time (s)
20	2,000,000	10%	102	14.682	4.287	18.969
20	2,000,000	11%	92	13.729	4.215	17.944
20	2,000,000	9%	112	14.723	4.969	19.692
20	2,000,000	5%	202	15.692	4.798	20.490

Figure 3. Performance difference by Internal memory size

4. Evaluation of the Performance

As we observe in Figure 1, in general, as Memory has more size, compared with 10MB and 20MB, the sorting duration becomes much faster. This tendency becomes more observable once the sample size becomes larger. For instance, when we have a sample size of 10,000,000 tuples, the total processing time is 148.150 seconds with a memory size of 10MB; whereas, it only takes 90.618 seconds for memory size of 20MB.

This means the proportion of total procession time for 10MB divided by the time for 20 MB getting higher as the sample size increases. For example, for 20,000 tuples, 0.485/0.393 = 1.23; however, for 10,000,000 tuples, 138.772/92.394 = 1.50. This result means when there are 20,000 tuples, 20MB operations are only faster 1.23 times than 10MB operation; however, when there are 10,000,000 tuples, 20MB operations are faster 1.50 times than 10MB operation. So we claim that we require more sorting operation due to larger data size, then larger memory size is definitely more efficient. It is because some other operations in order to run the programs are unrelated to sorting operations but still require running such as some system-related process. We also prone to presume that in Java it is more efficient to operate with fewer TreeMap-s of larger size then with more TreeMap-s of smaller size.

Notably, the disk IO for writing back to the disk is decreased because when we write to the disk, we have removed some duplicates from the sorted list. Therefore, the disk I/O for Phase 1 and Phase 2 are not necessarily the same unless there is no duplicate. Also, the number of runs is different by actual memory size.

Ultimately, we find it is important to note that our tests were carried out on a modern hardware such as solid-state devices (SSD) with large internal cache. Furthermore, modern operating systems totally isolate physical properties of hardware from processes running in a user space. As such, performance effects that “classic” DBMS are expected to exhibit might not be this noticable in our study. We presume that expected performance penalties could have been observed in case of significantly larger relation.

From Figure 2, as we observe, the number of runs (number of sorted sublists) are varying by Internal memory size and actual memory size. For example, by default internal memory proportion (10%), 10MB creates 4 sublists and 20MB creates 2 sublists with 20,000 tuples; however, it is also depending on internal memory proportion, as it increases, the number of runs is decreased, and as it decreases, the number of runs is increased.

From Figure 3, we also compare the performance by Internal memory size. By default, it takes 18.969 seconds, where when it’s 11%, it takes only 17.944 seconds. If the internal memory size is less, then the processing time is also increasing such that 9% for 19.692 seconds and 5% for 20.490 seconds. Please note that this factor usually affects to Phase 1, but not Phase 2, since Phase 2 does not deal with a number of runs, so that’s why it is possible that 5% is faster than 9% (4.798 seconds vs 4.969 seconds), but Phase 1 has a clear difference. Also, if we increase to 20% for example, the processing time becomes too slow (takes more than 4 minutes with given 2,000,000 tuples), and this is happening because 80% of Java operations are not enough so it needs to wait for previous operations; for example, the writing operation takes a longer time. Another note is that Disk I/O and Block I/O are unchanged, regardless of the internal memory size.