EECS 484 Project 4: Database Structure

Due: Thursday, April 14th, at 11:55 pm Eastern

In Project 4, you will implement a database structure – Grace hash join – using the C++ language. You may achieve a total score of 70 points. The code is to be submitted to the Autograder system and you may receive feedback on your submission three times per day.

Attention: If you are working in a team, do not make any submissions until the second member joins the team on the Autograder; otherwise, they will be prevented from joining!

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.

1. Implement Grace Hash Join (70 pts)

There are 5 test cases on the Autograder and no hidden tests.
For this part of the project, you will need to implement a 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 section 1.4 on GHJ for your reference.
Please read through 1.4 Key reminders thoroughly before implementing the grace hash join!

1.1. Starter code:

There are 6 main components for the 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.

1.1.1. constants.hpp:

This file defines three constant integer values used throughout the project.

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

1.1.2. Record.hpp & Record.cpp:

This file defines the data structure for an emulated data record, with two main fields, key and data. Several member functions from this class that you should use in your implementation include:

- partition_hash(): this function returns a 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 a 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.

1.1.3. Page.hpp & Page.cpp:

This file defines the data structure for an emulated page. Several member functions from this class that you should use in your implementation include:

- loadRecord(Record r): insert one data record into the page.
- loadPair(Record left_r, Record right_r): insert 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. record_id is in the range [0, size).
- reset(): clear all the records in this page.

1.1.4. Disk.hpp & Disk.cpp:
This file defines the data structure for an emulated disk. The only member function from this class you need to be concerned about is read_data(), which loads all the data records from a text file into the emulated "disk" data structure. Given a 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) Refer to 1.1.7 for more information.

1.1.5. Mem.hpp & Mem.cpp:
This file defines the data structure for emulated memory. Several member functions you should use in your implementation 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.

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

- add_left_rel_page(int page_id): add one disk page id of the left relation into the bucket
- add_right_rel_page(int page_id): add one disk page id of the 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. These variables are automatically updated when add_left_rel_page and add_right_rel_page are called, respectively.
- get_left_rel(): returns a vector of disk page ids. These disk pages contain the records from the left relation that are mapped to this bucket.
- get_right_rel(): returns a vector of disk page ids. These disk pages contain the records from the right relation that are mapped to this bucket.

1.1.7. Join.hpp & Join.cpp:
This file defines two functions: **partition**, **probe**, which is made up two main stages of GHJ. These two functions are the **ONLY** part you need to implement for GHJ.

- partition(): Given input disk, memory, and the disk page ID ranges for the left and right relations (represented as an std::pair <begin, end> for a relation, its data is stored in disk page [begin, end), where 'end' is excluded), perform the data record 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 the left and right relations in one specific partition.

- probe(): Given disk, memory, and 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.

1.1.8. Other files:

Other files you may find helpful to look over 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 code. 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, which you can 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
... ...
```

1.2. Building and running

This project was developed and tested on a 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.

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

1.3. Files to submit

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

1.4. Key reminders:

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

Grace Hash Join

```
Input: relations $R$ and $S$, $B$ buffer pages
Output: relation joining columns $R.\rho$ and $S.\sigma$

// Hash relation $R$
foreach tuple $r \in R$ do
    put $r$ in bucket (output buffer) $h_1(r.\rho)$
flush $B - 1$ output buffers to disk

// Hash relation $S$
foreach tuple $s \in S$ do
    put $s$ in bucket (output buffer) $h_1(s.\sigma)$
flush $B - 1$ output buffers to disk

// Simple hash join for $R_k \bowtie S_k$
for $k = 1$ to $B - 1$ do
    foreach tuple $r \in R_k$ do
        put $r$ in bucket $h_2(r.\rho)$
    foreach tuple $s \in S_k$ do
        foreach tuple $r$ in bucket $h_2(s.\sigma)$ do
            if $r.\rho = s.\sigma$ then
                put $(r, s)$ in the output relation
```

In the figure above, a "bucket" refers to a page of the in-memory hash table. For more information regarding simple hash join and in-memory hash table. Go to https://rosettacode.org/wiki/Hash_join#C.2B.2B or course slides.

- Do not modify any starter code, except Join.cpp. Otherwise, you are liable to fail on the Autograder.
- In the partition phase, use the record class's member function partition_hash() for calculating the hash value of the record's key. DO NOT make any other hash function on your own.
- Similarly, in the probe phase, use the record class's member function probe_hash() for calculating the hash value of the 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 the 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 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. In other words, 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 a record from the left relation and a record of the 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 the join result, fill in as many records as possible. You must not make any optimization even 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 the Join.cpp that you turn in.

Submission Instructions

Please join your team before making any submissions if you are working in a team. You will need to email instructors on team issues.
Submit the **Join.cpp** file to the Autograder at autograder.io. The Autograder is up and ready for submissions.

The regular due date is Thursday, April 14th. Please refer to the course policy on the course website for information on the late submission penalty.

Appendix: sample output and explanation

The result of joining `left_rel.txt` and `right_rel.txt` provided in the starter code should look **similar** to the output below.

```
Size of GHJ result: 1 pages
Page 0 with disk id = 6
Record with key=0 and data=0l
Record with key=0 and data=0r
Record with key=1 and data=1l
Record with key=1 and data=1r
Record with key=1 and data=1l
Record with key=1 and data=11r
Record with key=1 and data=11l
Record with key=1 and data=1r
Record with key=1 and data=11l
Record with key=1 and data=11r
Record with key=1 and data=111l
Record with key=1 and data=1r
Record with key=1 and data=111l
Record with key=1 and data=11r
```

In the output above, each pair of records is a joined result. For example,

```
Record with key=1 and data=1l
Record with key=1 and data=1r
```

is the joined result of a record from left_rel.txt (notice how the data ends with an 'l') and a record from right_rel.txt (notice how the data ends with an 'r') where both records have the same key 1.

The order of these pairs (and the order within each pair) does not matter in the Autograder. Your code can output these 7 pairs of records shown above in a different order and still be correct.