COMS W4112 Project 2 Report

Varun Ravishankar, vr2263 and Jervis Munidi, jjm2190

May 1, 2013

1 Introduction

In this paper, we report on the performance of selectivities to a set of records, looking at execution time, branch misprediction rate, and instructions performed per CPU cycle using an algorithm described by Kenneth Ross in [Ross 2004]. As main memory sizes grow larger and less expensive to maintain, databases that reside entirely in RAM become more affordable and more attractive as primary databases. These databases must take into account processor characteristics like branching rates and cache sizes to be efficient, just as disk-based databases must take into account the characteristics of hard disks.

While disk-based databases use algorithms that optimize for random I/O and sequential I/O, algorithms that operate on data in main memory do not have the same I/O concerns. Since I/O is much faster when reading and writing to RAM than when operating on disk, the main concern is in making sure that the CPU continually has data to operate on so it can take advantage of the higher read and write rates. This means that algorithms that operate on main memory can be greatly affected if they cause cache misses or cause branch mispredictions, where "the CPU executes a conditional branch instruction and predicts the outcome incorrectly" [Ross 2004]. One algorithm designed to optimize database queries to minimize branch mispredictions by taking into account a cost model for the CPU in question is described in [Ross 2004] and is outlined below.

2 Background

CPUs have a pipelined architecture, and need the pipeline to be filled to maximize efficiency. When a CPU encounters a conditional branch like the common if/else condition, there are two possible outcomes and the CPU must use branch predictions to predict the most likely instruction sequence to be executed. This most likely sequence is then used to fill the pipeline. If the CPU predicted correctly, then all is well and the CPU continues executing instructions. However, in a branch misprediction, the CPU guesses incorrectly. It must then flush the pipeline, which now contains incorrect results based upon the incorrectly taken branch, and then start again. This means that a branch misprediction can cause significant processing latencies; [Ross 2004] notes that a Pentium II processor suffers a 17 cycle penalty for every branch misprediction.

Assume we have k conditions we want to evaluate, with provided selectivities (defined in [Ross 2004] to be the "proportion of records in the table satisfying the condition"). We want to evaluate these conditions to look at all the appropriate records in the tables and minimize the overall cost for doing so. Each of these conditions can be combined in a number of orders and a number of ways, all of which can affect the speed and overall selectivity for the overall plan. However, these different queries still have a similar structure for the equivalent C code, as described in [Ross 2004]:

```
/* Basic Algorithm Structure */
for(i=0; i < number_of_records; i++) {
    if(f1(r1[i]) AND ... AND fk(rk[i])) {
        answer[j++] = i;
    }
}
```

Here, the query is broken up into a number of conditions. Each condition is evaluated and the results are ANDed together. We then populate the answer array with the appropriate records. Also, it is important to

note that the ANDs here may be &&, labeled as branching AND, or &, defined as logical AND. Branching AND can result in less work done overall if $f1$ is very selective, but is translated to assembly with k conditional branches. This can result in many branches being executed if each selectivity is close to 0.5, which translates to more branch mispredictions. The algorithm for this implementation, called Algorithm Branching-And in [Ross 2004], is described below:

```
/* Algorithm Branching-And */
for(i=0; i < number_of_records; i++) {
    if(f1(r1[i]) && ... && fk(rk[i])) {
        answer[j++] = i;
    }
}
```

Algorithm Logical-And, which uses logical AND instead of branching AND, is also described in [Ross 2004]. This algorithm only has one branch, for the overall if statement, but does poorly if the first few conditions are very selective, since we evaluate all the conditions in the statement every time we execute the if statement. Algorithm Logical-And is displayed below:

```
/* Algorithm Logical-And */
for(i=0; i < number_of_records; i++) {
    if(f1(r1[i]) & ... & fk(rk[i])) {
        answer[j++] = i;
    }
}
```

[Ross 2004] notes that Algorithm Logical-And can result in a large branch misprediction if the combined selectivity of the query is close to 0.5, and describes an algorithm that does not branch at all. Algorithm No-Branch is described below:

```
/* Algorithm No-Branch */
for(i=0; i < number_of_records; i++) {
    answer[j] = i;
    j += (f1(r1[i]) & ... & fk(rk[i]));
}
```

These different algorithms perform well in different selectivity ranges, as described in [Ross 2004]. In that paper, Ross describes tests performed on a 750 Mhz Pentium III on Linux. He found that Algorithm Branching-And performs best for low selectivities, while Algorithm Logical-And performed best for intermediate selectivities. Once the combined selectivity got close to 0.5, Algorithm Branching-And and Algorithm Logical-And started to perform poorly, and Algorithm No-Branch started to outperform the other two. The graph for the tests performed in [Ross 2004] can be seen in Figure 1:

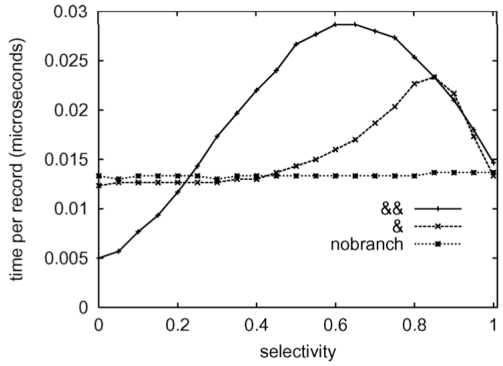

Figure 1: A comparison of the three algorithms on a Pentium III performed in [Ross 2004]

From this graph, we can conclude that an optimal algorithm, one that combined features from all three algorithms at the appropriate selectivities, could easily outperform any one algorithm. [Ross 2004] calls such an algorithm a Mixed Algorithm, and provides an example below:

```
/* A Mixed Algorithm */
for(i=0; i < number_of_records; i++) {
    if(f1(r1[i]) & f2(r2[i]) && f3(r3[i])) {
        answer[j] = i;
        j += (f4(r4[i]) & ... & fk(rk[i]));
    }
}
```

To create such a mixed algorithm, one could look at all the combinations of expressions in the if condition and then calculate their cost. However, this form considers too many combinations and takes too long to compute. Instead, [Ross 2004] proposes a normal form for algorithms and defines a cost model based upon the sum of the cost to evaluate each term in a plan, the cost of each & operation that is evaluated, and the cost of each && operation that is evaluated. [Ross 2004] then extends this first formulation to take into account the cost of a branch misprediction, the selectivities, the cost of writing the answer to the result array, and the cost of applying the function f to its argument. After defining a cost model, [Ross 2004] defines a lexicographical ordering for the terms in an optimal plan and creates two metrics, the c-metric and d-metric, for evaluating a given expression. These metrics help eliminate plans that are certainly unoptimal. Finally, [Ross 2004] uses these results to formulate a dynamic programming algorithm to determine the optimal plan. This algorithm builds up subsets of optimal queries and then combines optimal subsets to get the optimal plan. At each step, unoptimal queries are discarded based upon their c-metric and d-metric. Finally, the optimal plan is found, and is converted to the normal form and to the equivalent C-code. For more details on the normal form, cost model, c-metric, d-metric, and the full algorithm, see [Ross 2004].

3 Implementation

Our query optimizer was coded in Java and implements the algorithm in [Ross 2004]. The CPU configuration and the query selectivities are parsed to create an optimized query for each set of selectivities. We then use a set of shell scripts to take the output of our Java program and include it in a test program, branch_mispred.c, that executes the optimized plans and monitors the branch misses. This test program also calculates the approximate number of CPU cycles and instructions per cycle, letting us determine how well each of our optimized plans performs.

4 Experimental Setup

Testing of the algorithm is done through a supplied C-program named branch_mispred.c that simulates a workload of executing a selection filter for 100 million records and then measuring various performance metrics such as the number of branch mispredictions that occur as well as the amount of time elapsed in the query execution.

The program takes as input 4 parameters which represent the selectivities of 4 selection conditions. As a consequence of this, all the experiments we have done were for queries that involve 4 selection conditions. The format followed is that we would do main query each with 10 different runs of the selectivities such that there would be a range of combined selectivity usually going from 0 to 1.0.

The testing was done on the CLIC Lab Network of machines available in the Columbia University Computer Science department. The precise host used was `islamabad.clic.cs.columbia.edu`. It has 24GB of main memory present along with a Xeon X5550 Quad-Core CPU running at 2.67GHz and a cache size of 8MB.

The testing process was automated as much as possible so that many different data collections could be achieved. The benchmark tests were run multiple times and then averaged out to get the mean result. The code used to automate the benchmarking process can be found at online repository: `https://github.com/jervisfm/W4112_Project2`.

We plot many different graphs in analyzing the data collected from the benchmarking process. This includes looking at the elapsed time, instructions per cycle, and branch misprediction rate. As a baseline upon which to judge the performance of the results, we also look at the empirical performance of only singular plans involving all logical ands, all branching ands, and no branching done at all.

5 Results

To test the algorithm, we followed the setup described in 4 and generated selectivities. Our experiments and the associated queries are described below.

5.1 Experiment 1

To start off, we focused on testing the performance of the optimizer over the entire breadth of combined selectivity ranging from 0.0 to 1.0 with the constraint that each selection condition would be equally selective.

The exact subqueries that were given to our optimizer program implemented in stage 2 of the project were the following:

```
0 0 0 0
0.562341325 0.562341325 0.562341325 0.562341325
0.668740305 0.668740305 0.668740305 0.668740305
0.740082804 0.740082804 0.740082804 0.740082804
0.795270729 0.795270729 0.795270729 0.795270729
0.840896415 0.840896415 0.840896415 0.840896415
0.880111737 0.880111737 0.880111737 0.880111737
0.914691219 0.914691219 0.914691219 0.914691219
0.945741609 0.945741609 0.945741609 0.945741609
0.974003746 0.974003746 0.974003746 0.974003746
1 1 1 1
```

We combined all the results of many different runs for the above selectivities and produced the following graphs:

- Elapsed Time
- Instructions Per Clock Cycle
- Branch Misprediction Rate
- Predicted and Actual Performance

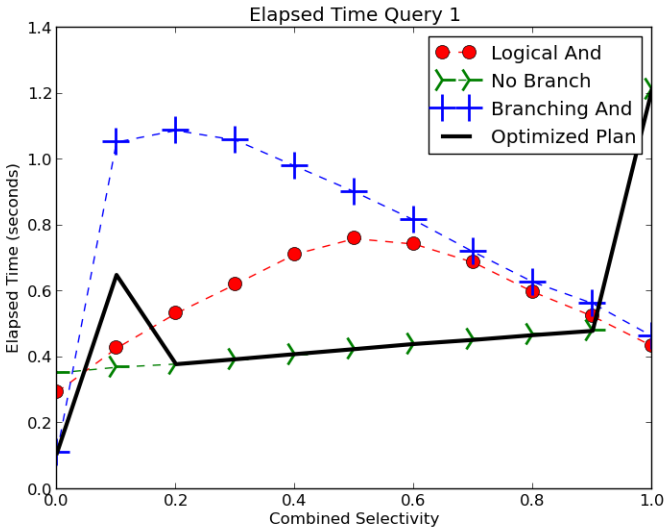

Figure 2: Elapsed time for Query 1

The plot in Figure 2 is a measurement of the actual time spent in evaluating the various selection conditions. We have also included the empirical results of using just one type of plan so that they can serve as a baseline for judgment.

We see that for the combined range of selectivity going from 0.2 to 0.9, the plan that the optimizer picked was efficient and took the smallest amount of time to complete. Using a single-only plan of just logical ANDs as an example, it would have been as much as twice as slow. Thus, we can see the benefit and importance here of picking a good plan which the optimizer has done.

That said, there are two points for which the optimized plan was not fastest. This occurs at the combined selectivity of 0.1 and 1.0. For the 0.1 case the plan the optimizer picked was a hybrid of regular branching AND and logical AND, so we see that the performance for that plan was in between those two. The point at combined selectivity of 1.0 is different in that it is not a hybrid plan at all. Given this, it is strange that the optimizer did not pick the other singular plans which as we see from the graph perform much better. We first suspected that there may have been an issue with an implementation of the optimizer itself in this particular case. We proceeded to check the code again and stepped through it for this particular case. We concluded that the optimizer code is doing the right thing in the context of information that it has. In other words, it sees the no-branch plan to be the best choice. This suggests then then that there may be a mismatch between predicted performance of the model and actual performance, which we discuss later in this section.

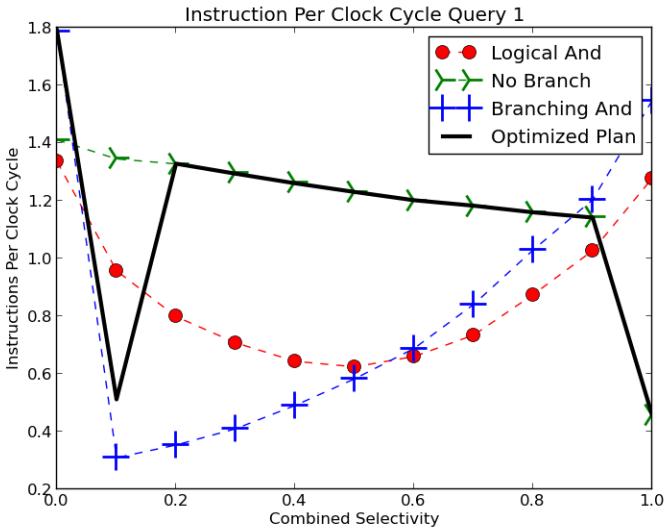

Figure 3: Instructions per Clock Cycle for Query 1

In Figure 3, we plot a measurement of the (average) instructions processes in a single CPU clock cycle. Again, we have also included the empirical results of using just one type of plan so that they can serve as a baseline for judgment.

Ideally, we want to have as many instructions in a single clock cycle as possible. We see that the optimized plan does the best in the range of combined selectivity 0.2 to 0.9. This really shows that the plan the optimizer picked did a good job of maximizing the throughput of instructions processed in a single cycle. This also helps to explain why in this range the plans took the least amount of time, as we have seen already.

However, for the two anomalous points at combined selectivities of 0.1 and 1.0, we see that there is a dip in instruction throughput. As hinted at before, a possible reason for this is a possible mismatch between the prediction performance derived from the cost model and the actual performance for this case.

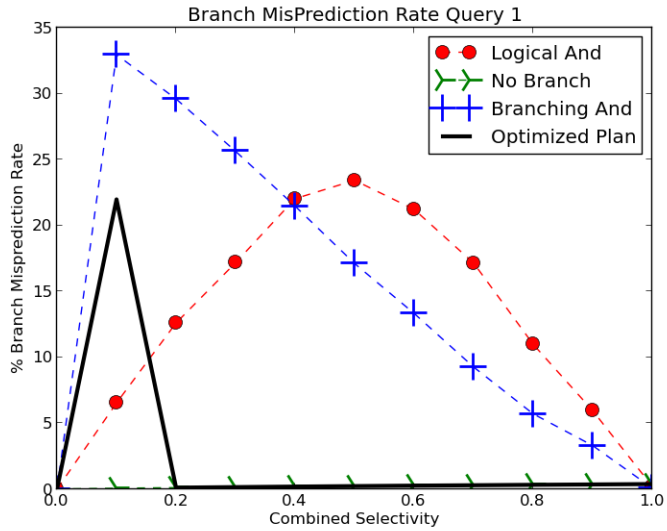

Figure 4: Branch Misprediction Rates for Query 1

In Figure 4, we plot the rate at which branch mispredictions occurred at the the various levels of combined

selectivity. We include the empirical results of using just one type of plan so that they can serve as a baseline for judgment.

Again, we see that the optimal plan picked by the optimizer does really well this regard. For combined selectivities ranging from 0.2 to 1.0 in this instance, we see that the branch misprediction rate is very small and close to zero. This shows the efficacy with which the optimizer was able to optimize and minimize the branch misprediction penalty.

We still have the anomalous result at combined selectivity of 0.1. This was a hybrid plan so we see that while the misprediction rate was high, it was not as high as that for the Branching AND plan.

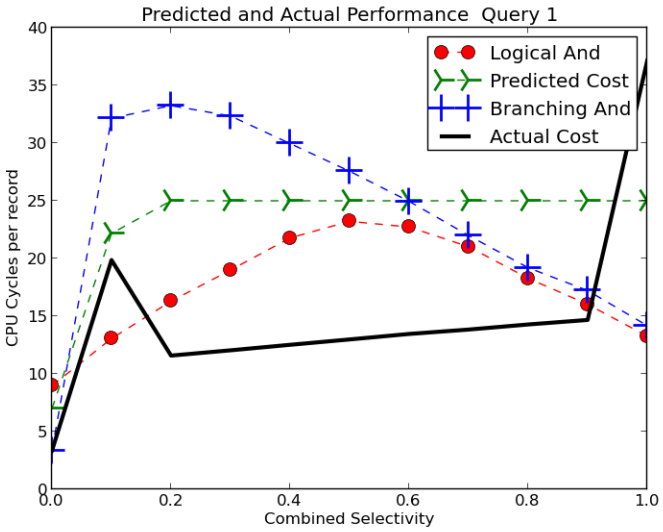

Figure 5: Predicted and Actual Performance for Query 1

To better understand and explain what is happening in anomalous results we have obtained such far, it is helpful to look at the plot of predicted and actual performance, as is shown in 5. The predicted performance was obtained from the result of the optimizer program. The other included graphs of logical-AND and branching-AND are empirical values obtained from experimentation.

In analyzing these anomalous results, it was suggested that a part of the reason for why they occurred was that in that particular case, the cost model did not turn out to be very predictive. It is possible that there is a mismatch and that what the optimizing algorithm determined to be the best plan based on the cost model turned out not to be the case in actual reality. This supposition is borne out by in 5, where we see there is a gap between the predicted performance and actual performance. The gap between the two is not insignificant, which suggests that perhaps there is an opportunity to further tune the parameters of the cost model in order to make it more predictive. Unfortunately, due to time limitations, this option was not pursued but would be an interesting avenue to consider in future works.

5.2 Experiment 2

For query two, we took the first query and then tweaked a pair of parameters such that the selectivity of one term was increased slightly while the selectivity of the other was decreased slightly as well. The exact selectivities used were the following:

```
0 0 0 0
0.590 0.536 0.585 0.541
0.702 0.637 0.695 0.643
0.777 0.705 0.770 0.712
0.835 0.757 0.827 0.765
```

```
0.883  0.801  0.875  0.809
0.924  0.838  0.915  0.846
0.960  0.871  0.951  0.880
0.993  0.901  0.984  0.909
0.993  0.955  0.984  0.964
1  1  1  1
```

We combined all the results of many different runs for the above selectivities and produced the following graphs:

- Elapsed Time
- Instructions Per Clock Cycle
- Branch Misprediction Rate
- Predicted and Actual Performance

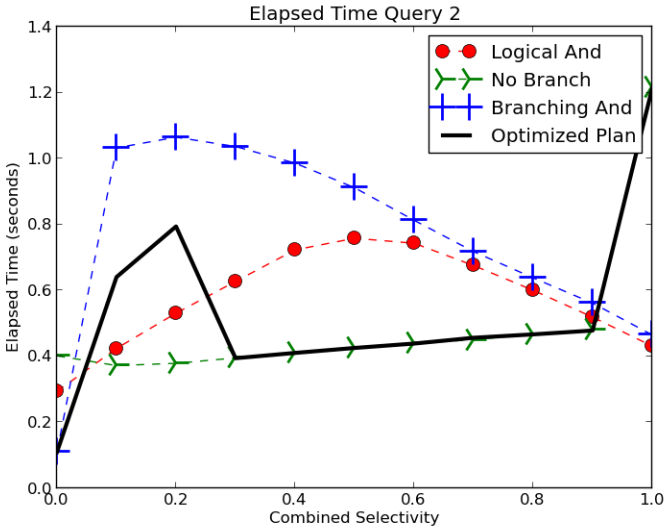

Figure 6: Elapsed time for Query 2

The plot in Figure 6 is a measurement of the actual time spent in evaluating the various selection conditions. We have also included the empirical results of using just one type of plan so that they can serve as a baseline for judgment.

We see that for the combined range of selectivity going from 0.3 to 0.9, the plan that the optimizer picked was efficient and it took the smallest amount of time to complete. Using a single-only plan of just branching ANDs as an example, it would have been more than twice as slow. The improvement over the simple logical-AND, which not as great in comparison to Branching And in certain cases, is still considerably slower. The difference between these is as much as a factor of two. Again, we can see the benefit and importance here of picking a good plan.

There are also two points at combined selectivity of about 0.1 and 0.2 where the optimizer picked hybrid plans consisting of a mixture of logical-AND terms and regular branching-AND terms. As such, the performance for these cases is between that of the single logical-AND and branch-AND plan.

Lastly, note that the anomalous result at the combined selectivity of 1.0 still remains here. Again, this is likely to have caused by a mismatch between the expected performance as predicted by the model and the true performance that actually happened on the test machine.

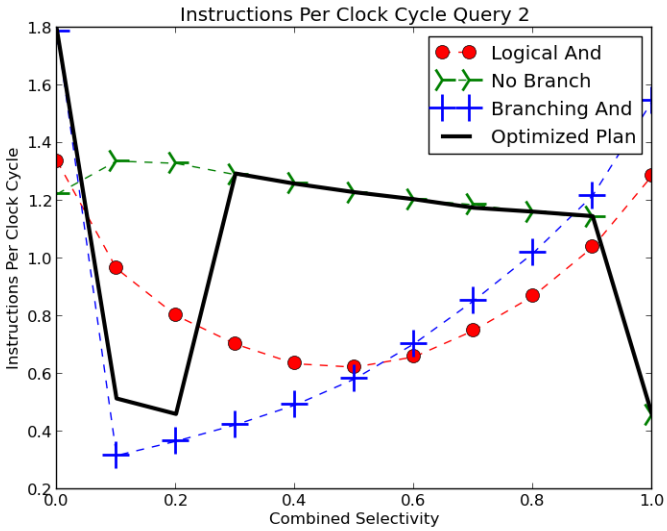

Figure 7: Instructions per Clock Cycle for Query 2

In Figure 7, we plot a measurement of the (average) instructions processes in a single CPU clock cycle. Again, we have also included the empirical results of using just one type of plan so that they can serve as a baseline for judgment.

In this graph, higher is better and ideally we would want to have the maximum number of instructions executed in a single clock cycle. We see that the optimized plan does the best for combined selectivities between 0.3 to 0.9. This shows that the optimizer was able to pick (in this instance) a good plan that had a high throughput in terms of number of instructions executed in a single cycle. This also helps to explain why in this range the optimal plan took the least amount of time, as we have previously seen.

However, for the anomalous point at combined selectivities of 1.0, we see that there is a dip in instruction throughput. As hinted at before, a possible reason for this is a possible mismatch between the prediction performance derived from the cost model and the actual performance for this case. There is also a dip in instruction throughput at combined selectivity of 0.1 and 0.2. This can be explained by taking into account the fact that these plans hybrid plans and so they have components of both logical-AND plans and branching-AND plans. The net result is that their performance ends up being somewhere in between these two.

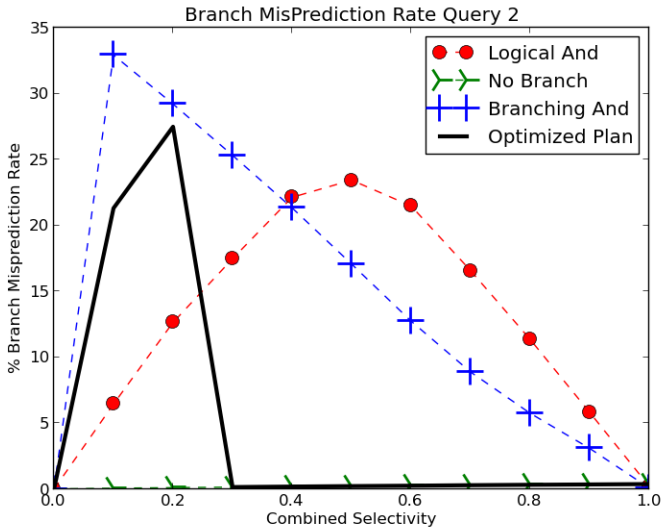

Figure 8: Branch Misprediction Rates for Query 2

In Figure 8, we plot the rate at which branch mispredictions occurred at the the various levels of combined selectivity. We include the empirical results of using just one type of plan so that they can serve as a baseline for judgment.

For the combined selectivities ranging from 0.2 to 1.0 in this experiment, we see that plan selected by the optimizer was able to achieve very low range of branch mispredictions. This again shows the efficacy with which the optimizer was able to minimize the number of branch mispredictions that occur.

We still have the odd results at combined selectivities of 0.1 and 0.2. These were hybrid plans so we see that while the misprediction rate was high, it was not as high as that for the Branching AND only plan.

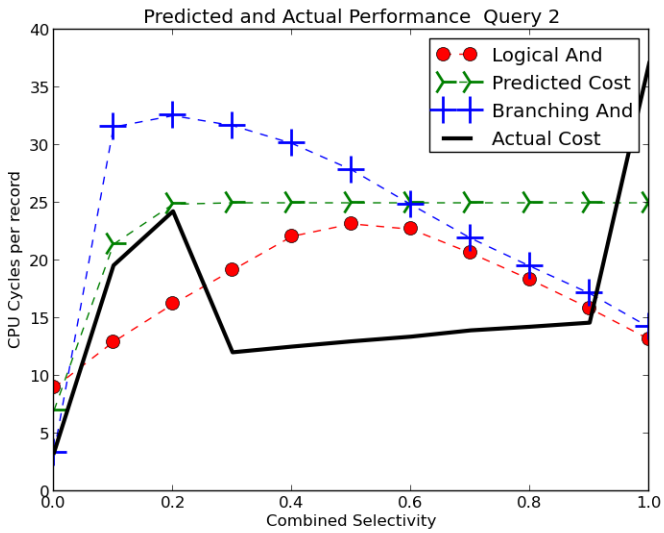

Figure 9: Predicted and Actual Performance for Query 2

To better understand and explain what is happening in some of the results we have obtained, it is helpful to look at the plot of predicted and actual performance, as is shown in 9. The predicted performance was

obtained from the results produced by the optimizer program. The other included graphs of logical-AND and branching-AND are empirical values obtained from experimentation.

One of the suggested reasons for the some of the odd results that we have seen is that they perhaps arose due to a lack of predictive power of the cost model in some particular instances. It is possible that there is a mismatch and that what the optimizing-algorithm determined to be the best plan based on the cost model turned out not to be the case in reality.

Again, this hypothesis was borne out by in 9, where we see a gap between the predicted performance and actual performance. The gap between the two is significant between the ranges of combined selectivity 0.3 to 1.0. It is much better for combined selectivity of 0.1 and 0.2 where there were hybrid plans involved. This indicates that the lack of predictive power of the cost model is worst for singular plans and it may be better for hybrid plans. All things considered, this is further evidence that there is a need to further tune the parameters of the cost model in order to make it more predictive. Unfortunately, due to time limitations, this option was not pursued.

5.3 Experiment 3

Lastly, we decided to focus on the performance whereby queries have very small selectivities to get further insight on the some of the result we have seen so far. The range we focus on was 0.0 to 0.1 and the precise selectivities used are given below:

```
0 0 0 0
0.316227766 0.316227766 0.316227766 0.316227766
0.376060309 0.376060309 0.376060309 0.376060309
0.416179145 0.416179145 0.416179145 0.416179145
0.447213595 0.447213595 0.447213595 0.447213595
0.472870805 0.472870805 0.472870805 0.472870805
0.4949232 0.4949232 0.4949232 0.4949232
0.514368672 0.514368672 0.514368672 0.514368672
0.53182959 0.53182959 0.53182959 0.53182959
0.547722558 0.547722558 0.547722558 0.547722558
0.562341325 0.562341325 0.562341325 0.562341325
```

Once more, we combined all the results of many different runs for the above selectivities and produced the following graphs:

- Elapsed Time
- Instructions Per Clock Cycle
- Branch Misprediction Rate
- Predicted and Actual Performance

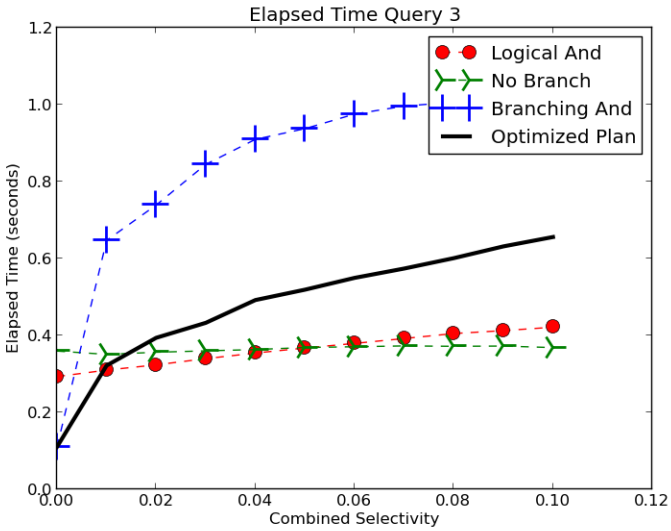

Figure 10: Elapsed time for Query 3

The plot in Figure 10 is a measurement of the actual time spent in evaluating the various selection conditions. We have also included the empirical results of using just one type of plan so that they can serve as a baseline for judgment.

We see that for the combined range of selectivity going from 0.0 to 0.01, the plan that the optimizer picked was efficient and took the smallest amount of time to complete. Using a single-only plan of just branching ANDs as an example, it would have been as much as twice as slow.

However, these selectivities perform poorly for a much wider range than before. The optimal plan performed poorly for selectivities between 0.02 and 0.1. For all these selectivities, the optimizer picked a hybrid plan between branching AND and logical AND, which turned out to perform poorly in comparison to just Algorithm Logical And or Algorithm No Branch. We thus concluded the same thing as before: the performance model must be overestimating the cost of Logical And and No Branch, and thus prefers a hybrid plan instead of a single-only plan. This was surprising because in the previous two queries, the optimal plan only performed worse for a few combined selectivities. However, in Query 3, the optimal plan performs worse for far more selectivities, and indicates that there may be serious shortcomings with the cost model used here.

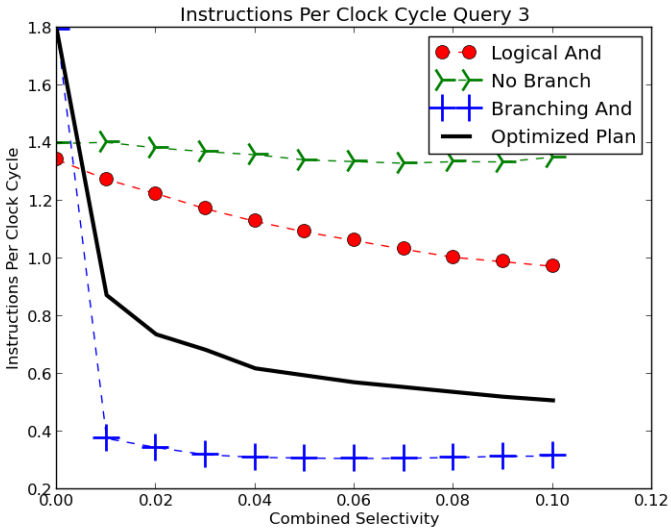

Figure 11: Instructions per Clock Cycle for Query 3

In Figure 11, we plot the instructions per clock cycle for Query 3. Again, we have also included the empirical results of using just one type of plan so that they can serve as a baseline for judgment.

The optimal plan, a hybrid plan, performs better than Branching And but worse than Logical And or No Branch. This again makes sense if we remember that the costs for a single-only plan are being overestimated. The optimal plan does do better for the combined selectivity 0.0, but this performance quickly drops off as the combined selectivity increases.

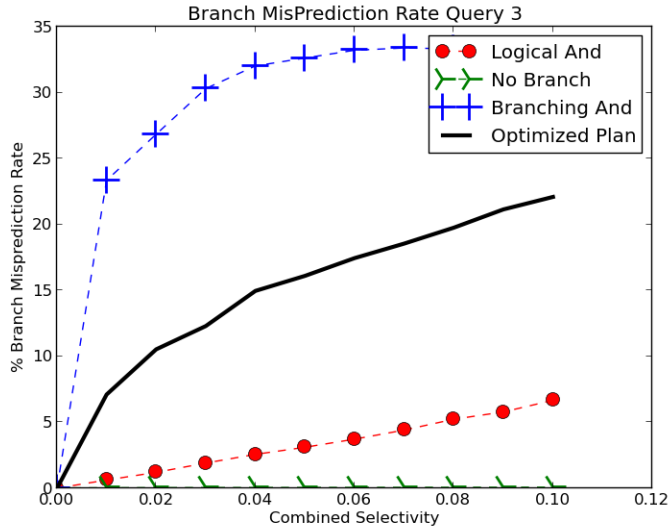

Figure 12: Branch Misprediction Rates for Query 3

In Figure 12, we plot the rate at which branch mispredictions occurred at the the various levels of combined selectivity. We include the empirical results of using just one type of plan so that they can serve as a baseline for judgment.

Here, we again see that Algorithm No Branch and Algorithm Logical And perform better than the optimized plan, which makes sense since the optimizer chose a hybrid plan for all the selectivities. However,

we do note that the optimized plan performs much better than Algorithm Branching And, which indicates the optimization algorithm is correctly estimating the comparative cost of Branching And, and properly chooses a mixed plan instead of a single-only Branching And plan.

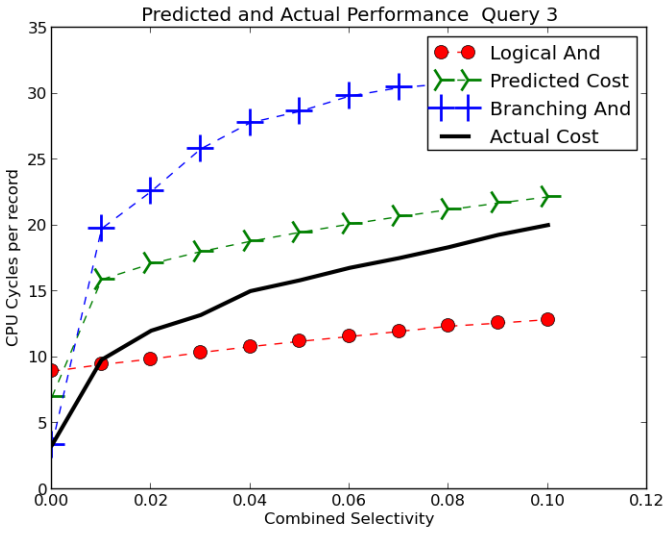

Figure 13: Predicted and Actual Performance for Query 3

Finally, we plotted the predicted and actual performance of Query 3, as shown in 13. We can see that our suspicions were confirmed: the optimization algorithm is overestimating the cost of the Logical And algorithm. In particular, Logical And performs very well for selectivities above 0.01, and this reflects the performance gap between the optimal plan and the lowest cost plan. The Branching And plan, on the other hand, always has a higher cost, which means that the algorithm properly chose a more optimal plan than Algorithm Branching And. Improvements to the optimization algorithm or the cost model to better account for the cost of Logical And and No Branch could perhaps lead the optimizer to chose a better plan.

6 Conclusion

From the experimentation we have done, we see that the proposed algorithm for optimizing the runtime of selection conditions that occur in main memory by Kenneth Ross in [Ross 2004] can be effective in picking plans which do well.

In Experiments 1 and 2, we looked at queries where the selectivities of the four conditions were equal and unequal respectively, and we looked at a range of combined selectivities from 0 to 1. We found that the optimizer was able to, for the most part, pick plans which were efficient in the amount of time they took to run. There were also a few instances for very small selectivities in which the optimizer did not pick the absolute best plan. Close inspection of these results led to the conclusion that they had occurred because of the mismatch between the predicted cost of a plan based on the cost model and the true cost of executing that plan.

In Experiment 3, we focused on queries with very small selectivities to better understand the odd results in Experiments 1 and 2. We found that in the case where we had fairly small combined selectivities, the plans the optimizer picked were not the best. Since the optimizer picked hybrid plans, the plans' performance was an amalgamation of the performance of the different pure plans.

The critical element here is the cost model and how predictive it is. Our experimentation indicates that the cost model for the specific parameters in use can have poor predictive power in correctly estimating the true cost of singular plans in comparison to hybrid plans. Ross does acknowledge that the predictive aspect of the model can be an issue, writing in [Ross 2004] that "for architecture-dependent reasons [one] cannot

expect [the] cost models to be exact cost estimates". In our case, we made the assumption that the supplied parameters we used in the cost model were valid for the CLIC machine where we did our experiments on. In future work, it would be worthwhile to validate these parameters and fine-tune them if necessary so that there is minimum disparity between computed and actual cost of plans.

Hence, we conclude that the optimizing algorithm described in [Ross 2004] can be effective with the condition that the cost model that is used to pick the best plan must be reflective of the true cost. Indeed, even in the experiments that had a disparity between actual and predicted cost, the performance we saw was still nonetheless respectable. Therefore, with an appropriately tuned cost-model, we expect that the performance would have been better still.

References

Kenneth A. Ross. 2004. Selection conditions in main memory. *ACM Trans. Database Syst.* 29, 1 (March 2004), 132–161. DOI:http://dx.doi.org/10.1145/974750.974755

Appendix

A Output from branch_mispred.c for Query 1

This output is averaged over multiple runs. The true raw output for each query can be seen on `https://github.com/jervisfm/W4112_Project2`, in the Benchmark/Results directory.

```
Benchmark name: result_optimal_q1 | Combined Selectivity 0.0
Elapsed Time: 0.110082507 seconds
Cpu Cycles: 335172211.5
Instructions: 600109923.5
IPC : 1.790649
Branch Misses: 5912.0
Branch Instructions: 200018954.0
Branch MisPredict Rate: 0.002956 %


==========
Benchmark name: result_optimal_q1 | Combined Selectivity 0.1
Elapsed Time: 0.6501464845 seconds
Cpu Cycles: 1986503530.0
Instructions: 1016492380.0
IPC : 0.511716
Branch Misses: 51585230.0
Branch Instructions: 234672681.5
Branch MisPredict Rate: 21.9817045 %


==========
Benchmark name: result_optimal_q1 | Combined Selectivity 0.2
Elapsed Time: 0.379477024 seconds
Cpu Cycles: 1158649378.5
Instructions: 1539377886.5
IPC : 1.3286925
Branch Misses: 133588.0
Branch Instructions: 106346747.5
Branch MisPredict Rate: 0.1256155 %


==========
Benchmark name: result_optimal_q1 | Combined Selectivity 0.3
Elapsed Time: 0.3944069145 seconds
Cpu Cycles: 1204375033.0
Instructions: 1558426564.0
IPC : 1.294044
Branch Misses: 180733.5
Branch Instructions: 109415115.0
Branch MisPredict Rate: 0.1651815 %


==========
Benchmark name: result_optimal_q1 | Combined Selectivity 0.4
Elapsed Time: 0.410035014 seconds
Cpu Cycles: 1252156602.5
Instructions: 1578518498.5
IPC : 1.260717
Branch Misses: 231022.5
Branch Instructions: 112651524.0
```

```
Branch MisPredict Rate: 0.205077 %

==========
Benchmark name: result_optimal_q1 | Combined Selectivity 0.5
Elapsed Time: 0.425188422 seconds
Cpu Cycles: 1298639166.5
Instructions: 1598045161.0
IPC : 1.2306215
Branch Misses: 279619.0
Branch Instructions: 115796849.0
Branch MisPredict Rate: 0.2414725 %

==========
Benchmark name: result_optimal_q1 | Combined Selectivity 0.6
Elapsed Time: 0.4407320025 seconds
Cpu Cycles: 1346190881.0
Instructions: 1618045845.5
IPC : 1.201971
Branch Misses: 329411.5
Branch Instructions: 119018484.5
Branch MisPredict Rate: 0.27677 %

==========
Benchmark name: result_optimal_q1 | Combined Selectivity 0.7
Elapsed Time: 0.453224897 seconds
Cpu Cycles: 1384220180.5
Instructions: 1637513066.5
IPC : 1.183031
Branch Misses: 375765.0
Branch Instructions: 122154149.5
Branch MisPredict Rate: 0.307614 %

==========
Benchmark name: result_optimal_q1 | Combined Selectivity 0.8
Elapsed Time: 0.4676160815 seconds
Cpu Cycles: 1428333270.5
Instructions: 1656827768.0
IPC : 1.1600325
Branch Misses: 424608.0
Branch Instructions: 125265505.5
Branch MisPredict Rate: 0.338966 %

==========
Benchmark name: result_optimal_q1 | Combined Selectivity 0.9
Elapsed Time: 0.480467081 seconds
Cpu Cycles: 1467555200.0
Instructions: 1675602202.5
IPC : 1.141843
Branch Misses: 470732.5
Branch Instructions: 128304394.0
Branch MisPredict Rate: 0.3668875 %

==========
Benchmark name: result_optimal_q1 | Combined Selectivity 1.0
```

```
Elapsed Time: 1.214532375 seconds
Cpu Cycles: 3713020065.5
Instructions: 1696655305.0
IPC : 0.4569545
Branch Misses: 523498.0
Branch Instructions: 131695677.5
Branch MisPredict Rate: 0.3975055 %
```

==========

B Output from branch_mispred.c for Query 2

This output is averaged over multiple runs. The true raw output for each query can be seen on `https://github.com/jervisfm/W4112_Project2`, in the Benchmark/Results directory.

```
Benchmark name: result_optimal_q2 | Combined Selectivity 0.0
Elapsed Time: 0.1100360155 seconds
Cpu Cycles: 334967474.0
Instructions: 600100158.0
IPC : 1.7915735
Branch Misses: 5680.0
Branch Instructions: 200017281.5
Branch MisPredict Rate: 0.00284 %

==========
Benchmark name: result_optimal_q2 | Combined Selectivity 0.1
Elapsed Time: 0.6406970025 seconds
Cpu Cycles: 1957630823.5
Instructions: 1007948878.0
IPC : 0.5148895
Branch Misses: 49598478.5
Branch Instructions: 232699001.5
Branch MisPredict Rate: 21.3123705 %

==========
Benchmark name: result_optimal_q2 | Combined Selectivity 0.2
Elapsed Time: 0.794161916 seconds
Cpu Cycles: 2426975102.0
Instructions: 1118984284.0
IPC : 0.461186
Branch Misses: 67838953.0
Branch Instructions: 246743206.5
Branch MisPredict Rate: 27.4899725 %

==========
Benchmark name: result_optimal_q2 | Combined Selectivity 0.3
Elapsed Time: 0.3946009875 seconds
Cpu Cycles: 1204626798.0
Instructions: 1559466832.5
IPC : 1.2946265
Branch Misses: 183836.0
Branch Instructions: 109583241.5
Branch MisPredict Rate: 0.167758 %

==========
Benchmark name: result_optimal_q2 | Combined Selectivity 0.4
Elapsed Time: 0.4105899335 seconds
Cpu Cycles: 1254010362.5
Instructions: 1578917932.0
IPC : 1.2591585
Branch Misses: 231653.5
Branch Instructions: 112715817.5
Branch MisPredict Rate: 0.20552 %

==========
```

```
Benchmark name: result_optimal_q2 | Combined Selectivity 0.5
Elapsed Time: 0.4256564375 seconds
Cpu Cycles: 1299697400.0
Instructions: 1598208229.5
IPC : 1.2297815
Branch Misses: 279921.0
Branch Instructions: 115823307.0
Branch MisPredict Rate: 0.241673 %

====
Benchmark name: result_optimal_q2 | Combined Selectivity 0.6
Elapsed Time: 0.438984394 seconds
Cpu Cycles: 1340273073.0
Instructions: 1615481725.5
IPC : 1.2054355
Branch Misses: 322246.0
Branch Instructions: 118605484.5
Branch MisPredict Rate: 0.2716865 %

====
Benchmark name: result_optimal_q2 | Combined Selectivity 0.7
Elapsed Time: 0.456543565 seconds
Cpu Cycles: 1394551419.5
Instructions: 1639696442.5
IPC : 1.1759025
Branch Misses: 381666.5
Branch Instructions: 122505994.0
Branch MisPredict Rate: 0.3115495 %

====
Benchmark name: result_optimal_q2 | Combined Selectivity 0.8
Elapsed Time: 0.466873407 seconds
Cpu Cycles: 1426069273.5
Instructions: 1657650299.5
IPC : 1.1624185
Branch Misses: 427403.0
Branch Instructions: 125398503.0
Branch MisPredict Rate: 0.340836 %

====
Benchmark name: result_optimal_q2 | Combined Selectivity 0.9
Elapsed Time: 0.4784954785 seconds
Cpu Cycles: 1461356767.5
Instructions: 1676010948.5
IPC : 1.1469295
Branch Misses: 471634.5
Branch Instructions: 128355741.0
Branch MisPredict Rate: 0.3674435 %

====
Benchmark name: result_optimal_q2 | Combined Selectivity 1.0
Elapsed Time: 1.2133735415 seconds
Cpu Cycles: 3709428843.0
Instructions: 1696493856.0
```

```
IPC : 0.457351
Branch Misses: 521415.0
Branch Instructions: 131661455.5
Branch MisPredict Rate: 0.3960275 %
```

==========

C Output from branch_mispred.c for Query 3

This output is averaged over multiple runs. The true raw output for each query can be seen on `https://github.com/jervisfm/W4112_Project2`, in the Benchmark/Results directory.

```
Benchmark name: result_optimal_q3 | Combined Selectivity 0.0
Elapsed Time: 0.1103219985 seconds
Cpu Cycles: 335398300.5
Instructions: 600101785.5
IPC : 1.789343
Branch Misses: 5660.0
Branch Instructions: 200017551.5
Branch MisPredict Rate: 0.0028295 %

==========
Benchmark name: result_optimal_q3 | Combined Selectivity 0.01
Elapsed Time: 0.3216691015 seconds
Cpu Cycles: 982102461.5
Instructions: 857361695.5
IPC : 0.873212
Branch Misses: 14967389.5
Branch Instructions: 210543873.5
Branch MisPredict Rate: 7.1088935 %

==========
Benchmark name: result_optimal_q3 | Combined Selectivity 0.02
Elapsed Time: 0.3938270805 seconds
Cpu Cycles: 1202704904.5
Instructions: 886503458.5
IPC : 0.7371505
Branch Misses: 22708188.5
Branch Instructions: 215503447.5
Branch MisPredict Rate: 10.5372745 %

==========
Benchmark name: result_optimal_q3 | Combined Selectivity 0.03
Elapsed Time: 0.4327659605 seconds
Cpu Cycles: 1321511353.5
Instructions: 903069682.5
IPC : 0.68337
Branch Misses: 26829895.5
Branch Instructions: 218080007.5
Branch MisPredict Rate: 12.3021375 %

==========
Benchmark name: result_optimal_q3 | Combined Selectivity 0.04
Elapsed Time: 0.4919844865 seconds
Cpu Cycles: 1502723894.0
Instructions: 930268938.5
IPC : 0.6191365
Branch Misses: 33267454.0
Branch Instructions: 222280512.0
Branch MisPredict Rate: 14.9664025 %

==========
```

```
Benchmark name: result_optimal_q3 | Combined Selectivity 0.05
Elapsed Time: 0.518921614 seconds
Cpu Cycles: 1585142115.0
Instructions: 942986256.0
IPC : 0.5948995
Branch Misses: 36087805.0
Branch Instructions: 224043641.5
Branch MisPredict Rate: 16.10735 %

====
Benchmark name: result_optimal_q3 | Combined Selectivity 0.06
Elapsed Time: 0.5496418475 seconds
Cpu Cycles: 1678967028.0
Instructions: 958411475.0
IPC : 0.570848
Branch Misses: 39522749.0
Branch Instructions: 226441151.5
Branch MisPredict Rate: 17.453086 %

====
Benchmark name: result_optimal_q3 | Combined Selectivity 0.07
Elapsed Time: 0.573868036 seconds
Cpu Cycles: 1753075544.5
Instructions: 971703656.0
IPC : 0.5543125
Branch Misses: 42359120.5
Branch Instructions: 228300288.5
Branch MisPredict Rate: 18.554118 %

====
Benchmark name: result_optimal_q3 | Combined Selectivity 0.08
Elapsed Time: 0.600888014 seconds
Cpu Cycles: 1835809996.0
Instructions: 986746691.5
IPC : 0.5374995
Branch Misses: 45540512.0
Branch Instructions: 230459345.5
Branch MisPredict Rate: 19.7604575 %

====
Benchmark name: result_optimal_q3 | Combined Selectivity 0.09
Elapsed Time: 0.6314884425 seconds
Cpu Cycles: 1929421487.5
Instructions: 1004950523.5
IPC : 0.5208585
Branch Misses: 49295871.0
Branch Instructions: 233112325.5
Branch MisPredict Rate: 21.1461565 %

====
Benchmark name: result_optimal_q3 | Combined Selectivity 0.1
Elapsed Time: 0.655854106 seconds
Cpu Cycles: 2003520247.5
Instructions: 1018276121.5
```

```
IPC : 0.5082455
Branch Misses: 51895519.5
Branch Instructions: 234894253.0
Branch MisPredict Rate: 22.0931245 %

=====
```