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

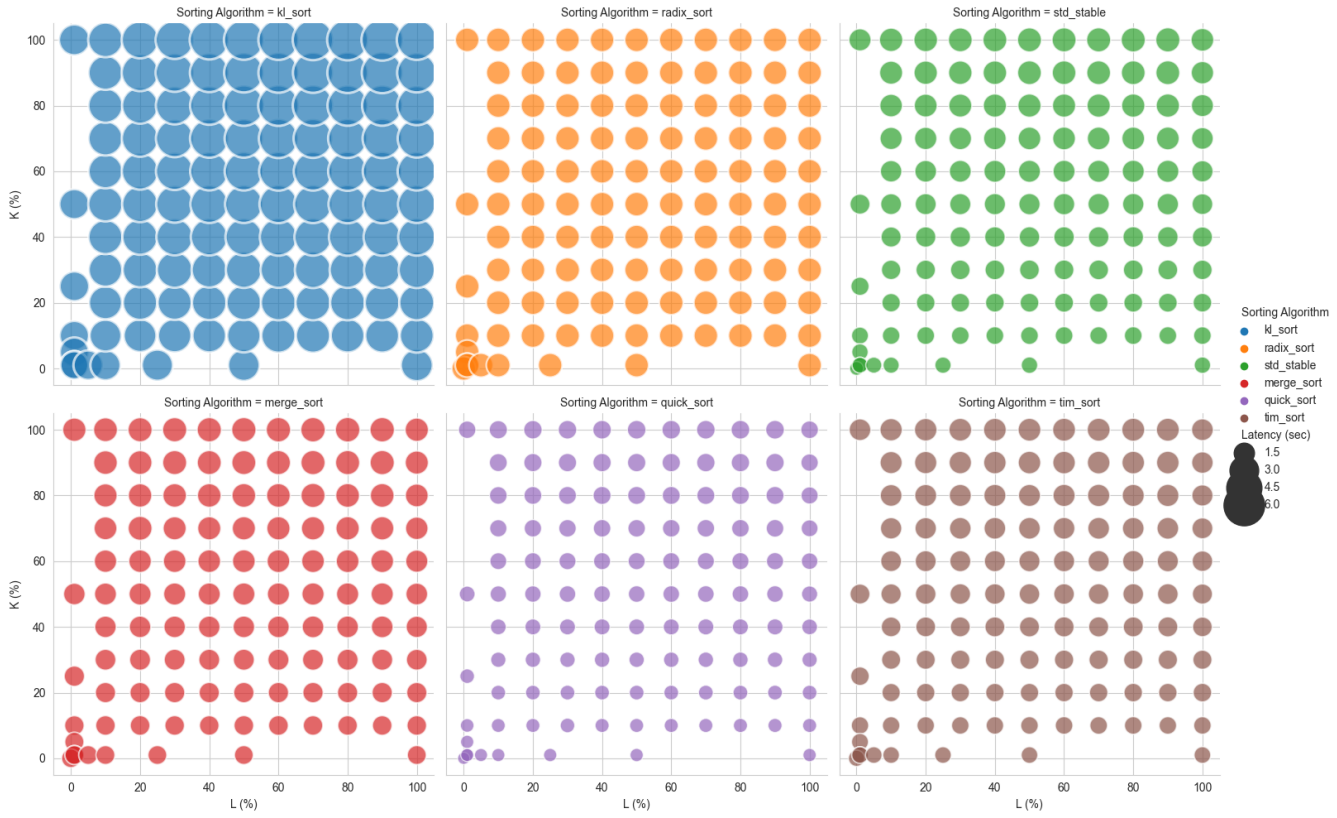

Figure 1: Caption.

ABSTRACT

Abstract here!

KEYWORDS

data system structure, index, nearly sorted, sortedness

1 INTRODUCTION

1.1 Motivation

Evaluating sorting algorithms with varying data sortedness is important because it allows us to understand the performance characteristics of these algorithms, and provide insights into the strengths and weaknesses of different algorithms under different conditions. This information can be used to make informed decisions about which algorithm to use for a particular dataset and optimize sorting operations' performance in different 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 TimSort, KL-adaptive sorting algorithm[1], Quicksort, MergeSort, 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 under different levels of sortedness in the input data stream. This will involve studying multiple adaptive sorting algorithms, including TimSort, (K, L)-adaptive sorting algorithm, Quicksort,

MergeSort, Insertion Sort, and std::stable_sort, and possibly adding a few more algorithms.

The next step will be implementing the API [1], we developed, that executes a selected algorithm for a given data stream. The performance of these algorithms will then be analyzed comprehensively by varying the sortedness of the input stream. The project also aims to explore the possibility of a new hybrid sorting algorithm that combines the radix sort and the KL-sorting algorithm. 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 take into account 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[2] 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 measure the extent of sortedness is by assessing the number of elements that are 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 crucial aspect of the proposed benchmark is the creation of data collections with varying degrees of sortedness. To achieve this, the BoDS data generator has been developed, which is 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 fraction of the total number of entries (N), in addition to the displacement distribution (on L), and the parameter to regulate the distribution maps (α, β). It is worth noting that a beta distribution maps to uniform when $\alpha = \beta = 1$. The size of the generated data collection can be controlled by specifying the number of entries to be produced (N) and the payload size (P), which is a randomly generated string with a given length.

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.

3 ARCHITECTURE

3.1 Sortedness Workload

Using BoDS, a Benchmark on Data Sortedness[2], we can generate corresponding workloads to evaluate. BoDS leverages the (K,L)-adaptive sorting algorithm and provides parameters for entries to generate, K, L, α, β, random seed, and payload size in bytes.

3.2 Sorting Algorithms

3.2.1 *Traditional sortings.*

std::stable_sort. $std :: stable\_sort$ is a function provided by the C++ Standard Library that sorts a given range of elements in ascending order according to the specified comparison function and it has the additional guarantee that the relative order of elements with equal values will remain unchanged after the sorting. std::stable_sort has a time complexity of O(n log n) on average.

Insertion Sort. Insertion Sort is a simple sorting algorithm that works by building a sorted array (or list) one element at a time. It is an in-place sorting algorithm and it has a time complexity of $O(n^2)$ in the worst case, where n is the number of elements in the array. It is efficient for small arrays or partially sorted arrays.

Merge Sort. Merge Sort is a divide-and-conquer sorting algorithm that works by dividing an unsorted array into two halves, sorting each half recursively, and then merging the two sorted halves into a single sorted array. It has a time complexity of $O(nlogn)$, where n is the number of elements in the array. However, merge sort is not an in-place sorting algorithm and it requires additional memory to store the subarrays during the sorting process.

Quick Sort. is a sorting algorithm that works by selecting a pivot element from the array, partitioning the other elements into two subarrays (one with elements smaller than the pivot and one with elements greater than the pivot), and then recursively sorting these subarrays. It has a time complexity of $O(nlogn)$ on average, where n is the number of elements in the array. However, in the worst case (when the pivot is selected poorly and results in unbalanced partitions), the time complexity can be $O(n^2)$.

Radix Sort. is a non-comparative sorting algorithm that sorts elements by grouping them according to the individual digits or other significant digits that make up the keys or values. It is commonly

[1] https://app.swaggerhub.com/apis/lin826/CS561-API/0.0.1

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

used for sorting integers or other data types that can be represented as a sequence of digits. It has a time complexity of $O(kn)$, where n is the number of elements in the array and k is the number of digits in the maximum element and it is also a stable sorting algorithm.

3.2.2 TimSort. is a sorting algorithm that is a hybrid of Insertion Sort and Merge Sort, and it is was designed to perform well on a wide variety of real-world data and is the default sorting algorithm in many programming languages, including Python and Java.

The algorithm works by first dividing the unsorted array into small subarrays, then sorting each subarray using Insertion Sort. The sorted subarrays are then merged using Merge Sort, with a key optimization that takes advantage of any pre-existing order in the data.

Tim Sort has a time complexity of $O(nlogn)$ in the worst case, but its performance on real-world datasets is often much better than that of other sorting algorithms with the same worst-case complexity, and it is also a stable sorting algorithm.

(K, L) sorting. The (K,L)-Sorting Algorithm is an algorithm that leverages the (K,L)-metric to sort a (K,L)-near sorted collection in a maximum of two sequential passes. However, this algorithm requires given K and L without any well-known estimation algorithm. This algorithm has a computational complexity of $O(Nlog(K+L))$ (assuming N entries) and requires a memory size of O(K + L). Using the adaptiveness of the (K,L)-sorting algorithm, we can efficiently organize data in the buffer.

3.3 Design Solution: Adaptive K-L Sorting

3.3.1 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.

First, we propose binary search on the minimal possible solution of K.

4 RESULTS

Describe hardware spec and the BoDS parameter setups.

Architecture: x86_64 CPU op-mode(s): 32-bit, 64-bit Byte Order: Little Endian Address sizes: 46 bits physical, 48 bits virtual CPU(s): 20 On-line CPU(s) list: 0-19 Thread(s) per core: 2 Core(s) per socket: 10 Socket(s): 1 NUMA node(s): 1 Vendor ID: GenuineIntel CPU family: 6 Model: 85 Model name: Intel(R) Core(TM) i9-10900X CPU @ 3.70GHz Stepping: 7 CPU MHz: 1200.074 CPU max MHz: 4700.0000 CPU min MHz: 1200.0000 BogoMIPS: 7399.70 Virtualization: VT-x L1d cache: 320 KiB L1i cache: 320 KiB L2 cache: 10 MiB L3 cache: 19.3 MiB NUMA node0 CPU(s): 0-19

MemTotal: 65539664 kB Buffers: 2228 kB Cached: 3317148 kB SwapCached: 548 kB SwapTotal: 4194300 kB

With three trials on each setup, all our experiment results include a shadow area of one standard deviation.

4.1 Observation on KL estimation

...

4.1.1 Underestimated K. ...

4.1.2 Underestimated L. ...

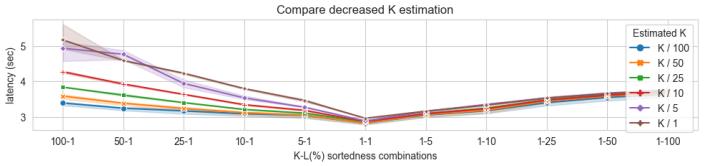

Figure 2: Caption.

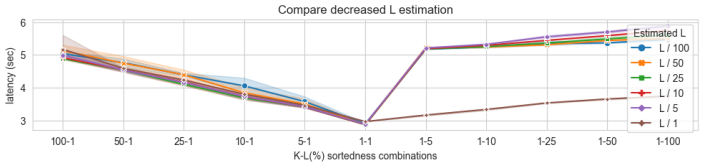

Figure 3: Caption.

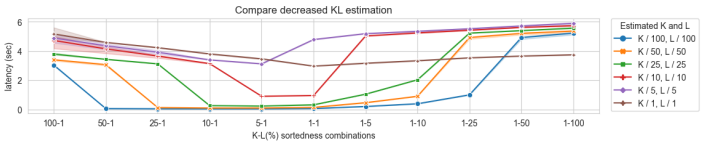

Figure 4: Caption.

4.1.3 Underestimated K and L.

4.2 Latency of Various Sortedness Workloads

TODO: Line chart of average latency with standard deviation.

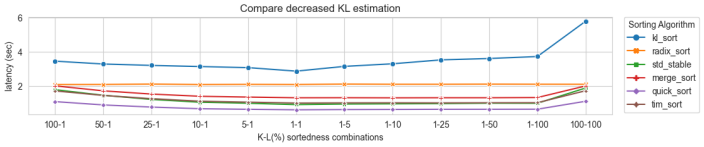

Figure 5: Caption.

4.3 Comprehensive Sortedness Simulation

TODO: Bubble chart

5 CONCLUSION

REFERENCES

[1] 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

[2] 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