Final Report

Moh Osman

December 2019

1 Design

1.1 Milestone One

The biggest obstacles when implementing milestone 1 were the load and select functions.

1.1.1 Load

For my load function I read the first line to determine the number of columns. For each column:

- client sends 1024 bytes (max number of bytes you can send through the socket) until the entire column is loaded.
- server receives these packets and loads each column accordingly.
- in terms of speed, sending the column data as ascii characters greatly hindered my load time, so I send ints over the socket

1.1.2 Variable Pool

Each query has an associated client context. Within this client context I store all results in a Result array. I include a find_result function which allows me to find previously stored results for fetch and arithmetic operations.

1.1.3 Selects

For milestone 1, I store all of my select results as a bitvector. The benefit of a bitvector is it saves a lot of space (1 byte per observation) while storing indices would require 4 bytes per observation. For point queries and low selectivity queries, the bitvector would not have better storage, and would subsequently slow down the fetch operation. My fetch operator has to scan the entire bitvector to find the corresponding result, so bitvectors would hurt the speed of my fetch's for point queries (and lowly selective queries)

I ran into an issue early on where my selects were by far my slowest operation. To optimize, I removed all if statements and replaced it with the ternary operator and gained a significant speed up. This is because the ternary operator's result will be used regardless if the condition is true or false. Meanwhile, the if statement relies on branch prediction, and if the branch predictor is incorrect, then the instruction buffer must be flushed.

Later in Milestone 3, I implement selects that store select results as IDs, which then allowed me to optimize my choice of select storage.

1.1.4 File Catalogue

For persistence of my data I implement a file catalogue system. I store each table as a separate csv file to make use of my load() function. In a file called "catalogue.txt" I store the number of tables in my database and a list of all the .csv's that store the database's tables. Once a client connects to the server it first reads "catalogue.txt" and then proceeds to load all the table csv's.

1.2 Milestone 2

1.2.1 batch query storage

Within my current database I add a char** so that when "batch_queries()" is called, I can store all incoming queries. Additionally I keep a count of the number of queries in the batch so my thread pool can then divide up the queries.

1.2.2 shared scans

Once batch_execute() is called I divide the selects in the batch as evenly as possible between the threads. I have three main functions that execute the queries in the batch.

- **dispatch_thread** - This function calls pthread_create and gives each thread a section of the chandle_table (my variable pool) to write the results of the selects.
- **selects_per_thread** - this function is input into pthread_create and each thread has a number of queries to execute and a section of the chandle_table to write results to. Since each thread is writing to a different section of the variable pool and the selects they are responsible are isolated from other threads', there is no need for any locking mechanism.
- **wait_all** This function makes sure all threads complete their task (parsing the select, executing the select, and writing to the variable pool) before returning.

Through experimentation I found that 2 threads was optimal, which makes sense since my computer only has two cores.

1.3 Milestone 3

I implemented an index as a union between an (value, id) array or an int b-tree.

1.3.1 Clustered Index

For a clustered index I store a copy of the base data within the table itself. I call the qsort function to sort an array of tuples, (value, id). By including the id in the sorting I can propogate the order of the index column to the rest of the columns. I can store multiple clustered indexes by keeping multiple copies of the base data within the table.

1.3.2 Unclustered Index

For an unclusterd index I store either a b-tree whose leafs are (value, id) tuples or I store a (value, id) array.

1.3.3 Btree Implementation and Loading

I tried to make my btree's as read friendly as possible. To do this I stored all my leafs contiguously in memory. Each generation is also stored contiguously in memory and each parent node points to a different part of the child array. Below is a drawing to help illustrate my implementation.

For loading I first sort all the data and fill up my leaf nodes. Then I take the maximum of each leaf node and store that in as many parent nodes as necessary. Each parent node points to a different section of the leaf array. This process continues until only one node (the root) is needed to store a child level's maximum values.

1.3.4 Clustered Selects and Fetch's

There are two types of clustered indexes:

- **array** if the index is an array then I use my own implementation of a binary search to find the position of the starting value in the range and the ending value.
- **btree** I implement a btree search which starts at the root. Since there is only one pointer in the root, I find the appropriate child node by index of the approriate value in the root node. This search continues until I find the appropriate value in the leaf node. I repeat this search for the upper value of the range query.

Once I have these two positions, I simply find the copy of the data with the column's order propagated and iterate from the starting position to the ending position in the column whose values we wish to project.

1.3.5 Unclustered Selects and Fetch's

- **array** My index is stored as an array of (val, id) tuples. Therefore, when I select I store the ID's, so that when I fetch I can index directly into the desired projection column.
- **btree** The leafs contain (val, id) tuples for an unclustered btree. This once again stores the ID's of the principle copy of the data.

1.4 Milestone 4

1.4.1 nested loop join

I implement the basic nested loop join. Given the initial selects, p1 and p2 (stored as id's). I iterate through and find unique matching pairs. If a pair match I record the ID's of the pair. As discussed in class, I make the optimization of choosing the smaller column to be the inner for loop.

1.4.2 simple hash join

1. build hash table on smaller column
2. iterate through other column and check if column values (which is key in hash table) yields any results in the hash table.
3. record matches (record the pairs of ID's if matches were found in the hash table)

The only cache tuning I do is making sure the inner for loop in a nested loop join is the smaller column. Similarly for the simple hash join I build the hash table on the smaller column to minimize cache misses. With more time I would have implemented the grace hash join and built and performed the join operation on subsets of my columns. The column subsets should be small enough so my hash table can always fit in the L3 cache.

1.5 Milestone 5

- **Inserts** - For inserts I create a buffer pool which stores items to be inserted. If I need to do a select on an index, I also have to check that the buffer pool if it is not empty. Once the buffer pool reaches capacity, it is flushed and the btree is rebuilt
- **Deletes** - For deletes I do lazy updates. I store a bitvector in my table which begins as a vector of 0's. As a delete comes in, I use the or operation between this bitvector and bitvector which stores the items selected for the delete. Then when I fetch, I check that the item has a 0 at the index in the delete bitvector, before I fetch it. Lastly, when the shutdown command is called I only write the rows that have a 0 in the delete vector, to disk.

2 Experiments

2.1 M1 - If Statement Versus Ternary Operator

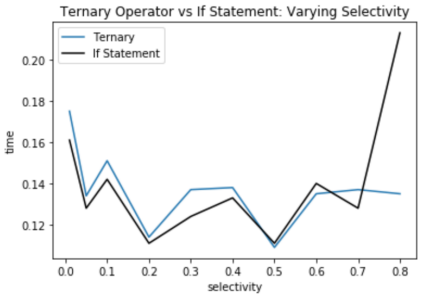
2.1.1 Motivation

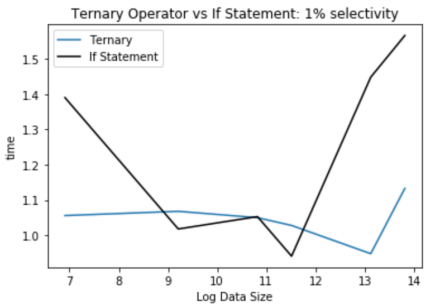
In class we often discussed the costly nature of if statements. The cost mainly comes from branch misprediction where the processor tries to guess which branch will be taken, so that it can load as many instructions as possible in the instruction buffer. However, if it guesses incorrectly then the entire buffer must be flushed and a new set of instructions must be loaded. The ternary operator, on the other hand, evaluates to a value and if you use always use this value, there is no need for branch prediction and no instructions must be flushed. In this experiment I aim to see the differences in run time between if statements and the ternary operator. This will give insight as to whether it is worth it to refactor my code to reduce if statements as many if statements as possible.

2.1.2 Setup

To test the difference in run time between these two logical operators I pre select a selectivity: 1%, 50%, 90%. Then I time how long it takes each operator to complete 100 select queries at the specified selectivity. As my indpendent variable, I vary the data size from 1000 elements to 1000000 elements. Furthermore I run an experiment to see how the run time changes as a function of the query selectivity.

2.1.3 Results





From the figures above we see the only conclusive graph was when the selectivity is 50%. For this graph we see that the ternary operator is consistently faster than the if statement. Another observation from the results is that the run time for the ternary operator seems much more stable. In contrast the if statement run time is all over the place and the gap between ternary operator run time and if statement run time is very volatile. This may indicate that

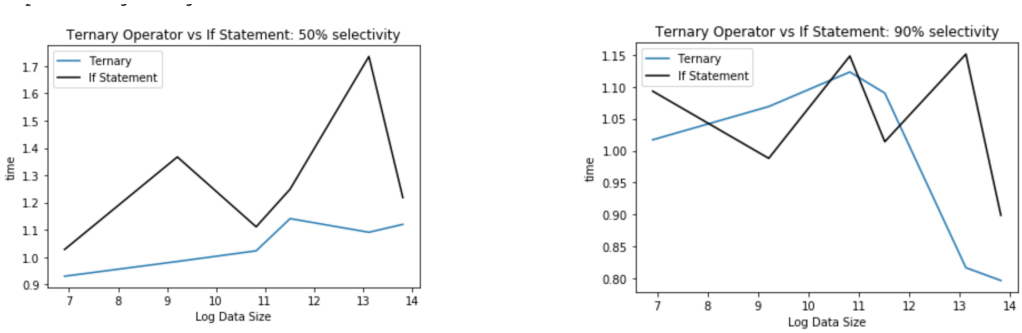


Figure 2: Ternary vs if statement

the if statement branch prediction gets lucky sometime and there is no observed difference between the two operators. However for extremely slow if statements, there were probably many mispredictions which slowed down the querying time.

2.2 M2-Optimal Number of Threads

2.2.1 Motivation

It is very wasteful to run a range query and load data into memory only to need that same memory for later selects. Therefore by batching the selects we can hopefully minimize cache misses. Furthermore we can parallelize our select queries so compute time can be significantly reduced. We should be careful, however, that we do not allocate too many threads, since after some point we will not gain any benefit. The goal of this experiment is to tune the number of threads to minimize run time and cache misses.

2.2.2 Setup

The setup of this experiment is similar to the comparison between test 16 and test 17 on the grading server. However, I do not include the cost of fetch's and print's so I can get a more accurate comparison between the batched selects and the normal selects. I batch 100 selects and record time and cache misses as a function of number of threads. My machine's CPU has 2 cores so I expect two threads to be optimal.

2.2.3 Results

From the figure we can see that there is a drastic drop from 1 thread to 2 threads. The actual minimum occurs around 4 threads. However, it is barely better than 2 threads and I suspect variance in run time accounts for this different. As the number of threads increases beyond 4, the run time is slightly higher but for the most part levels off. I did not include a figure for cache misses as a function of thread numbers because there were no differences in the number of cache misses

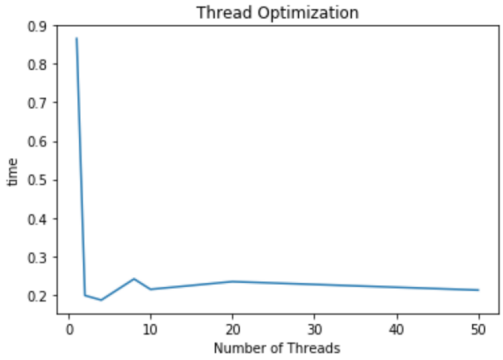


Figure 3: Run time as a function of thread number

when I ran the experiment. This might be due to a flaw in my setup, so my parallelization may not be tuned to minimize these cache misses.

2.3 M3 - unclustered btree select versus full scan

2.3.1 Motivation

In this experiment I aim to figure out when it is preferable to select using a btree and when I should do a full scan of the data instead. We discussed in class that if a range query is too selective we begin to lose the benefit of using a btree index for the select. For high selectivities we should just scan the data as there is no added benefit for using the index. After this experiment I will be able to optimize my select queries by proxying the selectivity. I proxy selectivity through sampling the column for the query and get a rough estimate of the selectivity. Given this information I can either do a full scan or use a btree index if available.

2.3.2 Setup

Using the milestone3 code I run test 26 (which fulfills range query with full select) and test 27 (unclustered btree select). I mess around with the the selectivity of my queries. I run 100 queries with the desired selectivity and compare the control (full scan) versus the btree select. These tests were ran on a test size of 100000.

2.3.3 Results

From the above figure we can see that from my experiment, for selectivities greater than 20% it is better to do a full scan instead of a btree select. In class we discussed that the threshold is actually around 1-3% for when you should scan the data instead of use the index. Perhaps my finding is the result of noise

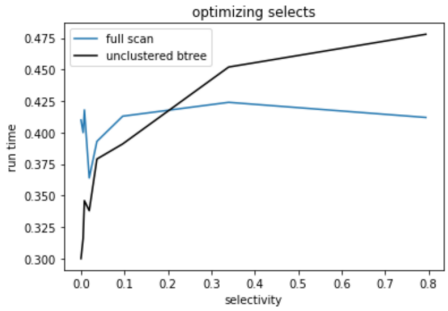


Figure 4: Caption

in the experimentation. Another thing to note, however is that the largest gap in time between a full scan and a btree select are for selectivities less than .05, which would seem to agree more with what we discussed in class.

2.4 M4 - Which column should I build the hash table on

2.4.1 Motivation

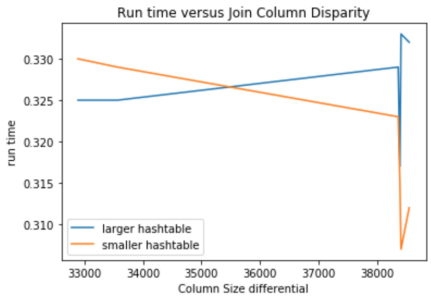
The motivation for this experiment is to see if it really matters which column we use to build the hash table. This experiment is mainly to compare the time benefits of choosing to build the hash table on smaller vs larger column. Furthermore I wish to explore how this choice in hashtable creation will affect the L3 cache misses.

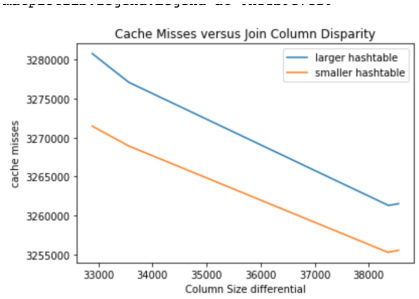
2.4.2 Setup

For this experiment I use test 36 and use a dataset of 50000 elements. I record the disparity between the length of each initial select query. Then I run test 36 with optimization (pick the inner column to build hash table) and run test 36 where I join on the larger table. I use cachegrind to measure my number of cache misses during testing.

2.4.3 Results

We can observe that using the inner column for the hashtable leads to fewer cache misses. However, it is surprising that the gap between the two decreases instead of increases as the disparity grows larger. There does not seem to be too much time benefit until we reach the largest disparity of 38000.





2.5 M5 - Tuning Size of Buffer Pool

2.5.1 Motivation

For my inserts I use a buffer pool which stores values inserted under a certain capacity. If I make the capacity too large, then having to do a scan of a large buffer will outweigh the cost of rebuilding the btree. Therefore the goal of this experiment is to tune the size of my buffer pool, so I can determine a capacity to reach before rebuilding my btree.

2.5.2 setup

I did not have enough time to run this experiment but my setup would involve running 100000 relational inserts per data point. For my independent variable I would change the capacity of the buffer pool (how many inserts before I rebuild my btree index).

3 Additional Experiments

3.1 Additional Experiment 1 - ID Store Versus Bitvector

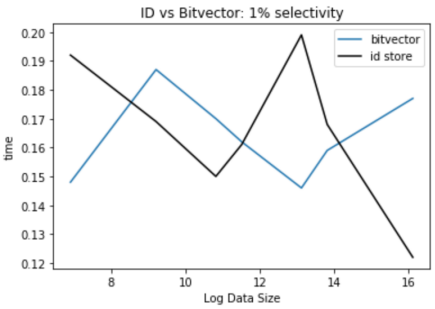
3.1.1 Motivation

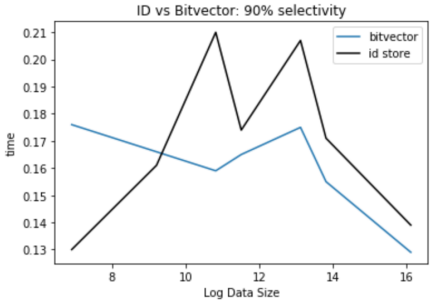
In this experiment I aim to figure out the run time differences between storing the results of select queries as an ID array versus storing it as a bitvector. Additionally, I wish to see how scaling my data size up affects the difference in run time for bitvector storage versus ID storage. This will then allow me to optimize whether I use a bitvector or an id array to fulfill select queries.

3.1.2 Setup

For this experiment I write a script that runs 100 select queries followed by 100 fetch queries. I run this experiment twice, once with 1% selectivity and then again with 90% selectivity. I scale my data up from 1000 items to 1000000 items and compared the times of the two payload storage methods.

3.1.3 Results





We can see a lot of volatility in the 1% selectivity graph. From the graph there are no conclusive patterns I can draw, and would need to run further experiments to get more substantail results. The 90% selectivity graph, on the other hand, shows that once the data size gets past 10000, the bitvector is faster than the id storage.

3.2 Additional Experiment 2 - M3 Tuning Size of Node

3.2.1 Motivation

Instead of tuning the tradition btree fanout, I instead tune the number of integers to be stored in each node. Since I implemented a cache conscious btree, each node only stores one pointer to the child nodes that are indexed by the parent. By experimenting with The size of the btree node I can further tune my btree index for main memory. One of the main goals in this milestone is to

minimize cache misses, so it is key to find which node size works best for my system

3.2.2 Setup

For this experiment I use test 27 (100 selects on an unclustered btree index) and record the number of cache misses as I vary my NodeSize parameter. I use the valgrind tool, cachegrind to get measurements of my L3 cache misses. I varied the pagesize between 500 and 12000 to find what was optimal.

3.2.3 Results

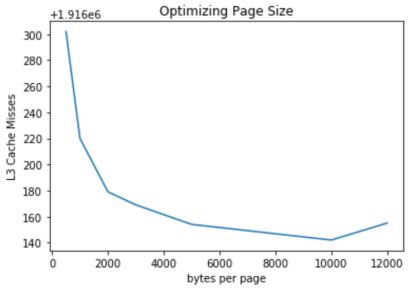


Figure 7: Caption

The number of cache misses seemed to be a decreasing function until the size becomes 12000 bytes, which might mean it is preferable to make my pagesize 10000. The only issue I have with this is that this will reduce the benefit of the btree in terms of I/O cost. As more items are included in a node, the more the select becomes a binary search instead. Therefore I might choose my size to be around 2000 since this is much smaller. Also it appears graphically to be where the most drastic drop in cache misses occurs.

3.3 Additional Experiment 3 - Clustered Sorted Index versus Btree Unclustered

3.3.1 Motivation

For this experiment I aim to compare the select performance of clustered sorted indexes versus a unclustered btree index. I expect the select time to be a lot less in the clustered index, but there is a lot more overhead in creating the copy of the data and propagating sort order throughout the columns. I plan to compare the two indexes across different data sizes. This can then be factored to guide preference in which index to optimize.

3.3.2 Setup

For each type of index I compare the time to run 100 selects. As my independent variable I vary the data size so I can see if propagating the order for clustered indexes will eventually outweigh the speed of the select.

3.3.3 Results

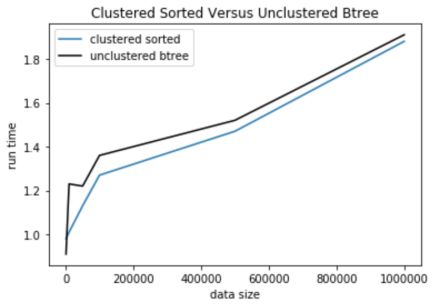


Figure 8: Caption

From the above figure we can see that the clustered index is always faster than the unclustered btree. However, as the data size grows to 1 million items, the gap between the two indexes decreases. This indicates that as the data size grows, the cost of sorting and propagating sort order to the rest of the columns becomes very costly.