EECS 484 W19 Project 3: Database Structure

EECS 484 W19 Staff

Due: March 29th, 2019 at 11:55 pm EST

In Project 3, you will implement three database structures – linear hashing index, external merge sort and grace hash join using C++ language. You may achieve a total score of 100 points. The coding part is to be submitted to the Autograder system and you may receive feedback on submission, once per day.

This project is to be done in teams of 2 students; individual work will not be permitted for this project except in extreme circumstances with the written permission of the professor. Students may participate in the same teams as in previous projects, or they may switch partners; students do not need any special permission to switch partners. However, if students have started to work together on the project, they are not allowed to dissolve the team and work with others. Both members of each team will receive the same score; as such, it is not necessary for each team member to submit the assignment.

Attention: Do not make any submissions until the second member joins the team on the Autograder, otherwise they will be prevented from joining!

Project 3 is due on Friday, March 29th at 11:55pm EST. If you do not turn in your project by the deadline, or if you are unhappy with your work, you may continue to submit up until Tuesday, April 2nd 11:55pm EST (4 days after regular deadline). For late day policy please refer to the course syllabus.

The University of Michigan College of Engineering Honor Code strictly applies to this assignment, and we will be thoroughly checking to ensure that all submissions adhere to the Honor Code guidelines. Students whose submissions are found to be in violation of the Honor Code will be reported directly to the Honor Council. You may not share answers with other students actively enrolled in the course, nor may you consult with students who took the course in previous semesters. You are, however, allowed to discuss general approaches and class concepts with other students, and you are also permitted (and encouraged!) to post questions on Piazza.

Part 1. Implement Linear Hashing Index

For this part of the project, you will need to implement a linear hashing index. The index should be able to take integer keys as input, and store them in accordance with the linear hashing index design described in this spec.

1.1. Logical Overview

1.1.1. Usage

On starting the program, the user can type in any of these commands:

- print: this will print the current state of the index.
- exit/Ctrl-d: this will exit the program.

Anything else the user types will be considered a key that that user wishes to insert into the index. Keys and commands are separated by whitespace characters.

1.1.2. Inserting

By default, the user can only insert integer keys. The keys are read as strings, but should be converted to integers using the given function custom_hash(). These keys will be hashed with a very simple function:

$H(x) = x \% (N * 2^L)$

In this formulation, x is the key to be hashed, N is the initial number of buckets, and L is the current level. If the result of this formula would insert the key into a bucket that has already been split at the current level, then it should be rehashed at level L+1 instead.

NOTE: all custom_hash() does by default is convert strings to integers. Any other hashing is up to you.

The key should be inserted into the bucket in the index's directory which matches its hash value. Whenever a key is inserted into a bucket, it should become the last value in that bucket. So, if 1 is inserted into:
[2 _]
It becomes:
[2 1]

Or, if 5 is inserted into :
[3 2]
It becomes:
[3 2] -> [5 _]

If you are given a key to insert, and that key already exists in the index, you should not insert the new key.

If the key would be inserted into a full bucket, it should be inserted into the overflow bucket instead. If the key is inserted into an overflow bucket, then the bucket pointed to by "next" will split, and "next" will move to the next bucket.

NOTE: It is possible to modify the program to accept arbitrary strings as keys, instead of just integers. If you wish to enable this, then set the value USING_HASH in *constants.h* to true. This will change the hash functions to:

$H(x) = h(x) \% (N * 2^L)$

Where h(x) in this case is std::hash. This does not change the overall logic of the index at all. When we grade, we will always have this set to false; it's only an option so that you can see how the index works with arbitrary data types.

1.1.3. Splitting

When you split a bucket, all of the keys in the old bucket and all of its overflow buckets should be removed, and then inserted into the new buckets from smallest to largest.

NOTE: The integers are stored as strings, so "smallest" and "largest" should be defined using string comparison.

For example, if you split the bucket:
[5 2]->[4 3]
Then these entries are re-inserted into the index in the order 2, 3, 4, 5.
This split should result in these buckets:
[3 5]
[2 4]

1.1.4. Printing

When the user types "print" (without quotes), the index should be printed. Printing is performed entirely using LinHashIdx::print and Bucket::print, which have been implemented for you.

As such, you do not need to- and should not- modify any of the printing functions.

1.2. Code Overview:

1.2.1. LinHashIdx class

The files *LinHashIdx.hpp* and *LinHashIdx.cpp* implement the LinHashIdx class. This class has three main functions:

- insert: insert a key into the index, as described above. You will need to implement this function.
- contains: Check if the index contains a given key. Our code does not directly call this function, so you don't necessarily have to implement it, but it's useful for implementing insert.
- print: Prints the current state of the index. Do not modify this function; it is entirely implemented for you, and we will grade your implementation based on how it prints the contents of your index.
- custom_hash: This is not a class function: this just hashes input keys and turns them into integers. If USING_HASH is true, this function calls std::hash. If USING_HASH is false, then this function directly converts the input string into an integer, if possible. This function has been implemented for you. Do not change it.

This class also has several variable to keep track of:

- directory: The directory of pointers to Buckets holding key values. The hash value of a key determines its index in the directory, which points to the Bucket that should hold the key.
- next: the index in the directory of the next Bucket to be split. Initially 0.
- level: the current level of the index. Initially 0.

Note that the print function uses all of these variables, so they should all be kept up-to-date as insertions occur.

You are permitted and encouraged to add your own functions and variables to the class if you feel the need for them.

1.2.2. Bucket class

The files *Bucket.hpp* and *Bucket.cpp* implement the Bucket class. This class has two main functions:

- insert: insert a key into this Bucket or one of its overflow Buckets, as described above. You will need to implement this function.
- print: print the current state of this Bucket. You should not need to modify this function; it is entirely implemented for you.

This class also has its own variables:

- keys: the keys contained in this Bucket. Any keys added to this Bucket should go on the end, as described above. The size of this vector should never be more than MAX_BUCKET_SIZE
- overflowBucket: If the Bucket overflows, then this points to the overflow Bucket that keys should be placed into. If there is no overflow, then this should be set to *nullptr*.

As with LinHashIdx, note that the print function uses all of these variables, so they should all be kept up-to-date as insertions occur.

You are permitted and encouraged to add your own functions and variables to the class if you feel the need for them.

1.2.3. Constants

The constants.cpp file includes the constants that your code should use.

- BITSET_LEN: This is only used by the printing functions. You should not modify it or worry about it.
- USING_HASH: As described in the Inserting section above, setting this to true will allow you to store arbitrary strings instead of just integers. You don't need to ever change this, but you can if you want to see how the index works.
- INITIAL_NUM_BUCKETS: The number of buckets that the index has when it starts. This number should always be a positive power of 2.
- MAX_BUCKET_SIZE: The maximum number of elements a bucket can hold before overflowing. This number must be positive.

We will test your code with various values of INITIAL_NUM_BUCKETS and MAX_BUCKET_SIZE.

NOTE: When changing values in constants.h, you will want to recompile all of your files with make -B.

1.2.4. Other files

You should not need to modify any of the other files provided.

FINAL NOTE ON CODE STRUCTURE:

We generally expect that students will use the code structure provided. However, you may modify the code in any way you desire, as long as it compiles with our files and produces the expected output.

1.3. Building and running Part 1

You can work on the project anywhere, but as usual, we recommend doing your final tests in the CAEN Linux environment. You can build the project by running make in your terminal. You can remove all extraneous files by running make clean. You can start the program by running ./linhash.exe, at which

point you can type commands. If you want to test using a list of predetermined commands, you can put the commands in any text file (for example, test.txt), and run the file with ./linhash.exe < test.txt.

1.4. Files to submit for Part 1

You will submit the files LinHashIdx.hpp, LinHashIdx.cpp, Bucket.hpp, and Bucket.cpp. These files should compile with the given makefile and constants.h file.

Part 2. Implement External Merge Sort

In this part, implement both **pass 0** and a **merge sort pass** of B-way external merge sort, where B is the number of buffer pages in the memory.You should use the sort.zip package as starter code and check correctness by the print result to standard output stream at the end of each pass.

2.1. Logical Overview

2.1.1. Usage

On starting the program, the user can type in any of these commands:

- print: this will print the current state of the data pages.
- exit/Ctrl-d: this will exit the program.
- sort: this will start B external merge sort and trigger print at the end of each sorting pass

Anything else the user types will be considered a data value that that user wishes to insert into the index. Keys and commands are separated by whitespace characters, or entered on different lines.

2.1.2. Sorting

- Pass 0: We only sort data on each page individually. For $B \geq 2$ buffer pages, we DO NOT produce length of B sorted pages. Data are sorted just as integer values would be.
- Pass 1~N: We attempt to merge as many sorted lists of data pages as possible, by leveraging all the buffer pages in the memory. Proper assignment of number of input buffer page(s) and output buffer page(s) is vital to getting correct sort result. However, you don't have to explicitly define buffer page data structure. Instead, use any data structure you want. At any point, if data on one input page has totally been written out, we will flush in the next page in that sorted list (if any). If the last page of one list has been written out, its corresponding input buffer page remains empty, until all the other lists to merge have reached their end and we begin merging new lists.

IMPORTANT: You may use any existing functions (e.g. std::sort), as long as the sort results printed at the end of each pass is the same as the result of the above algorithm.

2.1.3. Example

The following example is to demonstrate the expected behavior of B-way external merge sort:
For the given constants, TUPLE_SIZE is 32, PAGE_SIZE is 64 and MEM_BUFFER_SIZE is 256.
There should be 2 data tuples on each page, and 4 buffer pages in the memory. Each tuple comprises just one integer attribute.

We first insert 7 data records as integers, and print. Here, page boundaries are denoted with | .
The 4th page ended up half filled, and there is no space-holding symbol.
DO NOT change the implementation for print function in the starter code!

```
>7 6 5 4 3 2 1
>print
| 7 6 | 5 4 | 3 2 | 1  |
```

Then, we start external merge sort by entering sort. At the end of each pass, current data pages are printed.
Depending on number of input data, number of passes and lines printed may change.

```
>sort
```

```
 -->pass0
| 6 7 | 4 5 | 2 3 | 1  |
 -->pass1
| 2 3 | 4 5 | 6 7 | 1  |
 -->pass2
| 1 2 | 3 4 | 5 6 | 7  |
```

We may continue inserting data to the current page range and perform sorting without an exit.
An exit is necessary when you want to clear old data and start a new range of pages.
At current step, we may call sort function again on the sorted data pages. The implementation is expected to run pass0 to pass2 once again even though all data are sorted, and no actual work is done.

```
>sort
 -->pass0
| 1 2 | 3 4 | 5 6 | 7  |
 -->pass1
| 1 2 | 3 4 | 5 6 | 7  |
 -->pass2
| 1 2 | 3 4 | 5 6 | 7  |
```

2.1.4. Printing

When the user types "print" (without quotes), the data pages should be printed. Printing is performed entirely using ExtSortRange::print and Page::print, which have been implemented for you.

Again, you do not need to- and should not- modify any of the printing functions.

2.2. Code Overview:

2.2.1. ExtSortRange class

The files *ExtSortRange.hpp* and *ExtSortRange.cpp* implement the ExtSortRange class. This class has three functions implemented and two functions to complete:

- load: Load a stream of input as integer data. Suggested not modifying.
- print: Prints the current state of the data pages range. Do not modify this function; it is entirely implemented for you, and we will grade your implementation based on how it prints the data pages.
- extMergeSort: Calls pass0Sort once and continue calling passMergeSort until the latter returns false (denotes the end of sorting). Suggested not modifying.
- pass0Sort: TODO implement pass 0 of B way external merge sort.
- passMergeSort: TODO implement pass 1~N of B way external merge sort.

This class also has several variables to keep track of:

- pageRange: the vector of pointers to pages that hold data. Initially empty.
- groupSize: length of list of sorted pages. Initially 1.
- BUFFER_PAGE: TODO use constant values in constants.hpp to define number of buffer pages

2.2.2. Page class

The files *Page.hpp* and *Page.cpp* implement the Page class. This class has three functions implemented and one function to complete:

- load: Try load one integer data to current page and return false if full. Do not modify.
- print: Prints this data page. Do not modify.
- getData: Return data on this page as vector of integers.
- sort: (optional) TODO sort data on this page. You may use this as a helper function.

This class also has several variables to keep track of:

- dataVec: the vector of data on this page.
- TUPLES_PER_PAGE: TODO use constant values in constants.hpp to define number of tuples each page can hold. This is maximum of length of dataVec.

2.2.3. Constants

The constants.cpp file includes the constants that your code should use to define BUFFER_PAGE and TUPLES_PER_PAGE.

- TUPLE_SIZE: Size of each tuple.
- PAGE_SIZE: Size of each page.
- MEM_BUFFER_SIZE: size of the entire memory.

You are encouraged to try out different tuple size, page size and memory size. Because we will test your code on various sets of constants, DO NOT hard code numbers for BUFFER_PAGE in *ExtSortRange.hpp* and *Page.hpp*.

FINAL NOTE ON CODE STRUCTURE:

We generally expect that students will use the code structure provided. However, you may modify the code in any way you desire, as long as it compiles with our files and produces the expected output.

2.3. Building and running Part 2

You can work on the project anywhere, but as usual, we recommend doing your final tests in the CAEN Linux environment. You can build the project by running make in your terminal. You can remove all extraneous files by running make clean. You can start the program by running ./externalMergeSort.exe, at which point you can type commands. If you want to test using a list of predetermined commands, you can put the commands in any text file (for example, test.txt), and run the file with ./externalMergeSort.exe < test.txt.

2.4. Files to submit for Part 2

You will submit the files ExtSortRange.hpp, ExtSortRange.cpp, Page.hpp, and Page.cpp. These files should compile with the given makefile and constants.h file.

Part 3. Implement Grace Hash Join

For this part of the project, you will need to implement Grace Hash Join (GHJ) algorithm. There are two main phases in GHJ, **Partition** and **Probe**. Your job is to implement **Partition** and **Probe** functions in **Join.cpp**, given other starter code, in which we could simulate the data flow in the level of records in Disk and Memory and perform join operations between two relations.

There is pseudocode in the appendix to the back of this spec on GHJ for your reference.

3.1. Starter code:

There are 6 main components for GHJ part, including Bucket, Disk, Mem, Page, Record, Join, along with main.cpp, constants.hpp, Makefile, and two text files for testing. Code overview and key points for each component are discussed below.

3.1.1. constants.hpp:

This file defines three constant integer values used throughout GHJ part.

- RECORDS_PER_PAGE: the maximum number of records in one page
- MEM_SIZE_IN_PAGE: the size of memory in the unit of page
- DISK_SIZE_IN_PAGE: the size of disk in the unit of page

3.1.2. Record.hpp & Record.cpp:

This file defines the data structure for emulated data record, with two main fields, key and data. Several member functions you should use in implementing GHJ include:

- partition_hash(): this function returns the hash value(h1) for the key of the record. To build the in memory hash table, you should do modulo (MEM_SIZE_IN_PAGE - 1) on this hash value.
- probe_hash(): this function returns the hash value(h2 different from h1) for the key of the record. To build the in memory hash table, you should do modulo (MEM_SIZE_IN_PAGE - 2) on this hash value.
- Overloaded operator ==: the equality operator checks whether the KEYs of two data records are the same or not. To make sure you use the probe_hash() to speed up the probe phase, we will only allow equality comparison on 2 records within the same h2 hash partition.

3.1.3. Page.hpp & Page.cpp:

This file defines the data structure for emulated page. Several member functions you should use in implementing GHJ include:

- loadRecord(Record r): insert one data record into the page.
- loadPair(Record left_r, Record right_r): inset one pair of data records into the page. This function is used when you find a pair of matching records from 2 relations. You can always assume the size of pages in records is an even number.
- size(): return the number of data records in the page
- get_record(unsigned int record_id): return the data record, specified by the record id, which is in the range [0, size).
- reset(): clear all the records in this page.

3.1.4. Disk.hpp & Disk.cpp:

This file defines the data structure for emulated disk. The only member function of disk about which you may be concerned is read_data(), which loads all the data records from text file into emulated "disk" data structure. Given the text file name, read_data() returns a disk page id pair <begin, end>, for which all the loaded data is stored in disk page [begin, end). ('end' is excluded)

3.1.5. Mem.hpp & Mem.cpp:

This file defines the data structure for emulated memory. Several member functions you should use include:

- loadFromDisk(Disk* d, unsigned int disk_page_id, unsigned int mem_page_id): reset the memory page specified by memory_page_id and load one disk page specified by disk_page_id into the memory page specified by memory_page_id
- flushToDisk(Disk* d, unsigned int mem_page_id): write one memory page specified by memory_page_id into the disk and reset the memory page. This function returns an integer that refers to the disk page id for which it writes into.
- mem_page(unsigned int page_id): returns the pointer to the memory page specified by page_id.

3.1.6. Bucket.hpp & Bucket.cpp:

This file defines the data structure, Bucket, which is used to store the output result of Partition phase. Each bucket stores all the disk page ids and number of records for left and right relations in one partition. Several member functions you should use include:

- add_left_rel_page(int page_id): add one disk page id of left relation into the bucket
- add_right_rel_page(int page_id): add one disk page id of right relation into the bucket
- Notice that the public member variables, num_left_rel_record, num_right_rel_record, indicate the number of left and right relation records in this bucket, which are maintained by add_left_rel_page and add_right_rel_page functions. These two variables will be helpful in Probe phase.

3.1.7. Join.hpp & Join.cpp:

This file defines two functions **partition**, **probe**, which compose of two main stages of GHJ. These two functions are the **ONLY** part you need to implement for GHJ.

- partition(): Given input disk, memory, the disk page ID ranges for left and right relation (Given pair <begin, end> for a relation, its data is stored in disk page [begin, end), where 'end' is excluded), perform the data records partition. The output is a vector of buckets of size (MEM_SIZE_IN_PAGE - 1), in which each bucket stores all the disk page IDs and number of records for left and right relations in one specific partition.
- probe(): Given disk, memory, a vector of buckets, perform the probing. The output is a vector of integers, which stores all the disk page IDs of the join result.

3.1.8. Other files:
Other useful files you may find helpful to look into or use include:

- main.cpp: this file loads the text file (accepted as command line arguments) and emulates the whole process of GHJ. In the last step, we provide you with a function that outputs the GHJ result.
- Makefile: run **make** to compile the source codes. Run **make clean** to remove all the object and executable files.
- left_rel.txt, right_rel.txt: these are two sample text files that store all the data records for left and right relations, for which you could use for testing. For simplicity, each line in the text file serves as one data record. The data records in the text files are formatted as:

```
key1 data1
key2 data2
key3 data3
... ...
```

3.2. Building and running Part 3

The GHJ part is developed and tested based on Linux environment with GCC4.9.4. You can work on the project anywhere, but as usual, we recommend doing your final tests in the CAEN Linux environment. You can build the project by running make in your terminal. You can remove all extraneous files by running make clean.bucket

To run the executable file, run command as ./GHJ left_rel.txt right_rel.txt, where left_rel.txt and right_rel.txt represent the two text file names which contain all the data records for joining relations.

3.3. Files to submit for part3

The only file you need to submit is **Join.cpp** which implements functions declared in Join.hpp. Please make sure you can compile and run the whole GHJ part in the CAEN Linux environment. Otherwise, you are liable to fail on the autograder testing.

3.4. Key reminders:

- For a complete algorithm to do GHJ please refer to the following pseudocode

Figure: Grace hash join.

```
/* Hash relation R */
foreach tuple r ∈ R do
   put r in bucket (output buffer) k = h1(r.A)
od
flush output buffers 1, ..., m to disk

/* Hash relation Q */
foreach tuple q ∈ Q do
   put q in bucket (output buffer) k = h1(q.B)
od
flush output buffers 1, ..., m to disk

/* Simple hash join for R_k ⋈_C Q_k */
for k = 1 to m do
   foreach tuple r ∈ R_k do
      put r in bucket no. h2(r.A)
   od
   foreach tuple q ∈ Q_k do
      foreach tuple r in bucket no. h2(q.B) do
         if r.A = q.B then
            put r ∘ q in the output relation
         fi
      od
   od
od
```

In the figure above, a "bucket" refers to a page of in-memory hash table.
For more information regarding simple hash join and in-memory hash table. Go to
http://wweb.uta.edu/faculty/sharmac/courses/cse5331And4331/Spring2005/Query%20Optimization/classnotes/chap12-Hash1slidePerPage.pdf

- Do not modify any starter code, except Join.cpp. Otherwise, you are liable to fail on the autograder.
- In the partition phase, use record class's member function partition_hash() for calculating the hash value of record's key. DO NOT make any other hash function on your own.
- In the probe phase, use record class's member function probe_hash() for calculating the hash value of record's key. DO NOT make any other hash function on your own.
- When writing the memory page into disk, you do not need to consider which disk page you should write to. Instead, just call Mem class's member function flushToDisk(Disk* d, int mem_page_id), which will return the disk page id it writes to.
- You can assume that the any partition of the smaller relation could always fit in the in-memory hash table. In other words, no bucket/partition in h2 hash_function will exceed one page. Or you can always assume, for all test cases we will be testing you on, there is no need to perform a recursive hash.
- In the partition phase, do not store record of left relation and record of right relation in the same disk page. Do not store records for different buckets in the same disk page.
- In the probe phase, for each page in join result, fill in as many records as possible. You do not need to do any optimization if one partition only involves the data from one relation.

- You do not need to consider any parallel processing methods, including multi-threading, multi-processing, although one big advantage of GHJ is parallelism.
- DO NOT call any print() function or cout any text in Join.cpp that you turned in.

Submission Instruction for Part 1, 2 and 3

Create a folder and copy over LinHashIdx class files, Bucket class files, ExtSortRange class files, Page class files (all class files both .hpp and .cpp) and Join.cpp.

cd into your folder and run the following command, and submit project3.tar.gz to the Auto-grader grading system:

```
tar -czvf project3.tar.gz LinHashIdx.* Bucket.* ExtSortRange.* Page.* Join.cpp
```

Please join team before making any submissions. Or you will need to email instructors on team issues.