COMS 4112: Database Systems Implementation
Project 2

In this project, you will be asked to implement parallel (multi-threaded) versions of in-memory hash joins and group-by aggregations. The goal of the project is to give you an idea of how large analytical queries run on modern database systems that execute most queries in main memory.

You are given the following relational schema:

```
orders(…, item_id, store_id, quantity, …)
items(id, price, …)
```

The id column in table items is the primary key of the table and is unique. Both tables would normally have additional columns but we omit them since they will not participate in the query.

You are given the following SQL query (let's name it "q4112"):

```
select avg(temp.avg_order_value) from
    (select avg(items.price * orders.quantity) as avg_order_value
    from orders, items
    where items.id = orders.item_id
    group by orders.store_id) as temp;
```

This query finds the average order value per store (not the general average across all stores).

The project is split in 2 parts with separate deadlines. Eventually (part 2), you will write code to execute the previous query. But first (part 1) you will write code to execute the previous query with the additional assumption there is only one store. Thus, the query will search for the order value average for that store only. The SQL query can be simplified for part 1 as follows:

```
select avg(items.price * orders.quantity)
from orders, items
where items.id = orders.item_id;
```

You are given the following files: http://www.cs.columbia.edu/~orestis/4112_proj2_src

q4112.h	Common header file describing the API.
q4112_main.c	Main function that generates data in memory and runs the query.
q4112_gen.o	Generate data for the query in memory (multi-threaded code).
q4112_nlj_1.c	Run the simplified query using nested loop join on a single thread.
q4112_nlj.c	Run the simplified query using nested loop join on multiple threads.
q4112_hj_1.c	Run the simplified query using hash join on a single thread.
Makefile	Compile 3 executables for running the simplified query.

The header file contains two functions used in the main function. One function is implemented in "q4112_gen.o" and generates the data for the query in memory. The other function executes the query and is implemented on multiple files. Each executable you create can only use one source file for executing the query. The main function and the generation code are common.

Part 1 (40%)

You have to implement and submit “q4112_hj.c” in C (or “q4112_hj.cpp” in C++), which is the multi-threaded version of “q4112_hj_1.c”. Your code must execute the simplified version of the query, like “q4112_hj1_.c”, “q4112_nlj_.c”, or “q4112_nlj.c”, and produce the correct average.

You will also run your code on CLIC machines with two 4-core CPUs with hyper-threading and 24 GB of RAM. There is a subset of CLIC machines under this specification. You will measure your code in these machines only. One example machine is “algiers.clic.sc.columbia.edu” but there are ~10 others as well. If you run the command “lscpu” on such a machine you will get:

```
Thread(s) per core:   2
Core(s) per socket:   4
Socket(s):            2
```

This shows that the machine has 16 threads: 2 CPUs, 4 cores per CPU, and 2 threads per code. If you run the command “free” on such a machine you will get (something like):

```
Mem:     24684388   8799768  15884620     11284    922320   6338564
```

This shows that the machine has 24 GB of RAM in total and that ~16 GB are currently available. Before measuring, make sure nobody else is running stuff there with “free”, “top”, “who” etc. Some CLIC machines with 16 threads and 24 GB RAM (recheck before using): algiers, baghdad, beirut, bucharest, hanoi, jakarta, jerusalem, kathmandu, nairobi, nassau, pretoria, santiago

For the measurements, you will submit a CSV file with the following parameters:

```
arg1, arg2, ..., arg10, repeat, nanoseconds
```

The arguments are used in the main function (see “q4112_main.c” source).
For the number of threads, the 10th argument (last), you will use 1, 2, 4, 8, and 16 threads.
You will run 5 repeats per experiment, and the repeat column must be: 1, 2, 3, 4, 5.
For the first 9 arguments, you must measure these 8 configurations (8*5*5=200 lines in CSV):

```
>./q4112_hj       100 1.0 99999 1000000000 0.5 99999 0 0 0.0
>./q4112_hj       100 1.0 99999 1000000000 1.0 99999 0 0 0.0
>./q4112_hj    100000 1.0 99999 1000000000 0.5 99999 0 0 0.0
>./q4112_hj    100000 1.0 99999 1000000000 1.0 99999 0 0 0.0
>./q4112_hj 100000000 0.5 99999 1000000000 0.5 99999 0 0 0.0
>./q4112_hj 100000000 1.0 99999 1000000000 0.5 99999 0 0 0.0
>./q4112_hj 100000000 0.5 99999 1000000000 1.0 99999 0 0 0.0
>./q4112_hj 100000000 1.0 99999 1000000000 1.0 99999 0 0 0.0
```

To create the CSV file, you can modify the main function or write a script. You can modify the main function to generate the data once for each configuration and execute the query multiple times and use a different number of threads. You could also rerun the program every time but this is discouraged. You will not submit scripts or modified source files, only the “q4112_hj.c” in C (or “q4112_hj.c” in C++) source file, the CSV “q4112_hj.csv”, and an optional documentation file. If you want to submit multiple source files, you must also include your own “Makefile” but be warned that all other source files will be replaced, since you have to use the same API.

The query for part 1 is a single join followed by a trivial aggregation of a single group. You must use a global hash table (same as "q4112_hj_1.c") that will be shared across multiple threads. You must insert the inner table (table items from schema) in the hash table using multiple threads. To avoid race conditions, you must use atomic instructions and be **lock-free**. You are **not** allowed to use locks, semaphores, or your own implementations of the above. Your code must avoid locking in general (including spin locks). You can extend the hash table code used in "q4112_hj_1.c", but you must use **compare-and-swap** instructions to avoid race conditions (e.g. using GCC atomics: https://gcc.gnu.org/onlinedocs/gcc-4.4.5/gcc/Atomic-Builtins.html). You can also use **pthread_barrier_*** methods to synchronize all threads, but only a small (O(1)) number of times (e.g. our implementation uses 1 barrier for part 1 and 4 barriers for part 2).

The nested loop join source files and the single-threaded hash join are provided as a reference. Avoid floating point numbers for computing any averages, we will only use integer precision. You can either submit a README file or even describe your algorithm in the source code file.

Part 2 (60%)

In part 2, you will write code to execute the complete query with multiple stores. Again, you will submit a source file with your code "q4112.c" in C (or "q4112.cpp" in C++) and a CSV file named "q4112.csv" with measurements. You must measure the following 21 configurations:

```
>./q4112       100 1.0 99999 1000000000 1.0 99999       100     0 0.0
>./q4112       100 1.0 99999 1000000000 1.0 99999     10000     0 0.0
>./q4112       100 1.0 99999 1000000000 1.0 99999   1000000     0 0.0
>./q4112       100 1.0 99999 1000000000 1.0 99999 100000000     0 0.0
>./q4112       100 1.0 99999 1000000000 1.0 99999 100000000   100 0.5
>./q4112       100 1.0 99999 1000000000 1.0 99999 100000000   100 1.0
>./q4112       100 1.0 99999 1000000000 1.0 99999 100000000 10000 1.0
>./q4112    100000 1.0 99999 1000000000 1.0 99999       100     0 0.0
>./q4112    100000 1.0 99999 1000000000 1.0 99999     10000     0 0.0
>./q4112    100000 1.0 99999 1000000000 1.0 99999   1000000     0 0.0
>./q4112    100000 1.0 99999 1000000000 1.0 99999 100000000     0 0.0
>./q4112    100000 1.0 99999 1000000000 1.0 99999 100000000   100 0.5
>./q4112    100000 1.0 99999 1000000000 1.0 99999 100000000   100 1.0
>./q4112    100000 1.0 99999 1000000000 1.0 99999 100000000 10000 1.0
>./q4112 100000000 1.0 99999 1000000000 1.0 99999       100     0 0.0
>./q4112 100000000 1.0 99999 1000000000 1.0 99999     10000     0 0.0
>./q4112 100000000 1.0 99999 1000000000 1.0 99999   1000000     0 0.0
>./q4112 100000000 1.0 99999 1000000000 1.0 99999 100000000     0 0.0
>./q4112 100000000 1.0 99999 1000000000 1.0 99999 100000000   100 0.5
>./q4112 100000000 1.0 99999 1000000000 1.0 99999 100000000   100 1.0
>./q4112 100000000 1.0 99999 1000000000 1.0 99999 100000000 10000 1.0
```

The full query requires both a join and a group-by aggregation of multiple groups. The final aggregation is still for a single group but multiple groups must be computed first before the final count and sum are computed. You will use a global shared hash table to compute the aggregates again without locks. All additions in the global table can use atomic operations provided by GCC as in part 1 (you will need **atomic-add** in addition to **compare-and-swap**).

Notice that number of groups is not known during execution, thus we do not know how many buckets we need for the aggregation hash table. The largest number of groups could be the same as the number of outer tuples. Allocating a hash table for this many groups will not fit in ~24 GB of RAM. Allocating a large hash table in each thread will also not work due to memory constraints (and is also not allowed by the specifications that require a global hash table).

To find how big a global aggregation hash table needs to be, we can estimate the number of groups (e.g. using https://en.wikipedia.org/wiki/Flajolet–Martin_algorithm). We will discuss further extensions on Piazza. Another approach is to resize the global hash table "on the fly", but we must "stop the world" using synchronization barriers to avoid race conditions and pay the overhead of rehashing all partial aggregates, which must be done in parallel by all threads.

Part 2 Bonus (20%)

When multiple threads atomically update the same buckets of the global aggregation table, **contention** occurs. Contention is not very important during hash joins, because if the number of inner tuples is small, we do not care about the rate at which we insert each tuple anyway. For aggregation, however, contention can be very important. For example, assume that we aggregate 1 billion tuples and we have only 100 aggregates. Since all threads will update the same 100 memory locations, the speed per tuple will degrade rapidly for all 1 billion tuples.

In x86 architectures, it is hard to "detect" contention. Instead, we use small hash tables that are private to each thread and hold a few recent partial aggregates. When the local hash tables become full, we "flush" the partial aggregates to the global hash table. The local hash tables can also be thought as a "cache" (e.g. using direct mapping also does not require any LRU-style replacement policy). The local hash tables should remain cache-resident in each thread to avoid excessive overhead if the number of aggregates is large and there would be no contention.

To get some bonus grade, your implementation must include local hash tables used to cache the recent aggregates. There are multiple approaches here and the implementation details are up to you. We will track how well your code avoids contention through some configurations in the CSV. To get the full bonus grade, your code must be correct, you must clearly describe your approach, and the code must be efficient in the configurations that would generate contention.

Grading Policy

The grade (in each part) of the project is dependent on:

Correct code (50%)	Does your code work? Is it parallel? Does it not overuse memory?
Correct CSV (10%)	Did you do the measurements asked & submit the CSV file?
Code quality (10%)	Is your code well written? Ideally, use (or convert to) Google style.
Code clarity (10%)	Is your algorithmic approach clear? This includes documentation.
Code efficiency (20%)	Is your code efficient & scalable? We will compare against our code.

You can get up to 40 points for part 1 and up to 60 + 20 = 80 points for part 2 with the bonus.