
# Demonstrating Indexing for Near-Sorted Data

Aneesh Raman
Boston University
aneeshr@bu.edu

Philip Chindris
Boston University
pchin10@bu.edu

Anwesha Saha
Boston University
anwesha@bu.edu

Teona Bagashvili
Boston University
teona@bu.edu

Manos Athanassoulis
Boston University
mathan@bu.edu

## ABSTRACT

Indexing sits at the heart of database systems due to its ability to support efficient queries. Classical indexes like B$^+$-trees accelerate lookups by imposing order at the leaf level of the index, thereby adding *structure* (or *sortedness*) to otherwise unstructured data. However, when data arrives pre-sorted or *nearly-sorted*, B$^+$-trees fail to leverage the intrinsic order and perform *redundant* indexing effort. Near-sorted data is increasingly common in modern applications like the stock market or sensor readings. In such applications, the *redundant indexing effort* leads to sub-optimal performance.

In this paper, we demonstrate recently proposed indexing techniques for near-sorted data, highlighting the performance implications of textbook ingestion in B$^+$-trees as well as how our method achieves better ingestion performance. The demonstration integrates tail-leaf optimization - a widely deployed production technique for efficient ingestion, alongside SWARE and QuIT - our recently proposed index designs that harness sortedness to improve performance. Users can visualize differently sorted data workloads, view the complete index ingestion cycle, as well as analyze performance metrics that measure the impact of intrinsic order.

**PVLDB Reference Format:**
Aneesh Raman, Philip Chindris, Anwesha Saha, Teona Bagashvili, and Manos Athanassoulis. Demonstrating Indexing for Near-Sorted Data. PVLDB, 19(12): 4694 - 4697, 2026.
doi:10.14778/3827998.3828099

**PVLDB Artifact Availability:**
The source code is made available at https://github.com/BU-DiSC/sortedness-demo. The demo can be viewed at https://disc-projects.bu.edu/sortedness/.

## 1 INTRODUCTION

Indexes in database systems offer fast access to selection predicates in the base data through point or range queries. This comes at the expense of *constructing and maintaining the index*, that adds structure (or sortedness) to otherwise, unstructured data. In cases where such structure already exists, e.g., with *near-sorted* data, one would expect indexes to benefit and reduce their indexing cost.

**Near-Sortedness in Practice.** In practice, there exist many applications that generate data that is *nearly sorted* or *close to* being fully sorted. For example, stock market applications and event-based applications like sensor failures [7, 15], frequently generate or collect data that is nearly sorted. Near-sorted data can also occur as correlated attributes in large tables [2], or as intermediate results of join operations [3].

**Identifying Near-Sorted Data.** There exist several metrics that quantify sortedness in a data stream, to determine its intrinsic structure [5, 8, 9]. Simpler metrics include, *inversions*, or *exchanges*, that count the number of unordered pairs of entries or the number of swaps required to establish total order, respectively. Recent work [12] proposes to quantify sortedness using two parameters: *the number of unordered entries (K)* and *the maximum displacement among the unordered entries (L)*. When combined, the metric helps generate data through a continuum - from fully sorted to scrambled.

**Problem.** We argue that *higher the intrinsic structure or sortedness, lower the indexing cost should be*, as indexes should perform less effort to establish complete order. However, classical indexes like B$^+$-trees only optimize index construction when the data is fully sorted (e.g., through bulk loading [1, 6]). Otherwise, they fall back to standard ingestion through the root node (i.e., *top-inserts*), that leads to redundant indexing effort [11, 12, 14] for near-sorted data. This bottlenecks performance when serving applications that place an increasing demand for real-time data analysis.

**Our Approach.** Our prior work [13, 14] introduces three new indexing designs that are *sortedness-aware*, i.e., they adapt to the sortedness of incoming data to reduce the indexing effort and improve performance. The first design is a Sortedness-Aware Paradigm [14] that *intelligently* buffers and *opportunistically* bulk loads data to an underlying B$^+$-tree to improve performance. Next, we present an optimization to B$^+$-trees that allows insertions to the *last-insertion-leaf* ($\ell i\ell$-B$^+$-tree) to achieve sortedness-awareness. Lastly, we demonstrate the Quick Insertion Tree (QuIT) [13] that offers sortedness-adaptivity through a fast-path mechanism termed the *predicted-ordered-leaf* ($po\ell e$). QuIT packs a lightweight design that requires minimal tuning, enabling a more practical adoption.

In this work, we demonstrate a visualization of differently sorted data, how the sortedness-aware designs achieve index construction, and the performance implications of near-sorted data on the indexing data structures. We use the data generation algorithm from BoDS [12], while also presenting a comparative data stream generated with a simpler (and legacy) metric like *exchanges*. Interested researchers and practitioners can benefit from this tool to gain useful insights into the working of both indexing designs and assess their adoption for specific use-cases.

**Demonstration.** Participants in the conference can interact with the tool to visualize and compare differently sorted data streams as well as analyze the index construction cycle. The visual tool allows participants to see how a particular sortedness metric and its parameters affect workload patterns. The tool also allows users to see in action the complete insertion cycle in different indexing data structures - textbook B$^+$-tree, a B$^+$-tree with an optimization


Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828099

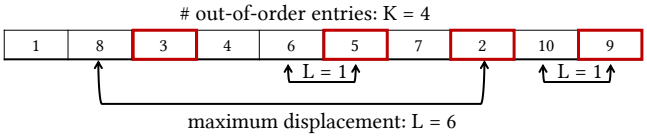


**Figure 1: The *K-L*-sortedness metric quantifies sortedness by how many elements are out-of-order and by how much.**

that allows for insertions to the tail-leaf when inserting sorted data (we call this the tail-B$^+$-tree), a $\ell i \ell$-B$^+$-tree, QuIT, and SWARE. The demonstration also captures (and plots) various metrics that affect the sortedness-awareness of the index designs (e.g., #fast inserts).

## 2 QUANTIFYING DATA SORTEDNESS

There exist a plethora of metrics to quantify data sortedness [3, 9]. Popular metrics include, *inversions* that count the number of pairs in incorrect order, or *exchanges* that count the least number of swaps needed to establish complete order in the data. In prior work [12], we introduced a more intuitive way of quantifying sortedness (inspired by Ben-moshe, et. al. [3]) through two parameters: $K$ that counts the entries in incorrect order, and $L$ that counts the maximum displacement among the unordered entries. A generalized beta distribution $(\alpha, \beta)$ controls the displacement of the out-of-order entries bounded by $L$, allowing the metric to model both uniform and skewed disorder. We illustrate the *K-L*-sortedness metric through an example in Figure 1. The metric allows us to generate a complete spectrum of data collections with varying degrees of sortedness, ranging from fully sorted ($K$=0%), nearly sorted (low $K$ and $L$), less sorted (high $K$ and $L$), and fully scrambled ($K$=$L$=100%), where both $K$ and $L$ represent a fraction of the total entries. The *K-L* metric also allows to capture near-sorted data having high $K$ but low $L$ or vice-versa. To instantiate differently sorted workloads, the data generator induces disorder by sampling $K/2$ unique source entries and swapping each entry with a destination position within $L$ positions according to the $(\alpha, \beta)$-distribution.

## 3 SORTEDNESS-AWARE INDEXES

In this section, we present a brief background into the indexing data structures that harness sortedness.

### 3.1 Tail Leaf Insertions

When inserting data in-order in a B$^+$-tree, entries are always inserted to the right-most leaf node (i.e., the tail leaf). Hence, maintaining a pointer to the tail-leaf allows direct insertions to the tail leaf node if the entry is in-order, alleviating the need for tree traversals. While tail-leaf optimization helps avoid tree traversals when inserting data in-order, it fails as soon as the leaf node has too many *outliers* or out-of-order entries [13] even with near-sorted data.

### 3.2 *Last-Insertion-Leaf*

Most entries when inserting near-sorted data are likely to follow a common insertion path. The *last-insertion-leaf* ($\ell i \ell$) optimization [13] uses this principle to attempt insertions into the leaf that accepted the most recent insert. If the entry cannot be inserted into the $\ell i \ell$-node, it is *top-inserted* and its metadata is immediately updated. While $\ell i \ell$ significantly covers the drawbacks of the tail leaf optimization, we still pay a penalty for every missed fast insertion - a top-insert will update $\ell i \ell$ to a different node which will incur another top-insert to reset.

### 3.3 Quick Insertion Tree

Ideally, an index that is fully capable of harnessing sortedness should only perform *top-inserts* for out-of-order entries, while ingesting all in-order entries using the fast-path. The Quick Insertion Tree (QuIT) [13] achieves this through the *predicted-ordered-leaf* ($po\ell e$) optimization that uses an *In-order Key estimatoR* (IKR) to determine when to update it's fast-path. Unlike $\ell i \ell$, QuIT only updates its fast-path during splits, where *IKR* determines the maximum acceptable key that can exist within the $po\ell e$-node for it to qualify as a fast-path. *IKR*'s estimation uses Equation (1) as follows: assuming $p$, $q$ and $r$ are the smallest keys in the node preceding $po\ell e$ ($po\ell e\_prev$), $po\ell e$, and the newly split node from $po\ell e$ respectively, then $po\ell e$ is updated if:

$$r \leq q + \frac{q-p}{po\ell e\_prev_{size}} \cdot po\ell e_{size} \cdot fuzzy\_scale \qquad (1)$$

If Equation (1) does not hold, then *IKR* quantifies $r$ as an *outlier*. Since this is the smallest among all keys in the newly split node, QuIT retains $po\ell e$ as is to continue with in-order insertions. To avoid reaching a *stale-state*, QuIT is also equipped with a reset mechanism that temporarily switches the fast path to the last insertion leaf after the index fails to utilize the fast path beyond a particular threshold.

### 3.4 Sortedness-Aware Paradigm

The Sortedness-Aware Paradigm (SWARE) [14] proposes a framework that can be applied to any tree-based index to achieve sortedness-aware data ingestion. To do so, SWARE uses an in-memory buffer that captures the sortedness of the data stream as it arrives, with the help of Zonemaps [10] - maintained at a granularity of every buffer page. When the buffer is full, SWARE only *opportunistically* flushes the part of the buffer that arrives fully-sorted; the remaining entries are retained in the buffer and sorted post flushing the sorted component. The *opportunistic* partial flushing of the buffer helps capture any overlapping entries that may arrive between subsequent buffer cycles. When flushing a page from the buffer, it is either *bulk loaded* onto the index when its range is in-order with the tree; otherwise, entries are *top-inserted*. To alleviate the read penalty, SWARE also augments the buffer with a collection of Bloom filters [4], that together with zonemaps reduce the amount of data scanned in the buffer.

## 4 DEMONSTRATION

**Overview.** The primary user interface of demonstration is shown in Fig. 2. The interface consists of an input panel (A-C), a data visualization panel, a control panel, index visualization panel (D-E), as well as a performance panel. The general flow of operation in the tool is as follows: (i) select the sortedness configuration for the data stream in the input panel; (ii) click "Visualize Workload" to plot the data and compare against a comparable metric; (iii) click "Visualize Index" to select two index designs to start emulating and comparing their ingestion algorithms; and (iv) compare metrics

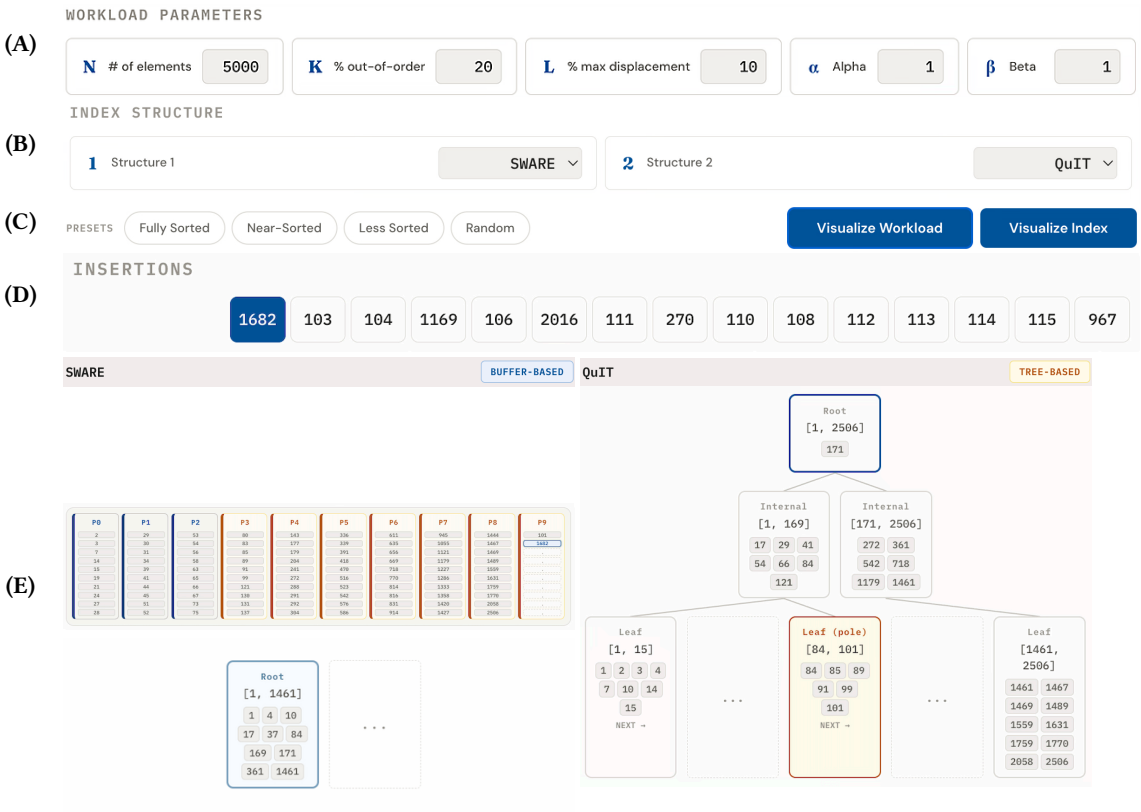


**Figure 2: The user interface allows to: (A) customize the input workload; (B) select upto two index designs; (C) utilize preset configurations to visualize the workload; (D) view the stream of inserted entries; and (E) visualize the two index insertions.**

captured during ingestion in the performance panel. The full demo can be viewed at: https://disc-projects.bu.edu/sortedness/.

**Input Panel.** The input panel shown in Fig. 2(A-C) allows users to customize the input workload, i.e., the sortedness of the input data. The interface allows the users to select the number of entries ($N$), $K$, $L$ and the sortedness distribution ($\alpha$, $\beta$). When choosing to visualize index ingestion, the input panel also provides the option for choosing up to two index designs to compare ingestion cycles against (Fig. 2(B)). In addition to allowing users to enter sortedness combinations of their choosing, the demo interface also provides preset configurations for differently sorted data (Fig. 2(C)).

**Data Visualization Panel.** The data visualization panel (Fig. 3) displays two plots that visualize the input workload as selected from the input panel. The first plot shows the data generated using the $K$-$L$ sortedness metric and the next plot shows a similar data collection generated using *exchanges* - a uni-dimensional metric.

**Control Panel.** Users can control the visualization via *Play*, *Pause*, and *Next* buttons (iterating through insertions one entry at a time).

**Index Visualization Panel.** We visualize iterative ingestion into the indexes in a dedicated panel, as shown in Fig. 2(D-E). We show the ingestion stream in small rectangular boxes, each denoting a single entry. In SWARE, entries are first inserted into the buffer containing multiple rectangular boxes - each representing a buffer page. During ingestion, out-of-order entries or pages that overlap within the buffer (in case of SWARE) use a red accent, while the non-overlapping pages in the buffer are marked by a blue accent. The other data structures directly ingest data into the index. We build each index one node at a time, and denote the range of keys and a small subset of keys that they store. Each index is highlighted in green when they accept an entry for ingestion. Fast inserts mark the respective leaf page in green, while *top-inserts* mark the insertion path in red. Additionally, the *fast-path*, (tail-leaf, $\ell i \ell$, or $po\ell e$) is labeled at the appropriate leaf node.

**Performance Panel.** The system collects various metrics that quantify the sortedness-awareness of the index. Users can observe these metrics as well as plots that are updated in real-time (Fig. 4). In each plot, the x-axis represents a running count of # inserted entries while the y-axis captures a particular metric (e.g., # fast-inserts).

## 5 DEMONSTRATION SCENARIOS

Users can interact with the demo interface to explore differently sorted workloads, their ingestion into the index, as well as analyze the impact of sortedness through performance metrics. Below, we outline four scenarios to demonstrate interactions with the system.

**Scenario 1: Visualize Data Generated Using the $K$-$L$-Metric.** We consider the following settings: $N = 5,000$ with $K = 20\%$ and $L = 10\%$, with $(\alpha, \beta) = (1, 1)$. This generates $K$-$L$-sorted data with 5,000 entries where the unordered entries are uniformly distributed. After users click on the *Visualize Data* button, they are presented

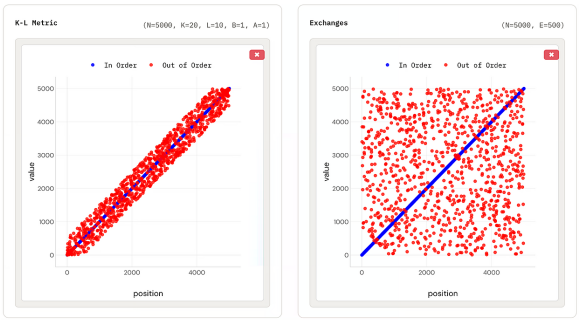


**Figure 3: The data visualization panel displays two plots for near-sorted data - the *K-L* metric (left) and *exchanges* (right).**

with two plots (Fig. 3) - the $x$-axis denotes the position of every entry while the $y$-axis denotes its value. Users can zoom in/out of the graph, and analyze how sortedness affects the data stream.

**Scenario 2: Compare Against Another Sortedness Metric.** The system also presents a second plot of a comparable data collection generated by a simpler, but uni-dimensional metric - *exchanges*. The features of the plot remain the same. In this example, we observe that the *K-L* metric is able to effectively generate entries that better qualify as nearly-sorted than a uni-dimensional metric like *exchanges* where the out-of-order entries are uniformly distributed over its entire domain.

**Scenario 3: Visualize Ingestion in Different Indexes.** We iteratively insert the data into different indexes and visualize the ingestion lifecycle. Users can see how the textbook B$^+$-tree always performs *top-insertions*, while the tail-B$^+$-tree briefly offers fast-insertions before falling back to *top-insertions* entirely. Similarly, users can also observe how the pointer to the fast-path in $\ell i \ell$-B$^+$-tree frequently switches between nodes, while the fast-path in QuIT using the $po\ell e$ optimization is relatively more stable. In SWARE, users can observe the intelligent buffering mechanism that identifies overlapping and non-overlapping/unsorted pages in the buffer, as well as opportunistic flushing of non-overlapping pages to bulk load the index. Occasionally, SWARE may perform *top-inserts* if the flushed page is not in-order with the tree.

**Scenario 4: Examine the Performance Impact.** Through metrics collected in real-time, users can analyze the impact of data sortedness during ingestion (Fig. 4). Sortedness-aware indexes ingest a larger fraction of entries employing their respective optimizations when inserting near-sorted data. This means increased opportunistic bulk loads in SWARE, while the $\ell i \ell$-B$^+$-tree and QuIT perform more fast insertions.

## 6 CONCLUSION

In this paper, we present a demonstration for indexing near-sorted data, showing impact of intrinsic data order in widely used tree indexes like the B$^+$-tree and tail-B$^+$-tree, as well as the more recent state-of-the-art sortedness-aware designs in SWARE, $\ell i \ell$-B$^+$-tree and QuIT. The demonstration tool can help users visualize differently sorted data, as well as understand how ingestion in indexes can be accelerated *harnessing* sortedness as a resource. Through various performance metrics, the demo can help users quantify the


**Figure 4: The user interface allows users to analyze the performance impact of sortedness on the indexing designs.**

potential improvement by applying the sortedness-aware optimizations for their respective use-cases.

## ACKNOWLEDGMENTS

This work was funded by NSF Grants No. IIS-2144547 and CCF-2403012, a Facebook Faculty Research Award, and a Meta Gift.

## REFERENCES

[1] Daniar Achakeev and Bernhard Seeger. 2013. Efficient Bulk Updates on Multi-version B-trees. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1834–1845.
[2] Manos Athanassoulis and Anastasia Ailamaki. 2014. BF-Tree: Approximate Tree Indexing. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1881–1892.
[3] 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 International Conference on Database Theory (ICDT)*. 256–267.
[4] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.
[5] Svante Carlsson and Jingsen Chen. 1992. On Partitions and Presortedness of Sequences. In *Acta Informatica*. 267–280.
[6] Jochen Van den Bercken and Bernhard Seeger. 2001. An Evaluation of Generic Bulk Loading Techniques. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 461–470.
[7] Nikolaus Glombiewski. 2023. *Robust Stream Indexing*. Ph. D. Dissertation.
[8] Donald E Knuth. 1997. *The art of computer programming, Volume I: Fundamental Algorithms (3rd Edition)*. Addison-Wesley.
[9] Heikki Mannila. 1985. Measures of Presortedness and Optimal Sorting Algorithms. *IEEE Transactions on Computers (TC)* 34, 4 (1985), 318–325.
[10] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 476–487.
[11] Aneesh Raman, Andy Huynh, Jinqi Lu, and Manos Athanassoulis. 2024. Benchmarking Learned and LSM Indexes for Data Sortedness. In *Proceedings of the International Workshop on Testing Database Systems (DBTest)*. 16–22.
[12] Aneesh Raman, Konstantinos Karatsenidis, Subhadeep Sarkar, Matthaios Olma, and Manos Athanassoulis. 2022. BoDS: A Benchmark on Data Sortedness. In *Performance Evaluation and Benchmarking - TPC Technology Conference (TPCTC)*. 17–32.
[13] Aneesh Raman, Konstantinos Karatsenidis, Shaolin Xie, Matthaios Olma, Subhadeep Sarkar, and Manos Athanassoulis. 2025. QuIT your B+-tree for the Quick Insertion Tree. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 451–463.
[14] Aneesh Raman, Subhadeep Sarkar, Matthaios Olma, and Manos Athanassoulis. 2023. Indexing for Near-Sorted Data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 1475–1488.
[15] Marc Seidemann, Nikolaus Glombiewski, Michael Körber, and Bernhard Seeger. 2019. ChronicleDB: A High-Performance Event Store. *ACM Transactions on Database Systems (TODS)* 44, 4 (2019).