Review and Comparison of Binary Search Algorithms

Benji Ofori, Gabe Richner, Jonathon Ross

Ohio State University

Dr. John Paparrizos

CSE 5242: Advanced Database Management Systems

December 1, 2024

Abstract

This paper takes an experimental look into increasing the performance of binary search and band join. We looked at six versions and binary search and two versions of Band Join. Each function takes its own attempt to increase performance by limiting slow instructions and taking advantage of CPU parallelism. We also briefly discuss how the CPU cache affects our test results. We found that for most datasets, you will use low_bin_nb_4x binary search to do four searches at a time for the best performance, as well as use regular band join when doing a band join. While our results were dependent on the hardware of the computer the tests were carried out on, it demonstrated that through the CPU's parallelism, you can gain a significant increase in performance when carrying out programs.

Table Of Contents

I. Introduction

This paper explores innovative ways for optimizing binary search algorithms that use CPU parallelism, caching, and sophisticated hardware capabilities to improve computational efficiency. Binary search and band join algorithms are critical for tasks such as large-scale data processing, real-time applications, and range queries in databases and analytics. Efficient binary search algorithms make it easy to rapidly locate elements within sorted arrays, whereas band join methods facilitate fast and precise range-based operations. To ensure a thorough comprehension, this paper begins with an introduction of hardware and fundamental algorithm ideas before delving into advanced optimization strategies. The paper begins with a basic binary search implementation and progresses to more advanced implementations. The first optimization, known as low bin search, minimizes inner-loop comparisons to optimize the search process for elements meeting specific conditions. Adding to this, the low bin NB arithmetic technique replaces conditional statements with arithmetic operations to improve efficiency. Additional developments include the low bin NB mask approach, which incorporates bitwise operations to reduce computational complexity, and the low bin NB 4x methodology, which allows for simultaneous execution of four searches at the same time for significant speed gains. The paper then talks about the usage of AVX2 SIMD instructions in the low bin NB SIMD implementation which shows how modern hardware capabilities can be used to provide parallelized and efficient searches. Beyond binary search, the paper introduces optimized band join, which uses branch elimination and AVX2 extensions to speed up range query operations. These functions are then tested with varying datasets and conditions to better understand each implementation's runtime, leading to a comprehensive conclusion on the findings and implications of each algorithm’s implementation.

II. Hardware Overview

For our testing, the hardware we execute our code on is an important factor to consider. To ensure our testing is uniform we do all of our testing on OSU Student Linux Servers. This allows us to run our tests on the same set of hardware every time regardless of who is executing the program. Here is a list of specs allocated for a user on the OSU Servers:

Components	**Specification**
Architecture	x86_64
CPU	AMD EPYC 7713 64-Core Processor
CPU MHz	1996.287
L1d Cache	32 KB
L1i Cache	32 KB
L2 Cache	512 KB (0.5 MB)
L3 Cache	32,768 KB (32 MB)

This list shows the amount of cache space we as a user have access to when running our program. The amount shown is not the full amount the processor can produce, which means we are limited to a certain chunk of memory space. For smaller data sets, our program should perform at its peak efficiency

and be fine. However, for large to very large data sets, being limited to this amount of cache space may impact performance.

III. Function Overview

This section of the paper provides an overview of the different implementations of binary search and band join. Each implementation will highlight the differences and importance of those changes to promote better optimization and efficiency for the algorithms.

A. Simple Binary Search

This is the baseline algorithm for the project. Binary Search is an algorithm that finds the target value within a sorted array. The function takes the data array, the size of the array, and the target value and returns the index value if it's found in the array. The function initializes two “pointers” variables at each end of the array called left and right. The next variable is a midpoint variable. The function contains a loop that won’t break until the left and right endpoints are next to each other or the target value is found. Every loop calculates the midpoint of the right and left and compares it to the target. If the midpoint value is the target we return the value and are done. Otherwise, if the midpoint is less than the target we set left to midpoint + 1, and conversely, if it's greater we set right to the midpoint - 1. This reduces the search by half every iteration. This allows the algorithm to have a run time of O(log(n)) making binary search an effective method of finding a particular item in a sorted dataset.

B. Low Bin Search

Low bin search is a function that is similar to simple binary search. The purpose of binary search is to find the exact match to the target value within a dataset whereas low_bin_search finds

the first element in the sorted data that is greater than or equal to the target value. Like Binary Search, Low bin has two pointer arrays named left and right that start at each end of the array. The function enters a loop that continues until the left variable is less than the right variable. In the loop, we calculate the midpoint value in the array and compare it to the target, and adjust left and right to narrow the search just like simple binary search. The key difference is we loop until the left is no longer less than the right. Once completed the right point will hold the value of the first element greater than or equal to the target value and return that index. Returning the index of the value that is greater and equal allows us to find the starting index point for ranged queries. It also is fast as the run time is O(log(n)) because we narrow the searches in half every iteration. The function performs one comparison in the inner loop during each iteration, optimizing the performance of large data sets. Since this is a binary search, the function takes O(log N) as its time complexity.

C. Low Bin NB Arithmetic

The goal of this function is to search an array to find a target. This function takes in three arguments. A pointer to the data set, the size of the data set, and the target. The function defines three main variables. The left variable sets it to 0, the right variable sets it equal to the size of the array, and lastly a mid variable. The function continues with an iteration on the condition that while the left variable is less than the right variable, it keeps iterating over the array provided. First, the value for mid is calculated by the mid variable is calculated by adding the left variable to the result of subtracting the left variable from the right and performing one right shift on the result. Mid represents the index of the value in the middle of the dataset. The function does no comparisons in the inner loop by converting control dependencies into data dependencies using arithmetic operations. By doing this, we reduce branch mispredictions and enhance parallelism. What makes this function unique from Low Bin Search is that it eliminates conditional branching in the inner loop. In doing this, it avoids branch misprediction with branches especially when the

input data patterns are unpredictable. This optimization reduces such penalties, hence improving performance.

D. Low Bin NB Mask

The goal of this function is to identify the position of the first element in the array that is greater than or equal to the target value. If the target is larger than all elements in the array, the function returns the size of the array. This implementation optimizes binary search by eliminating traditional conditional statements like if within the inner loop. Instead, it leverages efficient bitwise operations, such as AND and OR, to update the left and right variables based on the search condition. What makes this function more efficient in comparison to Low Bin NB Arithmetic is that its approach is branch-free and enhances performance by reducing control dependencies and improving predictability on modern CPUs.

E. Low Bin NB 4x

This function uses a parallelized binary search to find the first index in a sorted array where the values are greater than or equal to a target for four different target values at the same time. If a target is greater than the array's total number of elements, the function returns the array's size as the result. This function avoids the use of inner comparison replacing it with the use of bit masking operations. It is designed to leverage cache efficiency and out-of-order processor execution by interleaving four binary searches. What makes this function more efficient in comparison to Low Bin NB Mask is that it performs 4 individual searches simultaneously and uses bit masking operations.

F. Low Bin NB SIMD

This function uses AVX2 SIMD intrinsics to do a binary search for four target values simultaneously. This implementation has the same functionality as low_bin_nb_4x, but it is designed to use SIMD (Single Instruction several Data) instructions. By doing so, it allows the CPU to complete several binary search operations in a single instruction cycle. This considerably improves performance, especially for huge datasets.

G. Band Join

This function implements an in-memory band join operation. The function is designed to find all matching pairs of indices between two sorted tables, inner and outer. If values from the inner table fall within a specific range close to values from the outer table, they are matched. The defined range is [p - bound, p + bound] for each value in the outer table. This function utilizes low_bin_nb_4x to narrow down the search range in the inner table reducing the number of comparisons needed for each outer value. If the outer_size is not a multiple of 4, low_bin_nb_mask is used to narrow down the remaining records in the inner table reducing the number of comparisons needed for each outer value. The results are stored as pairs of indices in the inner_results and outer_results arrays, representing the matching rows from both tables.

H. Band Join SIMD

This function has the same goal as Band Join except it uses AVX2 intrinsics to process data in parallel and performs a highly optimized in-memory band join using SIMD (Single Instruction Multiple Data) techniques. This function achieves improved performance over scalar implementations by leveraging SIMD to process four outer values at a time. Any remaining elements that are not divisible by four are processed using a fallback scalar approach, ensuring correctness.

IV. Performance Testing

A. Testing Procedures:

Our testing involved two main sections; Queries with various functions and Band Joins. These two sections performed slightly different tasks and therefore were tested separately. The Queries section consisted of the functions: *simple_binary_search, low_bin_search, low_bin_nb_arithmetic, low_bin_nb_mask, low_bin_nb_4x, and low_bin_nb_simd*. The Band Join section tested the functions *band_join and band_join_simd.*

The functions for the Queries took two inputs: The data set size (N), and the number of times to repeat the search (R). To test these functions, we ran two experiments on them. Experiment 1 tested each function at a fixed data size of N = 1,000,000 and varied values of R from 1-32. In Experiment 2, we fixed the repeats at R = 32 and varied the data size from 10-10^7. These two experiments should test how functions react with various repeat sizes and how they compare with each other over various data sizes.

The Band Join functions test took four inputs: The inner table size (N), the outer table size (X), The size of the output table (Y), and the band size (Z). For our test, we thought of the example of joining a table with itself for some condition. For example, comparing employee salaries with other employees. Therefore, in our testing, we set X = N and varied N from 10:10^7 in increments of factors of 10. To ensure we captured all possible outputs within our range, we set Y = N. While it's unlikely all values of the tables want to join, we wanted to make sure we did not lose any amount of output. Finally, we wanted to keep our band small so we set Z = 100. Since our queries are numbers in the ballpark of 10^10, we felt that a size of 100 would be sufficiently small. To ensure our comparison between Band_Join and Band_Join_SIMD was fair, we used the same inputs for both functions.

B. Testing results:

To start off, let's take a look at the results of Experiment 1. The Figure below shows the results produced for each function at the set values.

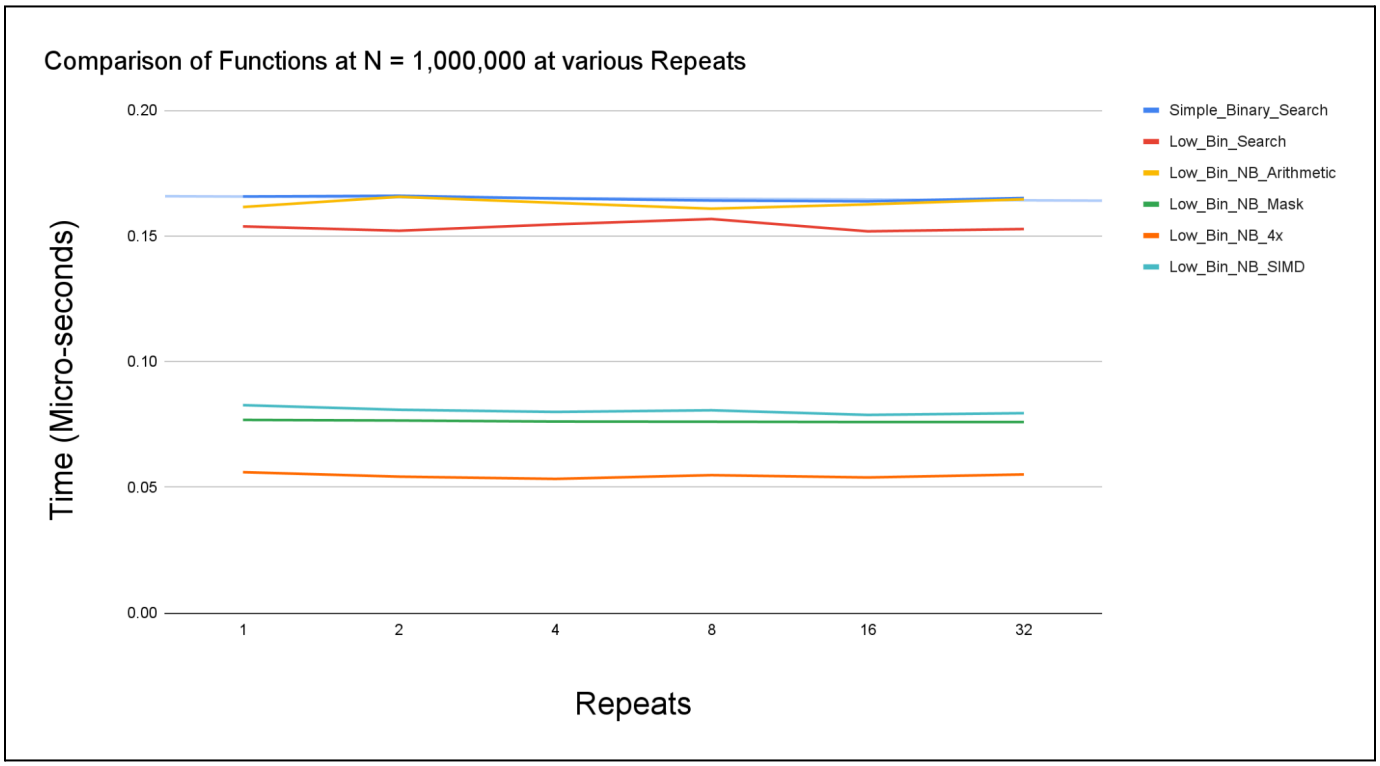

Figure 1

From this figure, we can first gather that with a large data size of N = 1,000,000 there is limited change in performance with a fixed data size and varying number of repeats. With this, we can rank the functions from fastest to slowest:

1. Low_Bin_nb_4x - Averaging 0.055 MicroSeconds
2. Low_Bin_nb_Mask - Averaging 0.076 MicroSeconds
3. Low_Bin_nb_SIMD - Averaging 0.081 MicroSeconds
4. Low_Bin_Search - Averaging 0.154 MicroSeconds
5. Low_Bin_nb_Arithmetic - Averaging 0.163 MicroSeconds
6. Simple_Binary_Search - Averaging 0.165 MicroSeconds

Next, let's take a look at the results of experiment 2 and how the functions compare to each other at different data sizes.

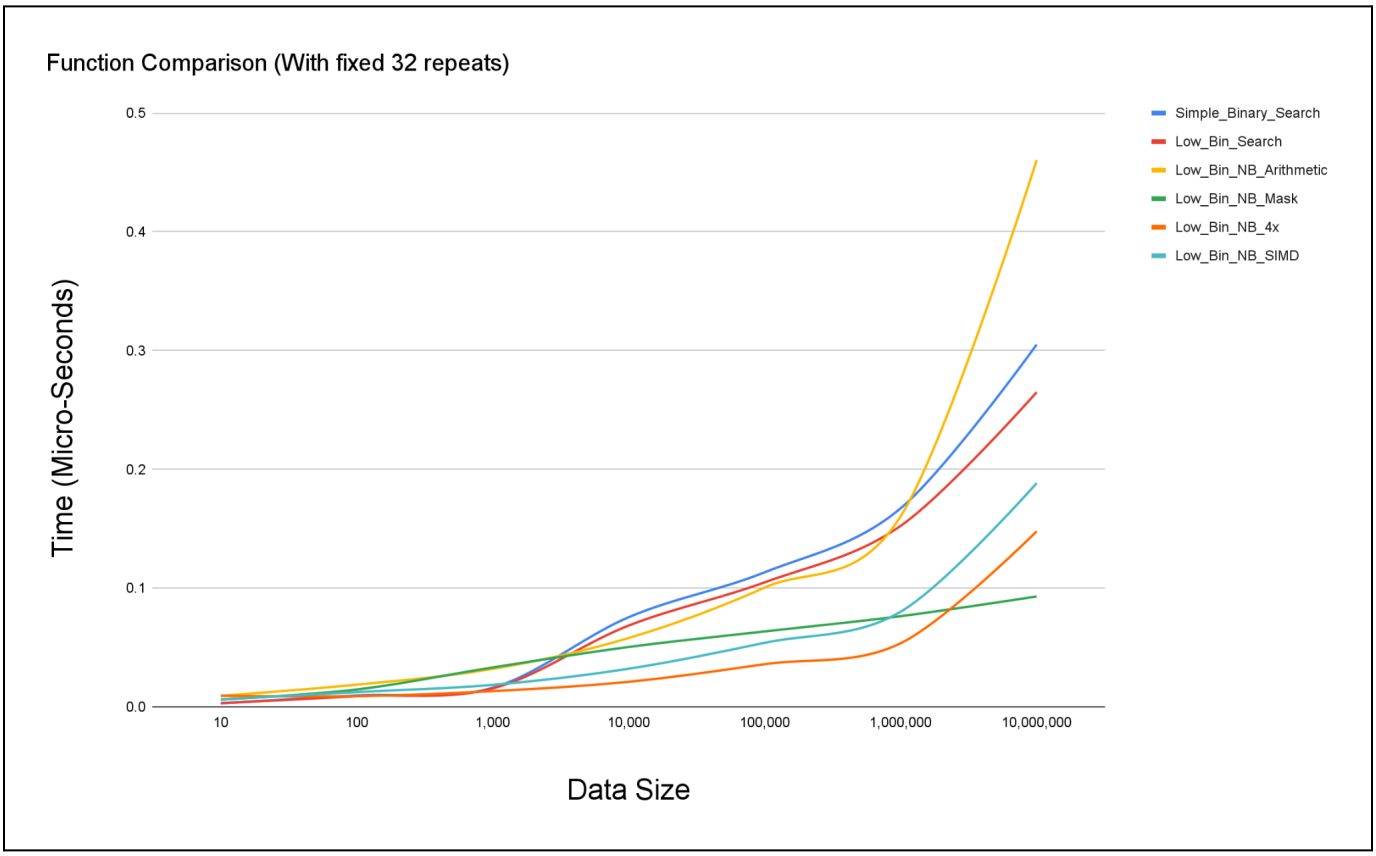

Figure 2

This figure shows us several key details about the performance of our functions and even the machine the experiments were carried out on. First, there is not much of a difference in the overall performances of these functions in the data range of 10-100. Around 1,000-1,000,000 is when we see a big shift in the performances and an overall ranking can be seen. In this data size range, we can clearly distinguish which functions are faster than others. Once the data set hits a size of 10^7, the average time of each function jumps up significantly. This is likely due to the hardware the experiment is running on. When the data size is less than or equal to 10^6, the computer is able to fit the data in memory which keeps the runtime relatively low. Once the data size hits 10^7, the computer can no longer store all of the data in memory

which significantly raises the run time of the functions. One interesting thing to note is that the function Low_Bin_nb_Mask has a runtime that is very linear. No matter the data size, its runtime seems to proportionately increase in a linear fashion. This is different from the rest of the functions as they have a lot more variance to them depending on the data set size.

Finally, let's take a look at the third experiment which compares the results of the two band join functions with various data sizes.

Figure 3

The results of the two joins show that both are quick and have similar performances. At low data sizes, Band_Join_SIMD tends to compute the data faster than Band_Join. With data sizes between 100 and 1,000,000, the joins show their best performances, with Band Join being slightly faster than Band Join Simd. Similar to our test of the functions, once the data size hits 10^7, the computation time spikes

up likely due to the lack of memory space. These results show that for small amounts of data, it's best to use Band Join Simd due to not needing to set up the join. Otherwise, it's best to use Band Join for the best performance possible.

V. Conclusion:

In conclusion, there are many steps you can take to increase the performance of code. By limiting unnecessary or slow instructions while taking advantage of your hardware you can dramatically increase performance. When taking steps to optimize code, it's important to understand what your goals and requirements are. You should also understand how your hardware will affect your code. In our case, we noticed how a lack of memory would severely affect runtime with very large datasets. Based on our results, when performing a binary search it's best to construct something like our low_bin_nb_4x code that does 4 searches at once to take advantage of the CPU parallelism. However, if you are performing searches on very large datasets then we recommend low_bin_mask due to its growth rate being linear, thus slower than other methods. When performing band joins, we found that it's best to do it directly instead of trying to take advantage of the AVX-256 instructions. While our results might differ if the tests were to be redone on different hardware, we found that in general, these functions are the best way to optimize binary search and band join.