Profiling the Performance of Querying Algorithms

CSE 5242 Project 2 Report

Jonathan Adams, Yukun Duan, Jake Smith

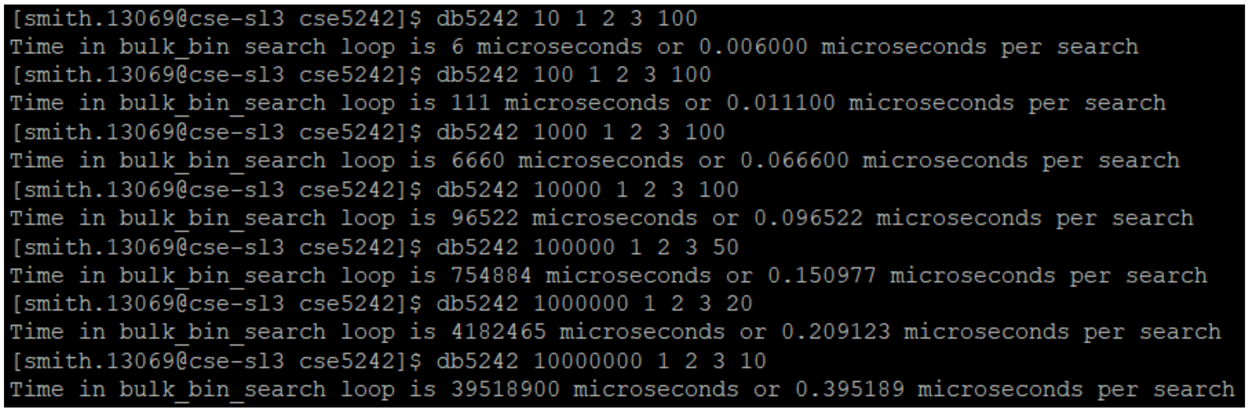
Introduction

When building a database system, the choice of how to query results affects how well the system will run. Each querying algorithm can have different types of benefits. Some reduce the time complexity needed to perform queries while others reduce the amount of memory or I/O operations needed during the querying process, and these are just a few of the possible obtainable benefits. When considering the workload that the system will be performing, it is possible to choose an algorithm that will maximize the benefits. Things like the average number of entries being queried over in the workload or whether or not indices exist for the most common queries are a couple factors that affect which algorithm performs the best. Beyond the workload, the hardware that is used is another factor that affects the efficiency of querying algorithms. By analyzing the options for how querying over entries in a database is performed, it's possible to get more insight into the types of workloads that each algorithm is more suited to take on. In this report, both searching algorithms that look for individual entries and band join algorithms that find a range of entries in joined tables will be examined.

Search Performance

low_bin_search

The first search algorithm examined is a simple binary search algorithm, provided by the instructor in the function low_bin_search. It utilizes if/else branching to determine how to narrow down the remaining values until the value that's closest to the query without going under is found. Since if/else branches are used, the query can skip over instructions that would try to update parameters that won't change during an iteration of the outer while loop. The performance times of the algorithm over multiple test queries are as follows:



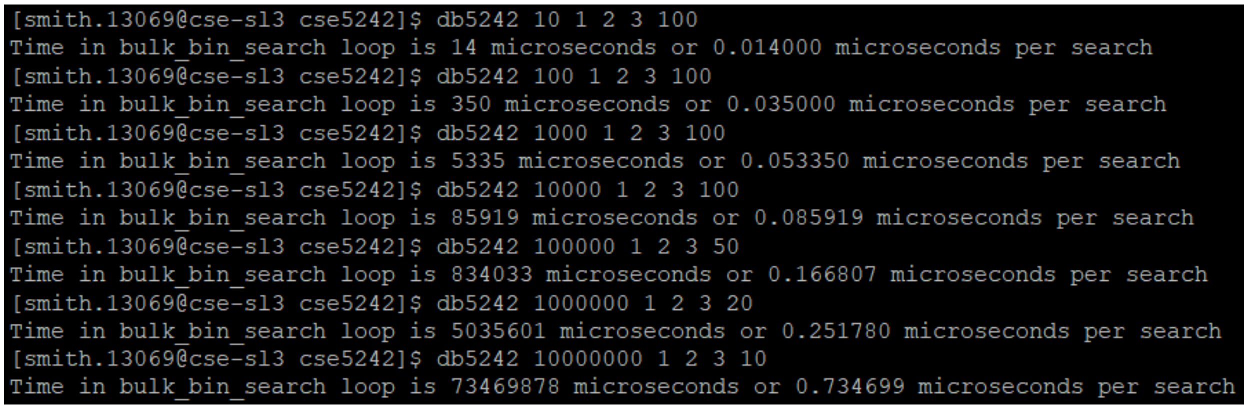
```
[smith.13069@cse-sl3 cse5242]$ db5242 10 1 2 3 100
Time in bulk_bin search loop is 6 microseconds or 0.006000 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 100 1 2 3 100
Time in bulk_bin_search loop is 111 microseconds or 0.011100 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 1000 1 2 3 100
Time in bulk_bin_search loop is 6660 microseconds or 0.066600 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 10000 1 2 3 100
Time in bulk_bin_search loop is 96522 microseconds or 0.096522 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 100000 1 2 3 50
Time in bulk_bin_search loop is 754884 microseconds or 0.150977 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 1000000 1 2 3 20
Time in bulk_bin_search loop is 4182465 microseconds or 0.209123 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 10000000 1 2 3 10
Time in bulk_bin_search loop is 39518900 microseconds or 0.395189 microseconds per search
```

For an algorithm that searches for one entry at a time, the time cost of each individual search is kept relatively low, with only 0.006 microseconds needed on average for searching a 10 entry array and 0.395 microseconds needed on average to search a 10^7 entry array. It also has the lowest average search time for arrays of 100 entries and smaller out of all the observed algorithms. The largest jump in time is going from a 100 entry array to 1000 entries, where the cost jumps by 6 times the amount. In all other cases, the time to search an array that is 10 times larger than the previous takes no more than double the average search time.

The increase in search time going from 100 to 1000 entries is the largest jump percentage wise across all algorithms, although it is not the largest jump in total cost. It's likely that the algorithm sees an increase in overhead for medium to large size datasets due to the use of if/else branching. Despite this, the average search times for medium and large size datasets is still kept reasonably low, with the performance being the best out of all the observed search algorithms that only do one search at a time. Given this information, the low_bin_search algorithm performs the best on small datasets while suffering a drop in performance on larger datasets, but despite the lower performance it is still able to search at a decent time cost.

low_bin_nb_arithmetic

In the function low_bin_nb_arithmetic, if/else branching is replaced by the use of arithmetic operators which update each of the variables in a binary search. The updated variables narrow down the number of values that still need to be searched. However, not every variable changes in each iteration of the outer while loop, but the algorithm will still perform the mathematical operation attempting to update each variable, resulting in a slight overhead. The test queries over the algorithm resulted in the following performance times:



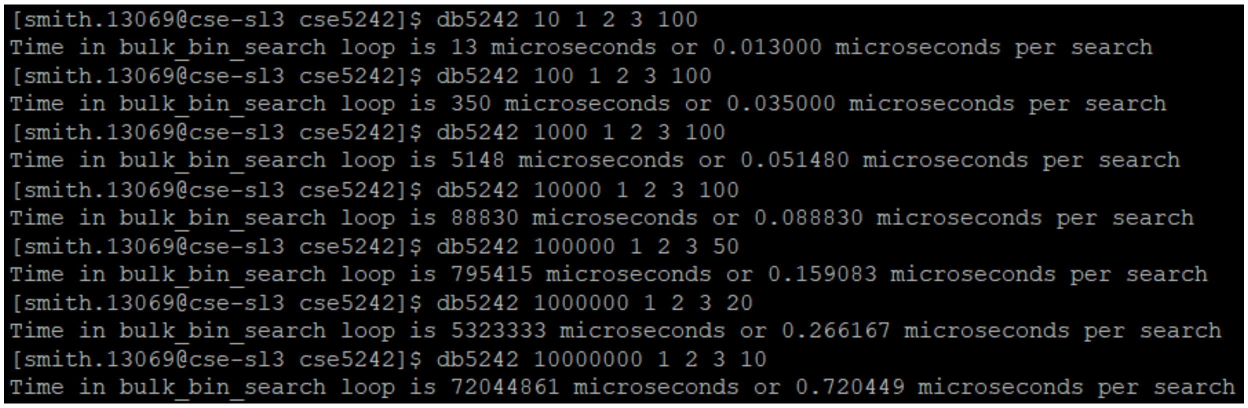
```
[smith.13069@cse-sl3 cse5242]$ db5242 10 1 2 3 100
Time in bulk_bin_search loop is 14 microseconds or 0.014000 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 100 1 2 3 100
Time in bulk_bin_search loop is 350 microseconds or 0.035000 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 1000 1 2 3 100
Time in bulk_bin_search loop is 5335 microseconds or 0.053350 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 10000 1 2 3 100
Time in bulk_bin_search loop is 85919 microseconds or 0.085919 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 100000 1 2 3 50
Time in bulk_bin_search loop is 834033 microseconds or 0.166807 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 1000000 1 2 3 20
Time in bulk_bin_search loop is 5035601 microseconds or 0.251780 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 10000000 1 2 3 10
Time in bulk_bin_search loop is 73469878 microseconds or 0.734699 microseconds per search
```

For smaller size arrays containing 100 entries or less, the time cost of the queries on average was relatively high, with 0.014 microseconds needed on average to search a 10 entry array and 0.035 microseconds for 100 entries. When compared to low_bin_search, the time cost of queries over smaller datasets was over double. This also held true for large data sets, where the algorithm needed 0.735 microseconds on average to search through an array of 10^7 entries. However, for arrays of size 1000 to 10^6 entries, the performance of the algorithm was comparable to that of low_bin_search, sometimes even performing better on average.

The time increase from test to test followed a similar pattern to that of the average time cost, where the highest increases in average search time were between the small datasets and when going from 10^6 entries to 10^7, the latter seeing an increase of almost 3 times the cost. Also, the increase in cost going from 10^6 to 10^7 entries was the largest total time increase out of every trial across all algorithms. Overall, the low_bin_nb_arithmetic algorithm performed well on medium sized datasets, but due to a noticeable drop in efficiency when the dataset was too large or too small it ended up performing the worst on non-medium sized datasets.

low_bin_nb_mask

The algorithm in low_bin_nb_mask further expands upon binary search by replacing the use of most arithmetic operators with bitwise operands. This keeps the functionality of the replaced arithmetic operators while reducing the complexity of the operations done since bitwise operands only need to compare bits to calculate the results. The bitwise operands update the variables that narrow down the search, and each variable still needs to be updated every iteration of the outer while loop since if/else branching is not used. The performance results of the test queries are as follows:



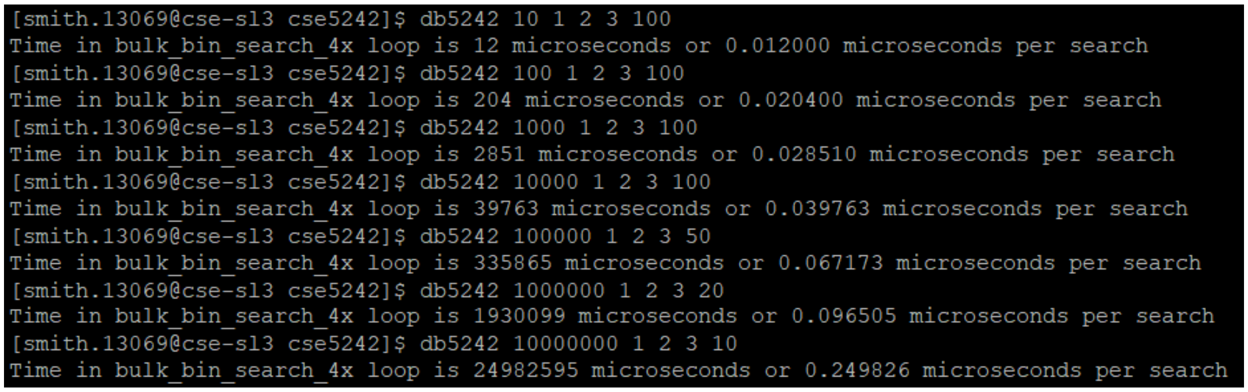
```
[smith.13069@cse-sl3 cse5242]$ db5242 10 1 2 3 100
Time in bulk_bin_search loop is 13 microseconds or 0.013000 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 100 1 2 3 100
Time in bulk_bin_search loop is 350 microseconds or 0.035000 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 1000 1 2 3 100
Time in bulk_bin_search loop is 5148 microseconds or 0.051480 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 10000 1 2 3 100
Time in bulk_bin_search loop is 88830 microseconds or 0.088830 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 100000 1 2 3 50
Time in bulk_bin_search loop is 795415 microseconds or 0.159083 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 1000000 1 2 3 20
Time in bulk_bin_search loop is 5323333 microseconds or 0.266167 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 10000000 1 2 3 10
Time in bulk_bin_search loop is 72044861 microseconds or 0.720449 microseconds per search
```

The time cost of using the algorithm in low_bin_nb_mask was high for datasets with 100 or less entries and for the 10^7 entry test. The average time to search a 10 entry array was 0.013 microseconds, and 0.720 microseconds were needed on average for the 10^7 entry array. For the other tests, the time complexity was respectable, with 10^3 entries needing 0.051 microseconds on average for a search and 10^6 entries needing 0.266 microseconds. The jumps in time were also higher when the dataset was larger or smaller, with the jump from 10 to 100 entries seeing over double the time cost and the jump from 10^6 to 10^7 entries seeing a similar level of increase. The increase in time across the rest of the trials was less than double when increasing the number of entries by a factor of 10.

The average search times for this algorithm are comparable to the search times for low_bin_nb_arithmetic. This could be a result of the compiler automatically optimizing the code for both functions, causing the executable code to run in a similar way when using each of low_bin_nb_arithmetic and low_bin_nb_mask. Likewise, this algorithm has similar applicability to low_bin_nb_arithmetic where it works best on moderately sized datasets and sees a decrease in its efficiency when the datasets go above or below certain thresholds.

low_bin_nb_4x

The algorithm in low_bin_nb_4x uses the same concepts of bit masking that are used in low_bin_nb_mask, but sticks them in a for loop that does 4 searches at the same time. Each iteration of the outer while loop will narrow down the remaining values being searched for each of the 4 searches being done. This is done by using bitwise operands to update the variables that keep track of the search parameters. When the algorithm was applied to multiple tests on different queries, these were the resulting time performances:



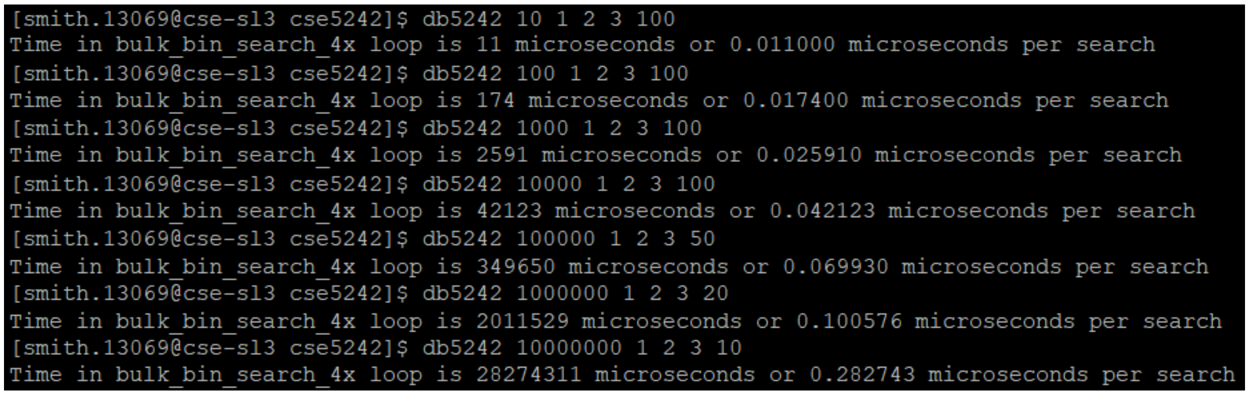
```
[smith.13069@cse-sl3 cse5242]$ db5242 10 1 2 3 100
Time in bulk_bin_search_4x loop is 12 microseconds or 0.012000 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 100 1 2 3 100
Time in bulk_bin_search_4x loop is 204 microseconds or 0.020400 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 1000 1 2 3 100
Time in bulk_bin_search_4x loop is 2851 microseconds or 0.028510 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 10000 1 2 3 100
Time in bulk_bin_search_4x loop is 39763 microseconds or 0.039763 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 100000 1 2 3 50
Time in bulk_bin_search_4x loop is 335865 microseconds or 0.067173 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 1000000 1 2 3 20
Time in bulk_bin_search_4x loop is 1930099 microseconds or 0.096505 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 10000000 1 2 3 10
Time in bulk_bin_search_4x loop is 24982595 microseconds or 0.249826 microseconds per search
```

The cost of the algorithm ranged from an average of 0.012 microseconds per search for a 10 entry array to 0.250 microseconds per search for a 10^7 entry array. When the dataset was small, the time complexity was relatively high, albeit being ever so slightly lower than low_bin_nb_arithmetic and mask. As the number of entries in the array increased, the time cost increased at a noticeably shallower rate. The time increase per search was no more than double the average time it took to search an array that was 10 times smaller, apart from the jump from 10^6 entries to 10^7. In this case, the time cost was over twice as large, and this jump is likely due to the size of the caches of the system (which will be examined later).

Compared to all the search algorithms that only do 1 search at a time, low_bin_nb_4x saw a lower average time cost per search for every trial apart from the 10 and 100 entry array trials of low_bin_search. It also had a noticeably lower rate of increase between trials apart from the jump from 10^6 to 10^7 entries. Another aspect of this algorithm is that the number of cores the system has plays an effect on the performance since it can split up the work of the inner for loop. Overall, this algorithm seems to work best on large datasets and when run on multi-core hardware.

low_bin_nb_simd

In the function low_bin_nb_simd, the algorithm still performs 4 searches simultaneously, but uses AVX2 to control the multiple searches as opposed to a for loop like in low_bin_nb_4x. Since AVX2 inherently utilizes SIMD, or single instruction multiple data, it is able to utilize parallelism to improve its performance even without the need of multiple cores. The algorithm's methods utilize bitwise operations to manipulate variables that narrow the search down within the iterations of an outer while loop. This is similar to how low_bin_nb_mask performs its search, with the main difference being that the variables are contained in different data types and are manipulated through function calls due to the use of AVX2. The time performances of the test queries on low_bin_nb_simd are given:



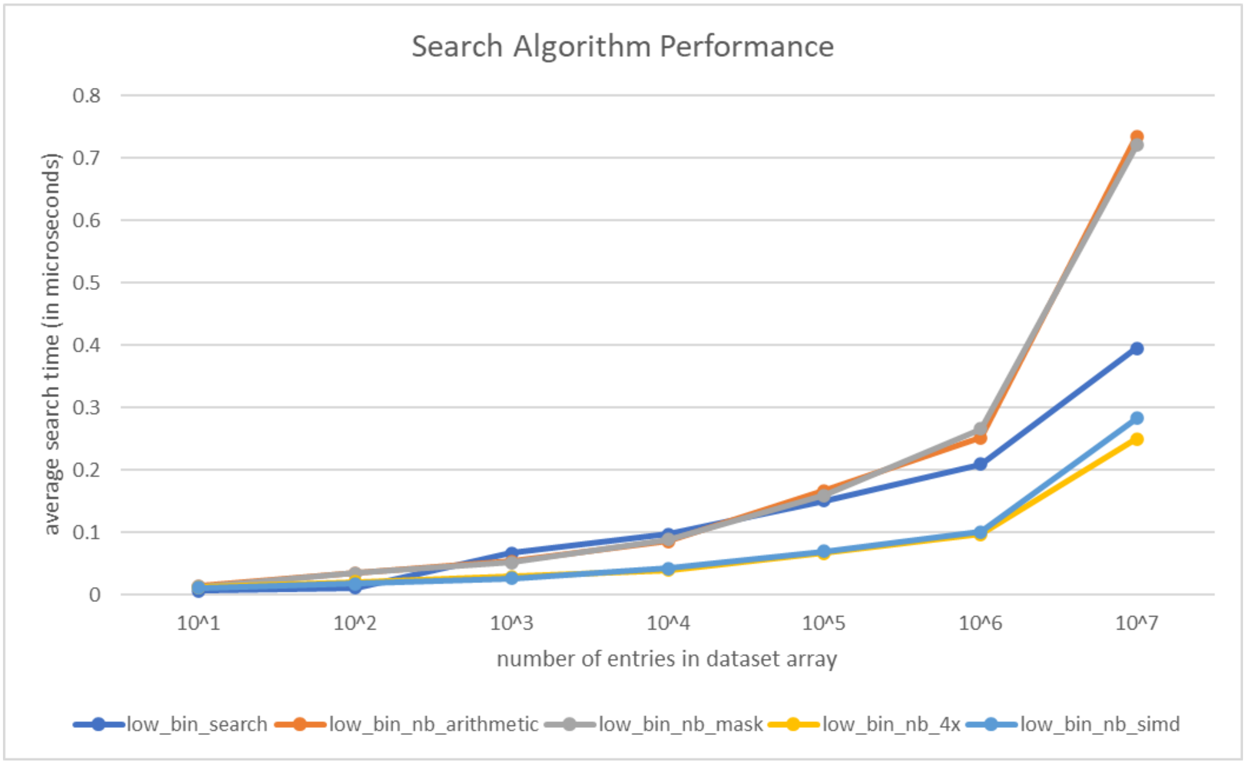
```
[smith.13069@cse-sl3 cse5242]$ db5242 10 1 2 3 100
Time in bulk_bin_search_4x loop is 11 microseconds or 0.011000 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 100 1 2 3 100
Time in bulk_bin_search_4x loop is 174 microseconds or 0.017400 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 1000 1 2 3 100
Time in bulk_bin_search_4x loop is 2591 microseconds or 0.025910 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 10000 1 2 3 100
Time in bulk_bin_search_4x loop is 42123 microseconds or 0.042123 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 100000 1 2 3 50
Time in bulk_bin_search_4x loop is 349650 microseconds or 0.069930 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 1000000 1 2 3 20
Time in bulk_bin_search_4x loop is 2011529 microseconds or 0.100576 microseconds per search
[smith.13069@cse-sl3 cse5242]$ db5242 10000000 1 2 3 10
Time in bulk_bin_search_4x loop is 28274311 microseconds or 0.282743 microseconds per search
```

In the smaller sized arrays, the cost of a search on average was respectably low, with a 10 entry array needing 0.011 microseconds on average and a 100 element array needing 0.017 microseconds. For larger sized arrays, the time cost was also low, with an array of 10^7 entries only needing 0.283 microseconds per search on average. The increase in time cost when the number of elements in the array increases by a multiple of 10 is quite shallow, with only the jump from 10^6 to 10^7 entries being over double the time cost.

Compared to low_bin_nb_4x, the average search cost of each trial is pretty similar. The costs to search the smaller arrays was a bit smaller, and the search cost for larger arrays was a bit larger, but overall the times were comparable with no corresponding trials being over 0.05 microseconds apart. Compared to the single search algorithms, the time cost is significantly better for larger arrays. It's also mostly better for smaller arrays, with low_bin_search being the only algorithm to have a lower average cost per search on arrays of size 100 and lower. Given the collected data, the use of AVX2 in low_bin_nb_simd is ideal for large size databases and isn't hindered by a fewer number of cores.

Search algorithm comparison graph

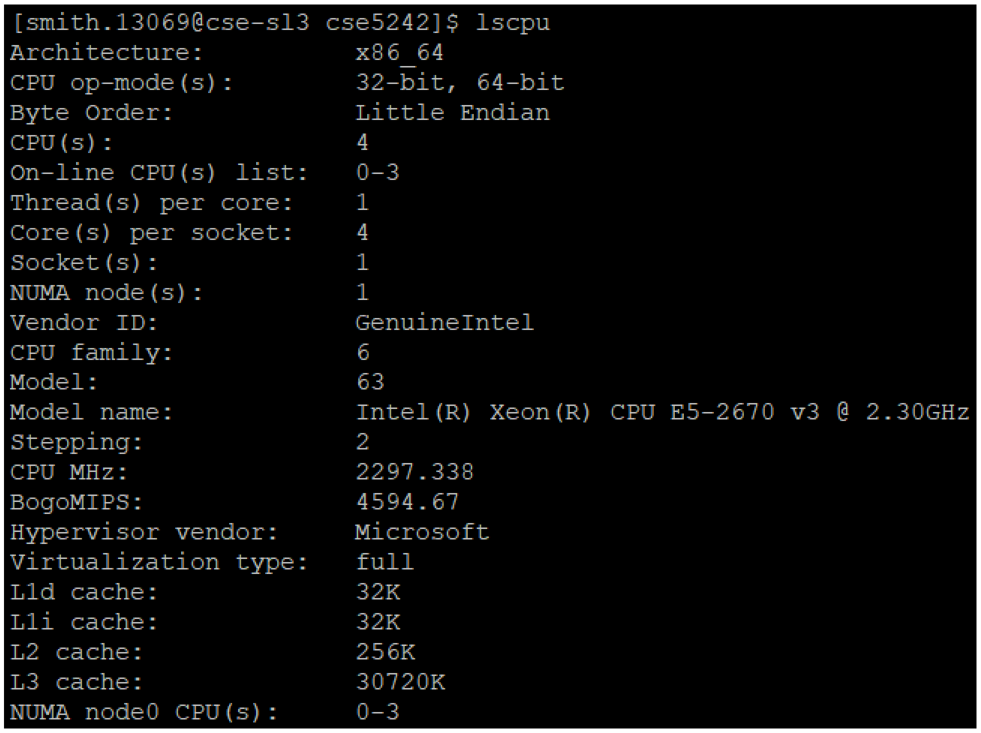
Given is a graph visualizing the performance times of all the querying algorithm functions that were observed:



The graph gives a good idea of the rate at which each of the algorithms increase in time cost. It also shows the gap in time cost between the algorithms at each level. Looking at the graph, it's logical to conclude that both low_bin_nb_4x and low_bin_nb_simd have the slowest growth rate for cost, resulting in them both being noticeably lower in cost than the other algorithms when the datasets had 10^5 or more entries. While low_bin_search had the lowest cost for datasets of 100 entries or less, the time saved is not nearly as much as the time that the 4x and SIMD algorithms save for larger datasets, although it's still worth mentioning. For a database that has an unknown number of entries that wants to do search queries in a reliable timeframe no matter how small or large the dataset being searched over is, the use of parallelism in low_bin_nb_4x and low_bin_nb_simd have observable benefits which makes both of them a solid choice.

Effects of the CPU specs

Information about the machine used to perform all the search algorithm test queries is provided below. Included is information about the CPU and caches that the machine utilizes.



```
[smith.13069@cse-sl3 cse5242]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz
Stepping:              2
CPU MHz:               2297.338
BogoMIPS:              4594.67
Hypervisor vendor:     Microsoft
Virtualization type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              30720K
NUMA node0 CPU(s):     0-3
```

One of the biggest factors in the performance of the querying algorithms is the size of the caches that are used. If the algorithm and data can fit in a cache, the cache can perform the searches quicker than memory, with the L1 caches being the fastest and L3 cache being the slowest of the caches. The system used for the test queries has caches ranging in size from 32 KB for the L1 caches to 30720 KB for the L3 cache. During the test queries, most algorithms saw the largest jump in time cost when the array went from 10^6 to 10^7 entries. Since each entry in the array was 64 bits, this jump occurred when the size of the array went out of bounds of the caches. In other words, the 10^7 entry array no longer fits in the L3 cache. The large increase in time cost is most likely due to the algorithms needing to perform the searches outside of cache. Therefore, having a cache large enough to fit the data used in a database's common queries is optimal for performance, but comes at a steeper monetary cost as the cost per byte of cache is significantly more expensive than the cost per byte of memory.

Another factor that affects query performance is the number of CPUs used in the querying process. The system used has 4 usable CPUs, and this factor has a significant impact on low_bin_nb_4x, as multiple cores being used in unison allow segments of code that don't share variables to be processed simultaneously instead of needing to be processed one after the other. Since each iteration of the inner for loop of the 4x algorithm manipulates different variables, the 4 inner loop iterations that happen per iteration of the outer while loop can be processed at the same time. This allows low_bin_nb_4x to perform with a similar time cost to low_bin_nb_simd, but at the cost of needing to utilize more cores when searching the datasets. This factor would also cause the performance of the 4x algorithm to decrease in the cases where the system has less usable CPUs.

There are other hardware factors that can affect query performance, like the number of threads per core and the CPU MHz. However, these factors were utilized in a constant fashion across all observed algorithms and test queries. As a result, they didn't have an observable effect which would cause any one algorithm to perform better than another.

Effects of the size of the dataset

One fact that remained the same for every observed algorithm is that as the number of entries in the dataset array (call this N) increased, the average time cost of a search also increased. This is due to the nature of binary search, which every observed algorithm is based around. Binary search continues to half the number of entries being searched through until only 1 entry remains. For these algorithms, that entry is the one closest to the query without going under. Given how binary search works, the time complexity of a simple binary search algorithm is in $O(\log_2 N)$.

While this is not the exact time complexity of every observed search algorithm, the time complexity of binary search shows that the main factor in the algorithm's runtime is the number of entries in the dataset. Therefore, as N increases, the time cost of a search should also increase, and that is exactly what happened in the test queries.

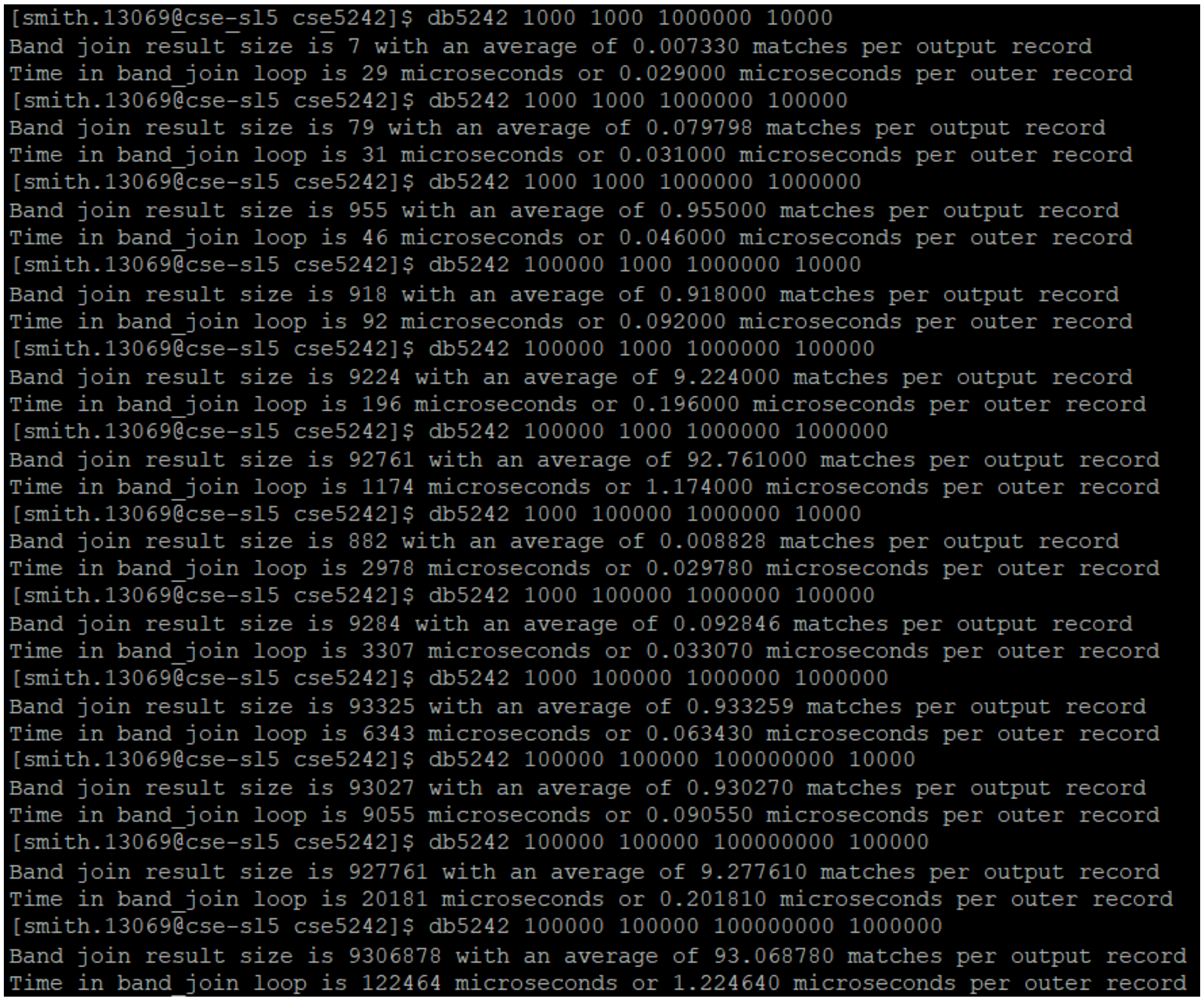
Band Join Performance

Choice of table sizes

The values for the inner sorted table N and outer table X changed between multiple trials, with one trial having both at 1000 entries, another with both at 100000, and then 2 more where one was at 1000 and the other was at 100000. These values were chosen because they can represent the difference between small and large tables, and their use can be used to examine the difference in time cost and result size when the table sizes change between small and large.

band_join

The results of the test queries on band_join are as follows:



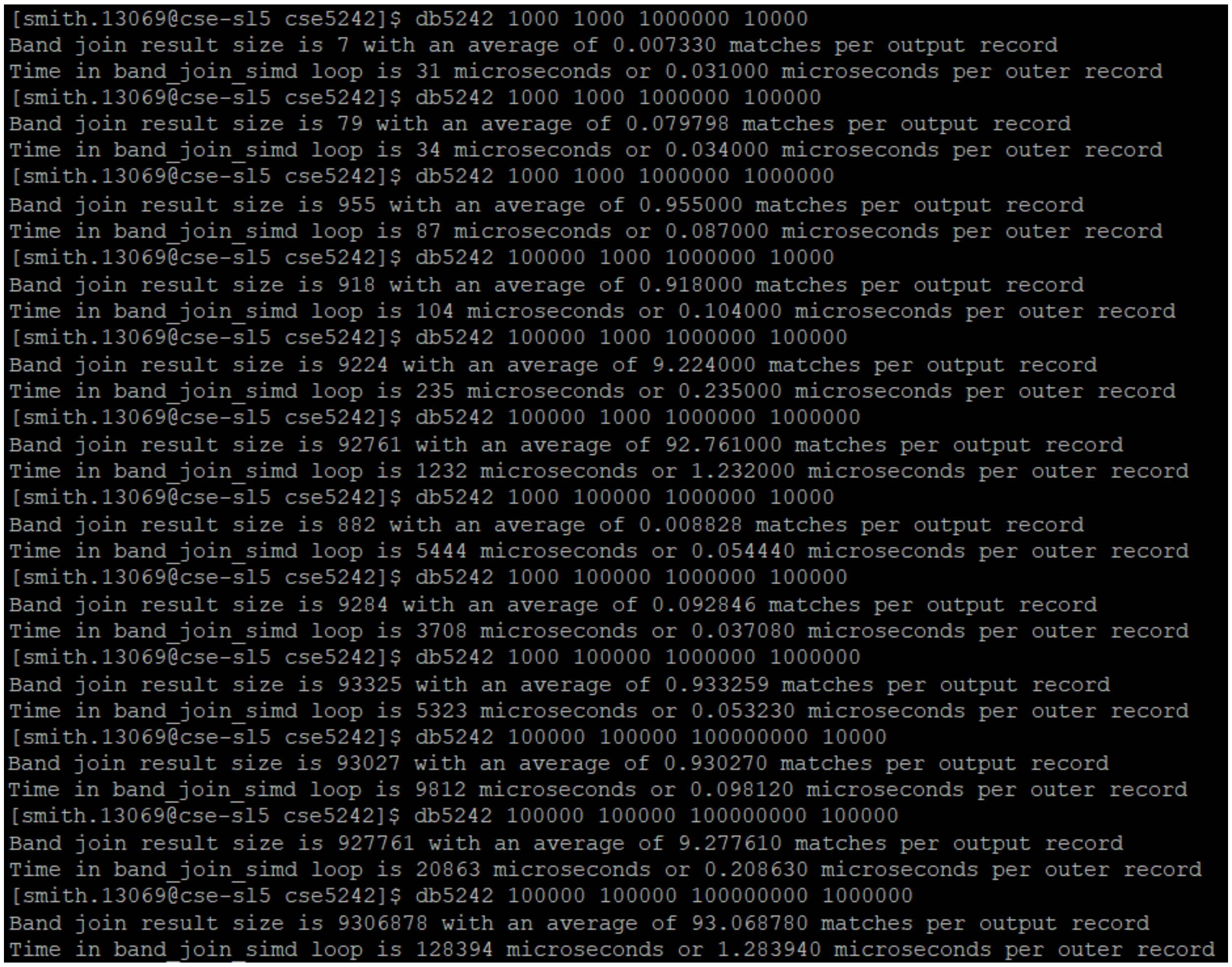
```
[smith.13069@cse-sl5 cse5242]$ db5242 1000 1000 1000000 10000
Band join result size is 7 with an average of 0.007330 matches per output record
Time in band_join loop is 29 microseconds or 0.029000 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 1000 1000 1000000 100000
Band join result size is 79 with an average of 0.079798 matches per output record
Time in band_join loop is 31 microseconds or 0.031000 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 1000 1000 1000000 1000000
Band join result size is 955 with an average of 0.955000 matches per output record
Time in band_join loop is 46 microseconds or 0.046000 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 100000 1000 1000000 10000
Band join result size is 918 with an average of 0.918000 matches per output record
Time in band_join loop is 92 microseconds or 0.092000 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 100000 1000 1000000 100000
Band join result size is 9224 with an average of 9.224000 matches per output record
Time in band_join loop is 196 microseconds or 0.196000 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 100000 1000 1000000 1000000
Band join result size is 92761 with an average of 92.761000 matches per output record
Time in band_join loop is 1174 microseconds or 1.174000 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 1000 100000 1000000 10000
Band join result size is 882 with an average of 0.008828 matches per output record
Time in band_join loop is 2978 microseconds or 0.029780 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 1000 100000 1000000 100000
Band join result size is 9284 with an average of 0.092846 matches per output record
Time in band_join loop is 3307 microseconds or 0.033070 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 1000 100000 1000000 1000000
Band join result size is 93325 with an average of 0.933259 matches per output record
Time in band_join loop is 6343 microseconds or 0.063430 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 100000 100000 100000000 10000
Band join result size is 93027 with an average of 0.930270 matches per output record
Time in band_join loop is 9055 microseconds or 0.090550 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 100000 100000 100000000 100000
Band join result size is 927761 with an average of 9.277610 matches per output record
Time in band_join loop is 20181 microseconds or 0.201810 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 100000 100000 100000000 1000000
Band join result size is 9306878 with an average of 93.068780 matches per output record
Time in band_join loop is 122464 microseconds or 1.224640 microseconds per outer record
```

As the values used for the size of the inner table N, the outer table X, and the bound Z increased, the number of results returned also increased in every scenario. When the bound was increased by a multiple of 10, the average number of matches per outer record also increased by approximately a multiple of 10. Increasing the value of N had a similar effect where a value that was 100 times larger had an average number of matches that was also 100 times larger. On the other hand, when the value of X was increased, the number of matches was relatively the same.

The average time spent in the band join loop per outer record followed a similar pattern to that of the average number of matches per outer record. It increased when either N or Z increased, but the increase in average time cost was much more significant when N increased compared to when Z increased. When X increased, the average time cost per outer record was similar, which makes sense because the size of X should not have any noticeable effect on the average time cost per outer record.

band_join_simd

Given below are the results of the test queries on band_join_simd:



```
[smith.13069@cse-sl5 cse5242]$ db5242 1000 1000 1000000 10000
Band join result size is 7 with an average of 0.007330 matches per output record
Time in band_join_simd loop is 31 microseconds or 0.031000 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 1000 1000 1000000 100000
Band join result size is 79 with an average of 0.079798 matches per output record
Time in band_join_simd loop is 34 microseconds or 0.034000 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 1000 1000 1000000 1000000
Band join result size is 955 with an average of 0.955000 matches per output record
Time in band_join_simd loop is 87 microseconds or 0.087000 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 100000 1000 1000000 10000
Band join result size is 918 with an average of 0.918000 matches per output record
Time in band_join_simd loop is 104 microseconds or 0.104000 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 100000 1000 1000000 100000
Band join result size is 9224 with an average of 9.224000 matches per output record
Time in band_join_simd loop is 235 microseconds or 0.235000 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 100000 1000 1000000 1000000
Band join result size is 92761 with an average of 92.761000 matches per output record
Time in band_join_simd loop is 1232 microseconds or 1.232000 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 1000 100000 1000000 10000
Band join result size is 882 with an average of 0.008828 matches per output record
Time in band_join_simd loop is 5444 microseconds or 0.054440 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 1000 100000 1000000 100000
Band join result size is 9284 with an average of 0.092846 matches per output record
Time in band_join_simd loop is 3708 microseconds or 0.037080 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 1000 100000 1000000 1000000
Band join result size is 93325 with an average of 0.933259 matches per output record
Time in band_join_simd loop is 5323 microseconds or 0.053230 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 100000 100000 100000000 10000
Band join result size is 93027 with an average of 0.930270 matches per output record
Time in band_join_simd loop is 9812 microseconds or 0.098120 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 100000 100000 100000000 100000
Band join result size is 927761 with an average of 9.277610 matches per output record
Time in band_join_simd loop is 20863 microseconds or 0.208630 microseconds per outer record
[smith.13069@cse-sl5 cse5242]$ db5242 100000 100000 100000000 1000000
Band join result size is 9306878 with an average of 93.068780 matches per output record
Time in band_join_simd loop is 128394 microseconds or 1.283940 microseconds per outer record
```

The data shows the total results size increasing when N, X, and Z increase, but the average number of matches per outer record only has a noticeable increase when N or Z increase. Since the functionality of band_join_simd is the same as band_join, it makes sense that both the result size and average matches change in the same ways as they do in band_join when N, X, and Z change. Similarly, the average time cost per outer record saw a noticeable increase for higher values of N and Z but not X, which matches how average time cost changed in band_join.

Band join algorithm comparison

The `band_join` function is more like the traditional binary search method and employs `low_bin_nb_4x` for batch processing of the external table. The function first uses it to determine the starting position of each search range and then performs a linear scan of the internal table to find all elements falling within that range. When the size of the external table is not a multiple of four, the function will use `low_bin_nb_mask` to handle the remaining records individually.

Theoretically, `band_join_simd ` introduces SIMD and utilizes the AVX2 instruction set which will speed up the parallel data processing. This approach allows the simultaneous calculation of search range boundaries for four elements in the external table, significantly improving the efficiency of range calculations. `band_join_simd ` has the advantage of a faster liner scan process for each range compared to the band_join function.

However, through multiple tests in the university remote Linux virtual machine, we found opposite outcomes, ` band_join_simd ` taking the same or even longer time to search than `band_join `, according to the testing environment, possible reasons are: non-parallelized linear scan, system and hardware constraints, compiler optimizations, etc.

Effects of the size of the bound

When the size of the bound Z was increased from 10^4 to 10^5 and 10^6, the time per outer record would increase alongside it. This can be explained by examining the functionality of band join. The band join algorithm needs to find every pair of values from the inner table N and the outer table X that match the query. It does this by looking through N separately for each value in X to find all the query matching (N,X) pairs. Since N is sorted, the algorithm can stop checking N for a given X value once it finds the first value in N that is too large for a query match. Therefore, the average time spent in the band join loop for one X value is determined by the average index of the first value in N that is too large for the query. This in turn is determined by the average index of the first query-matching N value and the average number of query-matching N values for one value in X.

Both of these factors are affected by a change in the value of Z. The average number of query-matching N values is proportional to the bound size and the number of possible (N,X) pairs, or O(Z*(N*X)). Therefore, as Z increases, the number of query-matching N values also increases. Further, the average index of the first query-matching N value will usually be around half the query-matching N values before the midpoint of N, or O(N/2 - 0.5(Z*(N*X))). Combining these results in a time cost within O(N/2 + 0.5(Z*(N*X))), and this value increases as Z increases. This means that a larger bound should result in a lengthier time per outer record, and the results of the test queries on the band join functions match this analysis.

There was one test query (in the tests on band_join_simd with N=1000 and X=100000) where the average time per outer record actually went down when the bound increased. However, this is a lone outlier that was likely due to some extra overhead in the test query with the smaller bound. This overhead could have been a result of the average number of query-matching N values being extra high, the average index of the first query-matching N value being extra late, or both.

Conclusion

Looking at the results across every examined algorithm, the times when forms of parallelism were used had the cheapest time cost in the majority of trials. Overall, the use of technology like AVX2 to make use of SIMD when processing queries can lower the time complexity it takes for their processing. It can even allow for multiple similar queries to be executed in parallel which can accommodate multiple clients at the same time. The use of AVX2 can also be combined with splitting workloads amongst multiple CPUs to allow for higher levels of parallelism. In conclusion, when building a database system that prioritizes the speed of queries and processing multiple queries in unison, the use of AVX2 technology is a very strong choice with measurable benefits in time cost and how fast time cost grows.

Group Member Contributions

Jake wrote the code for the low_bin_nb_arithmetic and low_bin_nb_mask functions. Jake also reviewed and debugged the code for the low_bin_nb_4x and low_bin_nb_simd functions. Jake profiled the performance of all 5 search algorithms for the report, and wrote the entirety of the Search Performance section in the report. Jake also wrote the Introduction and Conclusion sections of the report. Jake collected the data used to profile both band join functions, wrote the analyses for the collected data, and wrote the segment explaining the choices in values for the band join table sizes. Jake also wrote the section of the report explaining how the size of the bound in the band join functions affects the average time per outer record. Lastly, Jake typed up the README provided which explains how to compile the code. Jake did not use AI when writing his code or the report, but did use the suggestions provided by Google docs when editing his writing in the report.

Jonathan revised some part of the code to make sure the band join work properly.

Yukun Duan finished the low_bin_nb_4x, low_bin_nb_simd, band_join, band_join_simd functions and analyzed the differences between band_join and band_join_simd functions and possible reasons why the outcome was not as expected in the report. Yukun did not use AI when writing his code or the report but did use the suggestions provided by Google Docs when editing his writing in the report.