SQL Query Executor

February 21, 2020

1 Introduction

In this project we simulate a query compiler / executor that compiles and executes SQL queries containing 2 types of predicates.

This project was split into 2 stages.
During the first stage, we analyzed and executed query batches that were used on SIGMOD 2018 Contest.
During the second stage, we tried to optimize the time needed to execute these batches using reordering and multi threading.

2 Methodology

The first type of predicates is join operator based on equality between 2 relations. (e.x. R.A = S.A). The second type of predicates is basically a filter performed on the values of 1 relation. (e.x. $R.A >= 5000$). These relations are loaded in memory from disk at the start of the program and remain till the end.

The method used for the join operator is called SortMerge Join which basically means that we have to sort each relation before joining them. This makes the join operator perform much faster than $O(n^2)$. More specifically, its complexity is O(n*k) where k is the size of a group with the same key.

2.1 Parsing

Each query followed the same format. An example is this query:
0 2 4| $0.1 = 1.2\&1.0 = 2.1\&0.1 > 3000$ |0.0 1.1

Which is translated as follows in SQL:

SELECT SUM("0".c0), SUM("1",c1)
FROM r0 "0", r2 "1", r4 "2"
WHERE $0.c1 = 1.c2$ and $1.c0 = 2.c1$ and $0.c1 > 3000$

Therefore, before processing we had to perform parsing.

2.2 Sorting and Merging

As mentioned above, before performing join operator on the 2 columns of these relations, we had to sort these relations based on these specific columns.
Using the example above, we had to sort r0 on column c1 and sort r2 on column c2.

The sorting algorithm used for this purpose is a variation of radix sort.
The basic idea is to split the data into buckets, and, if these buckets fit into L1 Cache, use quicksort to sort them, else split these buckets into more buckets until they fit into L1 Cache. (bucket size is smaller than 32KB)

The way the data are split into buckets is by iteratively grouping the keys based on their current byte value.
This will be better explained with a small example :
Consider 5 4-byte integers with the following bytes :

Byte 0	Byte 1	Byte 2	Byte 3
2	6	87837	787
4	545	18744	7560
2	545	778	7507
2	7	78	5415
4	88	788	6344

In the first iteration, we group the numbers into 2 buckets.
The first 3 keys are in the same group (group ”2”) because their first byte has the same value and the last 2 keys also possess the same byte 0 so they fit into the same group (group ”4”).
Before the second iteration, assume something like this :

Byte 0	Byte 1	Byte 2	Byte 3
2	6	87837	787
2	7	78	5415
2	545	778	7507
4	545	18744	7560
4	88	788	6344

Now, we move on the second iteration where we split each bucket formed based on the byte 1.
This splitting is done iteratively until a bucket has size less than 32KB, then it gets quick-sorted.
After sorting the relations based on their respective columns used for join, we have to merge these relationg.
Merge algorithm is as follows :

```
join_result merge_relations(relation *relR, relation *relS) {

    join_result join_res;

    uint64_t *results_r = NULL;
    uint64_t *results_s = NULL;

    size_t pr = 0;
    size_t s_start = 0;

    while (pr < relR->num_tuples && s_start < relS->num_tuples) {

        size_t ps = s_start;
        int flag = 0;
        while (ps < relS->num_tuples) {
            if (relR->tuples[pr].key < relS->tuples[ps].key) {
                break;
            }

            if (relR->tuples[pr].key > relS->tuples[ps].key) {
                ps++;
                if (flag == 0) {
                    s_start = ps;
                }
```

```
            }
            else {

                buf_push(results_r, relR->tuples[pr].key);
                buf_push(results_s, relS->tuples[ps].key);

                flag = 1;
                ps++;
            }
        }
        pr++;
    }

    join_res.results[0] = results_r;
    join_res.results[1] = results_s;

    return join_res;
}
```

2.3 Intermediate Results

Lets consider this query : 0 2 4| $0.1 = 1.2\&1.0 = 2.1\&0.1 > 3000$ |0.0 1.1
Our first job is to determine in which order to execute this query. There are many complicated algorithms that use heuristics to determine the optimal order to execute a number of consecutive joins, and this will be a concern for us in stage 2 where we try to optimize our query compiler / executor. For now, we assume that the optimal way is to execute filters before joins in order to remove the keys that will be joined afterwards.

When a predicate is executed on a relation **A**, we have to store the rows of this relation that satisfy this predicate in a structure. The structure that holds the relations and their respective rows after executing a predicate on them is called Intermediate Results Structure. For example, consider the filter-predicate $0.1 > 3000$. After executing this predicate, we need to store only the rows of 0.1 that have keys > 3000.

Therefore, we need a structure called "Intermediate Results" which temporarily stores the result (the rows of the relation that meet the criteria) of each predicate.

When we encounter a join predicate, we have to categorize it based on if the relations participating on this join are in the Intermediate Results Structure.

- **Classic Join** : Both relations are not found in the Intermediate Results Structure, thus we load all their keys **from memory** and perform SortMerge Join as mentioned above. Then, we store the rows of each relation in the Intermediate Results Structure.
  As an example, consider this join : 0 2| **0.1=1.2**$\&0.1 > 3000$ |0.0
  We see that this is the first operation performed on relation "0" and relation "1" so we assume that they are not present on the Intermediate Results Structure, thus we load their key-row pairs directly from memory.

- **IR Join** : One of the 2 relations already exists in the Intermediate Results Structure. As a result, we append the second relation in the structure too.
  Consider this example :
  0 2 4| $0.1 = 1.2\&1.0 = 2.1$
  After executing **0.1=1.2**, relation "0" and relation "1" are present in the Intermediate Results Structure. In order to execute **1.0=2.1** we have to access the Intermediate Results Structure, pick the rows of relation "1" and access relation "2" from memory in

order to perform the SortMerge Join. After joining these relations,we append relation's "2" rows in the Intermediate Results Structure.

- **Scan Join**: Both of the 2 relations are found in the Intermediate Results Structure, so instead of perfoming SortMerge Join we perform a scan join where we scan both relations one time and return the rows that have matched keys.

3 Optimizing

In order to optimize our query executor we performed 2 tasks.
Our main task was to parallelize the program on 3 levels and the second task was to use a heuristic algorithm to find the optimal way to execute join operators.

Our way to parallelize the program was by multi-threading. We created a job scheduler that manages jobs by creating a thread pool (with FIFO policy).

- **Parallel query execution** : Each query in a batch is to be executed in parallel with all the other queries in the same batch. By using this approach we encountered a huge out-of-memory problem so we minimized the number of queries executing in parallel.
- **Parallel SortMerge Join execution** : When sorting the relations, we split them into different kind of buckets, until they are small enough to be quicksorted. A simple idea is to use these buckets to split the data and use multi threading approach to sort and join them.
- **Parallel SortMerge Join calls** : Before merging 2 relations, we have to perform 2 sorts, and we do this in parallel by adding 2 more jobs in the pool. The first job is to sort the first relation and the second job is to sort the second relation in parallel.

After implementing multi threading approach, we found an algorithm that helps us determine the optimal way to execute join operators. The algorithm is presented below :

```
for (i = 1; i <= n; ++i) {
  BestTree({R_i}) = R_i;
}
for (i = 1; i < n; ++i) {
  for all S ⊆ {R_1, ..., R_n}, |S| = i do {
    for all R_j ∈ {R_1, ..., R_n}, R_j ∉ S do {
      if (NoCrossProducts && !connected({R_j}, S)) {
          continue;
      }
      CurrTree = CreateJoinTree(BestTree(S),R_j);
      S' = S ∪ {R_j};
      if (BestTree(S') == NULL || cost(BestTree(S')) > cost(CurrTree)) {
          BestTree(S') = CurrTree;
      }
    }
  }
}
return BestTree({R_1, ..., R_n});
```

Figure 1: Join Enumeration Algorithm [1]

4 Discussion and Conclusions

This project helped us to better understand how things are performed "behind the scenes" when, for example, we execute a query in SQL on a database.

Moreover, it revealed us some "hidden truth" behind parallelizing tasks. More specifically, our attempts to increase performance using multi-threading made our program slower than before.
We believe that the 2 main reasons for this ostensibly weird behavior are :

- **High Memory Usage** : As we increase the number of queries running in parallel, we increase the memory needed in a specific time of execution, thus making our program slower.
- **Amdahl's law** : [2] Amdahl's law implies that in order to achieve higher performance when parallelizing a program, you have to parallelize nearly all program execution.

References

[1] Guido Moerkotte. "Building Query Compilers." In: (2019).

[2] Peter S. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.