Evaluating Sorting Algorithms with Varying Data Sortedness

I-Ju Lin
Boston University
Boston, Massachusetts, USA
liniju@bu.edu

Wei-Tse Kao
Boston University
Boston, Massachusetts, USA
kaowt@bu.edu

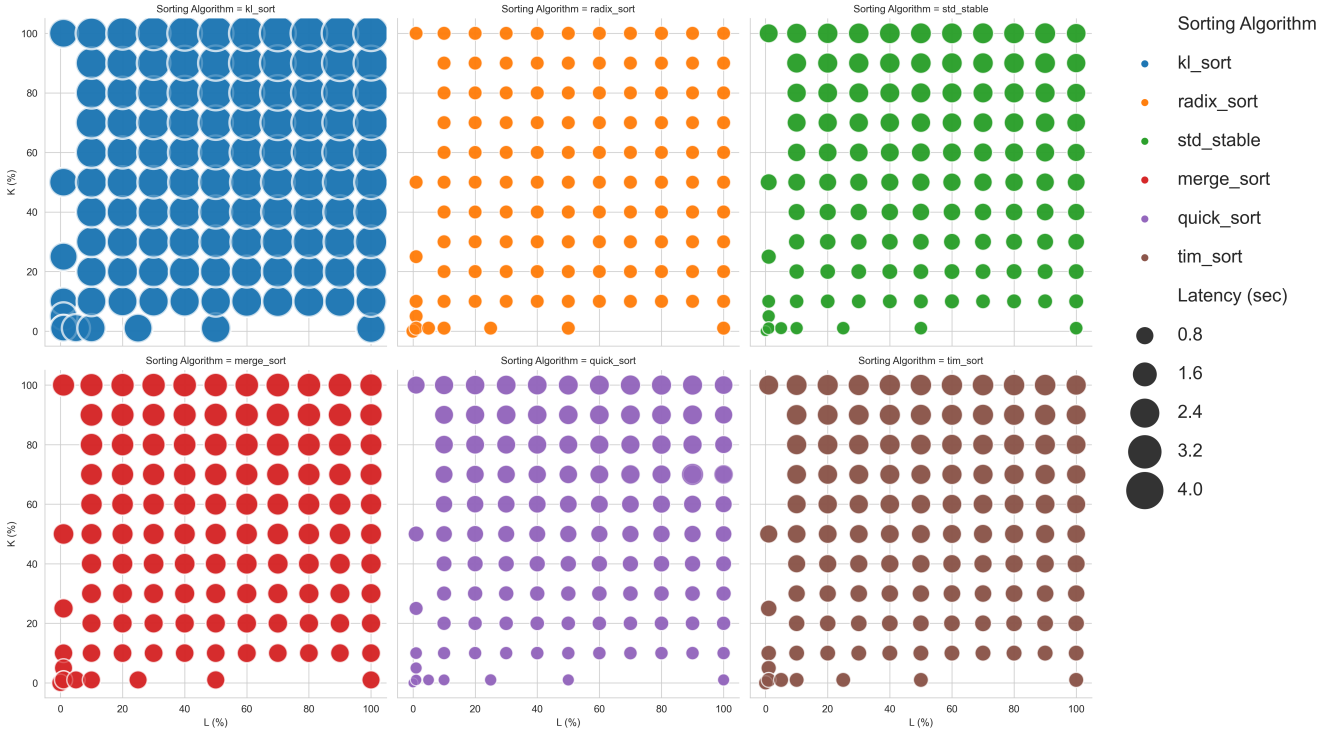

Figure 1: TODO: Caption.

ABSTRACT

TODO: Abstract here!

KEYWORDS

data system structure, index, nearly sorted, sortedness

1 INTRODUCTION

1.1 Motivation

Sorting the data can bring several benefits, such as faster reads, better performance in index-designed structures, and more accessible data analysis in data system structures. Therefore, evaluating sorting algorithms with varying data sortedness is an important topic. It helps developers understand these implementations and provides insights under different application conditions. This research can further inform decisions on which algorithm to use for data system designs and optimizations in various contexts.

1.2 Problem Statement

This project will evaluate the performance of multiple adaptive sorting algorithms using the BoDS benchmark workload generator. Specifically, we will analyze the performance of Tim Sort, (K, L) sorting algorithm[3], Quick Sort, Merge Sort, Insertion Sort, and std::stable_sort on the data generated by the workload generator. In addition, we will consider the inclusion of additional sorting algorithms to broaden the scope of our analysis. We aim to comprehensively compare these algorithms under various workloads and identify their strengths and weaknesses, which will contribute to developing more efficient sorting algorithms.

1.3 Contributions

This project aims to analyze the performance of various sorting algorithms, including merge sort, quick sort, std::stable_sort, and other classic ones, under different levels of (K, L) sortedness in the input data stream. This will involve studying three composed sorting algorithms requiring metadata, including Tim sort, Radix sort, and (K, L)-adaptive sorting algorithm. Furthermore, this can be the first paper proposing L approximating measurement methods.

We developed a C++ experimental system[1] that executes a selected algorithm with a given data stream. The performance of these algorithms is analyzed comprehensively by varying the sortedness. The project also aims to explore the possibility of an adaptive (K, L) Sort that estimates the sortedness automatically. The goal is generally to better understand the sortedness research problem through analysis and benchmarking.

2 BACKGROUND

Raman's work highlights that while indexes in data systems can improve data access, they are typically designed for either fully sorted or unsorted data and may not consider intermediate degrees of sortedness. For example, adaptive sorting algorithms help optimize performance when dealing with pre-ordered data. Among them, Insertion Sort is a well-known algorithm that can achieve $O(n)$ time complexity in the best-case scenario when the data collection/stream is fully sorted. The authors found that PostgreSQL cannot exploit data sortedness and identified opportunities for improving performance through sortedness-aware index designs.

The paper introduces the BoDS [5] benchmark, which measures the indexing performance of a data system as the degree of sortedness in the input data varies. To capture the level of data sortedness, various metrics have been suggested. One common approach to measuring the extent of sortedness is by assessing the number of elements out of place and the degree to which they are misplaced. The concept of (K, L)-sortedness metric is based on this notion and it characterizes nearly-sorted relations using two parameters, namely K and L. The parameter L denotes the level of roughness in the sorting process, where a higher value of L signifies that tuples can be more displaced from their original position before they are regarded as out of place. On the other hand, the parameter K indicates the number of tuples that are entirely out of order, i.e., the number of tuples that must be overlooked before the remaining relation complies with the L parameter.

The authors run the benchmark on PostgreSQL, a popular relational database system, and find that it must effectively exploit sortedness in the input data. However, the results also demonstrate the potential for improvement and provide a framework for experimenting with sortedness-aware index designs. Overall, the paper addresses an important issue in data systems and provides a valuable tool for benchmarking and improving indexing performance in the presence of intermediate degrees of sortedness in the input data.

To improve this research, exploring different sorting algorithms and comparing their performance when varying the sortedness in the input data stream could provide more comprehensive insights and help identify which algorithms are best suited for different levels of sortedness. This can also lead to the development of new hybrid sorting algorithms specifically designed to handle real-world data with varying degrees of sortedness.

2.1 Data Sortedness

Sortedness refers to the degree to which data is ordered. To qualify the degree of sortedness, we use (K, L) sortedness metric [2] in our experiments.

[1]https://github.com/lin826/Eval-Sort

2.1.1 (K, L) Definition. In (K, L) sortedness metric, there are two parameters in this metric: K is the number of the elements that are out of place, and L is the maximum positional displacement of the out-of-order elements.

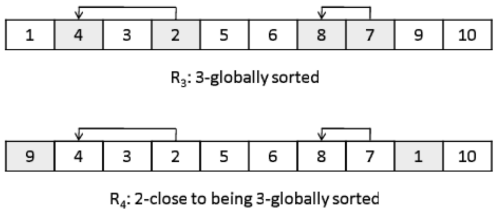

Figure 2: In the R_3 example, the maximum distance between the locations of any two unsorted tuples is always smaller than 3 in R_3, so it is 3-globally sorted. In the R_4 example, if we neglect the two out-of-ordered indexes, 9 and 1, the rest of the workload is 3-globally sorted as R_3. Thus, we call R_4 as 2-close to 3-globally sorted.

2.1.2 (K, L) Equivalence. K-close to being sorted means he size of unordered indices set is smaller or equals to K. L-globally sorted means the distance between the locations of any two unsorted tuples is always smaller than L.

2.2 Sortedness Workload

The crucial aspect of the proposed benchmark is the creation of data collections with varying degrees of sortedness. To achieve this, the BoDS data generator[2] has been developed, a synthetic data generator that produces data collections conforming to specific values of the (K, L)-sortedness metric. The data generator requires user-defined inputs for the K and L parameters of the sortedness metric, which are expressed as a percentage fraction of the total number of entries N, in addition to the index beta distribution maps by α and β. We chose $\alpha = \beta = 1$ with the simplest view to making index workload in a uniform distribution. The total size of the generated data collection, in summary, can be controlled by specifying the number of entries N and the payload size P, which is a randomly generated fixed length string using the random seed S.

3 SORTING ALGORITHMS

In the following sections, we assume an unsorted index table is uniquely ranging from the integer 1 to n in a total length of n. To inference the memory footprint, we suppose all index are stored in the data type of 32-bit integer array in C++ under an 64-bit operating system. That is, any pointer, recording a target memory address, takes 64 bits.

3.1 Classic Sorting Algorithms

3.1.1 Insertion Sort. is a simple sorting algorithm that builds a sorted array one element by one. It is an in-place sorting algorithm comparing each elements to the rest in each iteration. Thus, it takes

[2]https://github.com/BU-DiSC/bods

$n \times (n-1)$ operations in $O(n^2)$ in the worst case. Interestingly, because the inner loop stops whenever the rest comparisons can be skipped, it only takes $n = O(n)$ operations in the best case.

Regardless of the input sortedness, this sorting algorithm takes one temporary integer 4 bytes and one integer pointer in the memory 8 bytes. As a result, it takes 12 bytes, and the time complexity is $O(1)$.

3.1.2 Merge Sort. is a divide-and-conquer sorting algorithm. It works by recursively dividing an unsorted array into two halves, then merging the two sorted halves back into a single sorted array in a bottom-up manner. Accordingly, it is stable to the original order even with duplicate elements.

To calculate the time complexity, we need to understand the algorithm in details. First, it spends $\log_2 n$ steps for dividing in $size \in \{2, 2^2, 2^3, \ldots, n\}$. In each step, it further spends $\frac{n}{2size}$ iterations for each pair of subarrays and conquer them with two additional buffer assignments and a picking process up to $2size$ operations for taking the smallest elements back into the array. That is, a total operations of $\frac{n}{2size} \times (2size + 2size) = 2n$ in each step. Thus, the time complexity is $\log_2 n \times (2n) = 2(n \log_2 n)$ in $O(n \log_2 n)$.

The standard merge sort is not an in-place sorting algorithm. It requires buffers to store locally sorted subarrays during the merging process. The memory required for auxiliary data structures should be up to n times the size of integer data type: $(n \times 32)$ bits = $4n$ bytes. Additionally, for storing two sizes of subarrays: (2×32) bits, two integer pointers to the first considering elements (2×64) bits, and one integer pointer to the target position (1×64) bits, each memory takes another $(2 \times 32) + (2 \times 64) + (1 \times 64) = 256$ bits, which is 32 bytes. As a result, the memory footprint for the implementation takes $(4n + 32)$ bytes in an auxiliary space complexity $O(n)$.

3.1.3 Quick Sort. is a sorting algorithm that recursively selects one pivot element from the array, partitioning the rest into one smaller-element and one greater-element subarray and then sorting these two subarrays, respectively. Both the time and space complexity are ranges depending on the sortedness.

In the worst case, all elements fall into the same bin, and pivots are min and max in turns; each value compares and swaps in each iteration. The resulting time complexity would be $n^2 = O(n^2)$ without the leading coefficient 1. Occupying n integer locations to store pivots and one integer space to do a swap, the memory footprint is up to $(4n+4)$ bytes. On the contrary, the best case splits elements evenly into two subarrays in each iteration, the number of execution operations reduced to $n \log n = O(n \log n)$, and the needed memory is $(4 \log n + 4)$ bytes.

3.1.4 Radix Sort. is a non-comparative sorting algorithm that sorts elements by grouping them according to the individual or other significant digits that make up the keys or values. It is commonly used for sorting integers or other data types that can be represented as a sequence of digits. Notably, this sorting algorithm needs a predefined base b for dividing digits into buckets.

Because b is the maximum number of elements each digit can have, this method takes $\log_b n$ times the complexity of subroutines, which is usually the widely adopted counting sort implementation[3]. First, an integer array *count* with the size of b, initialized with zeros, counts the number of elements by one pass n. Then, the counts will sum up by b sum operations. The following loop directs each element to a buffer output array in n operations. The final loop dumps the buffer back into the array. As a result, the local time complexity is $n+b+2n = 3n+b$ with $4 \times b$ bytes for count[b] and $4 \times n$ bytes for the buffer. In the decimal numeral system, b would be 10 and thus the time complexity is $(3n+b) \log_b n = (3n+10) \log_{10} n = O(n \log_{10} n)$; the memory footprint is $(4 \times n) + (4 \times b) = 4n + 40$ bytes in the space complexity $O(n)$.

3.2 Composed Sorting Algorithms

3.2.1 std::stable_sort. is a function provided by the C++ Standard Library[4] that sorts a given range of elements in ascending order according to the specified comparison function and has the additional guarantee that the relative order of elements with equal values will remain unchanged after the sorting. *std::stable_sort* is regarded as an improved version of merge sort and thus has similar time and space complexity.

3.2.2 TimSort [4]. is a stable sorting algorithm hybrid of Insertion Sort and Merge Sort, designed to perform well on various real-world data and has become the default sorting algorithm in many programming languages since 2002.

The algorithm divides the unsorted array into subarrays with a predefined size of s, then sorts each subarray using Insertion Sort in $O(s^2)$ operations. The sorted subarrays are then merged with the conquer techniques in Merge Sort with the size $size \in \{s, 2s, 2^2 s, \ldots, n\}$ in $\log_2 (\frac{n}{s})$ iterations. Thus the total time complexity is $\log_2 (\frac{n}{s}) \times (2n) = 2n(\log_2 n - \log_2 s) = O(n \log_2 n)$, which is the same as the analysis in [1]. In our implementation of $s = 32$, the exact execution time would be $2n(\log_2 n - 5)$ operations.

3.3 Sortedness-oriented Sorting Algorithms

3.3.1 (K, L) Sort [2]. is the algorithm that leverages the (K, L)-metric to sort a nearly sorted collection. To avoid that dynamic memory allocation slowing the sorting, we implement with fixed memory footprint to its upper bound. It takes heaps S, G and arrays TMP, OUT, all as large as the input: $4 \times 4 \times n = 16n$ bytes.

This algorithm requires given K and L parameters without any well-known estimation algorithm for L. This sorting method depends on sortedness and has time complexity in a range. In the best case, elements are stored neither in the heap S nor G. They all append into the TMP array in the first pass and dump into the OUT array in the second pass. So a total $2n = O(n)$ operations.

However, if half of the elements are stored in a heap S before the first pass, it might be the worst case in a time analysis where inserting takes $\log_2 n$ for each. Inserting all the other half into the heap G, the sorting takes another $\frac{n}{2} \times \log_2 n = \frac{1}{2}(n \log_2 n)$ operations for heap insertions. Between two passes, the dump of heap S into TMP array takes another $\frac{n}{2}$ operations. In the second

[3] https://www.geeksforgeeks.org/radix-sort
[4] https://cplusplus.com/reference/algorithm/stable_sort

loop, the smallest element in the heap G might always dump to OUT array and consume one element from TMP into the heap G. In summary, the number of operations is $2\times\frac{1}{2}(n\log_2 n)+\frac{n}{2}+n\log n = 2(n\log_2 n)+\frac{1}{2}n = O(n\log_2 n)$.

3.4 Design Solution: Adaptive K-L Sort

Without meta-data provided by data system developers, we propose a design solution to automatically detect sortedness. Unlike predefined parameters in *Tim Sort* or *Radix Sort*. Our method is suitable to be adopted in online warehouse without reconfiguration as time pass.

3.4.1 Parameter Search. With rudimentary experiments, we found sortedness K plays the key role on the sorting performance. Reducing the estimated K significantly decreases latency. We decide to spends some effort on optimizing K estimation.

TODO:

K. is... We can easily adjust the sample size to make this search into an approximate approach for the parameter estimation.

L. is... We propose stream scan on the minimal possible of L. Assume the index array are a sequential range from 1 to n, ...

3.4.2 Estimation A.

3.4.3 Estimation B.

3.4.4 Estimation C.

3.5 Complexity Analysis

Through our rudimentary results in Table 1, we found the Big-O notation captures the characteristics close to infinite array size, but not for workloads in reality. Here we plot in Figure 3 the time complexity[5] with exact coefficients.

4 EVALUATION EXPERIMENTS

4.1 Evaluation System Architecture

Our system implementation includes C++ libraries: *algorithm*, *chrono*, *climits*, *cstdlib*, *fstream*, *iostream*, and *string*. The system accepts these arguments:

- ***Input workload path***
- ***Output CSV path***
- ***Sorting algorithm type***
- ***[optional] K divisor***
- ***[optional] L divisor***
- ***[optional] KL metadata estimation type***

4.2 Computing Device

Our experiments are conducted in Windows Subsystem for Linux. The hardware device features an Intel® Core™ i9-11900 CPU with eight cores with a base clock speed of 2.5 GHz and 24MB of CPU cache. In terms of memory, this computer has two 16GB DDR4 RAM.

[5]https://www.desmos.com/calculator/dctk3bzuds

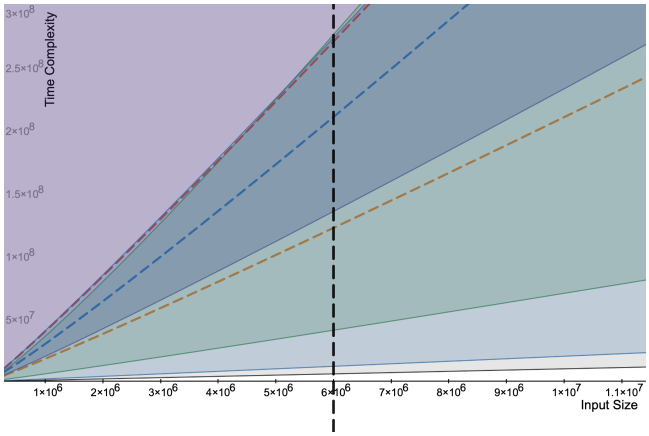

Figure 3: The dotted black line in the middle indicates the the maximum size of integer array n able to stored in our CPU cache without using RAM. Blue shadow area is the range of (K, L) Sort. Orange dotted line is Radix Sort. Green shadow area is the range of std::stable_sort. Red dotted line is Merge Sort. Purple shadow area is the range of Quick Sort. Blue dotted line is Tim Sort. Grey shadow area covers almost the whole figure is the range of Insertion Sort.

4.3 Testing Workload Dataset

To understand the ten million(10M) testing dataset for our comprehensive analysis, we plot workloads in scatter plots like Figure 4. Other than basic analysis of 1% on each parameter, we generated workloads by BoDS with K and L ranging from 0% up to 100% with 10% increment. The testing dataset[6] has approximately 21GB in total. Becasue of the misplacement definition of L is the upper bound rather than an exact value, the workload $K = 100\%$, $L = 100\%$ is not completely scrabled. Thus, in the following experiments, we still add the data from *std::iota* shuffles into considerations.

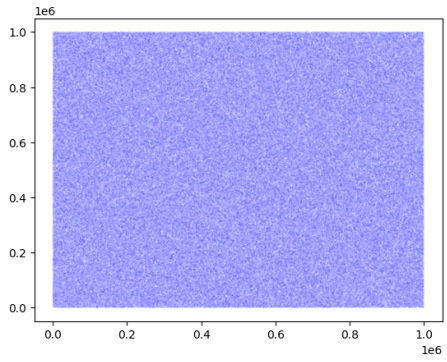

Figure 4: Use *C++*'s *std::iota* to generate the scrambled workload. X-axis indicates the position of entry in data, while Y-axis refers to the expected positions of that entry, which is the index value of that entry in our workloads.

[6]https://drive.google.com/drive/folders/1LjUzl99HMHKuHouqRdDr55fepV_9h4SN

	Time of Best Case	Time of Worst Case	Memory	Stable	1M Sorted Workload Latency (seconds)	1M Unsorted Workload Latency (seconds)
Radix Sort	$3n \log_{10}(n)$ = O(n)		O(n)	Yes	0.052707988	0.061692676
Quick Sort	$n \log(n)$ = O($n \log(n)$)	n^2 = O(n^2)	O(log(n))	**No**	0.025146049	0.104232642
std::stable_sort	O($n \log(n)$)	O($n \log^2(n)$)	O(n)	Yes	0.031312245	0.113842827
Tim Sort	$2n \log(n)$ = O(n log(n))		O(n)	Yes	0.047834768	0.120440994
Merge Sort	$2n \log(n)$ = O(n log(n))		O(n)	Yes	0.083934642	0.156695128
KL Sort	$2n$ = O(n)	$2n \log(n)$ = O(n log(n))	O(n)	Yes	0.000005962	0.287924236
Insertion Sort	n = O(n)	n^2 = O(n^2)	**O(1)**	Yes	0.003643681	368.8519107

Table 1: Summaries of sorting algorithms with rudimentary experiment results.

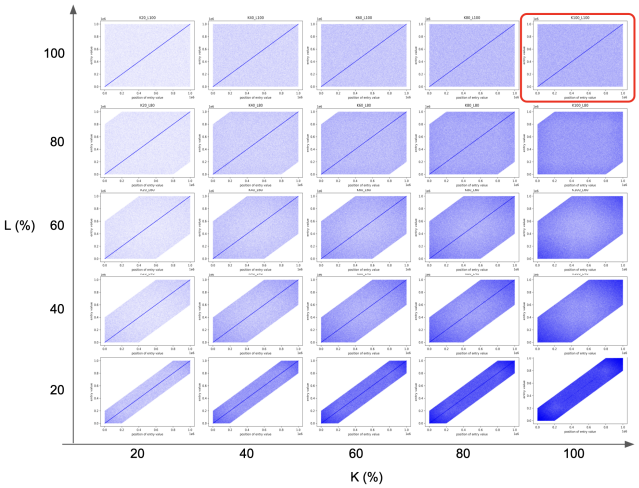

Figure 5: Workloads generated by BoDS with K and L ranging from 20% up to 100% with every 20% increment. In the red round square, the setup of K = 100% and L = 100% does not guarantee a completely scrambled dataset.

5 RESULTS

With three trials on each setup, all our experiment results include a shadow area of one standard deviation. All figures are plot by Python's Matploblib-based Seaborn library with Pandas CSV reader.

5.1 Rudimentary Experiment

Given an completly sorted index array in the size of one million(1M) and another completely unsorted one shuffled by *std::iota* shown in Figure 4, we measured the latency with *chrono::high_resolution_clock* in nanoseconds and showed the result in Table 1.

Because of hundreds of times of latency difference between Insertion Sort and others, our results focus on comparisons between the other six sorting algorithms: (K, L) Sort, Radix Sort, std::std_stable, Merge Sort, Quick Sort, and Tim Sort.

5.2 Observation on KL estimation

In order to show the benefits of (K, L) Sort, we design experiments to observe hyper parameter adjustments on the (K, L) values from the BoDS generator first.

5.2.1 *Underestimated K.* TODO: Explain Figure 6.

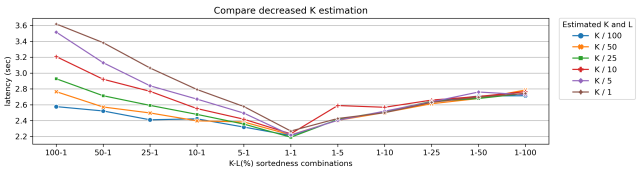

Figure 6: TODO: Caption.

5.2.2 *Underestimated L.* TODO: Explain Figure 7.

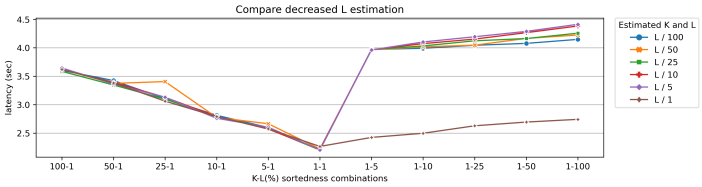

Figure 7: TODO: Caption.

5.2.3 *Underestimated K and L.* TODO: Explain Figure 8.

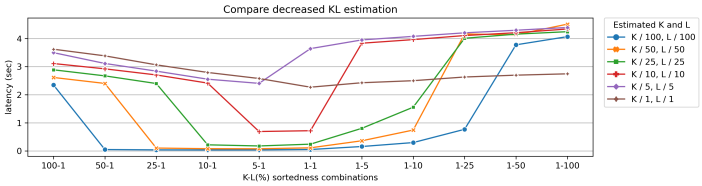

Figure 8: TODO: Caption.

5.3 Latency of Various Sortedness Workloads

We find that as K increased the latency of all sorting algorithm are increasing; however, the latency doesn't change obviously as L increased in all sorting algorithm in our experiment, so we should focus more on K than L. For K<1%, (K, L) sort might be a good way for sorting, but we still need to test the boundary that (K, L) sort will

outshine other sorting algorithms. For 25%>K>1%, quick sort has the best performance in our experiment. However, the quick sort can only apply on the workload with non-duplicate workload because it is not a stable sorting algorithm, so we might use std::stable_sort at this case. For K>25%, radix sort has the best performance in our experiment.

TODO: Explain the line chart Figure 9.

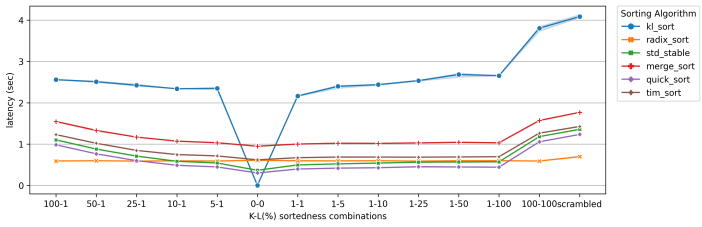

Figure 9: TODO: Caption.

5.4 Comprehensive Sortedness Simulation

TODO: Explain the bubble chart Figure 1.

6 CONCLUSION

TODO:

REFERENCES

[1] Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. 2019. On the Worst-Case Complexity of TimSort. arXiv:1805.08612 [cs.DS]

[2] Sagi Ben-Moshe, Yaron Kanza, Eldar Fischer, Arie Matsliah, Mani Fischer, and Carl Staelin. 2011. Detecting and Exploiting Near-Sortedness for Efficient Relational Query Evaluation. In *Proceedings of the 14th International Conference on Database Theory* (Uppsala, Sweden) *(ICDT '11)*. Association for Computing Machinery, New York, NY, USA, 256–267. https://doi.org/10.1145/1938551.1938584

[3] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426. http://dl.acm.org/citation.cfm?id=362686.362692

[4] Tim Peters. 2015. Timsort description. https://github.com/python/cpython/blob/main/Objects/listsort.txt

[5] Aneesh Raman, Konstantinos Karatsenidis, Subhadeep Sarkar, Matthaios Olma, and Manos Athanassoulis. 2023. BoDS: A Benchmark on Data Sortedness. In *Performance Evaluation and Benchmarking: 14th TPC Technology Conference, TPCTC 2022, Sydney, NSW, Australia, September 5, 2022, Revised Selected Papers* (Sydney, NSW, Australia). Springer-Verlag, Berlin, Heidelberg, 17–32. https://doi.org/10.1007/978-3-031-29576-8_2