EECS 484 Projects | p4-ghj

Project 4: Grace Hash Join

Worth	Released	Due
70 points	Date TBD	**June 15th at 11:55 PM ET**

Project 4 is due on **June 15th at 11:55 PM EST**. Please refer to the EECS 484 SP23 Course Policies for more information on penalties for late submissions, late day tokens, and sick days.

Introduction

In Project 4, we will implement a database algorithm – Grace Hash Join – using the C++ language. Instead of *utilizing* databases like in Projects 1-3, we will be *implementing* part of a database. This implies that this project does not require CAEN or SQL*Plus!

Submissions

This project is to be done in teams of 2 students (recommended) or individually. Be sure to create your team on the Autograder.

Honor Code

By submitting this project, you are agreeing to abide by the Honor Code: "I have neither given nor received unauthorized aid on this assignment, nor have I concealed any violations of the Honor Code." You may not share answers with other students actively enrolled in the course outside of your teammate, 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.

Starter Files

There are two main phases in GHJ, partition and probe. **Your job is to implement the partition and probe functions in Join.cpp.** You are given other starter files, which along with your code, will

simulate the data flow of records in disk and memory and perform a join operation between two relations.

Download the starter files (p4-starter_files.tar.gz).

There are 6 main components: Record, Page, Disk, Mem, Bucket, and Join. The files `constants.hpp`, `main.cpp`, `Makefile`, `left_rel.txt` and `right_rel.txt`, and `.clang-format` will also be used for testing and formatting. Code overview and key points for each component are discussed below.

Record.hpp and Record.cpp

These files define 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()` : 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()` : 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==` : this equality operator checks whether the **keys** of two data records are the same or not. To make sure you use `probe_hash()` to speed up the probe phase, we will only allow equality comparison of two records with the same h2 hash value.

Page.hpp and Page.cpp

These files define the data structure for an emulated page. Several member functions from this class that you should use in your implementation include:

- `size()` : returns the number of data records in the page.
- `empty()` : returns true if the page is empty.
- `full()` : returns true if the page is full.
- `reset()` : clears all the data records in the page.
- `get_record(unsigned int record_id)` : returns a data record, specified by the record id. `record_id` is in the range [0, `size()`).
- `loadRecord(Record r)` : inserts a data record into the page.
- `loadPair(Record left_r, Record right_r)` : inserts a pair of data records into the page. This function is used when you find a pair of matching records from two relations. You can always assume `RECORDS_PER_PAGE` is an even number.

Disk.hpp and Disk.cpp

These files define the data structure for an emulated disk. You do not need to use any member functions from this class in your implementation.

The only member function you need to be concerned about is `read_data(const char* file_name)`, which loads all the data records from a text file into the emulated "disk" data structure and returns a disk page id pair <begin, end>, for which all the loaded data is stored in the range of disk pages [begin, end). This function is used in the provided `main.cpp` file.

Mem.hpp and Mem.cpp

These files define the data structure for emulated memory. Several member functions you should use in your implementation include:

- `reset()` : clears all the data records in all pages in memory
- `mem_page(unsigned int mem_page_id)` : returns a pointer to the memory page specified by `mem_page_id`.
- `loadFromDisk(Disk* d, unsigned int disk_page_id, unsigned int mem_page_id)` : loads a disk page specified by `disk_page_id` into the memory page specified by `mem_page_id`.
- `flushToDisk(Disk* d, unsigned int mem_page_id)` : writes the memory page specified by `mem_page_id` into disk and resets the memory page. This function returns an integer that refers to the disk page id for which it writes into.

Bucket.hpp and Bucket.cpp

These files define the data structure for a 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 of one partition. Several member functions you should use in your implementation include:

- `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.
- `add_left_rel_page(unsigned int page_id)` : adds a disk page id of the left relation into the bucket
- `add_right_rel_page(unsigned int page_id)` : adds a disk page id of the right relation into the bucket
- Notice that the public member variables `num_left_rel_record` and `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.

Join.hpp and Join.cpp

These files define two functions: **partition** and **probe**, which make up the two main stages of GHJ. These two functions are the **ONLY** part you need to implement for this project.

- `partition(Disk* disk, Mem* mem, pair<unsigned int, unsigned int> left_rel, pair<unsigned int, unsigned int> right_rel)` : Given the disk, memory, and disk page id ranges for the left and right relations (represented as pair <begin, end>, where [begin, end) is a range of disk page ids), 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 of one specific partition.
- `probe(Disk* disk, Mem* mem, vector<Bucket>& partitions)` : Given the 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.

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 units of page

Other files

Other files you may find helpful to look over include:

- `main.cpp` : this file loads the text files and emulates the whole process of GHJ. It also outputs the GHJ result.
- `Makefile` : this file allows you to compile and run a test run of GHJ. See Building and Running.
- `left_rel.txt` , `right_rel.txt` : these two sample text files store all the data records for a left and right relation, 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:

```
1  key1 data1
2  key2 data2
3  key3 data3
4  ... ...
```

- `.clang-format` : this file aids with C++ formatting. You may choose to format your files in any way you choose, but this file offers a good starting point.

Building and Running

This project was developed and tested in a Linux environment with GCC 5.4.0 and C++14 (a few features are not supported in GCC 5.4.0). You can work on the project anywhere, but as usual, we recommend doing your final tests in the CAEN Linux environment.

To build the project and run the executable file, use the `Makefile`, where `left_rel.txt` and `right_rel.txt` represent the two text file names that contain all the data records for joining two relations.

```
$ make
$ ./GHJ left_rel.txt right_rel.txt
```

To remove all extraneous files, run

```
$ make clean
```

Grace Hash Join Pseudocode

Refer to the following pseudocode for a complete algorithm of Grace Hash Join:

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 ∈ R do
    put r in bucket (output buffer) h1(r.ρ)
flush B − 1 output buffers to disk

// Hash relation S
foreach tuple s ∈ S do
    put s in bucket (output buffer) h1(s.σ)
flush B − 1 output buffers to disk

// Simple hash join for Rk ⋈ Sk
for k = 1 to B − 1 do
    foreach tuple r ∈ Rk do
        put r in bucket h2(r.ρ)
    foreach tuple s ∈ Sk do
        foreach tuple r in bucket h2(s.σ) do
            if r.ρ = s.σ 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, visit Rosetta Code or the course slides.

Key Reminders

- Use the `Record` class's member functions `partition_hash()` and `probe_hash()` for calculating the hash value of the record's key in the partition and probe phases, respectively. **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, call the `Mem` class's member function `flushToDisk(Disk* d, unsigned int mem_page_id)`, which will return the disk page id it writes to.
- **You can assume that any partition of the smaller relation will fit in the in-memory hash table.** In other words, after applying the h2 hash function, **no bucket/partition will exceed one page. There is no need to perform a recursive hash.** Here, "smaller relation" is defined as the relation with the fewer total number of records.

- 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.**
- Do not make any optimizations even if one partition only involves the data from one relation.
- You do not need to consider any parallel processing methods, including multithreading and multiprocessing, although one big advantage of GHJ is parallelism.
- You may add your own helper functions, provided that `Join.cpp` still compiles with the partition and probe functions.

Submitting

The only deliverable is `Join.cpp`, which is worth 70 points. Submissions should be made to the Autograder. There are 5 public test cases on the Autograder and no private tests.

Remember to remove any print statements, as your submission will fail on the Autograder even if it compiles on CAEN.

Each team will be allowed 3 submissions per day with feedback; any submissions made in excess of those 3 will be graded, but the results of those submissions will be hidden from the team. Your highest scoring submission will be used for grading, with ties favoring your latest submission.

Appendix

The result of joining `left_rel.txt` and `right_rel.txt` provided in the starter files 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
```

```
15  Record with key=1 and data=111l
16  Record with key=1 and data=11r
```

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

```
1   Record with key=1 and data=1l
2   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 on the Autograder. Your code can output these 7 pairs of records shown above in a different order and still be correct. Your code can also have a disk id that is different from the one shown above and still be correct.

Acknowledgements

This project was written and revised over the years by EECS 484 staff at the University of Michigan. The most recent version was updated and moved to Primer Spec by Owen Pang.