Mini-1 reports

Team member: Thinh Bui thinh.bui@sjsu.edu

Introduction

In this report, a detailed analysis of my query engine is reported, together with the instruction to run it. My key finding is that delaying the loading of the data file from disk until the query time could increase the query performance significantly. Besides that, I also discovered that there is a difference between python and C++ multithread. While applying multi thread increased the C++ query efficiency, it made the python query engine slower. The document will be organized as follows. First, the baseline approach is presented, together with its measurement and explanation about the findings. Next, based on the findings from the baseline approach, a further optimization section digs deeper into the performance issue and presents some optimization and its results. The document concludes with a running instruction of the project.

Baseline implementation

To analyze the performance of query engines in different programming languages and scenarios, four different query engines are implemented in 2 languages, C++ and Python. Based on the performance analysis of four query engines, other optimized query engines are implemented for C++ and Python.

In this section, a detailed implementation of four query engines are presented. These query engines can be divided into two categories: single-threaded and multi-thread query engines. Since the algorithm to implement single-threaded and multithreaded query engines are the same on C++ and Python, I will present the algorithm on C++.

For a single-threaded query engine, the detailed algorithm is as follows: first, the CSV data is parsed, loaded to memory and transformed to columnar format. Since we are focusing on the filter query and its performance, co-locate the same column's value will reduce the time to fetch the whole column's values. Second, a predefined filter will extract the column its needs, loop through the data and evaluate each column's value based on the filter expression. A counter is used to keep track of the number of rows that returned true in the filter expression. Finally, the counter value is returned to the user.

For the multi-threaded query engine, its first and last step is similar to the single-threaded query engine. The only difference is the second step, where I distributed the filter workload to multithread by splitting the data into chunks and letting each thread process one chunk. In C++, I employed OpenMP[4] library to do multi-thread, while in Python, I used ThreadPoolExecutor for multi-threading the filter.

To test the performance, I run the query engine on the NYC Parking Dataset with the query “SELECT COUNT() FROM csv_file WHERE registration_state == CA” on my MacBook M1 pro. Details about the performance of Python and C++ query engine are reported in Table 1.

Language	Method	CSV parsing time (s)	Filter time (ms)	Note
C++	Single threaded	360 s	470 ms	
C++	Multi thread	360 s	73 ms	
Python	Single thread	412 s	20564 ms	
Python	Multi thread	412 s	Unable to record.	Process did not finish after 5 min

Table 1. Query engine running time on NYC Parking Violation Dataset

From the table [1], it showed that parsing the CSV file is the most time consuming part in both single-threaded and multithreaded query engines. Regarding the filter time, the C++ multithreaded implementation runs 6 times faster than the C++ single-threaded one. However, the Python multithreaded version is much slower than the single-threaded Python version. The slower reason related to Python's Global Interpreter Lock as it only allows one thread of a process to run at a given time, making it hard to increase the parallelism of code according to [Wikipedia]. Given the above observation, in the next steps, I decided to dig deeper into the parsing CSV steps to reduce the time to parse CSV. Furthermore, I also try to exploit the multi-process in Python to improve the performance of the filter.

Further optimization

For the CSV parsing optimization, I will focus on the C++ implementation. First, a profiling information of the code is obtained through gperftools [2]. Since we are interested in the performance of the CSV parsing part, we will only use the single-threaded implementation. Figure 1 is the profiling results collected while running the code on a sample 100k rows NYC Parking Dataset.

```
Unset
Total: 289 samples
     119  41.2%  41.2%      123  42.6% csv::CSVRow::get_field
      50  17.3%  58.5%       57  19.7%
csv::internals::CSVFieldList::emplace_back
      32  11.1%  69.6%       32  11.1% csv::internals::data_type
      14   4.8%  74.4%       14   4.8% 0x000000019d0bf344
```

```
      11   3.8%  78.2%       11   3.8%
std::__cxx11::basic_string::basic_string@b4da0
       8   2.8%  81.0%        8   2.8% 0x000000019d0bf35c
       7   2.4%  83.4%        7   2.4% 000000019cea4654
       7   2.4%  85.8%        7   2.4%
std::__cxx11::basic_string::basic_string@b2220
       4   1.4%  87.2%        4   1.4% __GLOBAL__sub_I_csv_row.cpp
       3   1.0%  88.2%        3   1.0% 0x000000019d0bef80
```

Figure 1. CPU profiling result of C++ query engine

From the CPU profiling result, it is clear that the operator[] of CSVRow from the csv-parser library consumes the most CPU. The excessive use of operator[] is because everytime I try to parse the field of csv rows, I use this operator to get the field value. Digging deeper into the implementation of this, I found that operator[] is not an O(1) operation. The operator[] calls a get_field method, which in turn calls several string processing functions. To reduce the usage of it, I delayed the parsing of CSV until the query execution phase. At that time, as we already know which column is needed for calculation, only the desired fields need to be parsed from the csv. After integrating this idea to C++ query engine and running the benchmark, the parsing time is reduced by 5 times, as shown in the table [2]. This led to my key finding: a delay of data fetching in a query-engine can significantly reduce the processing time.

Language	Method	CSV parsing time (s)
C++	Delay-loading data	75s
C++	Loading the whole CSV data	360s

Table 2. CSV parsing performance

Regarding the increasing Python parallelism through multiprocessing, as shown in Table1, the performance of multi-process implementation is worse than the multithreaded implementation. The performance is inefficient because the column data has to be copied when a new process is created. To further improve the multiprocess performance, the column data is shared between processes by using ShareableList [1]. After integrating the shared-memory, the performance of the multi-process shared-memory query engine is 4x times faster than the multi-process query engine. However, it was not as efficient as the multi-thread python implementation. I think it is because of the overhead of creating a ShareableList and setting up the process. The details about filter time are recorded in table [3]. Since the Python query engine can not produce results for the multithreaded filter, I used the 100k sample of Parking Violation Dataset for testing. To conclude, my Python query-engine did not gain any performance improvement when increasing parallelism through multi-thread or multi-process.

Language	Method	Filter time (ms)
Python	Multithread filter	314.9ms
Python	Multiprocess filter	2368.4ms
Python	Multiprocess shared memory	513.1ms

Table 3. Python filter time on 100k sample of Parking Violation Dataset

Build and run instructions:

Building instruction (for C++):

```
Unset
cd benchmark-cpp
mkdir build && cd build
cmake ..
make benchmark
```

Running instruction

```
Unset
For C++:
cd benchmark-cpp/build/bin
./benchmark /path/to/nyc_dataset /path/to/c++-json-query

The example of c++-json-query can be found at benchmark/query.json:
{
    "data_path": "../../data/nyc/parkingviolations/sample_100k.csv",
    "data_mode": "lazy-load",
    "filter": {
        "col": "Registration State",
        "op": "eq",
        "val": "CA"
    },
    "filter_mode": "multi-thread"
}
```

```
Unset
For Python:
cd benchmark-python
python3 src/benchmark.py /path/to/python-json-query

The example of python-json-query can be found at src/query.json:
{
    "data_path": "../data/nyc/parkingviolations/sample_100k.csv",
    "filter": {
        "col": "Registration State",
        "op": "eq",
        "val": "CA"
    },
    "filter_mode": "multi-thread"
}
```

Reference:

1. ShareableList https://docs.python.org/3/library/multiprocessing.shared_memory.html#multiprocessing.managers.SharedMemoryManager.ShareableList
2. Gperftools https://gperftools.github.io/gperftools/cpuprofile.html
3. Global Intepreter Lock https://en.wikipedia.org/wiki/Global_interpreter_lock
4. OpenMP https://www.openmp.org/