QuIT your B$^+$-tree for the *Quick Insertion Tree*

Aneesh Raman*
Boston University
aneeshr@bu.edu

Konstantinos Karatsenidis*
Boston University
karatse@bu.edu

Shaolin Xie†
University of Southern California
shaolinx@usc.edu

Matthaios Olma
MongoDB
matt.olma@mongodb.com

Subhadeep Sarkar
Brandeis University
subhadeep@brandeis.edu

Manos Athanassoulis
Boston University
mathan@bu.edu

ABSTRACT

Search trees, like B$^+$-trees, are often used as index structures in data systems to improve query performance at the cost of index construction and maintenance. Production data systems drastically reduce the index construction cost when the data arrives fully sorted by employing a *fast-path* ingestion technique to their B$^+$-tree that directly appends the incoming entries to the tail leaf. However, this optimization is only effective if the incoming data is fully sorted or has very few out-of-order entries. The state-of-the-art sortedness-aware design (SWARE) employs an in-memory buffer to capture near-sortedness to reduce the index construction cost when the data is nearly sorted. This, however, sacrifices performance during lookups and introduces additional design complexity.

To address these challenges, we present Quick Insertion Tree (QuIT), a new sortedness-aware index that improves ingestion performance with minimal design complexity and no read overhead. QuIT maintains in memory a pointer to the *predicted-ordered-leaf* (*pole*) that provides a sortedness-aware *fast-path* optimization, and facilitates faster ingestion. The key benefit comes from accurately predicting *pole* throughout data ingestion. Further, QuIT achieves high memory utilization by maintaining tightly packed leaf nodes when the ingested data arrives with high sortedness. This, in turn, helps improve performance during range lookups. Overall, QuIT outperforms B$^+$-tree (SWARE) by up to 3× (2×) for ingestion, while also offering up to 1.32× faster (than SWARE) point lookup performance and accessing up to 2× fewer leaf nodes than the B$^+$-tree during range lookups.

1 INTRODUCTION

Database indexes accelerate query processing by offering fast access to selection predicates. B$^+$-tree indexes [14, 22] are used as the primary index data structure by several popular data systems ranging from relational row-stores [35, 36, 38, 39] like Oracle, SQL Server, PostgreSQL and MySQL to NoSQL systems [15, 23, 34] like MongoDB due to their ability to allow efficient point and range queries. The improved query performance, however, comes at the cost of constructing and maintaining the index as new data is inserted, updated, or deleted from the database [3]. With modern applications requiring data systems to also support faster ingestion in addition to efficient query processing, the indexing cost becomes prohibitive for workloads with high ingestion rates.

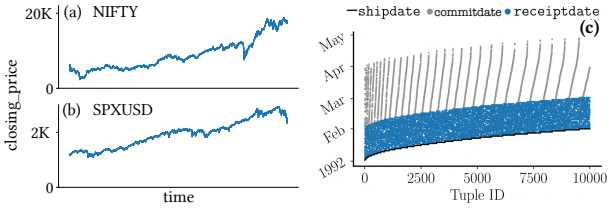

Figure 1: Real-world data can often carry implicit sortedness, for example, in closing prices of stock instruments like: (a) NIFTY, and (b) SPXUSD. (c) Near-sorted data can also occur as a result of correlated time attributes, e.g., date attributes in TPC-H data.

In a B$^+$-tree [14], ingesting a key involves accessing the root node, traversing the tree to find the appropriate leaf node whose range can fit the key, and inserting the key into that node in sorted order. This essentially adds *structure* to the data by establishing a sorted order to the data at the leaf level of the index. The tree traversal, coupled with the required splitting of nodes as the tree grows, heavily consumes the bulk of the indexing cost. There exist techniques that accelerate index ingestion when we have access to the entire dataset *a priori* (e.g., bulk-loading [1, 17]). However, for insertions taking place throughout workload execution, bulk loading is not feasible, and hence, data systems revert to standard ingestion using expensive top-to-bottom tree traversals.

Leveraging Data Sortedness. Our thesis is that since indexing *adds structure* to an otherwise unstructured data collection by creating a fully sorted version of the data, *the indexing effort should be minimal when the data arrives with some intrinsic order* [42]. The key idea here is identifying the correct location (i.e., leaf node), to insert a new key without performing expensive tree traversals. A simple case is when data arrives fully sorted – insertions are always right-deep and are applied only to the tail (right-most) leaf node of the index. Here, maintaining a pointer to the tail leaf suffices to alleviate the indexing effort. In fact, due to its simplicity, this technique is employed in popular commercial systems (e.g., *fast-path* optimization in PostgreSQL [38]). However, the *tail-leaf* optimization is only effective when data arrives fully sorted and fails to capture varying degrees of near-sortedness [41]. A small number of outliers, even as many as the capacity of a single leaf node, is enough to render the tail-leaf optimization useless, and the index reverts to expensive traversals.

In practice, several applications generate data that is *close-to* being fully sorted (i.e., *near-sorted*). Naturally occurring examples include aggregated time-series, log-based server data from data centers [12], event-based sensor data [19, 45], streaming applications that see data appear slightly out-of-order due to network delays, race conditions or intermittent machine failures [13], or even stock market data, as shown in Figures 1a and 1b. Here, we

*The first two authors have equal contribution.
†Author contributed while at Boston University.

© 2025 Copyright held by the owner/author(s). Published in Proceedings of the 28th International Conference on Extending Database Technology (EDBT), 25th March-28th March, 2025, ISBN 978-3-89318-098-1 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

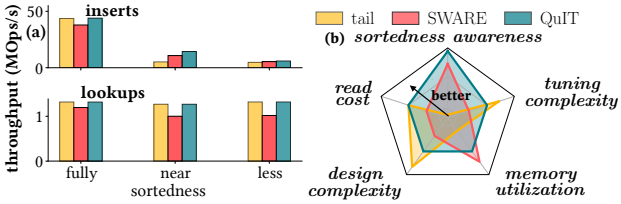

Figure 2: (a) QuIT significantly outperforms the existing baselines for any degree of sortedness; and (b) QuIT offers high sortedness-awareness with no additional read penalty while also striking a better balance between design complexity, tuning, and memory utilization.

plot the intra-day `closing_price` of the Indian Stock Market Index (`NIFTY`) and the Standard and Poor's 500 (`S&P 500`) ticker symbols at one-minute timeframes. Near-sorted data can also occur as a result of attributes in a table being correlated with the sort-key [2]. For instance, Figure 1c shows three correlated attributes `shipdate`, `receiptdate` and `commitdate` of the TPC-H [49] `lineitem` table along with their tuple ids. When the table is ordered by the `shipdate`, data in the two other date attributes occur as nearly sorted. Further, near-sortedness can also occur as intermediate results of query-evaluation in data systems - like joins or previously sorted files that received a small number of updates [5].

Recent work on sortedness awareness for indexing [42] proposes the SWARE paradigm that aims to exploit partial order in the incoming data by intelligently buffering entries and opportunistically bulk-loading them on the fly (into an underlying index, e.g., B$^+$-tree) to accelerate index ingestion. The advantages gained by applying the SWARE paradigm to a B$^+$-tree are illustrated in Figure 2a, which shows the average ingestion cost when ingesting 500M entries (integer key-value pairs) with varying data sortedness. We benchmark the B$^+$-tree with tail-leaf optimization enabled (henceforth referred to as *tail* or *tail*-B$^+$-tree) the SWARE design (equipped with a 40MB buffer), and our approach. Note that all experiments use the same underlying B$^+$-tree implementation to ensure a fair comparison.

While SWARE outperforms the tail-B$^+$-tree with near-sorted data, it falls back to a B$^+$-tree otherwise. The improved ingestion performance is beneficial for write-heavy workloads, however, accessing the buffer at query time makes SWARE slower for point queries. To reduce the read penalty, SWARE uses a more complex design for the buffer by employing additional data structures (i.e., Zonemaps [33] and Bloom filters [9]), still having up to 26% slower point queries than B$^+$-tree. This requires additional tuning, reduces maintainability, and as a result, hinders adoption.

An ideal tree index should offer high sortedness-awareness through minimal design complexity without hurting query performance.

Simplifying Sortedness-Awareness. In this work, we propose to exploit any intrinsic sortedness in the ingested data to optimize the indexing effort, with the specific goal of incurring minimal additional complexity in the index design. To that end, we propose two fast-path insertion optimizations named *last-insertion-leaf* (*$\ell i \ell$*) and *predicted-ordered-leaf* (*$po\ell e$*). The key intuition in both techniques is offering a predictor of in-order insertions and directing them to the appropriate leaf to avoid expensive tree traversals.

First, we replace the naïve predictor of *fast-path* insertions from the rightmost (*tail*) leaf to the *last-insertion-leaf* (*$\ell i \ell$*). This change allows near-sorted ingestion streams to quickly "come back" to the appropriate leaf if we have a small number of unordered entries. While this helps to increase the number of fast-path insertions, inserting an out-of-order entry results in up to two missed fast-path inserts - one top-insert (i.e., standard index ingestion) for the unordered entry and another to switch back to the correct leaf node for subsequent entries.

Ideally, we would like to only perform as many index traversals as the number of out-of-order entries ingested, and avoid the additional penalty. This is enabled by our second optimization, which is more sophisticated than *$\ell i \ell$* yet has minimal design complexity. Specifically, we maintain a pointer to the *predicted-ordered-leaf* (*$po\ell e$*), which is initiated to the tail leaf. When we receive out-of-order entries, we do not eagerly update *$po\ell e$*, rather, only when the node is split - we decide which of the two resulting nodes should be identified as *$po\ell e$* based on the ingested data.

Eliminating Overheads. In the worst case, both techniques easily fall back to regular index ingestion, which is exactly as efficient as the underlying B$^+$-tree. This way, we avoid a fraction of tree traversals without incurring any other overhead. The required metadata is also minimal: one pointer to the fast-path (*$\ell i \ell$* or *$po\ell e$*), the smallest and largest values that the fast-path node can accept, and only for *$po\ell e$*, the size and the smallest key of the previous leaf.

Quick Insertion Tree. We propose *Quick Insertion Tree* (QuIT), a lightweight indexing data structure that supports fast data ingestion using the *$po\ell e$* fast-path optimization. QuIT adapts to the sortedness of the incoming data to facilitate fast index appends. In the ingestion experiment shown in Figure 2a, QuIT outperforms the tail-B$^+$-tree and the SWARE design.

Since *$po\ell e$* is identified as a node receiving ordered entries, we use an *In-order Key estimatoR* (*IKR*) to guide its update policy (inspired by the inter-quartile range [16]). Using *IKR* enables QuIT to employ a variable split factor that better packs in-order entries in the *$po\ell e$*-node, in addition to redistributing entries between under-utilized leaf nodes. This helps the index to improve both space utilization when ingesting near-sorted data and read performance during range lookups. Further, lookups in QuIT are similar to B$^+$-tree, hence, *QuIT's benefits come with no read penalty.*

Figure 2b shows a qualitative comparison between tail-B$^+$-tree, SWARE, and QuIT based on sortedness-awareness (ingestion benefits), read cost, complexity, tuning, and memory utilization. Overall, QuIT outperforms both the tail-leaf optimization found in production systems and the SWARE paradigm for sortedness awareness without incurring any additional read penalty. QuIT achieves this with little to no tuning, minimal design complexity, and improved memory utilization. Further, QuIT allows for multi-threaded execution with some additional care for the metadata used. QuIT's lightweight design also allows for easy adoption into production data systems.

Contributions. Our work offers the following contributions:

- We propose two simple, yet powerful *fast-path* optimizations for B$^+$-trees: *$\ell i \ell$* that *reuses* the *last-insertion-leaf* to avoid full tree traversals, and *$po\ell e$* that exploits near-sortedness to predict which leaf should receive the next in-order entry (§3).
- We present an *In-order Key Estimator* (§4.1) that updates *$po\ell e$* (§4.2). *IKR* also allows for variable-split ratio and redistribution in leaf nodes to ensure higher space utilization (§4.3).
- We integrate the above techniques with minimal metadata and tuning (§4.4) and support for concurrent execution (§4.5) into

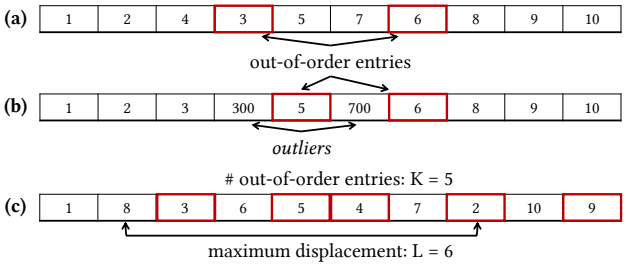

Figure 3: Examples of (a) out-of-order entries and (b) outliers in a data collection. (c) The K–L-sortedness metric captures the number of out-of-order entries as K and the maximum displacement of out-of-order entries as L.

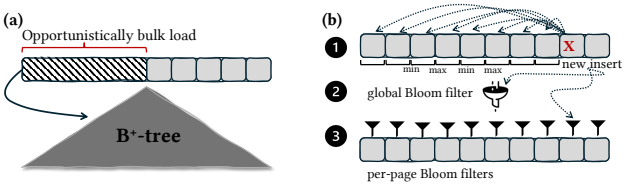

Figure 4: Internals of the SWARE paradigm: (a) SWARE employs an intelligent buffering scheme that opportunistically bulk loads pre-sorted data pages into the underlying index; however, (b) in addition to appending entries in the buffer, SWARE scans the zonemaps to update its metadata and also indexes the appended entry through two layers of Bloom Filters.

the Quick Insertion Tree (or QuIT), a general-purpose index that supports sortedness-aware fast ingestion.

- We extensively evaluate QuIT and its core components (§5). We show that QuIT significantly outperforms a state-of-the-art B$^+$-tree (with tail-leaf optimization) by up to 2.3× and SWARE by up to 2× during near-sorted data ingestion. QuIT is also 1.32× faster than SWARE during point lookups and improves its memory footprint by up to 49% when compared to B$^+$-tree.
- Finally, we demonstrate that QuIT scales well with data size and with concurrent execution, and make our artifacts available for exploration and reproducibility[1].

2 BACKGROUND AND MOTIVATION

In this section, we provide the necessary background to data sortedness and the associated index construction and maintenance cost. We discuss the existing techniques that attempt to exploit sortedness as a resource, and why they fall short.

Quantifying Data Sortedness. Data *sortedness* captures the difference between the arrival order and indexed order of data over the indexed attribute. Several metrics have been proposed in the literature that aim to quantify the sortedness of a stream of data [5, 11, 27, 32]. One way to quantify data sortedness is to count the number of entries that are out of order in a dataset [32]. By this quantification, a *fully sorted* data collection has no out-of-order entries, whereas a *scrambled* data collection has all or nearly all of its entries out of order. A *nearly-sorted* data collection has only a few out-of-order entries in an otherwise sorted data collection. In general, out-of-order entries are identified as those that are smaller than their preceding key in a monotonically increasing data stream, as shown in Figure 3a, or vice versa. Further, entries that deviate considerably from the overall expected value in a near-sorted data stream and that may (or may not) be in order with respect to their preceding entry are categorized as *outliers*. For example, in Figure 3b, although the entries 300 and 700 are in order with their respective preceding keys, they are considered *outliers* as they deviate significantly in magnitude with neighboring entries. Note that outliers are easy to identify when the data set is available in its entirety, however, identifying them accurately in an incoming data stream is challenging.

The K–L-Sortedness Metric. The K–L-sortedness metric [41] inspired by Ben-Moshe et. al. [5] more comprehensively quantifies data sortedness by accounting for both *the out-of-order entries* and *the distance by which they are out of order*. The number of out-of-order entries is denoted by K, and the maximum displacement of an unordered entry from its in-order position is denoted by L. An example of a nearly sorted data collection based on K–L-sortedness is shown in Figure 3c where K=5 entries are out of place and the out-of-place entries are displaced by at most of L=6 index positions.

Sortedness-Aware Indexes. Raman *et al.* [42] proposed the SWARE indexing paradigm that captures the sortedness of a data stream using an in-memory buffer and opportunistically bulk load (on-the-fly) incoming data to the underlying tree index (e.g., B$^+$-tree), as shown in Figure 4a. The benefits gained by buffering entries during ingestion come at the expense of query performance, as every query now has to first search the buffer. SWARE partially addresses this overhead by employing auxiliary data structures like Zonemaps [33] and Bloom filters [9] (BFs), in addition to a query-driven partial-sorting technique that is inspired by Cracking [24, 25]. Yet, point queries are up to 26% slower than the baseline - this cost becomes prohibitive when the fraction of reads in the workload increases.

More importantly, adding Zonemaps and Bloom filters to the design implies that insertions to the index are no longer simple appends to the buffer, as shown in Figure 4b. Every insert first checks if it is arriving in-order to the preceding key, and if otherwise, performs a linear scan of the Zonemaps to identify overlapping pages within the buffer. Additionally, the inserted key is also indexed through a couple of layers of Bloom filters [9] that require re-calibration during every buffer flush. The buffer and the metadata (including the auxiliary data structures) also increase the memory footprint (e.g., for indexing 1 TB of data, the memory requirement can be more than 10 GB). Thus, in addition to imposing a penalty during lookups, SWARE also requires careful tuning to ensure that the benefits of buffering outweigh tree traversals to guarantee overall performance improvement.

Tail-Leaf Insertion. Tail-leaf insertion in B$^+$-trees is a simple *fast-path* optimization that benefits from incremental in-order ingestion to the index. The tail-leaf *fast-path* essentially maintains one additional pointer to the rightmost leaf (tail) of the index along with its smallest allowed value. Any newly ingested key that is greater or equal to that value is directly inserted into the tail-leaf that is naturally cached, rather than traversing the tree. While this fast-path optimization is rarely useful when data does not arrive in sorted order, surprisingly, it also fails to accelerate index ingestion even when data arrives near-sorted. As soon as the number of outliers inserted exceeds one node worth of data, the tail-leaf only contains outliers, resulting in a "stale" fast-path. This results in future insertions (even for near-sorted

[1] https://github.com/BU-DiSC/quick-insertion-tree

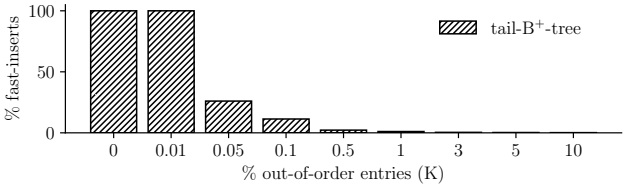

Figure 5: The tail-leaf optimization is effective only for an extremely high degree of sortedness and leads to no fast-inserts when 1% or of the entries are out of order.

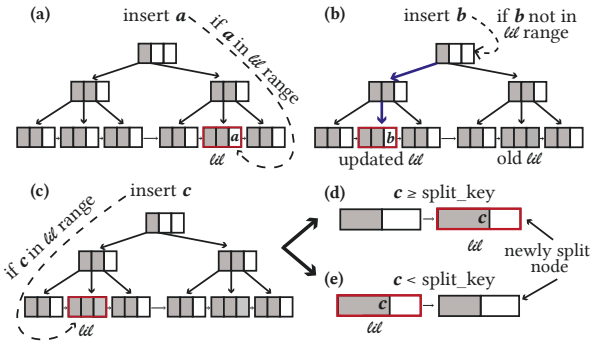

Figure 6: Fast-path ingestion using $\ell i \ell$: (a) a newly inserted key is added to $\ell i \ell$ if within $\ell i \ell$-range; (b) a top-insert updates $\ell i \ell$ pointer to the leaf node where we insert the key; (c) when an insert to $\ell i \ell$ causes a split, (d) $\ell i \ell$ is updated if $c \geq split_key$; otherwise (e) $\ell i \ell$ stays as is.

data) reverting to traditional incremental ingestion to the index (referred to as *top-inserts*) and missing the opportunity to utilize the fast-path (referred to as *fast-inserts*).

Tail-leaf is Only Helpful for Extremely High Sortedness. Figure 5 shows the fraction of fast-inserts when ingesting 5M integers into a tail-B$^+$-tree, as we vary data sortedness. Specifically, we vary the fraction of out-of-order entries, which are positioned uniformly and randomly in the workload. As expected, the tail-leaf optimization is effective for sorted data (i.e., 0% out-of-order entries) or extremely near-sorted data (i.e., very few out-of-order entries). Thus, while we get negligible top-inserts for 0.01% out-of-order entries, the tail-leaf optimization's efficiency drops to only 23% (11%) fast-inserts for 0.05% (0.1%) out-of-order entries, and, ultimately, to less than 1% fast-inserts for 1% out-of-order entries and beyond. This renders the tail-leaf optimization impractical for most near-sorted workloads as *fast-path* ingestion is very rarely used.

3 SORTEDNESS-AWARE FAST PATHS

We now propose *last-insertion-leaf* ($\ell i \ell$), a renewed fast-path optimization technique for B$^+$-trees that offers superior ingestion performance for near-sorted workloads, and lay the groundwork for *predicted-ordered-leaf* (*pole*), QuIT's key ingredient.

Tracing the Leaf of the Last Insertion. Contrary to the tail-leaf optimization, we maintain a pointer to the *last-insertion-leaf* or $\ell i \ell$ for short. At any point during a workload execution, $\ell i \ell$ points to the leaf node to which the most recent entry was inserted. A subsequent insert may be added to the $\ell i \ell$-node if it falls within its range as shown in Figure 6a. Otherwise, we revert to a *top-insert* followed by an update to the $\ell i \ell$-pointer (Fig. 6b). The $\ell i \ell$-pointer is also updated if a newly inserted entry results in splitting the

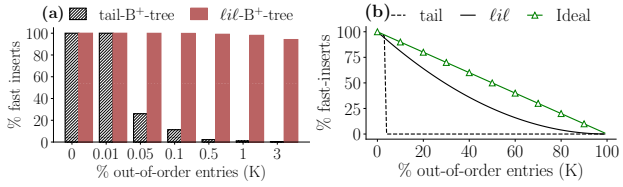

Figure 7: (a) The *last-insertion-leaf* optimization significantly outperforms tail-leaf insertions in B$^+$-trees for highly sorted data. (b) Simulation of the expected fraction of top-inserts using $\ell i \ell$ while varying data sortedness.

$\ell i \ell$-node (Fig. 6c). In this case, we update the $\ell i \ell$-pointer if the inserted key is placed into the newly created node from the split (Fig. 6d), or keep it unchanged otherwise (Fig. 6e).

Modeling $\ell i \ell$ Benefits. We quantify the expected efficiency of $\ell i \ell$ by estimating the fraction of data that would be fast-inserted into the tree as a function of the out-of-order entries. Ideally, a sortedness-aware index should fast-insert all in-order entries and perform top-inserts only for entries that are out of order. However, $\ell i \ell$ performs a top-insert both when (a) an out-of-order entry follows an in-order entry (outlier or not) and (b) an in-order entry follows an outlier. Conversely, $\ell i \ell$ would only succeed when we have two in-order entries in a row. We calculate the probability of two consecutive entries being in order as follows. We assume that we insert n entries into a B$^+$-tree and that a fraction, k, of those entries are out of order. Then, the number of in-order entries, y, is: $y = n \cdot (1 - k)$, and the probability of a fast-insert (which happens when two consecutive entries are in-order), FI, is given as:

$$FI = \frac{y}{n} \cdot \frac{y-1}{n-1} \underset{\text{for large } n}{\approx} \left(\frac{y}{n}\right)^2 = (1-k)^2 \quad (1)$$

Evaluating $\ell i \ell$. While the tail-leaf optimization results in virtually no fast-inserts even with only 1% out-of-order entries, Eq. (1) shows that $\ell i \ell$ should manage to achieve 98% fast-inserts in the same workload, which is corroborated experimentally. Figure 7a shows the fraction of fast-inserts (on the y-axis) when ingesting $5M$ entries (integer K-V pairs) as we vary the fraction of out-of-order entries k in the x-axis. Firstly, for fully sorted data both the tail and $\ell i \ell$ optimizations invoke their respective fast-path insertion routines and avoid expensive top-inserts altogether. We observe that $\ell i \ell$ is indeed able to perform 98% fast-inserts for a workload with k = 1%, 90% fast-inserts for a workload with k = 5% while also performing around almost no fast-inserts for k = 100%. The very few fast-inserts for the latter are due to consecutive out-of-order entries targeting the same node with a very low probability, which is slightly higher when the tree is small.

The superior benefits of $\ell i \ell$ against the tail-leaf optimization is because the latter works well only when an incoming entry can be correctly positioned in the tail leaf, the probability of which is very low. On the other hand, we observe a gradual decrease of fast-inserts performed in $\ell i \ell$ as we decrease data sortedness. While $\ell i \ell$ initially points to the tail leaf, it updates to the last insertion leaf as soon as we encounter an entry that is top-inserted elsewhere. This allows all subsequent in-order entries to be ingested through the fast path to the new $\ell i \ell$ or revert to the "correct" leaf node if $\ell i \ell$ is filled with outliers.

Headroom of Improving $\ell i \ell$. While $\ell i \ell$'s design offers the opportunity to perform better than the state-of-the-art tail-leaf optimization, there is still a lot of room for improvement. To quantify

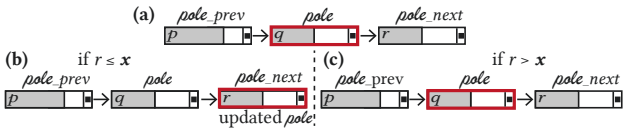

Figure 8: Updating *pole*: (a) post splitting *pole*, we use the smallest key (r) in the newly split node (*pole_next*) and compare to an estimation (x) from the *IKR*; (b) if $r \leq x$, we update *pole*; otherwise, (c) if $r > x$ we leave *pole* as is.

the headroom of improvement, we compare the expected number of fast-inserts performed by $\ell i \ell$ with the theoretical ideal design of a sortedness-aware fast-path optimization. We use Eq. (1) to estimate the number of fast-inserts as a fraction of the workload, while the ideal is all in-order entries. Figure 7b, shows that while $\ell i \ell$ (solid black line) is expected to clearly outperform tail-leaf insertion (in dashed black line) – from the experiment in Figure 7a, the number of fast-inserts quickly drops as the number of out-of-order entries increases. Instead, for an ideal sortedness-aware index (green line with a triangle marker), the fraction of fast inserts are expected to be linearly proportional to the in-order entries. The area between the solid black and green lines presents the headroom for improvement.

Focusing on the missed opportunity, $\ell i \ell$ performs two top-inserts per out-of-order entry: (i) one for an out-of-order entry, and (ii) one for the in-order after $\ell i \ell$ is updated to the wrong leaf.

Optimal sortedness-awareness should incur, at most, one top-insert per out-of-order entry.

Thus, we expect that the number of top-inserts (that can be 3-4× more expensive than fast inserts depending on the height of the tree) will be halved by such an ideal design, substantially reducing the overall cost of near-sorted data ingestion. Finally, when inserting near-sorted data we can improve the space utilization of the index by better packing leaf nodes with in-order entries and using a variable split ratio [42]. On the other hand, for tail-leaf optimization and $\ell i \ell$, the higher the data sortedness the lower the space utilization, since every node split will leave a half-full node that will never receive any future insert.

Next, we propose a new design that bridges the aforementioned performance headroom with a more robust fast-path optimization, while also offering better space utilization.

4 QUICK INSERTION TREE

We present *Quick Insertion Tree* (QuIT), an indexing data structure that is sortedness-aware *by design* and offers superior ingestion performance along with better space utilization when ingesting near-sorted data. At its foundation, QuIT is similar to B$^+$-tree – its root and internal nodes contain a list of keys and pointers, while the leaf nodes contain the data entries. However, QuIT employs a sortedness-aware fast-path optimization that benefits ingestion workloads that have at least some degree of intrinsic data sortedness. If the data is completely scrambled, QuIT effectively behaves like B$^+$-tree for writes. The key advantage of QuIT over other sortedness-aware counterparts [42] is the lack of any additional read penalty when compared to a B$^+$-tree. It also has minimal design complexity and requires little to no tuning. Through the rest of this section, we present the architecture of the Quick Insertion Tree.

4.1 A Robust Fast-Path Optimization

Predicting the Ordered Leaf. In Section 3 (Fig. 7b), we pointed out that an ideal sortedness-aware index would perform expensive top-inserts only for entries that are out of order, while all other entries should be ingested using the fast path. The fundamental limitation of $\ell i \ell$ is that it naïvely switches the fast-path access pointer (i.e., the $\ell i \ell$-pointer) based on the most recent insert, even if the entry ingested is out of order. We address this by replacing $\ell i \ell$ with a new leaf node pointer to the *predicted-ordered-leaf* (*pole*) node, that tracks the leaf that is most likely to accept the future in-order entries. Similarly to $\ell i \ell$, out-of-order entries are top-inserted. However, unlike $\ell i \ell$, the pointer to *pole* may be updated only when *pole* splits. The newly created node from the split will be identified as *pole* if its smallest key is not an *outlier*, while *pole* remains unchanged otherwise. A natural approach for updating *pole* can involve maintaining a running average of the deltas between all existing keys in the index and comparing it against the average of deltas in *pole*. However, through initial experiments (omitted for brevity), we observed that such a policy makes it increasingly difficult for the index to capture evolving degrees of sortedness, resulting in performance degradation. Thus, we build a simpler yet more adaptive outlier predictor called *In-order Key estimatoR* (*IKR*) as discussed below to guide the update policy of *pole*.

Identifying Outliers. Assume we are inserting entries into the index with keys following an increasing order. Let $pole_{size}$ and $pole_prev_{size}$ denote the respective number of entries in *pole* and its preceding node *pole_prev*. We want to identify if *pole* needs to be updated upon split. Now, let p and q be the values of the smallest key in the *pole_prev* and *pole*, as shown in Figure 8a. Since *pole* contains in-order entries, q is not an outlier, and necessarily p is also not an outlier as it precedes q. When splitting, we would like to identify whether the smallest key in the new node created from the split, r (i.e., the split key), is an outlier. This helps decide whether to keep the pointer to *pole* unchanged (if r is an outlier) or move *pole* to the new node (if r is not an outlier).

Our lightweight *IKR* estimator (inspired by *Interquartile Range* outlier detection [16]) calculates the maximum acceptable domain for a non-outlier key. Any key beyond this range is considered an outlier (denoted by x) as follows:

$$x = q + \left(\frac{q-p}{pole_prev_{size}}\right) \cdot pole_{size} \cdot scale \qquad (2)$$

The term $\frac{q-p}{pole_prev_{size}}$ calculates the density between p and q, i.e., two non-outliers. To ensure enough data for prediction, we bound $pole_prev_{size} \geq 50\%$ of node capacity (which is always true in traditional B$^+$-tree-node-splitting). The *scale* allows a small buffer to capture small deviations in density that are inherent in the data. Following standard practice [16], we use $scale = 1.5$. Consequently, from Eq. (2), we consider any $key > x$ as an outlier.

4.2 Fast-Path Insertion in *pole*

We now describe the application of *pole* as a fast-path ingestion technique in the Quick Insertion Tree and outline the steady-state insertion algorithm in Algorithm 1.

Initialization. The initial state of the index is represented by a single leaf node in the tree that is also its root. We also mark this leaf as *pole*. When this leaf first splits, we create a new root node and add the pivot pointers to its two children (similar to B$^+$-tree). We mark the leaf that received the latest insert as the *pole*-node.

Algorithm 1: Updating *predicted-ordered-leaf*

```
   Data: p = pole_prev_min, q = pole_min, entry = (key, value)
   Init: scale = 1.5
1  if q ≤ key < pole_max then                          // fast-insert
2      if pole is full then
3          pole_next ← pole.split();                   // return new leaf
4          r ← pole_next_min;
5          x ← q + ((q−p) / pole_prev_size) · pole_size · scale;
6          if r ≤ x then
7              pole_prev ← pole;
8              pole ← pole_next;
9      pole.insert(entry);
10 else
11     l_t ← top_insert(entry);                        // return selected leaf
12     if l_t = pole_next then                         // pole catches up
13         pole_prev ← pole;
14         pole ← pole_next;
```

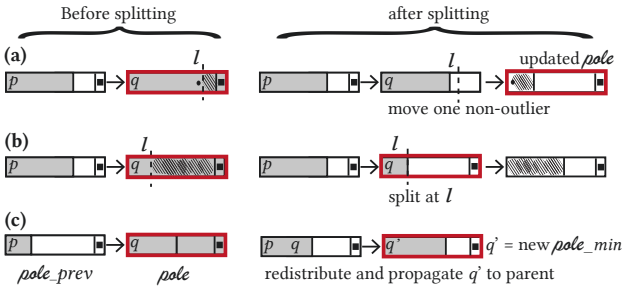

Figure 9: The default split position is in the middle of *pole* (50%). When $pole_prev_{size} \geq 50\%$, we use *IKR* to identify outliers. (a) If outliers occupy < 50% of *pole* ($l > def_split_pos$), we split at $l-1$, moving a non-outlier value to the newly split node and update *pole*; (b) Otherwise, we split at l (using the default *IKR*) and keep *pole*-pointer as is; (c) When $pole_prev_{size} < 50\%$, we redistribute entries to *pole_prev*, until $pole_prev_{size} = 50\%$.

Steady State. In steady-state, an entry inserted to the index utilizes *pole* if its key is within $[pole_{min}, pole_{max})$. When *pole* is also the tail-leaf, we omit the upper bound check on $pole_{max}$.

Splitting the *pole*-node. Once *pole* is full, we split it and update the pointer to *pole* as shown in Figure 8. We refer to the newly created node from the split as *pole_next*. We compare x from Eq. (2) with the smallest key in *pole_next*, denoted by r. We update *pole* to *pole_next* if $r \leq x$ (Fig. 8b), and leave it as-is otherwise (Fig. 8c).

Catching Up to Predicted Outliers. When $r > x$ (i.e., when r is an outlier), we infer that all entries of the new node are outliers. So, any future in-order entries will belong to the node that was split, and thus, we do not update *pole* after splitting. Eventually, if keys are largely in-order, entries in the *pole*-node may catch up to previously marked outliers in *pole_next*. Hence, when an entry is top-inserted to *pole_next*, we check if it is still an outlier using *IKR* and, only if not, update *pole* to *pole_next*.

4.3 Improving Space Utilization

The *pole* optimization offers a robust and sortedness-aware fast-path access to the leaf level of the index that avoids tree traversals for in-order data to boost ingestion performance. However, the current design retains the same worst-case space utilization as a B$^+$-tree when ingesting fully-sorted data. That is, due to consecutive right-deep insertions, every node will be exactly half full, wasting 50% of space. Ideally, a sortedness-aware index should exploit inherent data sortedness to improve its space utilization.

Finding Better Split-Points. QuIT exploits *IKR* and *pole* to further improve space utilization. First, in Algorithm 2, we detail a variable split strategy in the leaf nodes through which ordered data can be more tightly packed. We re-use *IKR* to identify the outliers and determine the optimal split-points for the node. Note that this is fundamentally different from Algorithm 1, where we first split *pole* by default at 50% and only use the *IKR* to update *pole*. Next, we discuss the major decisions in Algorithm 2.

Splitting a Node other than *pole*. Similar to a B$^+$-tree, we split a (non-*pole*) leaf node at 50% ($def_split_pos = leaf_{capacity}/2$).

Splitting *pole* when *pole_prev* is at Least Half-full. When *pole* is about to split and its preceding node (*pole_prev*) is at least half-full, we use *IKR* to determine the split position. We identify the position (l) of the first key greater than the largest estimated acceptable value in the *pole*-node in line 4.

When *pole* Contains Only Few Outliers. If *pole* mostly contains in-order entries ($l > def_split_pos$), we split the node (and update it as *pole*) at $l-1$ (lines 5-7), taking one in-order value to the newly created node from the split, as shown in Figure 9a. This ensures that the split leaf is at least half full while the new *pole* has more space to accommodate future fast-insertions.

When *pole* Contains Mostly Outliers. When $l \leq def_split_pos$, *pole* contains mostly outliers, and we split it at l and move all outliers to the newly created node (Fig. 9b). We do not alter the pointer to *pole* as it now has enough space for future fast insertions.

Redistribution When *pole_prev* is Less than Half-full. In case that at *pole*-splitting time, *pole_prev* is less than half full (e.g., due to an earlier variable split), using *IKR* may lead to an inaccurate estimation as it does not have enough data. Instead, we redistribute entries (line 10) from *pole* to *pole_prev* until the

Algorithm 2: Variable split strategy

```
   Data: q = pole_min, entry = (key, value)
   Init: scale = 1.5, def_split_pos = leaf_capacity / 2
1  if leaf ≠ pole then
2      leaf.split(def_split_pos);        // split leaf at 50%
3  else if pole_prev_size ≥ def_split_pos then // use IKR
4      l ← leaf.position(q + ((q−p) / pole_prev_size) · scale);
5      if l > def_split_pos then
           // take one non-outlier to new node
           pole_next ← pole.split(l − 1);
6          pole_prev ← pole;
7          pole ← pole_next;
8      else
           // move all outliers to new node
           pole_next ← lol.split(l);
9  else
       // redistribute entries from pole_prev
10     lol.redistribute(pole_prev);
```

latter is exactly half full (Fig. 9c). This allows future splits in the *pole* node to use the variable split strategy.

Resetting Fast-Path When *pole* Goes Stale. As with any other fast-path optimization, there exists a scenario where *pole* becomes *stale* due to certain workload characteristics, leading to unexpectedly many top-inserts. For example, this can be caused by workloads that do not exhibit any sortedness, or by corner cases of specific sequences of in-order and out-of-order insertions that may throw *IKR* off. We recover from the *stale* state by *resetting pole* to the leaf that accepted the latest insert. However, unlike *lil*, we do not adjust *pole* for every top-insert. Rather, we do so only if we have already performed a number of consecutive top-inserts. We set this threshold as $T_R = \lfloor \sqrt{leaf_{capacity}} \rfloor$ to offer a balanced outcome.

4.4 Other QuIT Operations

Point Lookups. Lookups in QuIT are identical to the classical B$^+$-tree. A point lookup-path starts at the root node and uses key comparisons to follow the pivot pointers, to navigate to the appropriate leaf node that may contain the target entry. A binary search on the keys in the leaf node returns whether the search key is present in the index or not.

Range Lookups. In QuIT, a lookup for keys in $[start, end]$ first performs a point lookup on $start$ to locate the first key $\geq start$. It then uses the pointers between leaf nodes to scan all entries until an entry $\geq end$ is found (similar to B$^+$-tree).

Deletes. Delete operations in QuIT are also exactly the same as the B$^+$-trees. Deletion of an entry begins with a point lookup on the target key. If the key is found, in QuIT, we typically remove the key from the corresponding leaf node and apply rebalancing to ensure that the leaf node and all internal nodes leading to the leaf node are at least half full. Only deletes of entries in the *pole*-node do not rebalance eagerly. In case the key marked to delete is the only key in *pole*, we reset *pole* to *pole_prev*.

4.5 Concurrency Control

Concurrency control protocols that have been widely studied for B$^+$-trees [4, 7] can be applied out-of-the-box to QuIT.

Locking Protocol for Ingestion. In B$^+$-trees, the internal nodes simply redirect searches to the next level that contains either an internal node or a leaf node. Hence, completely and exclusively locking the entire path is wasteful. We employ a simple protocol inspired by classical lock-crabbing [21].

Classical lock-crabbing [21] starts at the root node and descends down the tree, acquiring exclusive locks at every node along the insertion path. If a node along the path is not full, an insertion does not result in a split. Therefore, all the acquired locks in the preceding levels of the path are released. A lock on the entire path is acquired only when every node along the path is full. The locks are retained until the insertion completes, as a split in the lowest level of the tree (i.e., the leaf node) can potentially propagate to every internal node along the path, including the root node.

Granular Crabbing for QuIT. Our locking procedure is presented in Algorithm 3 and described in detail below. Every insert in QuIT first acquires a write lock on the fast-path metadata to verify if we can insert to *pole*. We do not optimistically lock the metadata (for read access) since we expect data to frequently arrive near-sorted and thus target *pole*. Optimistically granting read locks in such a scenario adds unnecessary effort due to restarting the procedure as most insertions will tend to conflict (inserting to *pole*). We now discuss what follows after locking the metadata.

Validating the fast-path. If the insertion key is within the range of fast-path access, we obtain a write lock on *pole* and verify again that the key is within its range. This check ensures that no other thread resulted in a split/reset/redistribution that would render the fast-path useless for this insertion key. If the fast-path has changed, we unlock *pole* and the metadata, and *top-insert* the entry.

The pole is full. If the key is within the range of fast-path access but *pole* is full, we top-insert the entry because the split may propagate to the higher levels of the tree and potentially up to the root node. After completing the top-insert, we unlock the fast-path metadata. That way, we avoid deadlocks due to concurrent inserts to the index, as they will first wait to lock the metadata.

Insertion can use fast-path. If the key is within the range of the fast-path access, and *pole* is not full, we update the size of the fast-path and unlock the metadata. This ensures that we do not hold the lock on the metadata for a long duration, so that it does not block concurrent insertion to the index. We then proceed with inserting the entry to *pole*, before unlocking it.

If key is not in *pole* range, we unlock the metadata before employing a top-insert for the entry.

Locking other nodes. Additionally, a split to *pole* would either result in the creation of a new node or a redistribution of entries from *pole* to *pole_prev*. In the former case, we only acquire a lock on the metadata of *pole_prev* (i.e., *pole_prev*_id, *pole_prev*_min, and *pole_prev*_size). For redistribution, we also acquire a lock on *pole_prev* and its associated metadata.

Optimistic top-inserts. During top-inserts, we obtain locks on nodes optimistically - we take read locks on internal nodes and a write lock on the appropriate leaf. In case the leaf node needs to split, we restart the procedure using traditional lock-coupling.

Locking Protocol for Lookups. Like simple lookups, concurrent lookups to QuIT essentially follow the same procedure as in a B$^+$-tree. We start from the root node, acquiring read locks on every node along the access path. Range lookups similarly

Algorithm 3: Granular Lock Crabbing in QuIT

```
1  metadata.writeLock();
2  if key within pole range then
3      pole.writeLock();
4      if key not in pole range then // change before locking
5          pole.unlock();
6          optimistic_top_insert();
7          metadata.unlock();
8      if pole is full then
9          optimistic_top_insert();
10         metadata.unlock();
11     else
12         fp_size++;
13         metadata.unlock();
14         pole.insert(key, value);
15     pole.unlock();
16 else
17     metadata.unlock();
18     optimistic_top_insert();
```

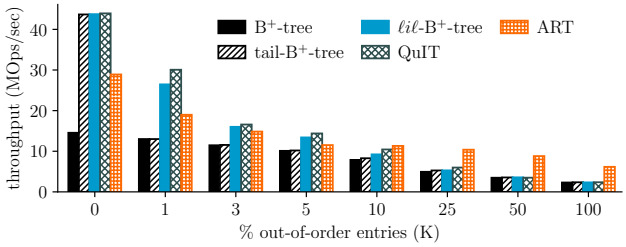

Figure 10: QuIT outperforms all B$^+$-tree baselines for any degree of data sortedness during ingestion, while it outperforms ART for fully sorted or nearly sorted data.

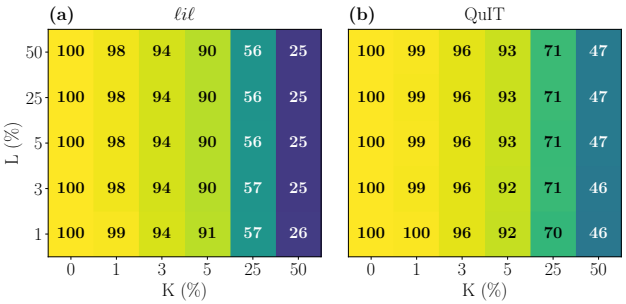

Figure 11: Comparing fraction of fast-insertions $\ell i \ell$-B$^+$-tree and QuIT when varying data sortedness in the ingestion workload: (a) $\ell i \ell$ performs best with high data sortedness; (b) QuIT maximizes fast inserts when compared to $\ell i \ell$.

obtain read locks up to the first leaf node that is accessed by the query and on subsequent leaf nodes.

5 EXPERIMENTAL EVALUATION

We now show the benefits of QuIT by comparing it to (i) a textbook B$^+$-tree (*classical B$^+$-tree*) that only performs top-inserts, (ii) a B$^+$-tree with tail-leaf optimization (*tail-B$^+$-tree*) inspired by the state-of-the-art design in PostgreSQL [38], (iii) a B$^+$-tree with the *last-insertion-leaf* optimization (*$\ell i \ell$-B$^+$-tree*), the Adaptive Radix Tree (ART) [30], and the state-of-the-art sortedness-aware index design (SWARE [42]). We test using multiple configurations, varying data sortedness and data size, stress-testing the fast-path. Finally, we compare with a full-feature B$^\epsilon$-tree and LSM-tree, and we experiment with real-world stock market data.

Experimental Setup. We run experiments using our in-house server with 128GB of DDR5 main memory at 4800 MHz and a 1.9TB NVMe SSD. The server runs Rocky Linux (version 9.3) and is equipped with two sockets of Intel Xeon Gold (6442*Y*) 2.6 GHz processors (24 cores), each supporting 48 threads. We run experiments on a single core and extend to multiple cores for concurrent execution in §5.3.

Index Design and Default Setup. We use an in-memory implementation of the B$^+$-tree (inspired by state-of-the-art [8]), and implement tail, $\ell i \ell$, and *poℓe* optimizations on this platform. We also implement QuIT on the same platform that includes features such as the variable-split, redistribute, and reset strategies. We use the same B$^+$-tree implementation as the underlying index when comparing with the SWARE paradigm by extending the API to support bulk loading. For the SWARE buffer, we deploy the open-sourced code [43] out-of-the-box and default to a buffer size equivalent to 1% of the total data size (same setting as the SWARE paper [42]). The default entry size in all our experiments is 8B (with 4B keys), and we use a 4KB page size that fits up to 510 entries in the leaf nodes. We set $T_R = \lfloor \sqrt{leaf_{capacity}} \rfloor = \lfloor \sqrt{510} \rfloor = 22$ to trigger a reset of the *poℓe* fast-path in QuIT. We make our code available on GitHub at https://github.com/BU-DiSC/quick-insertion-tree.

Workloads. To evaluate QuIT with varying degrees of sortedness, we use the workload generator from the Benchmark on Data Sortedness (BoDS) [41]. BoDS uses the K–L sortedness metric to create a family of differently sorted data collections. The generator takes as arguments (i) the number of entries (N) to ingest, (ii) the number of unordered entries (K) and the maximum displacement (L) both as a fraction of N, (iii) the distribution of data sortedness (α, β), and (iv) a seed value.

Default Workload. For our experiments, we set N=500M to generate datasets of size 4GB. The skew-parameters α and β are set to 1 to uniformly distribute the unordered entries. Our query workloads (executed post data ingestion) contain 5M (1% of total data size) point lookups that are generated uniformly and randomly on existing keys in the index. Further, our query workloads also contain 1000 range lookups that are generated for random ranges in the key domain with three levels of selectivity: 0.1%, 1%, and 10%.

5.1 Benefits of Quick Insertion Tree

We first demonstrate the benefits of QuIT against three baselines – classical B$^+$-tree, tail-B$^+$-tree, $\ell i \ell$-B$^+$-tree, and ART for ingestion performance. We also compare QuIT against other B$^+$-tree baselines for memory utilization and query performance. In these experiments, we vary the out-of-order entries (K) in the data collection while setting their maximum displacement (L) to 100%, and compile all indexes with the `-O3` optimization enabled.

QuIT Outperforms the State of the Art and $\ell i \ell$-B$^+$-tree. Figure 10 shows the ingestion throughput (y-axis) during ingestion for the B$^+$-tree, tail-B$^+$-tree, $\ell i \ell$-B$^+$-tree, and QuIT when varying data sortedness (x-axis). QuIT significantly outperforms the tail-B$^+$-tree and the classical B$^+$-tree in terms of ingestion performance. For fully sorted data, QuIT, $\ell i \ell$-B$^+$-tree and tail-B$^+$-tree offer $\approx 3\times$ better performance than a classical B$^+$-tree, as entries are directly inserted to the tail leaf using the fast-path optimization, rather than performing tree traversals. However, the performance of the tail-B$^+$-tree degrades very quickly as the data becomes even slightly unsorted, making it comparable to that of a classical B$^+$-tree. QuIT, on the other hand, exploits any inherent data sortedness and offers up to $\approx 2.3\times$ better throughput for near-sorted data ($K < 25\%$). Even for less-sorted data (K=25%), QuIT is still 1.23× better than both the classical B$^+$-tree and tail-B$^+$-tree, as both the baselines always perform top-inserts, accumulating a redundant indexing cost. The tail-B$^+$-tree performs better than the classical B$^+$-tree only with fully sorted data by allowing fast-path ingestion to the right-most leaf node in the index. This fast-path, however, becomes stale with even near-sorted data ($K \geq 1\%$), leading to performance degradation as its window to capture unorderedness in the data is too narrow (limited to the tail-leaf's range). Meanwhile, QuIT carefully adapts its fast-path (i.e., *poℓe*) to the data sortedness, thereby, delivering better performance.

We also observe that QuIT performs up to 14% better during ingestion than the $\ell i \ell$-B$^+$-tree. The $\ell i \ell$ optimization is simple and effective in improving ingestion performance over a tail-B$^+$-tree. Unlike $\ell i \ell$ that pays an additional penalty (as discussed in §3),

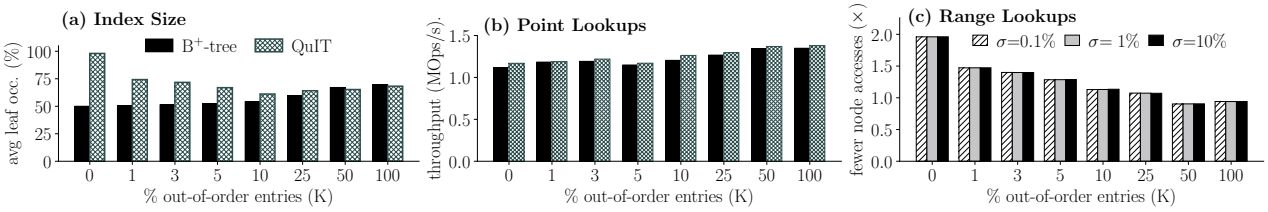

Figure 12: (a) Variable split factor in QuIT allows for higher occupancy in the leaf nodes; (b) Point lookups do not incur any read overhead; (c) Range queries are faster as they access fewer nodes during lookups.

QuIT reduces the standard index traversals to only as many as the number of out-of-order entries during ingestion. This, coupled with improved caching effects due to a more compact structure results in a better performance than the $\ell i \ell$-B$^+$-tree.

More Fast-Insertions Result in Improved Performance. The improved ingestion performance of QuIT is directly correlated to the increase in fast-insertions performed by adapting to data sortedness, as shown in Figure 11a and Figure 11b. QuIT performs approximately only as many top-inserts as there are out-of-order entries in the ingestion workload, very closely resembling the ideal sortedness-aware behavior. This benefit is pronounced when ingestion near-sorted and less-sorted data, as QuIT achieves up to 1.8× more fast-insertions for the same ingestion workload than the $\ell i \ell$-B$^+$-tree. This is because $\ell i \ell$ eagerly updates its fast-path access pointer based on the last insertion to the index (even during standard ingestion or *top-inserts*). Meanwhile, QuIT has a more robust maintenance scheme that utilizes $po\ell e$ as well as its reset strategies that help effectively and efficiently predict the fast-path to inserting the data.

QuIT Outperforms ART for High Data Sortedness. Figure 10 also shows that QuIT outperforms ART by up to 1.53×. QuIT benefits from its ability to avoid tree traversals for workloads having high sortedness, directing ingestion directly to the leaf level. On the other hand, insertions in ART still traverse the tree. As data sortedness decreases, QuIT's performance moves closer to the B$^+$-tree, and ART performs up to 2× better, as tree traversals of a B$^+$-tree-based index (including QuIT) are more expensive than those of ART. While ART performs better for workloads with lower degrees of sortedness during ingestion, it is not a drop-in replacement for B$^+$-trees because it does not favor efficient range queries due to the lack of pointers between leaf nodes. This results in more tree traversals during a range lookup, as opposed to a B$^+$-tree (or QuIT). QuIT, in turn, can act as a drop-in replacement for B$^+$-trees, as it performs significantly better during ingestion without adding any overhead during point or range lookups. Note that our goal was not to outperform prior designs for all use cases but rather to create a sortedness-aware drop-in replacement for B$^+$-tree. Extending other data structures, including ART, to be sortedness-aware is left as future work.

QuIT Improves the Overall Memory Utilization. The *IKR*-based variable node split strategy in QuIT is crucial to reduce the overall memory footprint. Table 1 compares the normalized memory footprint of the indexes, where lower is better.

Table 1: Space reduction of QuIT over all B$^+$-tree baselines. $\ell i \ell$-B$^+$-tree and tail-B$^+$-tree are omitted because they have the same memory footprint as the basic B$^+$-tree.

Index	% unordered entries							
	0%	1%	3%	5%	10%	25%	50%	100%
QuIT	1.96×	1.5×	1.41×	1.32×	1.16×	1.09×	1.01×	1×

The memory utilization of the tail-B$^+$-tree and the $\ell i \ell$-B$^+$-tree is identical to that of classical B$^+$-tree, as they all split a full node in half. We observe that QuIT's memory footprint is up to 1.96× smaller due to better space utilization on the leaf nodes, as shown in Figure 12a. Here, we vary the data sortedness on the x-axis and show the average leaf node occupancy (i.e., # entries in a leaf node) as a fraction of the leaf capacity on the y-axis. The worst-case space utilization for the classical B$^+$-tree (and consequently, all other baselines) occurs when ingesting fully sorted data, as every node is only half full due to right-deep insertions. QuIT alleviates this memory overhead by variable splitting guided by *IKR*. With near-sorted data (1%≤K≤10%), the average leaf occupancy of the classical B$^+$-tree is between 51-54%, while QuIT offers an average leaf occupancy between 62-74%. The variable split strategy in QuIT allows for tightly packing the leaf nodes that receive ordered inserts through the $po\ell e$. Further, redistribution also increases leaf occupancy when $po\ell e_prev$ is less than half full, utilizing previously unused space. For less sorted or fully scrambled data, QuIT is able to match the leaf occupancy of the classical B$^+$-tree.

QuIT Does Not Incur Any Overhead For Point Lookups. QuIT performs exactly as many accesses during point lookups as a classical B$^+$-tree, as the lookup algorithms are identical. Hence, QuIT does not incur any overhead for point lookups. From Figure 12b, we observe that point lookups in QuIT can in fact, be slightly faster (≈ 2% on average) as the tree is smaller due to better space utilization. This results in the overall size of the index nodes in QuIT being smaller than the system's cache, effectively leading to a marginal performance improvements.

QuIT is Faster for Range Lookups. Range lookups are faster in QuIT, as shown in Figure 12c, when ingesting near-sorted data. We observe that range queries access up to 2× fewer leaf nodes (≈ 1.3× on average) compared to the B$^+$-tree when the ingested data has high sortedness (K≤10%). This benefit gradually declines as sortedness decreases since space utilization in B$^+$-tree also improves. However, even with low data sortedness ($K \geq 25\%$), range lookups in QuIT access 1.15× fewer leaf nodes than the B$^+$-tree, directly correlating to improved performance. Overall, QuIT benefits from the variable split and redistribute strategies that tightly pack the leaf nodes, whereas, the B$^+$-tree experiences its worst-case space amplification when ingesting data with a high degree of sortedness.

5.2 Sensitivity Analysis

We now vary several aspects of the workload, particularly the data size, and different sequences of insertions that stress test fast-path ingestion in the Quick Insertion Tree.

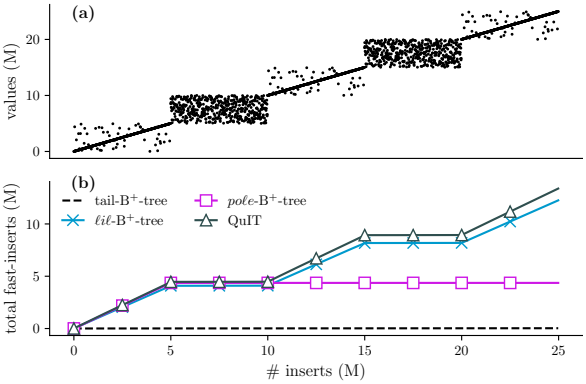

Figure 13: Stress testing the Fast-path optimizations under workloads with varying densities of data sortedness: (a) a workload that varies sortedness density by alternating between near-sorted and scrambled data in different data segments; (b) performance of tail, $\ell i \ell$, and $po\ell e$ optimizations in B$^+$-trees when comparing fast-insertions in QuIT.

5.2.1 Sensitivity Analysis on Data Size. Next, we analyze the scalability of QuIT by increasing the number of entries ingested into the index from 50M to 4B (scaling the data size from 40MB to 32GB). We pick candidate data collections reflecting three degrees of data sortedness - (i) fully sorted data (K=0%), (ii) nearly-sorted data (K=L=5%) that has a sizeable fraction of out-of-order entries, and (iii) less sorted data (K=L=25%) that has significantly many out-of-order entries. Table 2 summarizes our results when compared to the classical B$^+$-tree. We observe that QuIT's ability to utilize its fast-path and exploit sortedness in the ingested workload remains unaffected as the data size grows. The observed fraction of fast-inserts is 100% for fully sorted data, ≈ 95% for nearly-sorted data, and ≈ 75% for less sorted data. This aligns with the expected optimal performance for a sortedness-aware index from Figure 7b. The speedup during ingestion, however, is slightly amplified when scaling the data size since the number of levels in the tree grows. Scaling data size results in longer tree traversals and, therefore, an increase in indexing cost (top-insert) and maintenance. QuIT continues to employ fast-path insertions enabled by $po\ell e$, offering an amortized indexing cost. Overall, QuIT is at least 2× faster than B$^+$-tree when ingesting workloads having near-sorted data.

5.2.2 Stress Testing the Fast-Path. We now explore the performance of the fast-path optimizations in tail-B$^+$-tree, $\ell i \ell$-B$^+$-tree, and $po\ell e$-B$^+$-tree (i.e., QuIT without variable split, redistribute, and reset strategies), along with the full design of QuIT for workloads that alternate between near-sorted and fully scrambled. We anticipate that such a workload will be harder to predict and use it to stress-test all fast-path optimizations and *IKR*. We ingest into each index 25M entries divided into 5 segments (of 5M entries),

Table 2: QuIT scales with data size.

Sortedness	Metric	Data Size (GB)					
		0.4	2	4	8	16	32
fully sorted	Speedup	3.13×	3.19×	3.23×	3.25×	3.27×	3.31×
	% fast-inserts	100%	100%	100%	100%	100%	100%
nearly sorted	Speedup	2.43×	2.52×	2.56×	2.57×	2.61×	2.77×
	% fast-inserts	95.2%	95.2%	95.2%	95.2%	95.2%	95.2%
less sorted	Speedup	1.31×	1.32×	1.33×	1.34×	1.35×	1.35×
	% fast-inserts	74.6%	74.6%	76.4%	74.6%	74.6%	74.6%

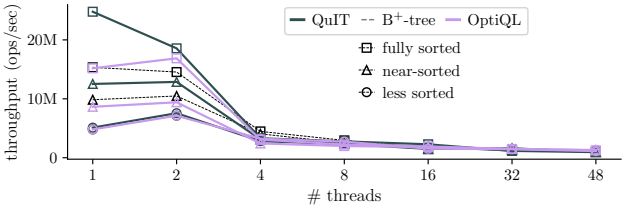

Figure 14: QuIT performs better than the baseline B$^+$-tree, and the state-of-the-art design in OptiQL for high data sortedness, under concurrent insertions into the index.

alternating between near-sorted data (K=10%) or scrambled data (K=100%), while we default L to 100%. We visualize this workload using the position (x-axis) and the value (y-axis) of the keys inserted in Figure 13a.

Figure 13b shows the number of entries ingested to the index using the fast path (on y-axis) at different snapshots on the x-axis (i.e., one for each segment). Note that a flat line in any segment implies that the index performed only top-inserts. We observe that the tail-B$^+$-tree very quickly reaches the stale state and fails to perform fast-path insertion as soon as the data becomes even slightly unordered. The $\ell i \ell$-B$^+$-tree continues to perform fast-path insertion when the data is nearly sorted, while only performing top-inserts for the scrambled data segments. Meanwhile, the $po\ell e$-B$^+$-tree marginally outperforms the $\ell i \ell$-B$^+$-tree in the first near-sorted data segment, and subsequently, only performs top-inserts for the remaining workload. This is because $po\ell e$ is trapped in a stale state after the first scrambled segment of data (between keys 5M and 10M). QuIT addresses the limitations of $po\ell e$ and recovers from this stale state with the help of its reset strategy (described in §4.3), allowing continued use of the fast-path when inserting near-sorted data. A robust fast-path ingestion strategy (like $po\ell e$) coupled with the reset strategy helps QuIT outperform $\ell i \ell$-B$^+$-tree by performing ≈ 11% more fast-path insertions.

5.3 QuIT under Concurrent Execution

We now benchmark QuIT and compare it with the classical B$^+$-tree as we have more threads concurrently using the two indexes. We also compare QuIT with the state-of-the-art optimistic locking design in OptiQL [46]. Here, we use optimistic lock coupling for B$^+$-tree, while we employ the locking protocol described in §4.5 for QuIT. We experiment with fully sorted data (K = 0%), near-sorted data (K = 5%), and less-sorted data (K = 25%), containing 500M entries (8B keys and 8B values).

QuIT Scales Better Than B$^+$-tree for Inserts. Ingesting data with a high degree of sortedness using multiple threads is expected to result in high contention because most insertions target the same leaf node. As a result, concurrently inserting with multiple threads hurts ingestion performance for both B$^+$-tree and QuIT. However, granular crabbing for QuIT leads to a shorter critical section, as we attempt to lock the $po\ell e$-node only when the specific insert uses the fast path. Figure 14 corroborates our expectations, showing that both indexes face contention as we increase the number of threads. Overall, QuIT offers up to ≈ 2× higher throughput than B$^+$-tree.

QuIT Scales Similar to OptiQL. Figure 14 further shows that QuIT outperforms the state-of-the-art optimistic lock coupling design used in OptiQL [46] when ingesting data with a high degree of sortedness for fewer than four threads while having

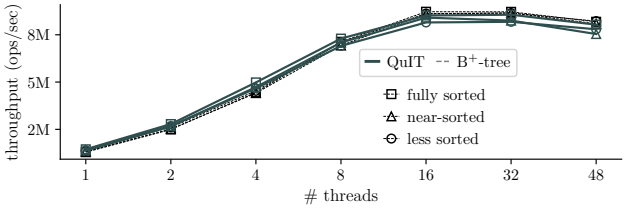

Figure 15: Concurrent Reads for both B$^+$-tree and QuIT have similar performance.

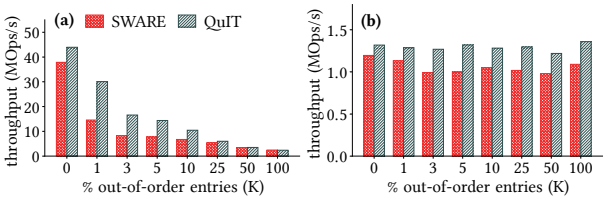

Figure 16: Comparing the SA-B$^+$-tree and QuIT: (a) A complex design allows SA-B$^+$-tree to opportunistically bulk load near-sorted data while QuIT maximizes fast insertions through index appends; (b) SA-B$^+$-tree incurs a read-overhead while QuIT is marginally faster than the B$^+$-tree.

similar performance for four or more threads. Note that in the OptiQL analysis [46], the authors observe that performance scales linearly as the number of threads increases. However, this experiment targets *updates* that virtually never alter the structure of the tree and, thus, rarely take exclusive locks for non-leaf nodes. On the other hand, the *insert-heavy* experiments we conduct significantly alter the structure of the tree and need to acquire exclusive locks when splitting internal nodes. As a result, lock coupling approaches like the ones employed by OptiQL and QuIT have long critical sections that hurt scalability when ingesting data with a high degree of sortedness.

QuIT Scales Similar to B$^+$-tree for Reads. As expected, read queries in both trees behave very similarly since QuIT's read path is essentially the same as the one of the classical B$^+$-tree. Figure 15 shows that both trees scale almost perfectly until 8 threads, while the scaling slows down for 16 threads and beyond.

5.4 Comparing with SWARE

We now compare QuIT with SA-B$^+$-tree [40], the state-of-the-art sortedness-aware index that employs the SWARE design. We ingest 500M entries (4GB) into both indexes and measure the throughput during insertions and 5M random point lookups (existing), when varying the fraction of out-of-order entries in the data collection (defaulting L=100%). Note that we utilize a more optimized B$^+$-tree (also used in QuIT) than the one provided with the SWARE codebase [43] as the underlying tree index in the SA-B$^+$-tree. SWARE originally packs a B$^\epsilon$-tree [6] that also functions as a B$^+$-tree when appropriately tuned. This, however, introduces orthogonal complexities and overheads that we avoid in our B$^+$-tree prototype. We compile both the index designs with the `-O3` optimization enabled.

QuIT Outperforms SWARE During Ingestion. Figure 16a shows that QuIT offers significantly better throughput than the SA-B$^+$-tree for any data sortedness. Even when ingesting fully sorted data, QuIT offers a 16% improvement over the SA-B$^+$-tree, as every insert can be directly appended to the fast-path node (*pole*). Despite SA-B$^+$-tree performing opportunistic bulk loading on-the-fly, it still pays a cost to index the data in the buffer through two levels of Bloom filters (the global and per-page Bloom filters) and the Zonemaps, in addition to costs associated with data movement in the buffer after every flush operation. Likewise, when ingesting near-sorted data ($K \leq 10\%$), QuIT is at least 1.55× (and 1.86× on average) better than SA-B$^+$-tree during ingestion. The additional costs incurred by the SA-B$^+$-tree to update the necessary metadata (i.e., identifying non-overlapping zones, Zonemaps, and Bloom filters) for facilitating opportunistic bulk loading far exceed the costs associated with maintaining QuIT's metadata, primarily due to the latter's lightweight index design. Thus, QuIT outperforms SA-B$^+$-tree during near-sorted data ingestion. Both indexes perform top-inserts for those entries that cannot utilize the fast-path optimization (opportunistic bulk loading in the case of SA-B$^+$-tree). As data sortedness decreases, top-inserts are more pronounced, thus, the average latency for an insertion increases for both indexes. QuIT and SA-B$^+$-tree are have comparable performance when ingesting less sorted ($K \geq 25\%$) or scrambled data.

QuIT's Benefits Come with No Query Penalty. QuIT does not incur any read overhead due to its lightweight design, while the SA-B$^+$-tree incurs an additional cost of scanning the buffer for the target key, as observed from Figure 16b. While the SA-B$^+$-tree employs additional data structures like Bloom filters and Zonemaps, as well as techniques like query-driven partial sorting to reduce this cost, it still pays an increased cost compared to QuIT, which only performs a B$^+$-tree lookup. Overall, QuIT outperforms the SA-B$^+$-tree by up to 32%.

5.5 Comparing with B$^\epsilon$-tree

Next, we compare QuIT with a textbook B$^\epsilon$-tree [6] previously open-sourced [43] with a workload with 500M insertions, executed using a single-thread, where we vary sortedness. We set up the B$^\epsilon$-tree with $\epsilon = 0.5$, and 4KB pages, similar to QuIT and report the observed throughput in Table 3. To ensure a fair comparison we exclude locking protocols in QuIT and integrate it with the buffer pool infrastructure of the B$^\epsilon$-tree prototype, only for this set of experiments.

Table 3: Throughput (MOps/sec) for the different indexes when varying data sortedness. QuIT offers higher throughput when compared to other baselines.

Index	% unordered entries							
	0	1	3	5	10	25	50	100
B$^\epsilon$-tree	4.60	2.23	1.7	1.58	1.31	1.11	0.86	0.71
QuIT	43.93	30.06	16.6	14.38	10.44	5.99	3.49	2.37

QuIT Outperforms the B$^\epsilon$-tree. We observe that QuIT significantly outperforms the B$^\epsilon$-tree during ingestion by up to 13×. While the B$^\epsilon$-tree asymptotically improves the ingestion cost by a factor of ϵ (amortized due to buffering entries in the internal nodes) when compared to the B$^+$-tree, entries are added to the index within the buffers of the internal nodes. Hence, it still incurs the cost of repeatedly sort-merging and flushing entries between its internal nodes. On the other hand, QuIT optimizes ingestion by utilizing a fast-path to directly place entries into the leaf level of the index when possible, reducing the index traversals. This results in significantly better performance than the B$^\epsilon$-tree. We would like to point out that the idea of sortedness-aware fast-path

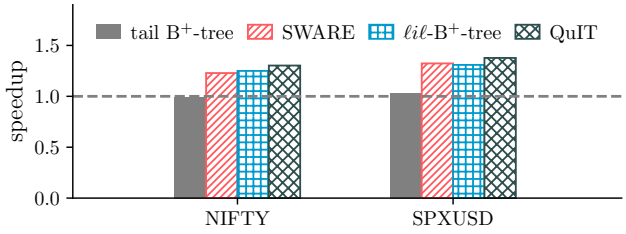

Figure 17: QuIT performs best for near-sorted ingestion on stock-market data, offering up to 1.3× speedup over the standard B$^+$-tree, which is higher than all other sortedness-aware designs (SWARE, tail-B$^+$-tree, and $\ell i \ell$-B$^+$-tree).

ingestion can also be extended to the B$^\epsilon$-tree, such that in-order entries are fast inserted. Out-of-order entries that would otherwise revert to top-insertions can take advantage of the amortized ingestion through the buffers, achieving the best of both worlds.

5.6 Indexing Real-World Data

Real-world data frequently exhibit a degree of near-sortedness that may be unknown or difficult to quantify, where the K-L metric may not be a natural descriptor, like data from stock-market applications, as discussed earlier in Figure 1a. We obtain the intra-day stock price data for the two tickers at one-minute timeframes (sources listed in footnote[2] and footnote[3]). Both datasets contain ≈ 1.4M entries and 2.2M entries, respectively. We see an overall upward trend that intuitively implies near-sortedness.

QuIT Offers Best Performance for Real-World Data. Figure 17 shows the speedup offered when ingesting the stock price data into the tail-B$^+$-tree, SA-B$^+$-tree, $\ell i \ell$-B$^+$-tree, and QuIT, normalized vs. the baseline B$^+$-tree. QuIT offers a ≈ 30% improvement on average during ingestion when compared to the tail-B$^+$-tree. In fact, our approach offers the maximum speedup among all baselines, even outperforming the state-of-the-art sortedness-aware index by ≈ 8% and 5% for the NIFTY and SPXUSD instruments, due to its lightweight design. Overall, tree indexes like B$^+$-tree-based designs benefit from sortedness-aware ingestion optimizations even when intrinsic data sortedness is hard to quantify or predict, as shown by the ingestion speedup offered by both SWARE and QuIT.

6 RELATED WORK

There is a plethora of B$^+$-tree variants optimizing for data ingestion. We recognize two design patterns aiming to reduce the cost of incremental data insertion: (i) approaches that optimize tree traversal and insertion operations by taking maximum advantage of modern hardware, and (ii) approaches that re-design the internal structure of the tree to amortize the cost of insertions across the workload. Specifically, the CSB-tree [44] resizes nodes to make all operations cache-conscious and minimize cache misses. The PLI-tree [48], BP-tree [50], T-tree [29], YATS-tree [26], Partitioned B$^+$-trees [20] and B$^\epsilon$-trees [6] adapt the data layout in nodes to their corresponding use case. For instance, B$^\epsilon$-trees trade-off fan-out for per-node buffers that can batch insertions. This allows the B$^\epsilon$-tree to amortize the ingestion cost, while the buffers are gradually flushed down the index. Finally, Bw-trees [31] are log-structured and take a latch-free approach using a delta update scheme with dynamically sized pages. The nodes of the Bw-tree are only logical and do not occupy fixed physical locations on main memory or storage. Bw-trees design eliminates thread blocking and is optimized for modern hardware.

These B$^+$-tree designs improve ingestion performance, however, they are unaware of prospective gains opportunities from taking advantage implicit sortedness of the incoming data. The SWARE indexing paradigm takes advantage of sortedness by using a combination of in-memory buffering and bulk-loading to optimize index ingestion [42]. However, its gains require a complex design that utilizes additional resources, adversely affecting the lookup performance. Meanwhile, QuIT offers sortedness-awareness index ingestion through a lightweight design and minimal metadata footprint.

Applicability to Data Streaming and Time Series. Time series indexing assumes that data ingestion follows an expected increasing order [28, 37, 51–53]. Streaming systems often use a buffer to capture the arrival skew within time-based windows [47] that allow for effective data series comparisons [10, 18]. QuIT eliminates the need for the additional buffer as in-order data will always be fast-inserted, leaving the arrival data skew to be captured by the fraction of top-inserts performed, as shown in §5.2.2.

7 CONCLUSION

Commercial data systems employ fast-path optimization techniques to amortize the cost of index construction during data ingestion. For B$^+$-trees fast-path ingestion helps avoid tree traversals, inserting entries directly to the tail leaf if the inserted data is fully sorted. However, this tail-leaf fast path becomes stale when the data is not fully sorted. We address this by proposing two new fast-path ingestion strategies for B$^+$-trees – $\ell i \ell$ and $po\ell e$ – that target near-sorted data. Further, we present QuIT, a lightweight index design that reduces the indexing cost proportionally to the sortedness of the indexed data. In addition, QuIT improves the memory footprint of the index, while also offering better lookup performance. Overall, QuIT outperforms the tail B$^+$-tree (and SWARE) by up to 2.3× (2×) when ingesting near-sorted data while offering on average 20% better space utilization when compared to the B$^+$-tree. The reduced memory footprint helps QuIT access up to 2× fewer nodes during range lookups, while it does not incur overhead for point lookups.

8 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive feedback. This work is funded by the National Science Foundation under Grant No. IIS-2144547, a Facebook Faculty Research Award, and a Meta Gift.

REFERENCES

[1] Daniar Achakeev and Bernhard Seeger. 2013. Efficient Bulk Updates on Multiversion 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] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 461–466.
[4] Rudolf Bayer and Karl Unterauer. 1977. Prefix B-trees. *ACM Transactions on Database Systems (TODS)* 2, 1 (1977), 11–26.
[5] 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.
[6] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. 2015. An Introduction to Bε-trees and Write-Optimization. *White Paper* (2015).

[2] https://github.com/aeron7/nifty-banknifty-intraday-data
[3] https://github.com/FutureSharks/financial-data

[7] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

[8] Timo Bingmann. 2007. STX B+ Tree. *https://github.com/bingmann/stx-btree* (2007).

[9] Burton H Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.

[10] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. Beyond Analytics: The Evolution of Stream Processing Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2651–2658.

[11] Svante Carlsson and Jingsen Chen. 1992. On Partitions and Presortedness of Sequences. In *Acta Informatica*, Vol. 29. 267–280.

[12] Badrish Chandramouli and Jonathan Goldstein. 2014. Patience is a Virtue: Revisiting Merge and Sort on Modern Processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 731–742.

[13] Badrish Chandramouli, Jonathan Goldstein, and Yinan Li. 2018. Impatience Is a Virtue: Revisiting Disorder in High-Performance Log Analytics. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 677–688.

[14] Douglas Comer. 1979. The Ubiquitous B-Tree. *Comput. Surveys* 11, 2 (1979), 121–137.

[15] CouchDB. [n. d.]. Online reference. *http://couchdb.apache.org/* ([n. d.]).

[16] Frederik M. Dekking, Cornelis Kraaikamp, Hendrik P. Lopuhaä, and Ludolf E. Meester. 2005. *A Modern Introduction to Probability and Statistics.* Springer London. 488 pages.

[17] 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.

[18] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2024. A survey on the evolution of stream processing systems. *The VLDB Journal* 33, 2 (2024), 507–541.

[19] Nikolaus Glombiewski. 2023. *Robust Stream Indexing.* Ph. D. Dissertation. Philipps-Universität Marburg.

[20] Goetz Graefe. 2003. Sorting And Indexing With Partitioned B-Trees. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

[21] Goetz Graefe. 2010. A survey of B-tree locking techniques. *ACM Transactions on Database Systems (TODS)* 35, 3 (2010).

[22] Goetz Graefe. 2011. Modern B-Tree Techniques. *Foundations and Trends in Databases* 3, 4 (2011), 203–402.

[23] Stratos Idreos and Mark Callaghan. 2020. Key-Value Storage Engines. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2667–2672.

[24] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

[25] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. 2011. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *Proceedings of the VLDB Endowment* 4, 9 (2011), 586–597.

[26] Chris Jermaine, Anindya Datta, and Edward Omiecinski. 1999. A Novel Index Supporting High Volume Data Warehouse Insertion. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 235–246.

[27] Donald E. Knuth. 1997. *The art of computer programming, Volume I: Fundamental Algorithms (3rd Edition).* Addison-Wesley.

[28] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2018. Coconut: A Scalable Bottom-Up Approach for Building Data Series Indexes. *Proceedings of the VLDB Endowment* 11, 6 (2018), 677–690.

[29] Tobin J. Lehman and Michael J. Carey. 1986. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 294–303.

[30] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 38–49.

[31] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 302–313.

[32] Heikki Mannila. 1985. Measures of Presortedness and Optimal Sorting Algorithms. *IEEE Transactions on Computers (TC)* 34, 4 (1985), 318–325.

[33] 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.

[34] MongoDB. 2023. Online reference. *http://www.mongodb.com/* (2023).

[35] MySQL. 2023. MySQL. *https://www.mysql.com/* (2023).

[36] Oracle. 2018. Introducing Oracle Database 18c. *White Paper* (2018).

[37] Themis Palpanas. 2015. Data Series Management: The Road to Big Sequence Analytics. *ACM SIGMOD Record* 44, 2 (2015), 47–52.

[38] PostgreSQL. 2023. PostgreSQL: The World's Most Advanced Open Source Relational Database. *https://www.postgresql.org* (2023).

[39] Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database Management Systems.* McGraw-Hill Higher Education, 3rd edition.

[40] 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.

[41] Aneesh Raman, Subhadeep Sarkar, Matthaios Olma, and Manos Athanassoulis. 2022. OSM-tree: A Sortedness-Aware Index. *CoRR* abs/2202.0 (2022).

[42] 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.

[43] Aneesh Raman, Subhadeep Sarkar, Matthaios Olma, and Manos Athanassoulis. 2024. *https://github.com/BU-DiSC/sware.*

[44] Jun Rao and Kenneth A. Ross. 2000. Making B+-trees Cache Conscious in Main Memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 475–486.

[45] Marc Seidemann, Nikolaus Glombiewski, Michael Körber, and Bernhard Seeger. 2019. ChronicleDB: A High-Performance Event Store. *ACM Transactions on Database Systems (TODS)* 44, 4 (10 2019).

[46] Ge Shi, Ziyi Yan, and Tianzheng Wang. 2023. OptiQL: Robust Optimistic Locking for Memory-Optimized Indexes. *Proceedings of the ACM on Management of Data (PACMMOD)* 1, 3 (11 2023), 1–26.

[47] Utkarsh Srivastava and Jennifer Widom. 2004. Flexible Time Management in Data Stream Systems. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. 263–274.

[48] Kristian Torp, Leo Mark, and Christian S Jensen. 1998. Efficient Differential Timeslice Computation. *IEEE Trans. Knowl. Data Eng.* 10, 4 (1998), 599–611.

[49] TPC. 2021. TPC-H benchmark. *http://www.tpc.org/tpch/* (2021).

[50] Helen Xu, Amanda Li, Brian Wheatman, Manoj Marneni, and Prashant Pandey. 2023. BP-Tree: Overcoming the Point-Range Operation Tradeoff for In-Memory B-Trees. *Proceedings of the VLDB Endowment* 16, 11 (7 2023), 2976–2989.

[51] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2014. Indexing for interactive exploration of big data series. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1555–1566.

[52] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2016. ADS: the adaptive data series index. *The VLDB Journal* 25, 6 (2016), 843–866.

[53] Kostas Zoumpatianos and Themis Palpanas. 2018. Data Series Management: Fulfilling the Need for Big Sequence Analytics. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 1677–1678.