Final Project Writeup 6.830

Benton Wilson Elizabeth Weeks

1 Introduction

Programming competitions are a great way to test our knowledge, since they force us to think outside the box and dive deep into understanding how to improve a given system. As such, we chose to participate in the 6.830 Programming Contest for spring 2021.

The structure of this competition was as follows: each team was given the same starting code, which was correct, but not heavily optimized for speed. The starting code worked, but the goal for each team was to make their code run as fast as possible. We were given 60 seconds before any of the queries began to do any preprocessing of the data. Queries were then fed in to the system, and timed to see how long they took to return the correct result.

A few constraints given for the environment were as follows (run on an AWS r5.4xlarge instance):

Processor	Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz
Configuration	8 cores / 16 hyperthreads
Main memory	128 GB RAM
OS	Ubuntu 18.04

One thing worth noting is the large amount of main memory that was available to us. It turned out that on the largest test case (there are 4 test official test cases ranging from "small" to "X-large"), the X-large test case only had a peak memory utilization of 60GB in the base code (just half of what is available). As such, part of the fun for the project was coming up with algorithms meant for in memory joins, and trying to adapt some of the ideas from class where we generally focused more on databases that cannot always do their joins in memory.

2 Overall Architecture for Optimizations

In general, our strategy for improving the speed of the code was to first try and find the bottlenecks in the code, and then loosen those bottlenecks by making use of concurrent execution, reducing the amount of work that needs to be done (like deferring operations until later), and also by combining strategies (since certain optimizations are better on certain inputs).

In the end, our code made a few major changes to implementation that seemed to have the largest effect. Unlike in the base code, we make extensive use of multi-threading to increase the amount of work that our system is able to do. Since there is an overhead to starting up and coordinating new threads, we also did some tuning, as it is sometimes more efficient to start fewer threads, or even just run the serial version of the code for smaller queries. We also tried to minimize, as much as possible, the computing of unnecessary work, employing techniques such as early return. There were also a few smaller optimizations that helped our code run faster (discussed in detail in the next section). Our code runs in around 112 seconds total on the AWS instance, spending around 66 seconds on the extra large test.

3 Specific Optimizations and Their Effectiveness

3.1 Join

Unsurprisingly, a large bottleneck in the base code came from doing joins between two different tables. As a result, we spent a large majority of our time trying to optimize our join algorithm.

In the base code, the join is implemented in `Join::run()` using a rather simple, serial algorithm. First, a hash table is built on the smaller table, and then the larger table is scanned, probing the hash table during the second scan.

One of the first optimizations we made here was to use the `std::thread` interface to parallelize the probing of the hash table. Unfortunately, this added one small problem; in the serial version, all of the values of the join were being put into a single `tmp_results_` variable, but in order to avoid race conditions, we needed each thread to have its own version of the temp results variable. In order to do this, we gave each thread one array of temporary results in a larger array of all the temporary results (called `all_tmp_results_`). Then, after each thread finished with its own temp results, we merge all of those results into the main `tmp_results_` variable. This final step is necessary, because later operators (higher up the query plan), require that the results are contiguous in memory (so they need to be merged eventually).

While this single optimization gave us a large improvement, we were still taking around 162 seconds to pass the extra large test. This was an exciting milestone since it was the first time we passed extra large but we knew we still had more work to do.

After making this optimization, we realized another problem; even though lookups into the hashtable were parallelized, the merging of all of the threads' individual temporary results was done in one thread, so one thread would still have to actually write the entire size of the output of the join. Thus, we thought about how we could also parallelize this part of the code (we can refer to this as the "merging" phase).

Since the database uses column store, we were actually able to solve this problem by having each thread be responsible for merging the values of one column into the single `all_tmp_results_`. After we did this, we were able to pass all of the test cases online, finishing the total tests in around 162 seconds.

However, one problem with this approach, which we realized later on, is that when there are a small amount of columns in the output (such as with the X-large test), we aren't actually spawning very many threads to do the joins. As a result, we decided to also parallelize the merging of the columns and then each thread was responsible for inserting values into a subsection of a column variable (the subsections were disjoint as to prevent threads from overwriting each other). After making this optimization, we were able to significantly reduce our runtime from 86 to 76 seconds on the extra large test.

Finally, upon doing both optimizations (of the probing and the merging of the temporary results), we notice that some of our test cases (particularly the smaller ones), got much slower (for reference, slower by about a factor of 2). We hypothesized that this was due to the overhead of starting threads when the work is small (like when joining two small tables). To fix this, we simply run the serial version of our code when the size of the tables is under a certain threshold, and we also reduce the number of threads that we spawn for some of the medium size joins. Doing this allowed us to recover

some of our speed on the small test case, but it did not give us a huge improvement on the X-Large test.

We also made a small optimization with ending early. Namely, if either table is of size zero in a join, then we know that we can just stop our join right there; there is no needed to do the rest of the join since we can't possibly match on anything.

3.2 Filter Scan and Self Join

Two other operators in the code base are `FilterScan` and `SelfJoin`. Filter scan is a scan of a relation with a filter applied, whereas Self Join handles the special case of joining a table with itself. In both cases, we employed the parallelization strategy from Join; that is, we partitioned the input, had each thread handle its partition separately, and then merged the results into the final results in parallel. Doing this gave us about a 10% boost in performance (though it seems like more of the boost came from parallelizing `SelfJoin::run()`).

3.3 Thread Pooling

In general, we know that spawning new threads can have a lot of overhead. As a result, we wanted to try using the common technique of having a shared thread pool to get around some of the overhead. We used a small library found on a github (https://github.com/progschj/ThreadPool), and using this for thread-pooling gave us about a 5% boost in performance on the large test.

3.4 Other Small Optimization

A rather small optimization that we did involved starting the left and right subtrees of a join in their own threads. This helped parallelize some of the different filter scans, and was a rather simple optimization that we did at the start, that seemed to help a bit.

Another, rather minor optimization that we made was in the `FilterScan` method for checking if a row matches a filter. Instead of looping through all the filters to see if a row matches, we break early. Before the code was continuing to evaluate the rest of the columns. While this optimization did not seem to make a huge change, it sometimes seems to help the code run faster locally, and definitely did not hurt performance, so we decided to keep it for the final.

4 Discussion

4.1 Optimizations that didn't work

So far, we have talked about the optimizations that we ended up using, but there were also many attempts at optimizations that either made no difference or actually hurt performance. Sometimes, this could be frustrating, but overall it helped us gain knowledge about what our bottlenecks were.

4.1.1 Query Plan Reordering

Throughout 6.830, we have talked a lot about the best way to order queries. The starter code simply sets up a left deep join tree with no full cross products, but other than that, there is no join ordering optimization done. This was actually one spot that we spent a decent amount of time trying to optimize at the start, but we weren't able to come up with a good way to choose the best ordering. Also, it is worth noting that even though we had some code for query plans from class, we chose not to port all of it over, since we didn't think that it would make as big of a difference (it might take longer to plan the query than to just do a lot of the smaller queries). Some very simple heuristics that we tried which did not end up working were to first sort the joins by smallest table to largest, or to try and give different weights to filters. Neither of these improved the performance of the code. We did determine that moving any joins on two primary keys to the beginning of the join ordering is beneficial since this join will only produce one row.

4.1.2 Efficient Temporary Results in Joins

As discussed earlier, we ended up parallelizing joins by giving each thread it's own vector of temporary results before merging all of the temp results. However, we saw one clear problem with what were were doing; if we needed to have 10 columns in our output, then we would be storing 10 `uint64_t` values in temp results for each row. However, if instead we simply stored the row number for the left and right table for each row in the output, we would only have to store 2 values in the temp results for each join. Then, we could just use the row number for each of the left and right tables to get the correspond data in the merge step.

Overall, we thought that this would help us not move as many values, during each join, but it turned out that it made the program slower. Upon investigation, we found that many joins used very few columns; if the number of columns in the output was 1 or 2, then the original algorithm would be faster. Though we think this would be an important consideration for a more broad suite of tests, it actually did not seem to help for the tests in this case.

4.2 General Challenges

4.2.1 C++

While we are both somewhat comfortable with C, C++ is a different beast, and as with C, debugging code can often be a process (compared to some other simpler languages like Java). However, we have both gained a lot of appreciation and knowledge about some of the internal workings of C++ types like `std::vector`. Also, one thing that really threw us off was that vectors are automatically copied by value upon reassignment unless marked copy by reference; this led to many hours of debugging, since we were used to aliasing by default. Also, since C++ is newer to us, it took longer than we would have liked to really get spun up on how the code base itself worked. However, once we understood the main code flow, we didn't have too much trouble figuring out how each part worked.

We also experienced some difficulty testing locally since our computers have different architectures and different amounts of RAM. Anything that ran on Elizabeth's computer was slower and caused her internal fan to start. This made comparing results between the two computers and the submission site difficult.

4.2.2 Implementing Things That Didn't Work

In general, it was discouraging to spend a lot of time debugging and implementing a new improvement in the system, only to have it not actually work in the end. Specifically, we had a lot of bugs when working on only storing the row numbers in the temporary results. Sometimes, it felt like something should definitely be an improvement, but despite all the effort, it didn't make a difference. At the same time, trying different optimizations is never guaranteed to work, and at the very least, we gained a better appreciation for some of the details about how our program was working.

4.2.3 Parallel Programming

As with any program, things get more complicated when using multiple threads, and it is often much more difficult to debug. This project was no exception. We often would have things working in one thread, but then when we scaled to more threads, we would end up with race conditions that could be difficult to track down.

5 Future Work

5.1 Preprocessing

One thing that we struggled with was coming up with a clear way to use the 60 second preprocessing time allotted to us before executing queries. Overall, we think that given the time to build out nice systems for storing data, we could have made better use of various aspects of the data. For instance, if we had the uniqueness and indices for various columns, we would be able to efficiently prob

into highly unique columns (for instance, it would have been easy to do lookups for primary keys). However, this would have required many larger parts of the program to be hugely modified in order to pass data from the preproccessing stage into the operators, so we opted not to focus on this until we ran out of ideas for optimizations in the joins. Also, we thought that it might be possible to simply precompute some of the hashtables, but we didn't end up having great luck with this when we tried it (it didn't really make much of a difference, and was prone to using up too much memory).

5.2 Storing One Column For Joins

Another optimization that we thought of towards the end, but could not implement was that we don't actually need to store both the left and right column for the columns we are joining on. This is because all joins are equi-joins; thus if we are joining on A.c1 = B.c1, we can only store one column, and it will contain the values for both A.c1 and B.c1. This would potentially save lots of wasted movement of data depending on how the queries are structured. We don't think that this would have much overhead to implement either, so it is definitely something we would have implemented given a bit more time.

5.3 Bushy Query Plans

We also think that it should be possible to do some sort of efficient bushy query plan (and we could just spawn a thread for each side of the tree). However, since we ended up not building out the infrastructure for histograms/various selection size estimators, we did not have a great way of actually choosing a bushy plan. Also, it seemed like many of the test cases often didn't use very many joins per query, so we didn't want to obscure results from other optimizations by creating a whole new structure for making the query plans.

5.4 Load Balancing Among Thread Partitions

Currently, when executing the probe phase of a join, we simply give each thread an equal number of rows from the larger table to probe with. However, it is possible that some threads do much more work than other threads. For instance, on thread may find many rows to put in the output, but then other threads may have much less work to do. This could lead to some threads sitting around idle, waiting for the slowest thread to finish.

We had few ideas for fixing this problem. One way would be to compute some simple statistics on the join inputs, and use heuristics to guess at how big the partitions would be. However, we weren't sure how to do this.

Another thing that we thought might be interesting was if we had threads set a flag when they finished. Then, other threads could check that the flag has been set, and if a thread still had say, more than 50% of its work left to do, it could ask the finished thread to continue doing work. Unfortunately, this is one area where we both feel that our knowledge of locking and threading in C++ is not strong enough to create an efficient implementation.

5.5 Two Way Partitioning On Joins

Finally, one thing that we really struggled with writing an algorithm for was efficient partitioning of both the left and right of our tables when doing a join. Currently, a lot of time is spent doing look ups into the hash table, which is built on the smaller of the left and right input to a join. However, we think that it might be possible to partition both the left and right tables into N partitions, based on the join column. Then, each thread could be responsible for joining only the partition of the left and the right that it was responsible for. This could greatly reduce the amount of comparisons that we need to do, which would improve the overall run time.

However, we also think that this approach would be greatly limited by uneven partitioning, so we would need an approach for getting around this (potentially by spawning more threads when one thread is stalled).

ACKNOWLEDGMENTS

We would like to give a huge thanks to the 6.830 staff for setting up the programming competition and for sharing their knowledge throughout the semester :).