Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?

Peter Van Sandt, Yannis Chronis, Jignesh M. Patel
Department of Computer Sciences, University of Wisconsin-Madison
{van-sandt,chronis,jignesh}@cs.wisc.edu

ABSTRACT

In this paper, we focus on the problem of searching sorted, in-memory datasets. This is a key data operation, and Binary Search is the de facto algorithm that is used in practice. We consider an alternative, namely Interpolation Search, which can take advantage of hardware trends by using complex calculations to save memory accesses. Historically, Interpolation Search was found to underperform compared to other search algorithms in this setting, despite its superior asymptotic complexity. Also, Interpolation Search is known to perform poorly on non-uniform data. To address these issues, we introduce SIP (Slope reuse Interpolation), an optimized implementation of Interpolation Search, and TIP (Three point Interpolation), a new search algorithm that uses linear fractions to interpolate on non-uniform distributions. We evaluate these two algorithms against a similarly optimized Binary Search method using a variety of real and synthetic datasets. We show that SIP is up to 4 times faster on uniformly distributed data and TIP is 2-3 times faster on non-uniformly distributed data in some cases. We also design a meta-algorithm to switch between these different methods to automate picking the higher performing search algorithm, which depends on factors like data distribution.

CCS CONCEPTS

• **Information systems** → **Point lookups**; *Main memory engines.*

KEYWORDS

In-memory search; Interpolation Search; Binary Search

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3300075

ACM Reference Format:
Peter Van Sandt, Yannis Chronis, Jignesh M. Patel. 2019. Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3300075

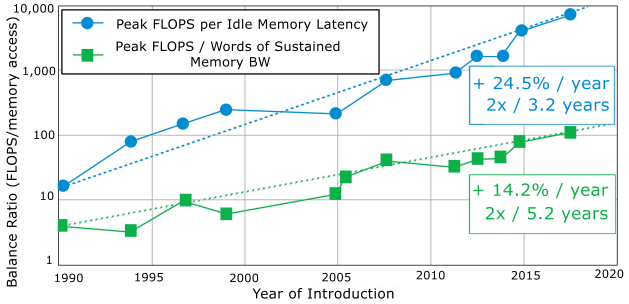

Figure 1: Speed comparison of representative processor and main memory technologies [27]. The performance of processors is measured in FLOPS. The performance of main memory is measured as peak FLOPS to sustained memory bandwidth (GFLOP/sec) / (Words/sec) and peak FLOPS per idle memory latency (GFLOP/sec) ∗ sec. In the conventional von Neumann architectural path, main memory speed is poised to become (relatively) slower compared to the speed of computing inside processors.

1 INTRODUCTION

Searching in-memory, sorted datasets is a fundamental data operation [23]. Today, Binary Search is the de facto search method that is used in practice, as it is an efficient and asymptotically optimal in the worst case algorithm. Binary Search is a primitive in many popular data systems and frameworks (e.g. LevelDB [25] and Pandas [30]).

Designing algorithms around hardware trends can yield significant performance gains. A key technological trend is the diverging CPU and memory speeds, which is illustrated in Figure 1. This trend favors algorithms that can use more computation to reduce memory accesses [4, 6, 16, 21, 27, 38]. The focus of this paper is on exploring the impact of this trend

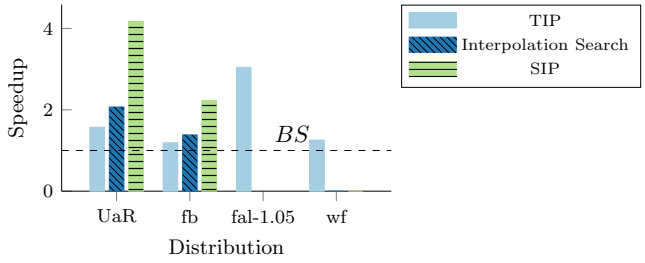

Figure 2: Speedup achieved by Interpolation Search methods over Binary Search on real datasets. The first two dataset *UaR* and *fb_ids* are uniformly distributed datasets, where as *fal-1.05* and *freq1* are skewed datasets. 8 Byte record are used. These datasets are described in detail in Section 5.1.3.

for Interpolation Search [31], a search algorithm that trades off computation to reduce memory accesses.

Interpolation Search has been studied in the past and despite its superior asymptotic complexity it has not seen broad adoption as a search algorithm for in-memory datasets. Interpolation Search has been found to underperform other search methods in practice due to its use of computationally expensive calculations [15]. Additionally, Interpolation Search was originally designed to search uniformly distributed data. It assumes a linear relationship between the values of the sorted data and their position within the dataset. This assumption enables the use of a linear interpolation, but it hurts the overall performance for datasets that don't follow such a distribution, namely non-uniformly distributed data.

This paper introduces two search algorithms to address the two shortcomings of the original Interpolation Search algorithm. Specifically, we introduce SIP (Slope reuse Interpolation) search, a collection of optimization techniques that includes re-using specific slope-related calculations, switching to fixed-point arithmetic from floating-point arithmetic (safely), and switching to a simpler search method on small ranges. These optimizations generally reduce the overall cost of Interpolation Search. SIP exploits hardware improvements [19] that accelerate the arithmetic operations. Essentially, SIP exploits the trend (shown in Figure 1) of the increasing divide between the costs of computation and memory accesses, with memory access poised to become relatively more expensive over time. SIP exploits this trend by using more complicated arithmetic to reduce the relatively more expensive memory accesses. As a result, SIP is more efficiently than Binary Search on uniformly distributed data.

We also introduce TIP (Three point Interpolation) search, an interpolation based search algorithm that uses a non-linear interpolation method to handle non-uniformly distributed data. It accurately estimates the position of the record(s) being searched on a broad range of non-uniform distributions by fitting linear fractions [20] to the distribution of the dataset.

We evaluate SIP and TIP over a variety of datasets against an optimized Binary Search implementation. Figure 2 highlights a key result by comparing the speedup of SIP, TIP and Interpolation Search to Binary Search on real and synthetic datasets. When searching the uniformly distributed datasets (*UaR* and *fb_ids*) all the interpolation-based search algorithms (Interpolation Search, SIP, TIP) outperform Binary Search, which is the baseline. SIP achieves a speedup of up to 4X. In the case of the non-uniformly distributed datasets (*fal-1.05* and *freq1*), SIP and Interpolation Search are dramatically slower than Binary Search, and do not show up in the chart as they are more than 6000X slower than Binary Search. TIP, on the other hand, outperforms Binary Search, and in one case by ~3X. SIP and TIP successfully capitalize on the diverging memory and processor speeds to outperform Binary Search. We expect the improvement of our algorithms to increase over time as the gap between memory and processor speeds continues to increase.

We also note that each algorithm (Binary Search, SIP and TIP) may outperform the others based on aspects such as dataset and hardware characteristics. To address this issue, we propose a sampling-based meta-algorithm to automatically switch between these methods.

The remainder of this paper is organized as follows: We review Interpolation Search in § 2. The SIP and TIP methods are described in §3 and §4, and evaluated in §5. Related work is presented in §6, and our concluding remarks are in §7.

2 INTERPOLATION SEARCH

In this section we describe Interpolation Search and present the shortcomings of the algorithm.

Algorithm 1 Interpolation Search

Input: V, y^* ▷ V: sorted array of size n, y^*: the target value
Output: position x^* of value y^*

```
1:  left ← 0
2:  right ← n−1
3:  while left < right do
4:      if V[left] == V[right] then
5:          if V[left]==y* then return left
6:          else return NotFound
7:      slope ← (right - left) / (V[right] − V[left])
8:      expected ← ⌊left + (y* − V[left]) ∗ slope⌋
9:      if V[expected] < y* then
10:         left ← expected + 1
11:     else if y* < V[expected] then
12:         right ← expected − 1
13:     else
14:         return expected
15: return NotFound
```

Given a sorted collection of values, V, and a target value, y^*, a search algorithm returns the position x^*, s.t. $V[x^*] = y^*$. Interpolation Search, Algorithm 1, iteratively reduces the search interval. The search interval is the part of the original collection that may contain the target value, possible containment can be checked as the values are sorted. For the first iteration, the search interval is initialized to contain the entire collection (lines: 1, 2). In each iteration Interpolation Search picks an expected position, *expected* (lines: 7-8), compares its value to the target value y^* and reduces the search interval accordingly (lines: 9-14). Each subsequent iteration considers only the reduced search interval. Lines: 4-6 handle the case where the reduced interval contains only one distinct value, this is a modification not usually included in Interpolation Search papers [31]. This algorithm is different from Binary Search in the way *expected* position is calculated.

Binary Search uses the middle position of the search interval, $(left - right) \div 2$, as the *expected* position. An interpolation based search fits a function f to V to characterize the distribution of the values and approximate the *expected* positions. Interpolation Search uses a linear function (line: 8 in Algorithm 1). It solves the equation $y^* = ax + b$ for x for the line that passes through the leftmost (smallest) and the rightmost (biggest) values in the search interval. The *expected* position, x, is calculated as :

$$expected = left + (y^* - V[left]) \frac{right - left}{V[right] - V[left]} \quad (1)$$

Throughout this paper, we describe collections by the distribution of their values. To discover the distribution of values, we correlate each value with its position in the sorted collection and plot them in two dimensions. For the y coordinate we use the value and for the x coordinate we use the position. In this method of visualization, a perfectly uniformly distributed collection of values appears as a diagonal line. Imperfections, like missing values, in uniformly distributed collections produce visualizations like the one shown in Figure 3. Conversely, a non-uniform dataset has points "far away" from the diagonal line. Interpolation Search assumes a linear relationship between the values and their positions within the sorted collection.

For the data visualized in the left in Figure 3, Interpolation Search will fit the line shown in the right in that figure. Algorithm 1 uses this line to calculate the *expected* position. In the first iteration, it computes *expected* as point 1 (which happens to have the value 5). The next step is to access the value of the collection at the *expected* position, namely $x = expected$ (point 2). If the value at that position equals y^* then the algorithm returns, otherwise it reduces the search interval and continues the search. The new search interval will use the *expected* position as one of its endpoints and will include the appropriate part of the previous search interval.

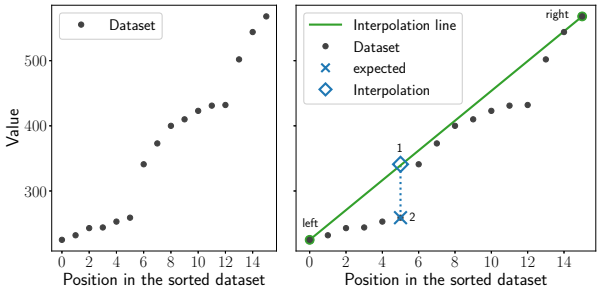

Figure 3: Left: a collection of values, Right: linear interpolation when searching for $y^* = 341$ and $x^* = 6$.

2.1 Performance for uniform data

Multiple analyses [31, 39] show that Interpolation Search performs an expected $\log\log N + O(1)$ iterations on values drawn from a uniform distribution, where N is the number of values in a collection. In the worst case (with non-uniform distributions), Interpolation Search is an $O(N)$ algorithm. In contrast, Binary Search is worst case optimal and performs $\log N$ iterations. Comparing the complexities of the two algorithms, one expects Interpolation Search to be faster on average as its complexity grows more slowly with N.

This asymptotic analysis considers only the number of memory accesses, as one iteration corresponds to one memory access. In an in-memory setting, the overhead of additional branches, arithmetic, and work that is done outside of the loop is significant. Interpolation Search performs more complex arithmetic and its control flow is more elaborate than that of Binary Search. Informally, this implies that if c_I is the average cost of an iteration in Interpolation Search and c_B is the average cost of an iteration in Binary Search, then a key ratio is $c = \frac{c_I}{c_B}$. Since, in general, Interpolation Search has to do more work per iteration, c is expected to be greater than 1. This extra work can wipe out the asymptotic advantage over Binary Search. Thus a key practical goal for Interpolation Search is to reduce c_I (and consequently c). In Section 3, we present various optimizations to address this issue, and build them into the SIP algorithm in Section 4.1.

2.2 Non-Uniformly Distributed Values

When values do not follow a uniform distribution, Interpolation Search can degenerate to searching values sequentially. Figure 4 shows the progression of Interpolation Search on a non-uniform collection of values. The *expected* positions (points 1 through 6) are calculated in each iteration. The progress made at each iteration is minimal as the use of linear interpolation is a poor approximation of the data distribution. To address this issue, we introduce a new interpolation algorithm called TIP in Section 4.2.

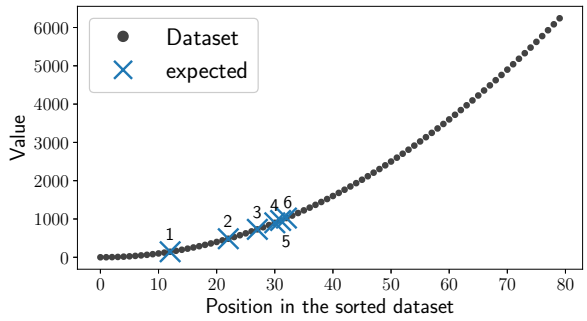

Figure 4: Progression of Interpolation Search on x^2 values, when $y^* = 32^2$.

3 FASTER INTERPOLATION SEARCH

This section presents optimizations that address the shortcomings of Interpolation Search identified in Sections 2.1 and 2.2. In Section 4, these optimizations are use to design the SIP and TIP algorithms.

3.1 Guard

The algorithmic advantage of Interpolation Search comes from its ability to quickly guide the search towards the target value by using information about the distribution of the data. The expected positions calculated by initial interpolations quickly converge to the target record, but the marginal benefit of further interpolations gradually diminishes. It is beneficial to stop interpolating when the progress made towards the target value does not warrant the cost of performing further interpolations. At that point, we can switch to a computationally cheaper search method, since we expect to be very "close" to the target record.

In this case, we choose to switch to using Sequential Search. Sequential Search visits every value and returns when it encounters the target key or when it determines that the target key is not contained in the dataset. Despite being a naive algorithm, it can be extremely fast when searching a very small number of records [9] by taking advantage of the spatial locality of its accesses and CPU parallelism.

An example of this behavior is shown in Figure 4. The first three interpolations (points 1-3) move the *expected* position close to the target position (point 6). But three more interpolations are required to find the target value. The progress made by the last three iterations does not justify the cost of interpolating, and switching to Sequential Search after the first three interpolations can be more efficient.

3.1.1 Optimizing Sequential Search. We can further optimize Sequential Search for the common case of searching a small set of elements. The key optimization we used are: loop unrolling and sentinels.

Loop unrolling takes several iterations of a loop and duplicates the code in each iteration, thus reducing the number of jumps and branch conditions, which in turn improves out-of-order execution. Overall, this method can improve the throughput/speed of the loop evaluation.

For each value that Sequential Search visits, it checks if the target is found and if the end of the data is reached. Sentinels are special values placed on either side of the data being searched. These values are chosen such that the condition that detects if the target key has been found can also be used to detect when the end of the collection has been reached, saving a comparison and branch in the loop of the algorithm. This technique requires a constant space overhead, but it only works if sentinel values are available, such as when the values in the dataset do not span the entire domain.

We evaluated the improvement of the two optimizations, loop unrolling and sentinels, using a microbenchmark designed to simulated the case of searching small arrays. The microbenchmark uses an array of 4096 elements and performs searches that search through 32 elements until they find the target. Combined the two optimizations improve the performance of Sequential Search by 150%.

3.2 Slope Reuse

As illustrated in Figure 3, each iteration during a search using a linear interpolation-based search method fits a line between the leftmost and the rightmost elements of the search interval. A line is characterized by its slope and intercept (Algorithm 1, lines: 7, 8). We can accelerate the interpolation computations by reusing the slope calculated in the first interpolation (Figure 14 in the Appendix illustrates the difference).

An alternative approach is to calculate a new slope every k iterations, instead of reusing the same slope throughout a search. In general, even for a very large collection of values, Interpolation Search performs a small number of iterations; e.g. $\text{loglog}10^9 \approx 5$ iterations for 1 billion values. Furthermore, approaches like this integrate multiple phases into the core loop and introduce additional branches, overhead and code size. This approach not only adds computational and control overhead, but it also impacts the degree to which loops can be overlapped due to the larger code footprint. In general, smaller loops are preferred for better cache efficiency and loop throughput [16].

3.3 3-Point Interpolation

Linear interpolation was designed and can successfully fit values that follow a uniform distribution. A more flexible interpolation method is necessary for an Interpolation Search algorithm to perform well on non-uniform distributions. The flexibility of fitting a variety of distributions will entail a

higher computational cost. However, if now fewer iterations are needed, then the overall search time can be improved.

We experimented with classical interpolating and spline fitting methods [1, 8], and we explored search as a root finding problem. For example, Interpolation Search is the regula falsi method described in [28, 34]. Polynomial functions, like cubic functions, are a common choice in numerical approximation [1], however, polynomial interpolation is not able to effectively fit fast growing distributions.

Jarratt and Nudds [20] describe a 3-point iterative root finding method using linear fractions, that fits a curve to the values of a dataset using three points. We combine Jarratt and Nudds's 3-point formula with bracketing, and introduce 3-Point Interpolation. To the best of our knowledge, this approach has not been proposed before, and as we show later, our new interpolation approach is able to fit a variety of non-uniform datasets.

Bracketing: A bracketing method maintains a search interval such that the target is contained within the interval. In each iteration, the search interval is reduced. Interpolation Search and Binary Search are examples of bracketing methods. The solid lines shown in Figure 15 in the Appendix, visualize how the 3-Point Interpolation method can fit non-uniformly distributed data and adapt to the relevant part of the data in each iteration.

During each iteration, the 3-Point Interpolation method fits a curve to three points of the search interval and calculates an *expected* position using Equation 2:

$$expected = x_1 + \frac{y_1(x_1-x_2)(x_1-x_0)(y_2-y_0)}{y_2(x_1-x_2)(y_0-y_1)+y_0(x_1-x_0)(y_1-y_2)} \quad (2)$$

$$y_i = V[x_i] - y^*, with\ V\ a\ sorted\ collection$$

The formula uses three points from the dataset that is being searched, namely *(position in the dataset, value):* $(x_0, V[x_0])$, $(x_1, V[x_1])$, $(x_2, V[x_2])$.

3.4 Fixed-point arithmetic

Multiplication by the slope of the interpolation line (the fraction in Equation (1)) accounts for most of the arithmetic cost in Interpolation Search. Equation (1) calculates the *expected* position, an integer. The slope can be represented as a floating-point number to avoid using integer division which is slow [19]. But, while floating-point multiplication is fast, the keys must be converted to floating-point and the product must be converted to an integer to use as an index. The calculation can be accelerated by using fixed-point arithmetic which saves conversion to and from floating-point. Fixed-point arithmetic may be less precise and requires the calculation of an approximation term. We can tolerate the loss of precision because interpolation is speculative, and we eliminate the calculation cost of the approximation term by only using fixed-point arithmetic in the first iteration where the approximation term can be precomputed.

Fixed-point arithmetic approximates multiplication by an arbitrary fraction with multiplication by a fraction with a known denominator. We use the symbol $\otimes$ for the approximation in Equation 4. Multiplication followed by division by 2^{64} can be fused together and done more quickly than multiplication by an arbitrary fraction. We exploit this behavior by finding an approximation, $s' = p' \div 2^{64}$, for an arbitrary fraction $s = p \div q$ such that $s' \approx s$. We multiply y by s' by returning the high word of the product of p' and y. The product of two 64-bit integers is a 128-bit integer with a high 64-bit word and a low 64-bit word. Returning the high word is equivalent to division by 2^{64}. To find p' we multiply p by 2^{64} and divide by q as seen in Equation 3.

$$\begin{aligned} y * s = y^* \frac{p}{q} &\approx \lfloor y^* \lceil \frac{2^{64}p}{q} \rceil \div 2^{64} \rfloor \\ &= \lfloor y * p' \div 2^{64} \rfloor, p' = \lceil \frac{2^{64}p}{q} \rceil \quad (3) \\ &= y \otimes s' \quad (4) \end{aligned}$$

The calculation of p' is too expensive to do in the core loop. For the first interpolation, we can pre-compute the value of p' using 128-bit arithmetic without loss of precision. We assume that $s < 1$, so p' fits in a 64-bit integer. This assumption is realistic because the denominator comes from the values in the array, which can be scaled if necessary, and the numerator comes from the length of the array, which is likely to be smaller than the maximum value of $q = 2^{64}$.

4 SIP AND TIP

This section introduces two new interpolation based search algorithms SIP and TIP. These algorithms address the shortcomings of the original Interpolation Search, by employing various optimizations (described in Section 3), as outlined in the following table:

Optimization	SIP	TIP
Guard	✓	✓
Slope Reuse	✓	
3 Point Interpolation		✓
Fixed-point arithmetic	✓	

Both algorithms use Guard conditions. SIP, or **S**lope reuse **I**nter**P**olation Search, uses the Slope Reuse and fixed-point arithmetic optimizations to reduce the cost of each iteration. TIP, or **T**hree point **I**nter**P**olation Search, uses the 3-Point interpolation method to fit a variety of datasets with non-uniformly distributed values.

The following subsections present how each optimization is incorporated into the two algorithms. We also quantify the benefits of each optimization in isolation by comparing the speedup that SIP or TIP achieves against a version of the

same algorithm with the evaluated optimization disabled. For this evaluation, we borrow the settings from Section 5.1.2, which includes executing each method mutiple times, and we use the *UaR* dataset. This dataset contains records with keys chosen uniformly at random from the interval [1,2^{63}]. (Section 5.1.3 describes this dataset in more detail.) We use the best configuration of guard size for each algorithm which is decided empirically.

4.1 SIP: Slope Reuse Interpolation Search

SIP aims to optimize the cost of linear interpolation, addressing the first shortcoming of Interpolation Search, its poor performance when searching in-memory, uniformly distributed, datasets. SIP reduces the cost of each interpolation iteration.

Algorithm 2 Slope reuse Interpolation Search (SIP)

Input: $V, y^*, guard\_size, slope$ ▷ V: sorted array of size n
Output: $position\ of\ value\ y^*$ ▷ y^*: the target value
1: $left = 0$ ▷ *slope*: precomputed slope
2: $right = n-1$ ▷ ⊗: fixed-point arithmetic multiplication
3: $expected = left + \lfloor (y^* - V[left]) \otimes slope \rfloor$
4: **while** *true* **do**
5: **if** $V[expected] < y^*$ **then**
6: $left = expected + 1$
7: **else if** $V[expected] > y^*$ **then**
8: $right = expected - 1$
9: **else**
10: **return** *expected*
11: **if** left == right **then**
12: **return** *NotFound*
13: $expected = expected + \lfloor (y^* - V[expected]) \otimes slope \rfloor$
14: **if** $expected + guard\_size \geq right$ **then**
15: **return** $sequential\_search(V, y^*, right)$
16: **else if** $expected - guard\_size \leq left$ **then**
17: **return** $sequential\_search(V, y^*, left)$

SIP, Algorithm 2, uses the same linear interpolation method as *Algorithm* 1 and assumes a linear relationship between the values and their positions. It iteratively calculates *expected* positions (lines: 3, 13) and reduces the search interval by comparing the corresponding values to the target value (lines: 5-10). SIP achieves a speedup of 1.5 to 3.7X against Interpolation Search on the *UaR* dataset for different dataset sizes, while Figure 2 presents the speedup achieved by SIP against Interpolation Search and Binary Search for the *fb_ids* dataset (details about the datasets can be found in Section 5.1.3).

Next, in subsections 4.1.1– 4.1.3, we present the optimizations employed by SIP. A detailed evaluation can be found in Appendix A.3.

4.1.1 Guard in SIP. We incorporate Guard conditions in each interpolation iteration (lines: 14, 16) that monitor the progress made towards the target record. If the new search interval does not differ significantly, the algorithm switches to sequential search from the endpoint of the search interval that violated the guard condition (line:15, 17). SIP achieves a speedup from 1.2X up to 3.7X when Guard is used compared to a variant of SIP without the Guard. The chosen *guard_size* for SIP, is empirically derived.

Guard, also, protects from the worst case of Interpolation Search, when the search degenerates to examining almost every record in the dataset sequentially, like in Figure 4.

4.1.2 Reuse Slope in SIP. SIP reuses the slope of the line fitted during the first iteration for every subsequent iteration (line: 13), like in Figure 14. This slope does not change when searching for different values over the same dataset, as it only depends on the search interval and for every first iteration, this search interval is the whole dataset, thus allowing us to precompute this slope for each dataset. When reusing the slope SIP achieves a speedup from 1.25X to 1.4X.

4.1.3 Fixed-point arithmetic in SIP. Linear interpolation is described in Equation 1 and calculates the *expected* position during each iteration. Linear interpolation can be split into the calculation of the slope:

$$slope = \frac{right - left}{V[right] - V[left]} \tag{5}$$

and of the *expected* position, which uses the slope:

$$expected = expected + (y^* - V[expected]) * slope \tag{6}$$

We consider integer arithmetic, floating-point arithmetic, and fixed-point arithmetic. Using integer arithmetic, we would have to rewrite the calculations as:

$$expected = expected + (y^* - V[expected]) \div \frac{V[right] - V[left]}{right - left} \tag{7}$$

This is necessary since $\frac{right-left}{V[right]-V[left]}$ truncates to zero because the denominator is larger. With integer arithmetic, we need to perform two slow integer divisions to find the *expected* position.

Floating-point multiplication is preferable to integer division because, depending on the number of queued operation, it requires close to 5 cycles, while integer division requires around 80 cycles [19]. If floating-point arithmetic is used, then we can use Equation 6, which is faster, but the necessary conversions to and from floating-point are on the critical path.

Fixed-point arithmetic transforms the division in the calculation of the *expected* position into a cheap multiplication by an approximation (line: 13). Because SIP reuses the same slope in each dataset, the divisor does not change, and the approximation can be precomputed. The Intel Architecture Code Analyzer [18] reveals that we begin loading the first

record in the search 24% sooner using fixed-point arithmetic rather than floating-point arithmetic.

Essentially, we amortize the cost of calculating the approximation of the fixed-point arithmetic by precomputing it, and we trade off accuracy for speed by eliminating floating-point conversions and division. SIP achieves a speedup of up to 5.5X when using fixed-point arithmetic compared to using integer arithmetic and up to 1.4X compared to using floating-point arithmetic.

4.2 TIP: Three Point Interpolation Search

Although SIP reduced the cost of each interpolation compared to Interpolation Search, the use of linear interpolation hurts the algorithm's performance on non-uniform distributions. We introduce a new search algorithm, called Three Point Interpolation Search (TIP), that employs the 3-Point Interpolation method to effectively characterize and interpolate on a variety of non-uniform distributions.

Algorithm 3 Three point Interpolation Search (TIP)

```
Input: V, y*, guard_size                    ▷ V: sorted array of size n
Output: position of value y*                ▷ y*: the target value
 1: left ← 0
 2: right ← n−1
 3: mid ← n÷2
 4: expected ← interpolate_R(y*,left,mid,right)        ▷ Eq. 8
 5: while true do
 6:     if abs(expected−mid) < guard_size then
 7:         return sequential_search(V, y*, expected)
 8:     if V[mid] ≠ V[expected] then
 9:         if mid < expected then
10:             left ← mid
11:         else
12:             right ← mid
13:         if expected + guard_size ≥ right then
14:             return sequential_search(V, y*, right)
15:         else if expected − guard_size ≤ left then
16:             return sequential_search(V, y*, left)
17:     mid ← expected
18:     expected ← interpolate_JN(y*,left,mid,right) ▷ Eq. 2
```

TIP follows the general structure of an Interpolation Search algorithm. It iteratively calculates *expected* positions (lines:4, 18) and reduces the search interval by comparing the value corresponding to the current *expected* position to the target value (lines: 9-12). *Figure* 2 presents the speedup achieved by TIP compared to Binary Search and Interpolation Search for the *fal* and *freq1* datasets.

4.2.1 Guard Condition in TIP. In TIP, Guard conditions are used in each iteration (lines: 6, 13, 15). The conditions monitor the progress made towards the target record and switch to sequential search if they progress is not significant (cf. Section 4.1.1). The *guard_size* is derived empirically.

4.2.2 3-Point Interpolation method in TIP. In addition to the Jarratt and Nudds formula, described in Section 3.3, TIP uses a simplification of the same formula, described by Ridders [36], which is easier to compute, but works only when the middle point is equidistant from the left and the right points.

TIP initializes the three interpolation points to the leftmost, mid and rightmost positions (lines: 1-3). We combine the two 3-Point formulas as follows: in the core loop (line: 18) we use the formula described by Jarratt and Nudds, Equation 2, because it allows us to reuse the most recent interpolation point and perform one memory access per iteration (*mid* at line:17 is assigned the value that *expected* had in the previous iteration). Ridder's formula requires two random memory accesses per iteration. For the first interpolation (line: 4), we use Ridder's simplified formula, Equation: 8. This method is cheaper to calculate but requires the three points to be equidistant, which can make it harder to produce them. But since we only use this formula for the first interpolation, we are able to precompute the three points.

$$expected = x_1 + \frac{y_1(x_1 - x_0)(1 + \frac{y_0 - y_1}{y_1 - y_2})}{y_0 - y_2 \frac{y_0 - y_1}{y_1 - y_2}} \qquad (8)$$
$$y_i = V[i] - y^*$$

Duplicate values can affect the 3-Point interpolation method. TIP protects against this case by not changing the endpoints of the search interval if $V[mid] = V[expected]$. Without this condition, one of the endpoints ($left$, $right$) would be set to *mid* (lines: 10, 12). This would result in two of the three points having the same value since $mid \leftarrow expected$ (line: 17). A curve calculated using 3 points with 2 distinct values would not be able to fit non uniform datasets effectively.

4.3 Discussion

To create SIP and TIP, we explored how the optimizations described in Section 3 could be combined to address the shortcoming of Interpolation Search.

Fixed-point arithmetic which accelerates divisions in SIP is not applicable to TIP because the formulas that TIP uses to calculate the *expected* positions contain fractions with the target value present in both the numerator and the denominator. The approximation required by the fixed-point arithmetic transformation would have to be calculated every time, which is a significant overhead.

We also investigated a simplification of the 3-point method that reuses calculations in subsequent interpolations, like what Slope Reuse does for SIP. We were not successful in

coming up with such a simplification, but we believe this is likely an interesting direction for future work.

Neither SIP nor TIP parallelize the search for a single value, as it does not yield any benefit. Parallelizing a single search requires partitioning the data to be searched and distributing them across multiple threads. Consequently, only one thread would have the segment of the original data that contains the target value. Because the data is sorted checking for containment is cheap, so none of the threads except one would contribute anything towards finding the target value. Searches for multiple values can be parallelized by assigning different search values to different threads, which is synergistic with our core task of speeding up the search task in each thread.

5 EVALUATION

In this section, we present results from an empirical evaluation of various in-memory search algorithms.

5.1 Experimental Setup and Methodology

5.1.1 Setup. The experiments presented in this paper were performed on a CloudLab[1] hosted server with two Intel(R) Xeon(R) CPU E5-2630 v3 running at 2.40GHz with 20MB of L3 cache, based on the Haswell architecture, and 128GB of main memory. The CPU Governor setting [5] is set to the "Performance" option. All algorithms were implemented in C++ and compiled using Clang 5.0.1 with maximum optimization settings, including unsafe floating-point math that assumes typical algebraic rules and similar arithmetic optimizations which might not comply with the standard. Our implementation is available on GitHub[2].

5.1.2 Experiments Methodology. In our experiments, we use datasets containing records sorted on an 8 Byte integer key. Each record contains an 8 Byte key and a fixed-length payload. The records simulate searching over database tables where the payload represents columns not participating in the search key. We populate our arrays with different numbers of records and differently-sized fixed-length payloads. By varying these parameters, we vary the part of the dataset that fits in each cache level, and the number of records that fit into a cache line, consequently changing the cost of memory accesses during search.

In Section 5.1.3, we describe how the values of the keys are chosen, we use real and synthetic datasets with uniform and non-uniform distributions. We vary the length of the synthetic datasets from 10^3 to 10^9, and the record size (key+payload) from 8 to 128 Bytes. The total size of the datasets we use ranges from 8KB to 32GB.

[1] https://www.cloudlab.us/

[2] https://github.com/UWHustle/Efficiently-Searching-In-Memory-Sorted-Arrays

	Record Size		
Dataset Size	8 Bytes	32 Bytes	128 Bytes
10^3	10%	4%	3%
10^4	7%	4%	4%
10^5	1%	1%	1%
10^6	4%	5%	3%
10^7	6%	5%	3%
10^8	2%	3%	2%

Table 1: Variance in the measurements: 90% of IQRs are below 10%.

We initialize the randomly-generated synthetic datasets with multiple seeds to account for the variability of random processes. In our results, we report this variance by showing the upper and lower quartiles of the measurements as error bars.

To measure the performance of an algorithm for a given dataset, we randomly permute the keys contained in the dataset and search for all of them, this is called a *run*. We measure the overall time to search subsets of 1,000 keys and collect at least 1,000,000 searches. We repeat this process multiple times to reduce variability across different runs.

To reduce possible variance in our measurements due to the contents of the cache and other hardware factors, we discard the first 30% of runs we perform for each experiment. We evaluate our measurement strategy by measuring 90^{th} percentile interquartile range between multiple runs of the same experiment, and present the difference between runs for all algorithms and datasets grouped by dataset and record size in Table 1. The small variance across our measurements strengthens the confidence in our results.

In our evaluation, we compare the algorithms based on the time to perform one search. This is calculated as the median of the measured times of the multiple runs. The search time for each run is an average over the time to search each subset.

5.1.3 Datasets. We experiment with eight different datasets that can be split into two groups. In the first group, the values of the datasets follow a uniform distribution. In the second group, the values of the datasets follow a non-uniform distribution. Both groups contain synthetic and real datasets. The values contained in the datasets range from 1 to 2^{63}. Duplicates exist in the datasets.

Uniformly Distributed Datasets: The uniform group contains three datasets *UaR*, *gap*, and *fb_ids*. The *fb_ids* dataset [11, 12] contains a uniform sampling of the ids of Facebook users. We used a subset of the dataset containing 289,000 records, as the rest of the dataset contains sequential values that can be fitted perfectly by a linear interpolation and thus does not provide any insights regarding the performance of the algorithm. The dataset *gap* is a synthetic dataset that captures the result of

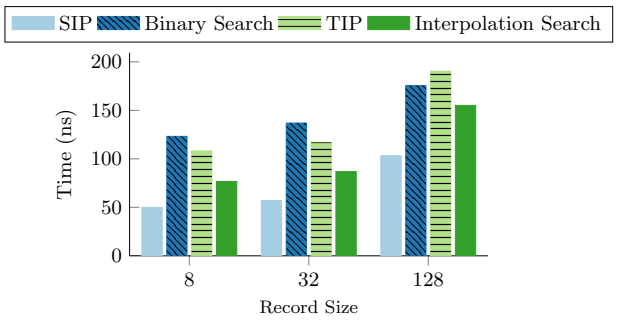

Figure 5: Comparison of SIP, TIP and Binary Search on the *fb_ids* dataset, over different record sizes.

a process where sequential ids are initially used, but gaps are created when records are removed uniformly at random. Using the shape parameter, we can choose how close the values are to perfectly sequential. The *UaR* (Uniform at Random) dataset contains values that are chosen uniformly at random from the interval $[1, 2^{63}]$.

Non-uniformly Distributed Datasets: We use three synthetic datasets, *fal, cfal, lognormal* and two real datasets *freq1* and *freq2*. The two real datasets describe the frequencies of words found in corpora. The *freq1* dataset contains the frequencies of the 2,076,000 unique words found in Wikipedia articles [3]. The *freq2* dataset lists the frequencies of 233,000 words from the dataset "Bag of Words" by Newman [4]. Values in both frequency datasets follow a Zipf distribution, which is observed in numerous real world phenomena, including population of cities, corporation sizes, and website visits [7, 26, 33]. The *fal* dataset also models values that follow a Zipf distribution. In the related dataset is *cfal*, the distance between consecutive values follows a Zipf distribution [32]. For the *fal* and *cfal* datasets, we specify how quickly the values grow by using a shape parameter, z. The shape parameter allows us to model distributions ranging from uniform to hyperbolic. Both *fal* and *cfal* are created using the original generator from [10], in which *fal:* N/r^z, *cfal: CumulativeSum(fal)*, where N=size of dataset, r = position in the sorted dataset, and z = shape. For our experiments we use multiple shapes like in [2]. Figure 21 in the Appendix visualizes the *fal* and *cfal* datasets for three different shapes. The values in the *log-normal* dataset follow the log-normal distribution. This distribution has been found to model various natural phenomena, such as the time spent by users reading articles online [40] and the size of living tissue [17]. To generate the log-normal values, we draw samples from a log-normal distribution. We parameterize the distribution with $\mu = 0,\ \sigma = 2$ as in [24].

5.1.4 Baseline Algorithm. To evaluate the performance of SIP and TIP, we compare them against Binary Search. We carefully looked at various optimizations that are possible over a textbook implementation of Binary Search and created an optimized Binary Search method. This optimized Binary Search method is described in more detail in Appendix A.2, and is used in all the subsequent experiments. The optimized version was significantly faster in many cases. For example, it was about 4X faster for small datasets with 1000 records.

[3] https://en.wikipedia.org/wiki/Wikipedia:Database_download
[4] https://archive.ics.uci.edu/ml/datasets/bag+of+words

5.2 Searching Uniform Datasets

This section evaluates SIP and compares it to Interpolation Search, TIP and Binary Search when searching datasets containing uniformly distributed keys. Figure 5 shows the time required to search for a key in the real dataset *fb_ids*, which contains keys from a uniform sampling of the ids of Facebook users. We evaluate the three algorithms over three different record sizes. SIP is more than twice as fast as Binary Search and TIP and around 50% faster than Interpolation Search.

The optimizations we introduced and applied to SIP, reduce the arithmetic cost of each iteration significantly relative to Interpolation Search and allow SIP to outperform it. The reduced iteration cost allows SIP to exploit the superior complexity of linear interpolation and outperforms Binary Search. The linear interpolation used by SIP is cheaper than the three-point fractional interpolation used by TIP which allows SIP to outperform TIP.

5.2.1 SIP on Uniformly Distributed Data. We evaluate how the speedup achieved by SIP against Binary Search generalizes as the cost of memory access/record change, which we indirectly control by using different dataset and record sizes. Larger datasets and bigger records increase the cost of each memory access (per record) as fewer records fit in each cache level and in each cache line. More expensive memory accesses benefit SIP because it uses more expensive calculations to save memory accesses.

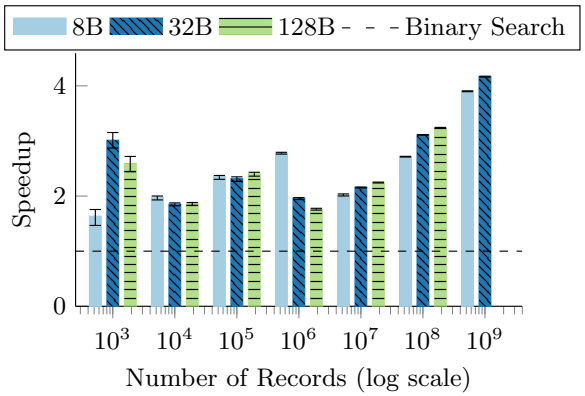

Figure 6: Speedup of SIP compared to Binary Search for different dataset and record sizes, on the *UaR* dataset. 10^9 records of size 128B exceed the memory capacity.

Figure 6 presents the speedup of SIP compared to Binary Search on the *UaR* dataset across multiple datasets and record sizes. We can observe that the speedup of SIP increases for bigger record sizes and for bigger dataset sizes. The error bars report the variability due to the multiple seeds used for the generation of the dataset. SIP achieves an up to 4X speedup, the results and trends for the *gap* dataset are similar. Crossing cache boundaries makes the graph "noisy" for smaller datasets, and a more clear pattern appears when the number of records grows relatively large.

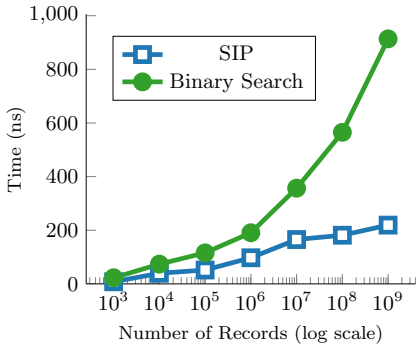

Figure 7: Time to perform a search for SIP and Binary Search for different dataset sizes, for the *UaR* dataset and record size = 32 Bytes.

Figure 7 plots the search times of SIP and Binary using the same data as shown in Figure 6, but just for the 32B record size. We would expect Binary Search, an $O(\log N)$ algorithm, to show as a straight line on a log-linear plot. The increasing slope of Binary Search is a result of the increasing cost of each memory access. Memory accesses are getting more expensive as the size of the dataset increases and consequently the part of the working set that fits in each cache level decreases. Our observation that the slope of SIP is decreasing suggests that we are realizing the theoretical $O(\log\log N)$ behavior in practice. We omit plotting the search times for all record sizes as they are similar.

SIP is able to outperform Binary Search on the uniform dataset across all records sizes, and successfully addresses the first shortcoming of the original Interpolation Search (namely, the poor performance of Interpolation Search). SIP benefits when memory accesses become more expensive as it accesses memory fewer times. We expect the performance difference between Binary Search and SIP to increase according to hardware trends, described in Figure 1, which bodes well for the future of SIP for such datasets.

5.2.2 Interpolation-Sequential Search. An interpolation-based search method uses information about the distribution of the data to calculate the *expected* position of the target value. An ideal interpolation method would be able to fit the data perfectly and with a single interpolation find the target value. Realistically we expect a very effective Interpolation Search method to guide us very close to the target at the first interpolation. The more effective the interpolation is, the smaller is the distance between the *expected* position and the target position, this distance is the *error of the interpolation*. When the error is small, a second interpolation would likely not offer any improvement that would warrant the computational cost.

Gonnet and Rogers [14] give an analysis of an Interpolation-Sequential search algorithm by Price [35] that uses a single interpolation to guide a subsequent sequential search which shows the expected number of sequential steps after the first interpolation to be $O(\sqrt{N})$. We have implemented Interpolation-Sequential Search which performs one interpolation and switches immediately to Sequential Search. As noted before, Sequential Search is generally faster than SIP, TIP, or Binary Search on very small numbers of records.

Interpolation-Sequential search can outperform SIP, if the first interpolation has a small error, such as when the values are nearly sequential. To evaluate this property, we use the *gap* dataset and vary the shape parameter. The shape parameter determines the number of elements that are randomly removed from a set of sequential elements; the bigger the shape parameter, the closer the dataset is to a collection of sequential values. Figure 8 shows the dataset size at which the performance of SIP and Interpolation-Sequential Search equalize. For smaller sizes, Interpolation-Sequential is faster. However, for larger sizes, SIP is faster for different record sizes. The equalization point changes as the record size changes. Different record sizes change the cost of memory access, as fewer records fit into a lower cache level and a cache line. When the shape parameter is 99.999% (in this case, the values are practically sequential), Interpolation-Sequential is faster for datasets with up to ten million records.

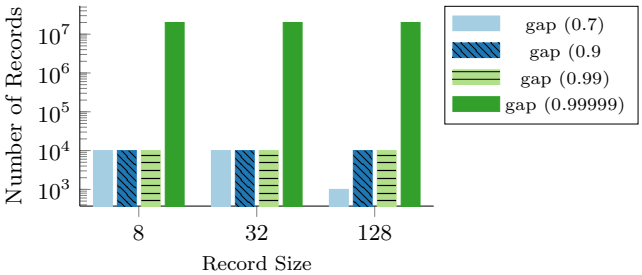

Figure 8: The dataset size (Number of Records) where SIP becomes faster than Interpolation-Sequential. For smaller sizes, Interpolation-Sequential is faster, for larger sizes, SIP is faster.

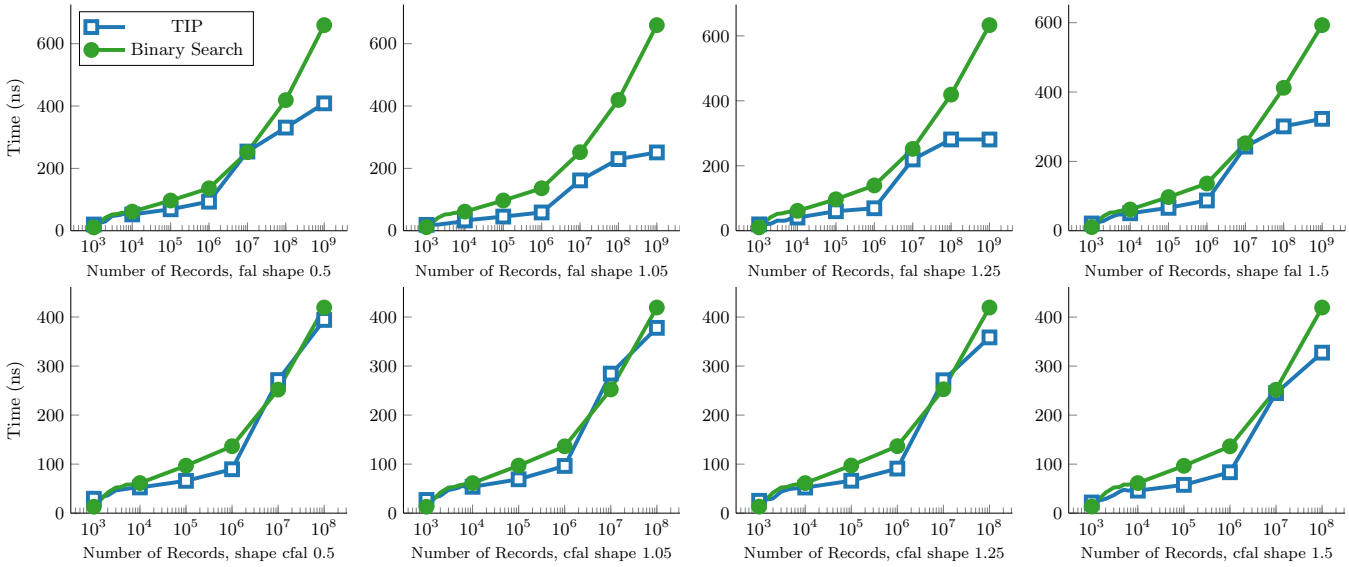

Figure 9: Time to perform a search for TIP and Binary Search for different dataset sizes on synthetic, non-uniform datasets. Record size is 8 Byte records, the results for other record sizes are similar and we omit them (X axis is in log scale).

We observe that the error of the first interpolation decreases with the dataset size. This suggests that even when the distribution is "less ideal," Interpolation-Sequential will outperform SIP because the absolute search distance is still small. SIP would have the same logical behavior when searching small datasets as Interpolation-sequential, but SIP would have to perform two interpolations so that the guard conditions could identify that the progress was not significant before switching to Sequential Search. The overhead of this extra work is substantial, especially when we expect searches to be compute bound. In Figure 10 we compare Interpolation Sequential to SIP for progressively smaller subset of the *fb_ids* dataset. Interpolation-Sequential is faster than SIP for small dataset sizes even if the fit in not ideal.

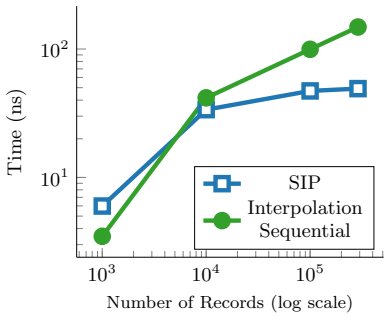

Figure 10: SIP compared to Interpolation-sequential for various sizes of the *fb_ids* dataset.

5.3 Searching Non-Uniform Datasets

This section evaluates TIP and compares its performance to Binary Search using datasets with values that follow non-uniform distributions. We designed TIP to address the second shortcoming of Interpolation Search, its inability to effectively fit non-uniform distributions.

5.3.1 TIP on Non-Uniformly Distributed Data. As highlighted in Figure 2 and discussed in Section 4, SIP is not well-suited for non-uniformly distributed datasets. On the other hand, TIP is designed to target this setting. TIP fits a range of distributions using a 3-point interpolation method instead of the linear interpolation used by Interpolation Search. We evaluate the ability of TIP to fit a wide range of distributions by using the *fal* and *cfal* datasets. These two datasets can represent a variety of distributions by varying their shape parameter.

Figure 9 compares TIP and Binary Search over different shapes and dataset sizes on *fal*. We use 4 shapes: 0.5, 1.05, 1.25, 1.5, similar to [2], which change the skew of dataset values as seen in Figure 21. For large numbers of records, TIP achieves up to a 2-3X speedup on some of the *fal* and *cfal* datasets (e.g. *fal* with shape parameter of 1.05).

We observe that Binary Search is faster on small dataset sizes, but that TIP stays competitive across all sizes and outperforms Binary Search across all shapes for datasets larger than 10^8. For the *lognormal* distribution we observe the same results, TIP is faster for sizes larger than 10^8 records. As described in Section 5.2.1, we would expect the performance of Binary Search to appear as a straight line when the time axis is linear and the dataset size increases logarithmically.

For TIP, the decreasing rate of increase shown in Figure 9 suggests sub-logarithmic runtime behavior in practice, similar to the $O(\text{loglog}N)$ behavior expected from Interpolation Search when it fits the distribution of the data.

We conducted further experiments across small dataset and record sizes to empirically determine when TIP starts becoming faster than Binary Search. Binary Search is faster than TIP on small data where the more costly arithmetic of the three-point interpolation is more expensive than additional random accesses. TIP becomes faster than Binary Search for datasets over 16KB, given the memory overhead for the benchmark, a 32KB L1 cache and cache conflicts this is likely when the working set exceeds the L1 cache.

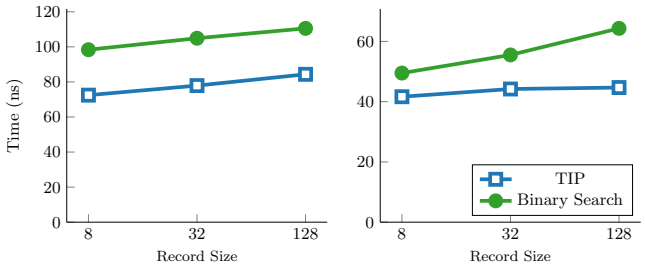

Figure 11: TIP compared to Binary Search on the $freq1$ (left) and $freq2$ (right) datasets for different record sizes.

Figure 11 shows the search duration of TIP and Binary Search on the $freq1$ and $freq2$ datasets, the keys of these two datasets describe the frequency of words found in corpora (Section 5.1.3). TIP outperforms Binary Search on these datasets even though they are still relatively small. Dataset $freq1$ contains 10 times more values than $freq2$ but exhibits shorter search times. This is likely caused by the relative dominance of duplicate entries in $freq1$. The number of unique frequencies is similar for the two datasets despite the 10X size difference. TIP benefits more than Binary Search from duplicate entries. TIP's use of sequential search allows it to terminate the sequential search earlier when a duplicate entry is encountered. On the other hand, our Binary Search is largely unaffected because it performs a constant number of iterations and a shorter sequential search.

5.4 Search time variance within a dataset

Because Binary Search reduces the search interval by half in each iteration, it performs a consistent number of iterations regardless of the distribution of the data. Interpolation methods are more susceptible to anomalies in the distributions of the data. This occurs when the data deviates from the interpolation function or when duplicates are present.

Dataset	Algorithm	Record Size		
		8 Bytes	32 Bytes	128 Bytes
fb_ids	SIP	1%	1%	2%
	Binary Search	1%	1%	1%
freq1	TIP	4%	5%	4%
	Binary Search	3%	3%	3%
freq2	TIP	2%	3%	4%
	Binary Search	2%	1%	2%

Table 2: Variance in searching for different keys.

In Table 2, we present the differences we measured across the times to search subsets of 1,000 keys, our base unit of measurements, within the same dataset, for all of our real datasets. The values we report refer to the differences of the upper and lower quartiles of our measurements. By studying this variance we can understand how anomalies in the distributions of the values affect the behavior of each algorithm. This summary shows consistent performance across searching for different keys. Duplicates in *freq1* and *freq2* explain higher variance in the duration of TIP searches.

5.5 Behavior Analysis of SIP

SIP combines two search methods, Interpolation Search and Sequential Search and switches between the two methods using Guard conditions. SIP employs two more optimizations (slope reuse and fixed-point arithmetic), as described in Section 4.1. Figure 12, presents the time SIP requires to search for one record and breaks down the time into the average number of interpolation iterations and average number of sequential steps performed for each dataset size. Each sequential step corresponds to examining one record by Sequential Search. The *UaR* dataset with record size 8B is used. This experiment allows us to compare the behavior of SIP with the theoretical analysis of Interpolation Search presented in Section 2. We observe that the increase of the number of interpolation iterations in relation to the size of the dataset is close to $\text{loglog}N$, which is the expected behavior according to the theoretical analysis of Interpolation Search [31].

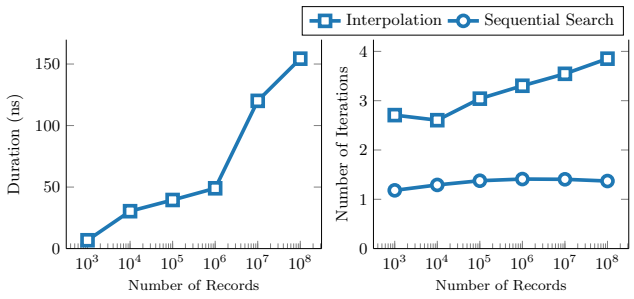

Figure 12: Time to search for one record, number of interpolations and sequential steps for SIP.

5.6 Alternative memory layouts

Binary and Interpolation Search require the data to be in arranged in sorted order. An alternative approach is to change the data layout for higher performance. Khunong et al [22] describe a method in which the data is viewed as being stored in a complete binary search tree. The values of the nodes in the tree are placed in an array in a left-to-right breadth-first traversal order. This format is called Eytzinger and the search algorithm simulates a search in this implicit binary search tree. The authors conclude that for small datasets binary search is faster, but for large datasets the proposed layout and search algorithm are faster. We implemented the two variations of the search algorithm proposed in [22] and compared them to SIP and TIP. We validate the results presented in [22] with our implementation of Binary Search. We replicated the experiments presented in Sections 5.2 and 5.3, and found that for uniformly distributed datasets when the Eytzinger layout is faster than Binary search, SIP is faster than both, outperforming Eytzinger by 10-250% depending on the dataset, number of records, and record size. For non-uniformly distributed datasets, Eytzinger and TIP are faster for different data distributions distribution, dataset and record sizes. The Eytzinger format can be faster up to 24% in some cases and TIP can be faster by up to 59%. Essentially, when Binary Search beats SIP and TIP, the Eytzinger also beats SIP and TIP.

An important point to note is that while the Eytzinger format offers performance improvements for "point" queries (in some cases), that format is not well-suited for range queries. Range queries need to scan (a portion of the sorted) data sequentially and the binary-tree based layout is not efficient for this access pattern. To investigate this aspect quantitatively, we created datasets containing from 10^3 up to 10^9 records with 8B keys and total record sizes ranging from 8B to 1024B. Then, we measured the performance to scan 10^2 to 10^7 records, starting from the middle record. The speedup achieved by using the sorted layout ranges from 1.6X to 3.8X, and shows that alternative memory layouts are disadvantaged when range queries are involved.

5.7 Integration with Numpy and LevelDB

We have also integrated SIP and TIP in Numpy [29] (version 1.16), which is used by Pandas [30], and LevelDB [25].

In Numpy, we extended the implementation of the *searchsorted*[5] method to use SIP and TIP in addition to Binary search. The *searchsorted* method returns the position a record should be inserted so that an array remains sorted. In the case of duplicates, an argument controls if the position at the end (or start) of the duplicates is returned. We compared SIP and TIP to Numpy's Binary Search by using all the datasets described in Section 5.1.3. For the synthetic datasets we varied the number of records from 10^3 to 10^9. The results validate our previous experiments and in most cases offer even better improvement as the Binary Search implementation used in Numpy is missing some of the optimizations present in our Binary Search (cf. Section A.2). SIP is up to 4.8 times faster when searching uniformly distributed datasets and TIP is up to 3 times faster when searching non-uniformly distributed datasets. Indicatively, the right panel in Figure 13 shows the speedup for the *UaR* and *fb_ids* datasets for different dataset sizes, record size is 8B. The left panel in Figure 13 shows the results for the same experiment shown in Figure 9 for the *cfal, shape=1.05* dataset and shows similar trends and performance improvement. (We omit detailed presentation of the reset of the experiments as they are similar to the ones presented earlier.) We note that TIP performs worse than Binary Search for the *freq* datasets by 15% to as high as 5X in one case, as Numpy handles the presence of duplicate values differently than our standalone implementation. With high number of duplicates, and Numpy's duplicate handling semantics, Binary Search is preferred. (The meta-algorithm in Section 5.8 can pick this case.)

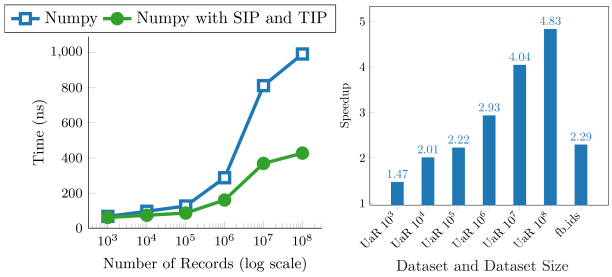

Figure 13: Left: Search times for the *cfal, shape=1.05*. Right: Speedup of Numpy with SIP and TIP on different sizes of the *UaR* dataset and on the *fb_ids* dataset.

We also integrated our search method into LevelDB [25]. LevelDB provides a benchmark [6], which we used in this evaluation. The dataset used in the benchmark follows a uniform distributions so only SIP was used in this experiment. We modified the block size to 2^{17} and also added 24 bytes and one key to each block to save reading the first and last keys in the compressed format. We run the *readrandom* LevelDB benchmark until the performance stabilizes and report the average of multiple runs after. For a database size of ~9GB, the benchmark performance improved by 38%, indicating that search time, which accounted for 56% of total execution time when using Binary search, improved by 146%. For a database size of ~35GB, the benchmark performance improved by 7%, indicating that search time, which accounted for 31% of total execution time when using Binary search, improved by ~3X.

[5] https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.searchsorted.html

[6] https://github.com/google/leveldb#performance

5.8 Choosing a search algorithm

Since each algorithm has a performance sweet spot, we need a method to choose between them. One approach is use a model of each method to predict the optimal algorithm. This exercise can be challenging as it requires accurately modeling data and hardware characteristics (and hardware characteristics are not easily exposed by hardware vendors).

Motivated by the the low variance in the search times of SIP and TIP (cf. Table 2), we propose a sampling-based meta-algorithm to select an appropriate search algorithm. When a new dataset is created we perform a small number (e.g. 10) random searches for each search algorithm, and pick the fastest for subsequent searches. The performance of Binary search is measured first and used as a threshold for the other methods. Appendix A.6 presents the algorithm.

During the lifetime of a dataset, records can be added or removed, potentially altering the distribution of the values, i.e. a uniformly distributed dataset may follow a non-uniform distribution after many updates/inserts. We implemented a safeguard for this case: we monitor the performance of the searches, and if they exceed a threshold (set to the difference between the performance of Binary Search and the fastest algorithm) we re-run the sampling algorithm. We implemented this meta-algorithm in Numpy and performed 10 searches (for the sampling component). The new meta-algorithm is able to select the fastest search algorithm for all the datasets that we presented in this paper. This method is effective as the time to perform each search is fast (tens to hundreds of nanoseconds) and the additional cost of sampling is negligible for workloads with a large number of search queries.

6 RELATED WORK

Searching sorted data is a fundamental problem in Computer Science. It is also a key data operation in databases systems and is used when traversing indices, accessing sorted files, etc. The most prominent algorithms, Binary and Sequential Search, have been studied extensively [23]. Sequential search examines all the elements of a dataset sequentially. It can be fast for searching smaller datasets and offers benefits when used in the final stages of more advanced methods [9, 13].

Interpolation Search has been used in database systems in special cases when the data exhibit a perfect distribution [15]. Binary Search is the de facto algorithm for searching over sorted datasets. It has predictable performance and its performance is independent of the distribution of the dataset, as it always reduces the search interval by half in each iteration. Previous work has explored hybrid methods that combine Interpolation Search with Binary Search to achieve the theoretical guarantees of Binary Search with the speed of Interpolation Search [3, 15, 37]. These methods have not been supported by experimental evaluations.

One could think indexing and searching methods as interchangeable solutions to the same problem, i.e. locating a record in a dataset. However, searching has advantages that an index can not offer: no space overhead, no preprocessing time, not affected by updates. Additionally, improving search methods can directly benefit the performance of an index. Indices use search methods to locate the target records at the final stage, i.e. inside a B-Tree node or as the model used at the final stage of a learned index [24]. Thus, these are separate problems, both worth independent examination.

7 CONCLUSION AND FUTURE WORK

Binary Search is the de facto algorithm for searching sorted data in-memory and a primitive in many systems. Diverging CPU and memory speeds motivate the use of algorithms that trade off computation for fewer memory accesses. Interpolation Search uses more complex arithmetic to reduce the number of memory accesses but is not widely used due to a number of shortcomings, including high computation overhead in each inner loop and poor behavior with non uniform data distributions. In this paper, we revisit Interpolation Search and introduce two new interpolation based search algorithms that address the shortcomings of the original algorithm. SIP and TIP, the two new algorithms, are designed to search data following uniform and non-uniform distributions respectively. Through a comprehensive experimental evaluation, we showed that SIP can be up to 4 times faster and TIP 2-3 times faster than a similarly optimized Binary Search implementation. The performance speedup of SIP and TIP over Binary Search is poised to increase as the gap between computation and memory access speeds widens.

There are a number of exciting directions for future work. NVMe storage devices (i.e. Intel Optane) are at the forefront of storage technology. They offer many benefits and can replace DRAM, but they offer slower access times. It will be interesting to examine the impact of our algorithms when data is stored primarily in these devices.

SIP, TIP, Sequential and Binary Search each have their performance sweet spots, which can change as the search interval changes. A dynamic algorithm that can adapt and use the best search method for the relevant part of the data (i.e. switch from one algorithm to the other in the same search), is also an interesting direction for future work.

ACKNOWLEDGMENTS

The authors would like to thank Eric Bach for his invaluable input. This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA and gift donations from Google and Huawei.

REFERENCES

[1] Boris Aronov, Tetsuo Asano, Naoki Katoh, Kurt Mehlhorn, and Takeshi Tokuyama. 2006. Polyline fitting of planar points under min-sum criteria. *International journal of computational geometry & applications* 16, 02n03 (2006), 97–116.

[2] Spyros Blanas, Yinan Li, and Jignesh M Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 37–48.

[3] Biagio Bonasera, Emilio Ferrara, Giacomo Fiumara, Francesco Pagano, and Alessandro Provetti. 2015. Adaptive search over sorted sets. *Journal of Discrete Algorithms* 30 (2015), 128–133.

[4] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the Memory Wall in MonetDB. *Commun. ACM* 51, 12 (Dec. 2008), 77–85. https://doi.org/10.1145/1409360.1409380

[5] Dominik Brodowski and N Golde. 2016. Linux CPUFreq–CPUFreq governors. *Linux Kernel.[Online]: http://www. mjmwired. net/kernel/Documentation/cpufreq/governors. txt* (2016).

[6] Carlos Carvalho. 2002. The gap between processor and memory speeds. In *Proc. of IEEE International Conference on Control and Automation*.

[7] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. 2009. Power-law distributions in empirical data. *SIAM review* 51, 4 (2009), 661–703.

[8] Carl De Boor, Carl De Boor, Etats-Unis Mathématicien, Carl De Boor, and Carl De Boor. 1978. *A practical guide to splines*. Vol. 27. Springer-Verlag New York.

[9] Paul M. Dorfman. 1999. Array Lookup Tecnhiques. Retrieved 11/01/2018 from https://analytics.ncsu.edu/sesug/1999/016.pdf

[10] Christos Faloutsos and HV Jagadish. 1992. On B-tree indices for skewed distributions. (1992).

[11] Minas Gjoka, Maciej Kurant, Carter T. Butts, and Athina Markopoulou. 2010. Walking in Facebook: A Case Study of Unbiased Sampling of OSNs. In *Proceedings of IEEE INFOCOM '10*. San Diego, CA.

[12] Minas Gjoka, Maciej Kurant, Carter T. Butts, and Athina Markopoulou. 2011. Practical Recommendations on Crawling Online Social Networks. *IEEE JSAC on Measurement of Internet Topologies* (2011).

[13] Gaston H Gonnet and Lawrence D Rogers. 1977. The interpolation-sequential search algorithm. *Inform. Process. Lett.* 6, 4 (1977), 136–139.

[14] Gaston H Gonnet, Lawrence D Rogers, and J Alan George. 1980. An algorithmic and complexity analysis of interpolation search. *Acta Informatica* 13, 1 (1980), 39–52.

[15] Goetz Graefe. 2006. B-tree indexes, interpolation search, and skew. In *Proceedings of the 2nd international workshop on Data management on new hardware*. ACM, 5.

[16] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.

[17] Julian Huxley, Richard E Strauss, and Frederick B Churchill. 1932. Problems of relative growth. (1932).

[18] Intel. 2017. IACA: Intel®Architecture Code Analyzer. Retrieved 11/01/2018 from https://software.intel.com/en-us/articles/intel-architecture-code-analyzer

[19] Intel. 2018. Intel® 64 and IA-32 Architectures Optimization Reference Manual. Retrieved 11/01/2018 from https://software.intel.com/en-us/articles/intel-sdm

[20] P Jarratt and D Nudds. 1965. The use of rational functions in the iterative solution of equations on a digital computer. *Comput. J.* 8, 1 (1965), 62–65.

[21] Kimberly Keeton. 2017. Memory-Driven Computing. USENIX Association, Santa Clara, CA.

[22] Paul-Virak Khuong and Pat Morin. 2017. Array Layouts for Comparison-Based Searching. *J. Exp. Algorithmics* 22, Article 1.3 (May 2017), 39 pages. https://doi.org/10.1145/3053370

[23] Donald Ervin Knuth. 1997. *The art of computer programming*. Vol. 3. Pearson Education.

[24] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 489–504.

[25] LevelDB. 2018. Retrieved 11/01/2018 from http://leveldb.org/

[26] Wentian Li. 2002. Zipf's Law everywhere. *Glottometrics* 5 (2002), 14–21.

[27] J. McCalpin. Invited talk at SC16, 2016. Memory Bandwidth and System Balance in HPC Systems. Retrieved 11/01/2018 from https://tinyurl.com/yanlv29r

[28] JM McNamee and VY Pan. 2013. Bisection and Interpolation Methods. In *Studies in Computational Mathematics*. Vol. 16. Elsevier, 1–138.

[29] Numpy. 2018. Sorted Search. Retrieved 11/01/2018 from https://docs.scipy.org/doc/numpy/reference/generated/numpy.searchsorted.html#numpy.searchsorted

[30] Pandas. 2018. Sorted Search. Retrieved 11/01/2018 from https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.searchsorted.html

[31] Yehoshua Perl, Alon Itai, and Haim Avni. 1978. Interpolation search a log log N search. *Commun. ACM* 21, 7 (1978), 550–553.

[32] Viswanath Poosala, Peter J Haas, Yannis E Ioannidis, and Eugene J Shekita. 1996. Improved histograms for selectivity estimation of range predicates. In *ACM Sigmod Record*, Vol. 25. ACM, 294–305.

[33] David MW Powers. 1998. Applications and explanations of Zipf's law. In *Proceedings of the joint conferences on new methods in language processing and computational natural language learning*. Association for Computational Linguistics, 151–160.

[34] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. 1996. *Numerical recipes in Fortran 90*. Vol. 2. Cambridge university press Cambridge.

[35] CE Price. 1971. Table lookup techniques. *ACM Computing Surveys (CSUR)* 3, 2 (1971), 49–64.

[36] C Ridders. 1979. Three-point iterations derived from exponential curve fitting. *IEEE Transactions on Circuits and Systems* 26, 8 (1979), 669–670.

[37] Nicola Santoro and Jeffrey B Sidney. 1985. Interpolation-binary search. *Information processing letters* 20, 4 (1985), 179–181.

[38] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News* 23, 1 (March 1995), 20–24. https://doi.org/10.1145/216585.216588

[39] Andrew C Yao and F Frances Yao. 1976. The complexity of searching an ordered random table. In *Foundations of Computer Science, 1976., 17th Annual Symposium on*. IEEE, 173–177.

[40] Peifeng Yin, Ping Luo, Wang-Chien Lee, and Min Wang. 2013. Silence is also evidence: interpreting dwell time for recommendation from psychological perspective. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 989–997.

A APPENDIX

A.1 Faster Interpolation Search

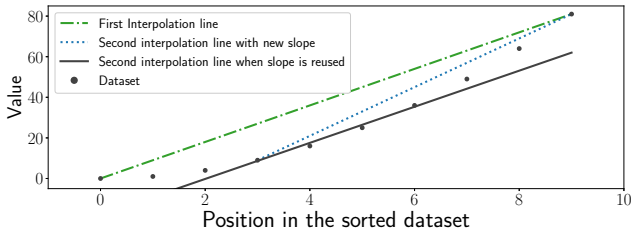

Figure 14: The lines of the first two interpolation when slope is reused and when it is not.

Figure 14 illustrates the lines of the first two linear interpolations when slope is reused and when it is not. The first interpolation line is the dotted-dashed, if the slope is reused the second line is parallel to the first (solid line), otherwise a new slope is used (dotted line).

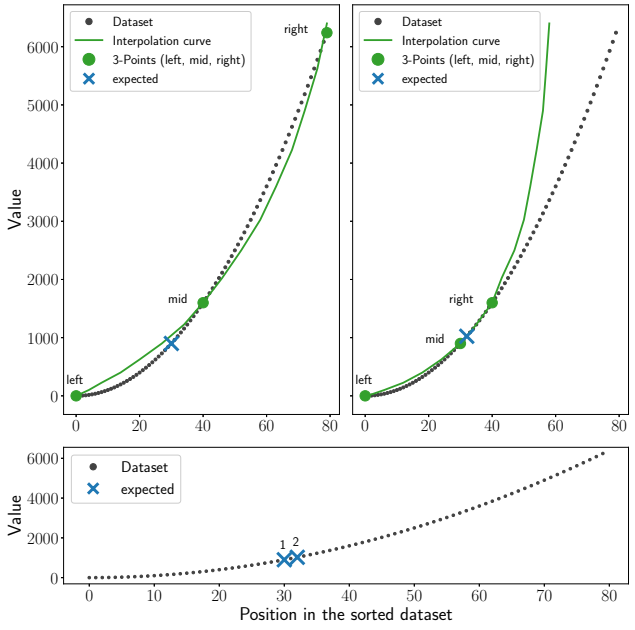

Figure 15: When the 3-Point interpolation method searches the x^2 distribution for $y^* = 32^2$ it will use the (solid line) curves presented in this figure to fit the data. Each subplot presents the curve fitted during each step of the search. The last plot present the progression of the expected positions until the target is found.

Figure 15 presents the steps performed by the 3-Point interpolation method during a search and the progression of the *expected* positions. The two plots at the top show the curves that the 3-Point interpolation method will fit to the x^2 distribution when we search for $y^* = 32^2$. The (solid line) curves are used to calculate the *expected* positions. We can observe that the curve fitted during the first interpolation matches the general distribution of the data but there are some areas for which the curve slightly diverges from the dataset, i.e. from position 0 to 35. The first interpolation guides the search near the target position. The second interpolation uses 3 points located much closer to the target position to fit a curve, which better matches the part of the dataset that contain the target value, and it locates the target value. For the same dataset and target value, the linear interpolation method used by Interpolation Search required 6 iterations to locate the target value, (Figure 4), while the 3-Point Interpolation method required 2 iterations (Figure 15 bottom, points 1, 2).

A.2 Optimized Binary Search

Algorithm 4 Optimized Binary Search

Input: $V, y^*, num\_iterations$ ▷ V: sorted array of size n
Output: $position\ of\ value\ y^*$ ▷ y^*: the target value

1: $left \leftarrow 0$
2: $t \leftarrow n$
3: **for** $i \leftarrow 0; i < num\_iterations; i{+}{+}$ **do**
4: $\quad mid \leftarrow \lfloor \frac{t}{2} \rfloor$
5: $\quad$ **if** $V[mid] \leq y^*$ **then**
6: $\quad\quad left \leftarrow mid$
7: $\quad n \leftarrow \lceil \frac{t}{2} \rceil$
8: **return** $sequential\_search(V, y^*, left + \lfloor \frac{t}{2} \rfloor)$

Algorithm 4, is the optimized implementation of Binary Search. It maintains the search interval implicitly as:

$$[left,\ left + \frac{t}{2}]$$

instead of explicitly tracking it, as *[left, right]* like in [23]. This saves one conditional evaluation because, instead of choosing which endpoint to update at every iteration, we update only $left$.

Guard in Binary Search: We terminate Binary Search with a Sequential Search, like in SIP and TIP. Instead of dynamically choosing when to switch to Sequential Search, we calculate the number of Binary Search iterations, $num\_iterations$, to be performed before switching to Sequential Search. This is possible because the size of the search interval after each iteration is known. The search interval is reduced by half in each iteration. We empirically calculate, similarly to Section 4.1.1, the size of the search interval where it is beneficial to switch to Sequential Search, $basecase\_size$. We precompute the number of iterations to be performed by Binary Search using $basecase\_size$ as:

$$num\_iterations = \lceil \log N - \log basecase\_size \rceil \quad (9)$$

where N is the number of records.

We also experimented with a prologue before the loop which reduced the size of the interval to a power of 2, saving the ceiling function in the core loop (line: 7). We found this degraded performance in all but the smallest dataset sizes.

A.3 SIP Optimizations

This section evaluates in detail the effect of the optimizations applied to SIP, Section 4.1. The *UaR* dataset and a record size of 8B are used for the evaluation.

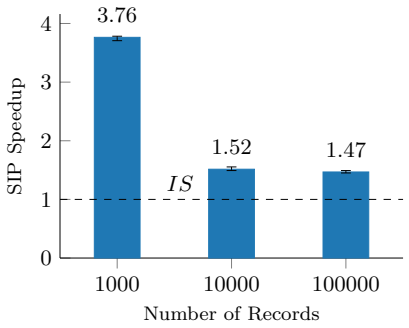

Figure 16: Speedup of SIP vs Interpolation Search (IS), Algorithm 1, on *UaR*.

Figure 16 presents the speedup of SIP against Interpolation Search for the *UaR* dataset for different dataset sizes, while Figure 2 presents the speedup achieved by SIP against Interpolation Search and Binary Search for the *fb_ids* dataset.

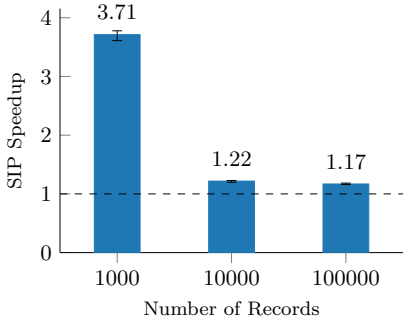

Figure 17: Evaluation of the effect of Guard on SIP. Presented is the speedup achieved by a version of SIP with Guard compared to a version of SIP without Guard.

Figure 17 present the improvement of SIP with the Guard condition by comparing it to a version of SIP without it. To set the *guard_size*, we measure the performance of SIP for various datasets and record sizes, like in Figure 18. Record sizes increase the cost of memory accesses because fewer records can fit in the processor's cache and in each cache line. This impacts the performance of the algorithm. We choose a guard size of 8 for SIP because it was the best all-around performer across the range of our experiments.

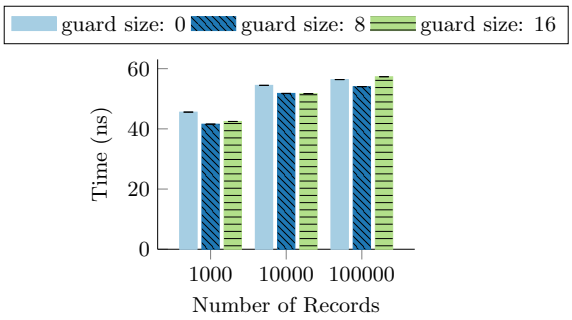

Figure 18: Performance comparison with different guard widths, for different record sizes.

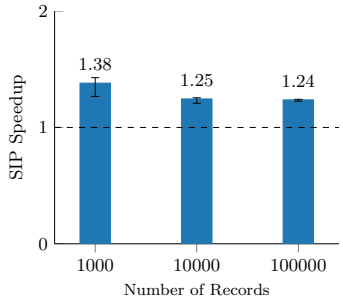

Figure 19: Speedup of SIP when slope is reused, on *UaR*.

Figure 19 present the improvement of SIP when the slope is reused compared to a variant of SIP which does not reuse the slope.

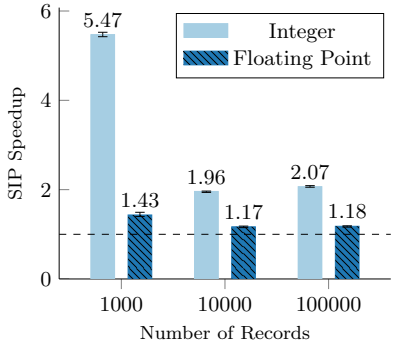

Figure 20: Speedup of SIP when fixed-point arithmetic is used compared to floating-point and integer arithmetic, on *UaR*.

Figure 20 compares the speedup in overall execution time of SIP when using fixed-point arithmetic to floating-point and integer arithmetic.

A.4 *fal* and *cfal* datasets visualized.

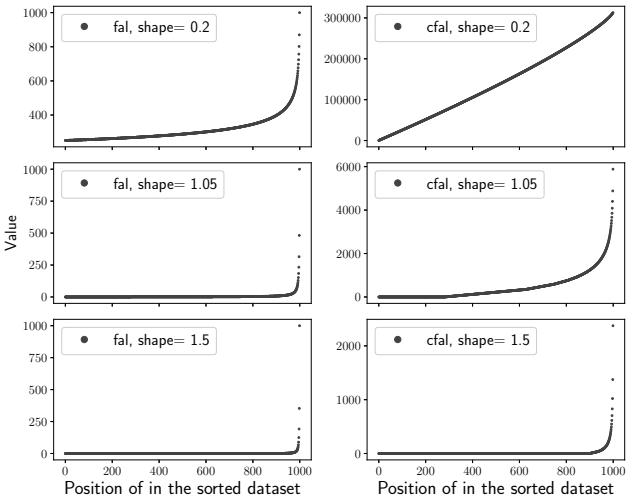

Figure 21: *fal* (left column) and *cfal* (right column) distributions. Each row presents datasets with a different shape: 0.2, 1.05, 1.5

A.5 Parallel Performance

SIP and TIP can be parallelized by searching for multiple records concurrently, as discussed in Section 4.3. To assess the impact of concurrent searches, we search for each record in a single dataset and vary the number of threads. Figure 22, presents the time to search for one record using SIP, using the *UaR* dataset (Section 5.1.3) with 100 million records and record size 8B. The time is the average of all the searches across all threads. We vary the number of threads from 1 to 32. We omit the corresponding experiment for TIP as the results are similar.

While the thread count increases to 16, SIP's performance is not affected. Almost all memory accesses are contained within the cache. When more than 16 threads are used, the time to search for one record increases. We believe that storing all the profiling information causes the performance degradation, up to a certain point. As we increase the thread count and the size of the dataset, we collect more data that has to be moved to memory. Smaller datasets, that produce less profiling information, present a smaller or no performance degradation after 16 threads. For all datasets and records sizes, the performance of SIP and TIP is affected significantly less than Binary Search when we use more threads. The memory access pattern of SIP and TIP likely help them perform better.

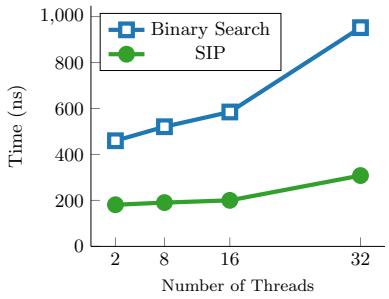

Figure 22: Performance of SIP and Binary Search across different thread counts.

A.6 Sampling Meta-Algorithm

The Meta-Algorithm, Algorithm 5, picks the fastest search method between Binary Search, SIP and TIP for a given dataset. The Meta-Algorithm is executed when a dataset is created and potentially in the case where the dataset's distribution changes after a significant number of updates, deletions or inserts. The algorithm's usage is described in Section 5.8.

Algorithm 5 Meta-Algorithm

Input: V, *noSamples* ▷ V: sorted array
Output: *fastestAlgorithm*

```
randVals ← noSamples random values from V
\\ Sample Binary Search
execTime ← 0
for y in randVals do
    execTime += time_d(BinarySearch(V, y), ∞)
algorithmPerformance.add("BS", execTime/noSamples)
\\ Sample SIP, TIP
deadline ← algorithmPerformance.get("BS")
for algo in SIP, TIP do
    execTime ← 0
    for y in randValues do
        exec_time +=
            time_d(algo(V,y),deadline)
    algorithmPerformance.add(algo, execTime/noSamples)
fastestAlgorithm ← min(algorithmPerformance)
return fastestAlgorithm
```

Function *time_d(func, deadline)* executes the function *func* and returns its execution time, if the execution time exceeds the *deadline func* is stopped and ∞ is returned.