**COMP 6521**

**Implementation Project**

**Team Members:**

Xiyue Li - 40048161

Natheepan Ganeshamoorthy - 29335838

1. Description of your code and architecture designs
   1. Tuple.java: data structure represents one input record.
   2. ProcessedTuple.java: data structure represents one output record (with computed order number and last date of order).
   3. Block.java: data structure represents one memory block containing at most 9 tuples.
   4. Reader.java: wrapper class for BufferedReader to read one or multiple blocks.
   5. Writer.java: wrapper class for BufferedWriter to write one or multiple blocks.
   6. Main.java: main part for TPMMS algorithm implementation.
      1. readAndSort: Read T1 block by block until buffer is full, and sort using quick sort algorithm. Then write to a file, and repeat until T1 is read. Then, repeat for T2.
      2. merge: Take sub-lists produced by sort phase as input files. Read the sorted lists based on memory limit and reserve one memory block as output. It will produce merged files (runs) every pass. The last pass will output only one file.
      3. processTuples: sequentially iterate the output file produced in merge phase. Merge the tuples with same client id into one and compute the number of orders and last date of order. Results are outputted into a txt file with tag “processed”.
   7. CreateData.java: randomly generate input records.
2. Highlights of your implementation features

* Using CreateData.java, users can create the required number of records for Table T1 and Table T2.
* Then, each text file is read 9 lines at a time, and each input record is used to create a tuple object, and 9 tuple objects are placed in one block object.
* In the first phase of reading and sorting, the quick sort algorithm has been implemented.
* Next, we do the merge, and all files that were generated in first phase of reading and sorting will be used. Result will be one big text file containing all records from T1 and T2.
* Finally, the required processing will be done.

1. Performance results and analysis

The data collect from our experiments using memory sizes of 10Mb and 20Mb, as well as files sizes of 0.5, and 1 million records are shown in the table below.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Read and Sort phase | | Total | | | |
|  | **# Disk I/O** | **Execution Time (sec)** | **# Disk I/O** | **Execution Time (sec)** | **# Records** | **# Blocks** |
| T1 & T2 Size: 500K  Memory: 10Mb | 222224 | 344s | 661271 | 790s | 951391 | 105711 |
| T1 & T2 Size: 1000K  Memory: 10Mb | 444448 | 756s | 1312539 | 1668s | 1812785 | 201421 |
| T1 & T2 Size: 500K  Memory: 20Mb | 222224 | 49s | 661271 | 247s | 951391 | 105711 |
| T1 & T2 Size: 1000K  Memory: 20Mb | 444448 | 161s | 1312539 | 524s | 1812785 | 201421 |

From this table, we can see that the number of disk I/O during the sort phase for 0.5 million records remain the same when the memory is changed from 10Mb to 20Mb. The same observations are noted for 1 million records.

The execution time during the sort phase is 7 times faster for 0.5 million records with 20Mb compared to 10Mb. It is 5 times faster for 1 million records with 20Mb compared to 10Mb.

The number of disk I/O to create the resulting table T for 0.5 million records remain the same when the memory is changed from 10Mb to 20Mb. The same observations are noted for 1 million records.

The execution time to create the resulting table T is 3 times faster for 0.5 and 1 million records with 20Mb compared to 10Mb.