EECS 484 F19 Project 4: Database Structure

EECS 484 F19 Staff

Due: Thursday, December 9th, 2019 at 6:00 pm EST

There are two parts in Project 4, you will implement a database structure – grace hash join using C++ language, and you will also use PostgresSQL to investigate query optimization. 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, three times per day; the Postgres part consists of short answer questions submitted through Google forms and will only be graded after the due date.

This project is to be done in teams of 2 students; individual work is permitted but not recommended for this project. 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 4 is due on Thursday, December 9th at 6:00 pm 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, December 14th , 6:00 pm EST (5 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 Grace Hash Join (70 pts)

This part is worth a total of 70 points, which can be earned on 5 public test cases and no hidden tests.
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.

1.1. 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.

1.1.1. 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

1.1.2. 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.

1.1.3. 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.

1.1.4. 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)

1.1.5. 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.

1.1.6. 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 you **should** manually maintain by yourselves. These two variables will be helpful in Probe phase.~~ don't need to anymore!

1.1.7. Join.hpp & Join.cpp:

This file defines two functions **partition**, **probe**, which is composed 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.

1.1.8. 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
... ...
```

1.2. Building and running

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.

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.

1.3. Files to submit

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.

1.4. 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 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

Please join team before making any submissions. Or 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.

Part 2 Postgres (30 pts)

The final part of this project involves using Postgresql's query optimization. It's worth a total of 30 points.

In this assignment, we will work with a small subset of an online e-commerce dataset called Dell DVD Store (http://linux.dell.com/dvdstore/). This dataset has information about customers, products, and orders. Although there are 8 tables in the dataset, our exercises will focus on the following two tables:

Customers (customerid, firstname, lastname, city, state, country, …)
Orders (orderid, orderdate, customerid, netamount, tax, totalamount)

We will be using PostgreSQL 9.2.18 (also known as Postgres) database system as it has very powerful tools for query performance analysis and optimization.

Instructions for logging into the Postgres server and insitailizing your database as well as some troubleshooting information can be found at:
https://docs.google.com/document/d/1L_GGltRACcl1FeAQQE9VpAxJP9yGvCDH-QvEBSGo5S4/edit?usp=sharing

To submit this part, please follow the instructions and fill out the answers in the Google form found at:
https://forms.gle/7Tbf2xzFix4UZ4HF6

No other submission is necessary for this part. If you are working in a group, make sure to indicate this in the form. Only one submission is necessary. You are free to re-submit as much as you want before the deadline.