Report: Assignment 2 (part 1)

Implementing a Sorted File (part 1)

Group Members:

1. Smarti Pandey, UFID: 4323-9459
2. Wins Goyal, UFID: 7357-1559

Instructions for Execution / Code compilation and to run tests:

I. To extract folders

a. Extract the contents of the folder, named "SmartiPandey_WinsGoyal_p2.zip".
b. Open the terminal.
c. Navigate to the extracted folder.

II. To create *binary files* from *tpch* files

a) Perform following these commands, first:

```
$ make clean
$ make a1test.out
$ ./a1test.out
```

Note: Before executing the above commands, generate the *tpch* files using the *dbgen* program. If needed, also change *the tpch, the dbfile output* and *catalog directories*, in the *a1test.cc* file.

b) Post-executing above commands gives a menu-based interface, which helps test the code through following three options. These can be selected as needed.

```
1. load          ### reads a 'tpch' file & outputs a 'binary heap' DBFile
2. scan          ### reads records from an existing 'heap DBFile'
3. scan & filter ### reads records from an existing 'heap DBFile'
                      and applies the filters using a CNF predicate
```

c) Now, choose Option: "1" (load) to generate a *binary heap file* in the *db-files* directory and go to selecting the needed (relation) table from the list of all tables/relations. This selected table shall output a binary heap file named '<*table_name*>.*bin*' in the *db-files* directory.

III. To run the tests

a) This step is same as the above II.(a) article. Just replace the '*a1test.out*' with '*test.out*' in the above commands. The same 'note' as above follows here as well.

b) Post-executing above commands gives again a menu-based interface, which helps test the code through following three options. These can be selected as needed.

```
1. sort
2. sort + display
3. sort + write
```

c) Now, to perform the tests, provide appropriate inputs to the menu-based interface, table name selected, run-length and sort-order, as needed.

Summarized Documentation of the implemented methods and functions:

Three data structures are used by the *BigQ* class:

a) *WorkerThreadData*: This data structure supplies the input arguments to the worker thread.
b) *RecordComparator*: This data structure makes use of the *OrderMaker* class for comparing two records as per the given CNF sort order.
c) *PriorityQueueItem*: This data structure uses a priority queue to maintain the records' order after comparison.

The *BigQ* class:

- The class constructor (of *BigQ*) spawns a worker class that implements the TPMMS algorithm.

- The TPMMS algorithm does following tasks in order:
  - Initializes the *WorkerThreadData*,
  - Generates sorted runs of run-length number of pages,
  - Merges these sorted runs into a final sorted output,
  - Executes the cleanup of the intermediate files.

  These intermediate files are generated during the sorting process.

- We have implemented the following methods and functions within the *BigQ* class:

a) *newThreadInitialize*: This initialize the pthread.

b) *bufferPageRecordAdd*: This checks how many records are being appended to the page in the file.

c) *addRecDSMerge*: After QuickSort has done its job, this function gets first (top) record of every first page from a number of pages in all the runs generated.

d) *divideRecToRuns*: This is the main function that implements the first phase of the algorithm. It calls the main helper function of Phase1. Briefly, it reads the remaining last pages from the pipe which did not completely fill the run length and puts the records from those pages in an unsorted array. Then, it calls the QuickSort function to perform the sorting as per the givenSortOrder and writes the sorted records into the files/pages. Eventually, the function terminates calling the merging process.

a) *QuickSort*: This takes the list of records to be sorted and a givenSortOrder as inputs. It calls the 'partition' function to get the right index of the random pivot selected from the record list. The function, then, makes recursive calls to itself until the records list is fully sorted as per the givenSortOrder.

e) *partition*: It is the key method for QuickSort function and takes as input a list of records to be sorted according to a given sortOrder. A record from the given list is randomly chosen as a pivot which will be placed at its correct index in the resulting sorted list. So, the method return this correct index, which will used by QuickSort function that called the partition function. The records that are less than the pivot according to the givenSortOrder will be to the left of the pivot and the rest will remain on the right of the pivot.

f) *combineRecordToRuns*: This implements the phase 2 of the algorithm by calling the function addRecDSMerge that gets the top records from all top pages in order for all the generated runs. This is done twice for getting the multiple records. The givenSortOrder of all those obtained records is input to startMerging function for the final merging process.

g) *phase1Helper*: This almost completes the phase 1 of the algorithm. It reads all the pages from the pipe until the run length is exceeded and puts the records from those pages in an unsorted array. Then, it calls the QuickSort function to perform the sorting as per the givenSortOrder and writes the sorted records into the files/pages. Eventually, the function terminates calling the merging process.

h) *writeRuns*: It appends the sorted runs to the pages of the file.

i) *initializePages*: It initializes the pages of the file to be written into it.

j) *startMerging*: It merges the sorted runs through initializing a priorityQueue data structure i.e. givenSortOrder. Top records from the different runs are taken one by one and compared according the order detailed in the priorityQueue so the records are placed accordingly. This results in a final merged sorted file.

k) *newThread*: It creates a new instances of the pthread initializer.

a) *freeUpData*: It executes the cleanup of the intermediate files.

l) *fetchRecordsfromTBL*: This is the testing function to the count the number of records in the file.

m) *fillBufferRecords*: It adds the records taken from the selected table file to the array to be sorted.

n) *writeSortedRunIntoFile*: It writes the sorted runs into the file. This happens in the Phase 1 helper, before the merging phase begins.

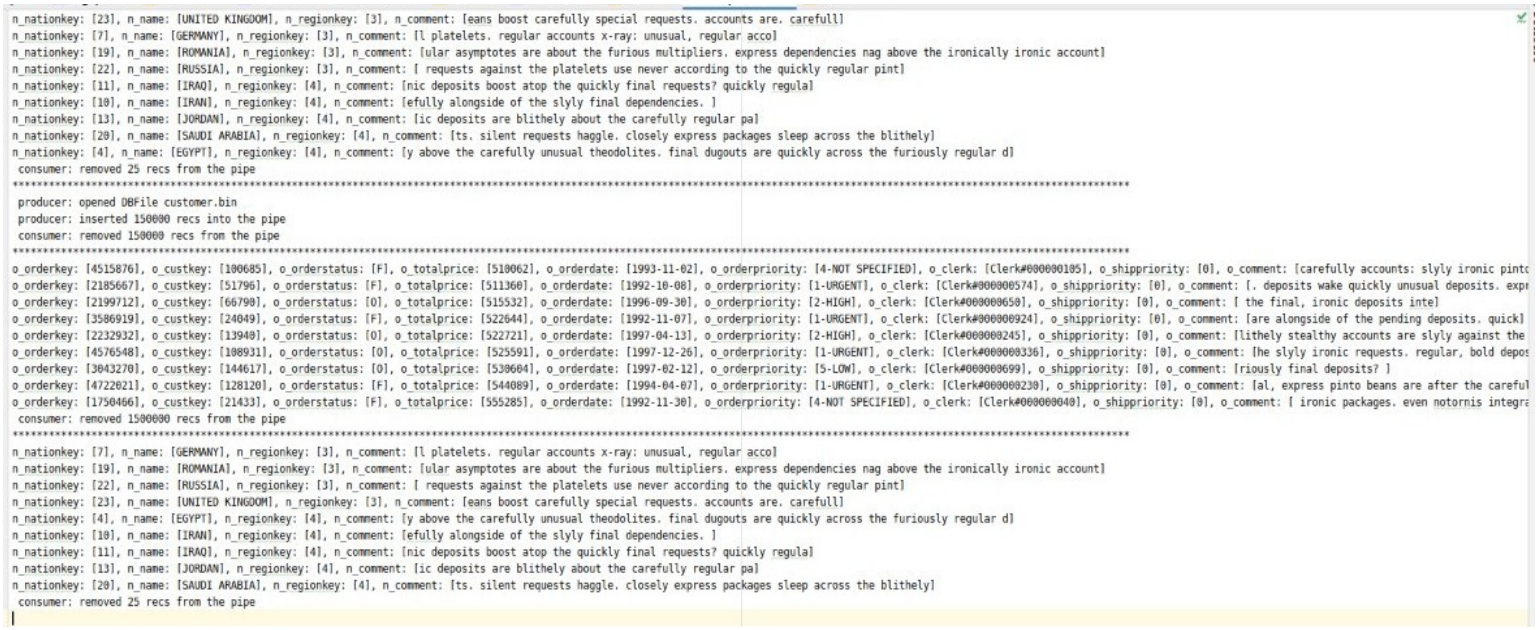
o) *resetBuffers*: It cleans the buffers to be used for next runs in the iteration in the Phase 1 helper.

Screenshots of Running All Test Cases

Perform following commands to run all the four test cases:

```
$ make clean
$ make test.out
$ ./runTestCases.sh
```

This generates output1.txt file. The following are the screenshots of this file.



The output of the output1.txt can be checked on the terminal, as shown:

```
smrati@smrati-XPS-15-7590:~/Downloads/a2-1test/a2test$ ./runTestCases.sh
 consumer: 25 recs out of 25 recs in sorted order
 producer: 100000
 consumer: recs removed written out as heap DBFile at customer.bin.bigq
 consumer: 150000 recs out of 150000 recs in sorted order
 producer: 100000
 producer: 200000
 producer: 300000
 producer: 400000
 producer: 500000
 producer: 600000
 producer: 700000
 producer: 800000
 producer: 900000
 producer: 1000000
 producer: 1100000
 producer: 1200000
 producer: 1300000
 producer: 1400000
 producer: 1500000
 consumer: 1500000 recs out of 1500000 recs in sorted order
 consumer: 25 recs out of 25 recs in sorted order
```

GTests:

- Perform following commands to executing the GTests:

```
$ make clean
$ make myGtest.out
$ ./myGtest.out
```

- Note: Before executing the above commands, generate the *tpch* files using the *dbgen* program. If needed, also change *the tpch, the dbfile output* and *catalog directories*, in the *a1test.cc* file.

a) *startPhase1*: This calls the 'initializePhaseOne' function in BigQ, to test whether the required the data structures are being properly created.

b) *freeUpdataTest*: This calls the 'freeUpData' function in BigQ and executes the cleaning of the unnecessary intermediate files creating during the sorting process.

c) *insertDataValTest*: This calls the 'bufferPageRecordAdd' function in BigQ and checks the count of how many records are being appended to the page in the file.

d) *countDataTest*: This calls the 'fetchRecordsfromTBL' function in BigQ. This checks and compares the count of records actually present in the selected tables and the count of records that are being consumed in the pipe during the scanning process. Both these counts should be equal to each other.

Following is the screenshots of all these four test cases in execution (terminal):