SJTU U-MICH JOINT INSTITUTE

VE482 Introduction to Operating Systems

Project 2 Documentation

Group No. 6

Jiayao Wu	ID: 517370910257	jiayaowu@sjtu.edu.cn
Rui Chen	ID: 516021910758	2016jichenrui@sjtu.edu.cn
Songqian Li	ID: 516021910193	himson@sjtu.edu.cn
Tianyu Meng	ID: 516370910065	xix12229@sjtu.edu.cn

Date: November, 2019

Contents

1 Objectives

The objective of this project is to design and build a database system, named "lemondb", that supports multi-threaded query processing for "Lemonion Inc.".

Based on the remaining source codes of the old database system, we first realized a single-threaded version. Then, we developed a new multi-threaded version that is able to faster executing the queries than the old system by having multiple threads working in parallel.

Finally, the speed of our lemondb will be tested in comparison with that of the old system in various query scenarios in order to find out whether or not the system has been improved.

2 Structure of Database

2.1 Flow Explanation

First, to start our database system, user should input a –listen argument to indicate the query file. Then our system will open the input query file and start reading queries one by one. For each read query string, it will be parsed into tokens separated by white spaces. After decoding the tokens, we will classify the query into different types (debug, management, and data types). Each classified query will then be executed accordingly and the result will be stored. In the end these results will be outputted in the same order as input. The overall flow diagram is shown in Figure 1.

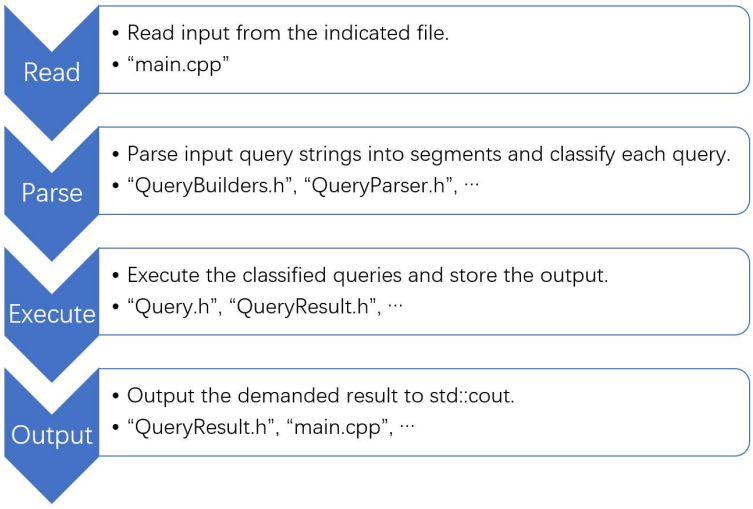

Figure 1: Flow Diagram.

2.2 Source Files

Figure 2 shows the structure of the "src/" folder which contains all our source codes. In "src/", "main.cpp" is the top layer of the system controlling I/O and file reading. "CMakeLists.txt" is used for building the database system.

The realization of the whole database and the tables is inside "src/db/".

In "src/query/", there are files for parsing and classifying query input and generating result output, as well as files for different classes of query executions: "src/query/management" is for database-level queries and "src/query/data" is for table-level queries.

The folder "src/thread/" contains information and operations about the number of threads that is available in running the system.

The folder "src/utils/" deals with formats and exceptions.

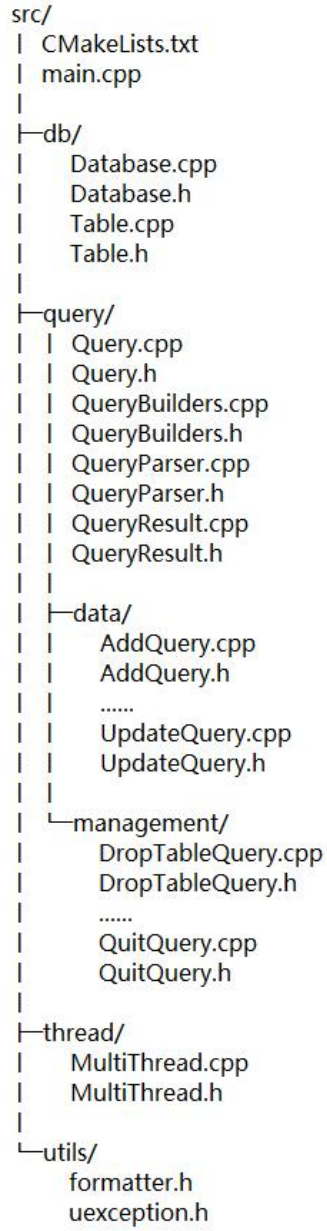

Figure 2: Source Files Structure.

3 Multi-threaded design

3.1 Overall Design

The advantage of using multiple threads is that these threads can run in parallel so that the speed of executing queries in our database system can be significantly improved.

After some observation we found that usually the tables have very large sizes, and other than table management queries (LOAD, DUMP, COPY, ...), most data queries (SELECT,

SUM, MIN, ...) must traverse the table row by row. So referring to Figure 1, we can conclude that the most time-consuming job is the execution of each query.

Then we figured that we can divide a very large table by the number of available threads to form some small table sections, and then have each thread take care of one section of the table. In this case all available threads can work in parallel to execute the query as shown in Figure 3.

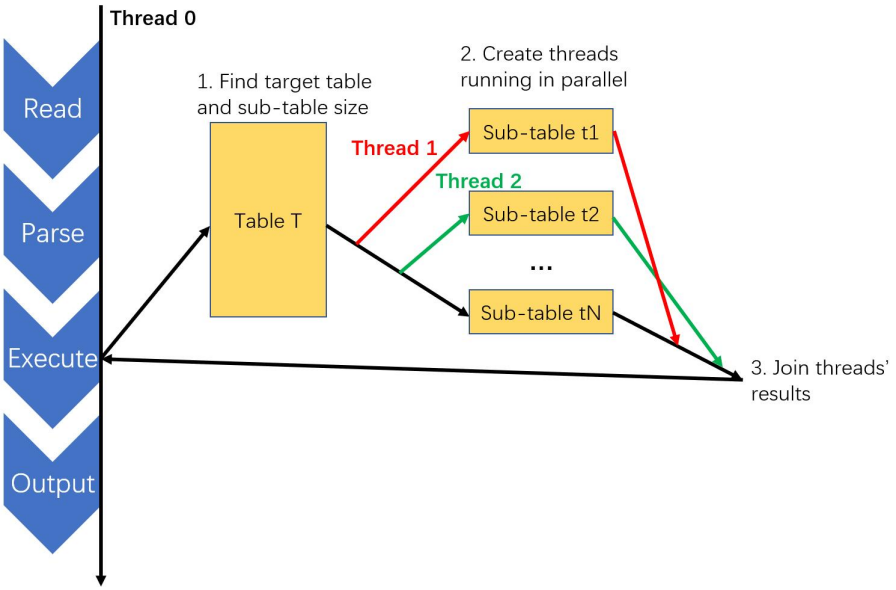

Figure 3: Multi-threaded Design.

Taking COUNT query as an example, we will show you the code that creates and joins the threads in Figure 4. After calculating the range, which is the size of each sub-table, we have all available threads executing the helper function CountQueryHelper() to perform counting on each sub-table. And in the end we join these threads.

```
range = (table->size())/(cq_thread_num);
pthread_t *t=new pthread_t[cq_thread_num - 1];
thread_counter = 0;
int i;
for(i = 0;i < (int)cq_thread_num - 1;i++){
    pthread_create(t+i, NULL, CountQueryHelper, NULL);
}
CountQueryHelper(NULL);
for(i = 0;i < (int)cq_thread_num - 1;i++){
    pthread_join(*(t+i), NULL);
}
delete [] t;
```

Figure 4: Creating and Joining Threads.

3.2 Critical Region and Locks

Since our strategy requires the target table to be perfectly partitioned so that there will be no overlapping between sub-tables and that not any row will be neglected. After this is implemented, we no longer need to add locks to the table or the content of the table.

However, we do need some locks to protect the shared variables among threads when reading or modifying them. And the locks we used are mutex from pthread.h.

Again using COUNT query as an example, shown in Figure 5 are the static shared variables. A mutex used for locking critical regions, a common target table, the result of counting, a pointer to this current query, cq_thread_num (total available threads for this COUNT query), the size of each sub-table and thread_counter, counting the already activated threads.

```
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static Table* table;
static int affected;
static ComplexQuery* qryptr;
static unsigned int cq_thread_num;
static unsigned int range;
static unsigned int thread_counter;
```

Figure 5: Shared Variables.

Among these variables, only "affected" and "thread_counter" could be modified by the execution functions while all the others are read-only variables. So the critical region and locks arrangement are clear now: add locks when the execution functions of each thread tries to modify "affected" or "thread_counter".

The implementation of the helper execution functions for each thread is shown below in Figure 6. It mainly contains 4 steps:

1. Read the "thread_counter" to know how many threads have been created and increment it to let others know this thread has started running.
2. Based on the already existing threads, calculate the sub-table position that this thread should be responsible of.
3. Doing counting within this sub-table, record the count result in a local variable.
4. Adding the local result to the final overall counting result.

```
void * CountQueryHelper(void *a) {
    // get and then increment thread counter
    pthread_mutex_lock(&mutex);
    int i = thread_counter;
    thread_counter++;
    pthread_mutex_unlock(&mutex);

    // locate my sub-table
    auto start = table->begin() + (i * range);
    auto end = start;
    if(i == (int)cq_thread_num - 1)
        end = table->end();
    else
        end += range;

    // count within my sub-table
    int temp = 0;
    for(;start != end;start++){
        if(qryptr->evalCondition(*start)){
            temp++;
        }
    }

    // adding my count to the final result
    pthread_mutex_lock(&mutex);
    affected += temp;
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

Figure 6: Execution on Each Sub-table.

4 Discussion and Conclusion

In this multi-threaded database system, we implemented a very naive strategy to have one query being executed by a number of threads in parallel. And in the end, the speedup comparing to the old single-threaded version is not very clear.

For the future work, we will focus on how we can glue some queries together into a big query so that we can reduce the number needed to traverse the table.