2023 IEEE 39th International Conference on Data Engineering (ICDE) | 979-8-3503-2227-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICDE55515.2023.00117

# Indexing for Near-Sorted Data

Aneesh Raman
*Boston University*
aneeshr@bu.edu

Subhadeep Sarkar
*Boston University*
ssarkar1@bu.edu

Matthaios Olma
*Microsoft Research*
maolma@microsoft.com

Manos Athanassoulis
*Boston University*
mathan@bu.edu

***Abstract*—Indexing in modern data systems facilitates efficient query processing when the selection predicate is on an indexed key. As new data is ingested, indexes are gradually populated with incoming entries. In that respect, *indexing can be perceived as the process of adding structure* to incoming, otherwise unsorted data. Adding structure, however, comes at a cost. Instead of simply appending the incoming entries, we insert them into the index. If the ingestion order matches the indexed attribute order, the ingestion cost is entirely redundant and can be avoided altogether (e.g., via bulk loading in a B$^+$-tree). However, classical tree index designs do not benefit when incoming data comes with an implicit ordering that is *close to* being sorted, but *not* fully sorted.**

**In this paper, we study how indexes can exploit *near-sortedness*. Particularly, we identify *sortedness as a resource* that can accelerate index ingestion. We propose a new sortedness-aware (SWARE) design paradigm that combines *opportunistic bulk loading*, *index appends*, *variable node fill and split factors*, and an *intelligent buffering scheme*, to optimize ingestion and read queries in a tree index in the presence of near-sortedness. We apply SWARE to two state-of-the-art search trees (B$^+$-tree and B$^\epsilon$-tree), and we demonstrate that their Sortedness-Aware counterparts (*SA B$^+$-tree* and *SA B$^\epsilon$-tree*) outperform their respective baselines by up to $8.8\times$ (*SA B$^+$-tree*) and $7.8\times$ (*SA B$^\epsilon$-tree*) for a write-heavy workload in the presence of data sortedness, while offering competitive read performance, leading to overall benefits between $1.3\times$ – $5\times$ for mixed read/write workloads with near-sorted data. Overall, we highlight that SWARE can be applied to other tree-like data structures to accelerate index ingestion and improve their performance in the presence of data sortedness.**

***Index Terms*—Indexing, Data Sortedness, B$^+$-tree, B$^\epsilon$-tree.**

## I. Introduction

Database indexing sits at the heart of almost any data system ranging from full-blown relational systems [45] to NoSQL key-value stores [30]. Indexes help accelerate query processing both for analytical and transactional workloads by allowing efficient data access of selective (range or point) queries. Essentially, database administrators decide to build and maintain indexes on frequently queried attributes to improve query performance. This comes at the expense of space and write amplification [5], and the time needed to update the indexes.

**Indexing Adds Structure to Facilitate Queries.** We pay the cost of index construction and maintenance because it *adds structure* to the data, which in turn, allows for efficient queries. As shown in Fig. 1 with the thick black line, every data organization technique exhibits a fundamental tradeoff between its *read* and *write* cost. To achieve efficient *logarithmic search time* for point queries, a classical index would insert data in their correct position (*in-place insertion*, bottom right part of the figure). On the other extreme, if read queries are infrequent, then scanning is acceptable, and instead of inserting entries to an index, we can simply *append* them (leading to *scans*, in the top left part of the figure). Indexes like B$^+$-tree [25] and B$^\epsilon$-tree [8], or even simple online sorting via in-place insertion, navigate the read vs. write tradeoff. Since all data organization efforts essentially *add structure to an otherwise unstructured data collection*, one would expect they benefit when such structure - *data sortedness* - already exists.

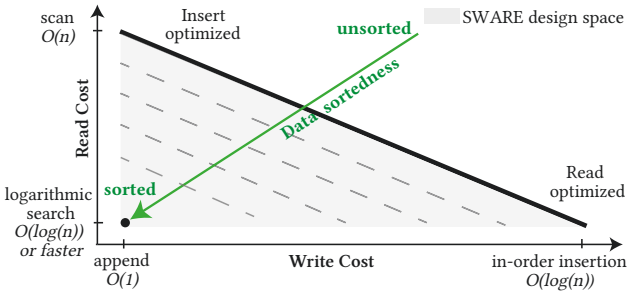


Fig. 1: State-of-the-art indexing and data organization techniques pay a higher write cost in order to store data as sorted (or, in general, more organized) and offer efficient reads. Since the goal of indexing is to store the data as sorted, we ideally expect that ingesting *near-sorted* data would be more efficient, which is not the case. We introduce the SWARE meta-design that offers better performance as data exhibit higher degree of sortedness.

**Data Sortedness.** There have been several efforts to quantify data sortedness [7, 16, 39], all of which essentially capture the difference between the *indexed order* and the *arrival order* of the indexed attribute (key). In practice, data entries may arrive as *near-sorted* in several real-world cases. Consider the TPC-H [51] `lineitem` table that has three date-related attributes. Fig. 2a depicts the first $10,000$ values of `shipdate`, `commitdate`, and `receiptdate` of the `lineitem` table, and shows that when data arrives as sorted on `shipdate`, the other two attributes are also very close to being sorted. There are several other scenarios that lead to near-sorted data collections. For example, (i) a relation that was sorted, but a few new arbitrary updates took place [7], (ii) data that has been created based on a previous operation, e.g., a join [7], (iii) data that is sorted based on another naturally correlated attribute (like the TPC-H example above) [4], or (iv) the timestamp attribute of an incoming data stream that has a few data packets arriving out of order due to network congestion.

**Problem: Lack of Sortedness-Awareness.** In this work, we show that state-of-the-art indexes like B$^+$-trees do not take advantage of existing order to improve ingestion performance

2375-026X/23/$31.00 ©2023 IEEE
DOI 10.1109/ICDE55515.2023.00117

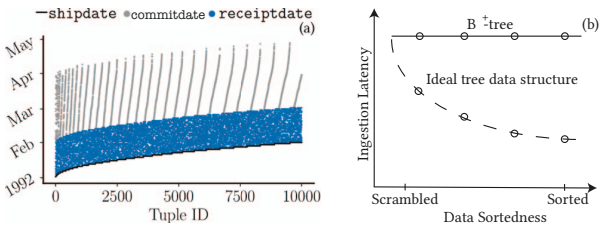


Fig. 2: (a) TPC-H implicit clustering between `shipdate`, `commitdate`, and `receiptdate` leads to near-sorted columns when the data is sorted based on one of them. (b) Ideally, index insertion performance should improve when inserting already sorted or near-sorted data.

and exhibit poor utilization of memory space. While indexes can already benefit from inserting a fully sorted data collection via bulk loading [1, 19], this is assuming that data is available at its entirety and its full sortedness is known *a priori*. However, for most practical use-cases that ingest data on the fly and build indexes online, bulk loading is infeasible, as the indexes are agnostic to the sortedness of the incoming (and future) data. We point out that when inserting data to an index, the *higher the data sortedness, the lower the insertion cost should be* for an *ideal tree data structure*, as shown in Fig. 2b. Note that indexes like B$^+$-trees do not exhibit any performance improvement (by design) when inserting near-sorted data. In fact, if data is inserted largely in near-sorted order a B$^+$-tree would have high space amplification, since, in the absence of bulk loading, every node will be exactly half full (as inserts are right-deep). In contrast, a *sortedness-aware index* should achieve better read vs. write tradeoffs (dashed lines in the shaded region of Fig. 1) as well as better space utilization when ingesting data with *increased sortedness* (that is, as we move closer to the origin of the green axis of Fig. 1).

*We envision a new class of index data structures that exploit sortedness to pay "less" indexing cost for near-sorted data.*

**Our Approach: SWARE.** Toward this, we establish data sortedness as a *fundamental resource* that can accelerate index ingestion. We propose a new index design paradigm that can exploit data sortedness to improve insertion performance without hurting query latency. We achieve this by using an ensemble of techniques that, when combined appropriately, lead to a better performance improvement than any one of them would do alone. Specifically, we employ an *intelligent buffering* mechanism that is periodically partially flushed to capture near-sortedness, combined with *opportunistic bulk loading* and *merging* techniques to create a *sortedness-aware* index. However, as is, the proposed approach comes at the cost of a nominal increase in the query latency, since every query may have to additionally search the buffer component. To alleviate this cost, we augment the buffer with a collection of Zonemaps [40], Bloom filters (BFs) [12], and *query-driven partial sorting*, that amortize query cost close to the baseline. The proposed sortedness-aware (SWARE) paradigm can be applied to any state-of-the-art tree index to form its sortedness-aware counterpart. We illustrate this by applying the SWARE paradigm to a B$^+$-tree and a B$^\epsilon$-tree to form their sortedness-aware equivalents: *SA B$^+$-tree* and *SA B$^\epsilon$-tree*. In a nutshell, they both buffer incoming data to opportunistically bulk load by reorganizing the out-of-order entries in memory, while reverting back to insertion from the root (*top-inserts*), otherwise. Further, by adaptively sorting buffered data during queries, *SA B$^+$-tree* and *SA B$^\epsilon$-tree* avoid the overhead of sorting large data chunks. To facilitate efficient query processing, they use interpolation search for the sorted parts of the buffer and pay only a constant cost of scanning a small amount of (unordered) data. Note that the SWARE paradigm can make any tree-based data structure amenable to data sortedness, and we use the B$^+$-tree and B$^\epsilon$-tree as two examples. SWARE is not a new index *per se*, rather, ***a new framework for creating sortedness-aware counterparts for any tree-based index.***

**Contributions.** Our work offers the following contributions.

- We identify *sortedness as a resource* that can be harnessed to ingest data faster in tree indexes.
- We propose a new index *meta-design* that employs buffering, partial bulk loading, and merging to enhance ingestion by exploiting data sortedness.
- We augment this design to propose the *SWARE paradigm* that encompasses query-driven sorting, merging, Zonemaps, and hierarchical Bloom filters to offer competitive performance for point and range queries.
- We apply the SWARE paradigm to a state-of-the-art B$^+$-tree, and we show that the *sortedness-aware B$^+$-tree* (*SA B$^+$-tree*) can achieve up to $8.8\times$ faster data ingestion with competitive read query performance leading to performance benefits of up to $5\times$ in mixed read/write workloads.
- Further, we also apply the SWARE meta-design to a B$^\epsilon$-tree to highlight that *SA B$^\epsilon$-tree* can achieve up to $7.8\times$ relative performance benefits against its standard counterpart.
- By applying the SWARE paradigm to a B$^+$-tree, we reduce space utilization by up to $48\%$.

## II. Background on Sortedness

In this section, we provide a brief background regarding data sortedness and sorting algorithms used in this work.

**Data Sortedness Metrics.** Multiple metrics have been proposed to capture data sortedness [2, 7, 16, 18, 22, 28, 39]. A natural way to quantify the *degree of sortedness* is to measure *how many elements* are out of place, and *by how much*. The $(K, L)$-sortedness metric [7] follows this idea using two parameters: $K$, which captures the number of elements that are out of order, and $L$, which captures the maximum positional displacement of the out-of-order elements.

**Sorting Algorithms.** Few sorting algorithms take advantage of the existing order in the input to optimize their performance [7, 15, 20, 34, 37, 44]. The *(K,L)-Sorting Algorithm* [7] is one such adaptive algorithm that uses the $(K, L)$-metric to sort a $(K, L)$-near sorted collection in at most two sequential passes. The algorithm has a complexity of $O(N \cdot log(K + L))$ (assuming $N$ entries) and uses memory of size $O(K + L)$. We take advantage of the adaptivity of the $(K, L)$-sorting

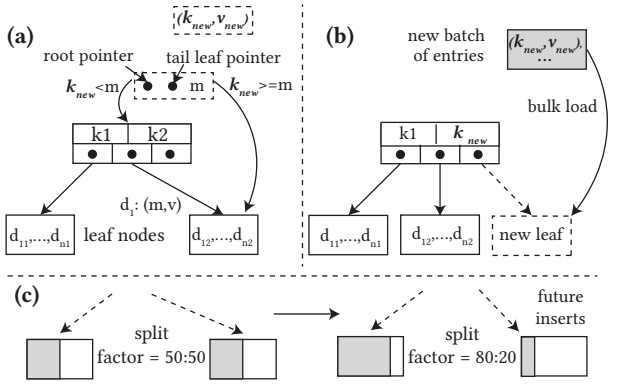


Fig. 3: Design elements aimed at exploiting data sortedness. (a) An in-order entry can be directly inserted at the tail leaf if $k_{new} \geq m$. (b) If we have a batch of new in-order inserts, they can be bulk loaded to the tree. (c) The node split factor can be adjusted, so that a newly created node from the split reserves more space for following inserts.

algorithm to quickly structure data in the buffer. Next, we discuss how the various degrees of sortedness, used in the paper, map to the the $(K, L)$-metric's domain.

**Degrees of Sortedness.** While a data collection may have arbitrary values for $K$ and $L$, we qualitatively define four degrees of data sortedness for the majority of our experiments: *fully-sorted* (where data arrives completely in order), *near-sorted* (where data arrives mostly in order albeit with a few out-of-place elements), *less-sorted* (where data is arriving mostly out-of-order but still contains few sorted entries) or *completely-scrambled* (where data is arriving completely out-of-order). A data collection is *completely sorted* if either $K{=}0$ or $L{=}0$, while *near-sorted* data collections usually have low values for $K$ and $L$, high values for either $K$ or $L$ but a low value for the other. The latter is also near-sorted since either the unordered entries are very close to their actual positions or the unordered entries occur far from their actual positions but are very few. Data with high values for both $K$ and $L$ are *less-sorted*, and in extreme cases, are completely scrambled.

## III. Design Elements

We now present the four fundamental design elements, which when appropriately combined, allow a tree-index to enhance ingestion performance by exploiting data sortedness. We decompose the design elements into (a) elements improving ingestion for sorted data, and (b) elements that enhance ingestion for intermediate data sortedness. The first three (illustrated in Fig. 3), i.e., right-most leaf insertion, bulk loading, and fill/split factor adjustment, benefit as-is, a fully sorted data ingestion. When combined with buffering, it leads to a design that can exploit any intermediate (degree of) data sortedness in the ingestion workload to improve performance.

**Right-Most Leaf Insertion.** When inserting data in-order, we can avoid the logarithmic tree traversal cost by always maintaining a pointer to the right-most leaf node (tail leaf), as shown in Fig. 3a. Every in-order insert to the minimum key in the tail leaf, can directly be inserted into the tail leaf. This costs only $O(1)$, instead of $O(log_F(N))$ due to tree-traversal.

**Bulk Loading.** If the data is *fully sorted*, we can perform better than in-order insertion, by bulk loading [19], as shown in Fig. 3b. This way, we can avoid accessing a node for every entry, and amortize the insertion cost across $F$ entries. While bulk loading gives great index creation time if data is fully sorted, it cannot exploit intermediate data sortedness.

**Split Factor/Fill Factor Adjustment.** We can further optimize the shape of the tree by adjusting *how we split the internal and leaf nodes* when ingesting data. Specifically, for fully sorted data, the classical node split creates two half-full nodes, the left of which never receives any future inserts and leads to poor space utilization. In some cases, splitting the nodes evenly may increase the index height as well, leading to asymptotically increased access cost. Instead, if we anticipate data to arrive fully sorted (or near-sorted), we can decide to split leaving more space for future inserts. Suppose we split at an $80:20$ ratio, $80\%$ of the entries stay on the original split node and the newly created one will only hold $20\%$ of the data, as shown in Fig. 3c. This also allows the nodes to have a higher fill factor on average. Adjusting the split factor reduces the number of overall splits, and the resulting higher fill factor reduces the overall number of nodes, thus, improving insertion performance as well as reducing the memory footprint.

**Buffering.** The above techniques offer benefits only if the data is fully sorted. To maximize the benefits for intermediate sortedness, we need to buffer incoming data and propagate to the tree only those inserts that are in order. In §IV, we discuss how to employ intelligent buffering and in-memory sorting to provide efficient ingestion without hurting read performance, outlining the crux of the SWARE paradigm.

## IV. SWARE Meta-Design

We now present the *SWARE meta-design* that exploits data sortedness to enhance ingestion into tree-based indexes. We demonstrate its usefulness on a B$^+$-tree and a B$^\epsilon$-tree.

### A. Sortedness-Aware Ingestion

Right-most leaf insertions and bulk loading help when ingesting fully sorted data, however, having any intermediate sortedness reduces their benefit. Thus, we intercept all index insertions and add them to a dedicated in-memory buffer. The buffer is a principal component of the SWARE framework and sits on top of the basic index (shown in Fig. 4). Specifically, the buffer performs opportunistic bulk loading via partial flushing. Below, we describe the buffering mechanism in detail.

**The SWARE-buffer.** The *SWARE-buffer* is an intelligent in-memory buffer that maintains all recently inserted data and checks whether the entries are in order. To ensure its contents are always in memory we pin its pages in the system's bufferpool. The data in the buffer is eventually inserted into the index either through *bulk loading* in an opportunistic manner (for entries with higher values than the data already in the index), or through *top-inserts* via the root node of the tree. By having this design, we can already guarantee that if data is inserted in order, they will be efficiently bulk loaded. We now

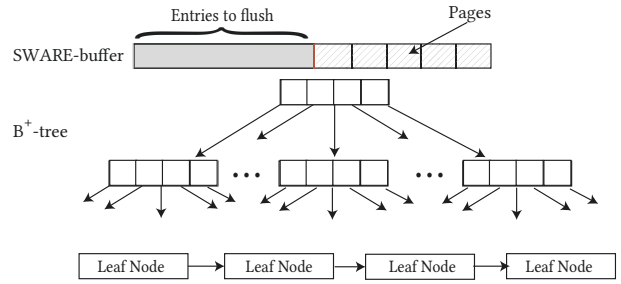


Fig. 4: The design of *SA B$^+$-tree* with a buffer atop a state-of-the-art B$^+$-tree.

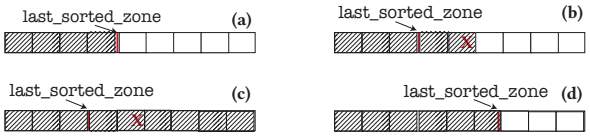


Fig. 5: The lifecycle of an insert in SWARE-buffer. (a) `last_sorted_zone` moves as new ordered entries are inserted. (b) An out-of-order entry moves `last_sorted_zone` to the left. (c) Newer inserts are added and `last_sorted_zone` only moves if required. (d) After a flush, the remaining entries are sorted and `last_sorted_zone` is reset.

discuss the buffer flushing strategies that optimize ingestion performance in the presence of a varying degree of sortedness.

**Flush Strategy.** Once the buffer is full, we *flush* it to the index either by *bulk loading* or through *top-inserts*. Our goal is to maximize the data inserted via bulk loading. When the buffer is fully sorted by virtue of pre-existing data sortedness, we bulk load the buffer contents with no sorting effort. Otherwise, we would need to sort it before flushing. At this point, (i) if the tree has strictly smaller values than the (now sorted) buffer, we bulk load as many pages as possible, and then (ii) top-insert overlapping entries through the root node.

Another decision we make at every flush cycle (i.e., every time the buffer is full) is *what proportion of the buffer to flush*. If we flush the entire buffer, we may insert entries that overlap with future inserts (if data does not arrive in fully sorted order). This would increase the number of top-inserts in subsequent flush cycles. Thus, we only flush a fraction of the buffer. We always flush the first eligible fraction of the buffer, and after a flush, the remaining buffered entries are sorted and moved to the beginning of the buffer to accommodate new inserts.

**Zonemaps to Identify Overlaps.** After a flush, the buffer is partially full (at least half) and its contents are fully sorted. Hence, we mark the last page containing sorted data as the `last_sorted_zone` (Fig. 5a). Due to the potential displacement, a new entry may either (i) overlap with data in earlier pages, hence moving the `last_sorted_zone` to the left (Fig. 5b), (ii) overlap without having to move the `last_sorted_zone` (Fig. 5c), or (iii) strictly greater, thus will move the `last_sorted_zone` to the right (Fig. 5d). Zonemaps maintained at the granularity of a buffer page help in performing a quick overlap test after every insertion to avoid unnecessary sorting at every flush. This amortizes the cost of sorting already ordered non-overlapping incoming data entries.

Once the buffer is full, we use the `last_sorted_zone` to decide how much to flush. If the `last_sorted_zone` is in the first half, we flush all pages up to this marker and attempt to bulk load. This way, we avoid the sorting cost *before* flushing. Otherwise, we flush half the pages in the buffer. By *partially* and *opportunistically* flushing non-overlapping entries, we amortize the potential top-inserts and maximize bulk loading, thus, improving the ingestion performance.

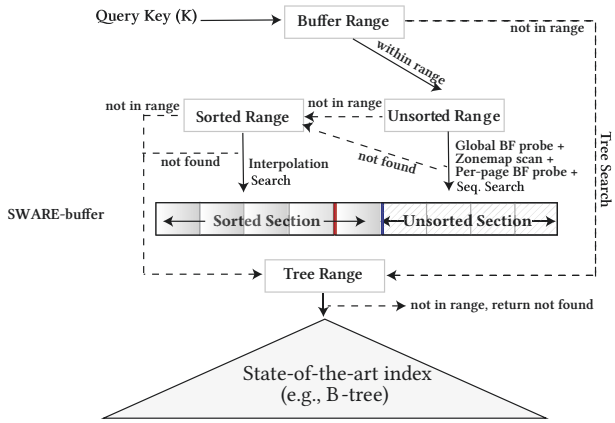


Fig. 6: The lifecycle of a point query.

### B. Optimizing Read Queries

While the SWARE-buffer helps to harness the sortedness by maximizing opportunistic bulk loading, its presence adds an overhead to the read performance. Specifically, every query first searches the buffer, and if it did not terminate in the buffer, it performs a tree search. In the worst case, a read query will need a full scan of the buffer. We now discuss how to reduce this query cost overhead to something nominal.

**Scanning the Unsorted Section First.** In steady-state, the SWARE-buffer can be in one of two states: (i) fully sorted or (ii) with a sorted portion and an unsorted portion. Note that even if the `last_sorted_zone` is moved to the beginning of the buffer (due to an out-of-order entry overlapping with the tree), we still have at least half of the buffer sorted (as we flush at most half the buffer). So, for any point query, we only need to scan the unsorted portion of the buffer that contains the most recent data. If the target key is not found in this part of the buffer, we continue to efficiently search the sorted section of the buffer, and if the lookup has still not terminated, we search the tree. However, for a key that is found in the buffer, we terminate early and avoid the cost of searching the tree, as the most recent version of the key would be the one in the buffer. The lifecycle of a point query is shown in Fig. 6.

*A BF to Skip the Unsorted Section.* The unsorted section is up to half of the SWARE-buffer, and thus, holds a small fraction of the overall data (residing in the buffer and the tree). As a result, queries without any temporal locality, have a low expectation to terminate in the buffer. Thus, to avoid the cost of unnecessary scanning the unsorted section, we maintain a BF for this section, that is continuously updated as new entries are inserted. This drastically reduces the cost of queries that do not terminate in the unsorted section.

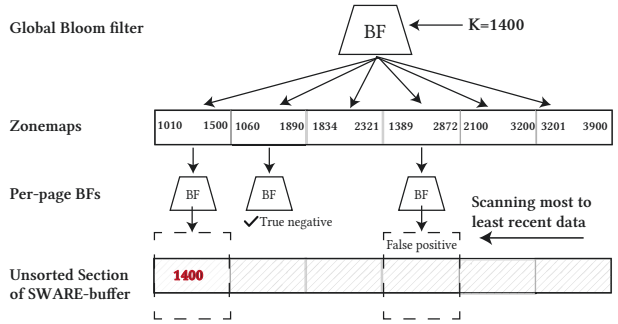


Fig. 7: In the unsorted section, a query first probes the global BF. If the probe returns positive, it checks the Zonemaps and then the qualified per-page BFs, before it scans any page.

*Zonemaps to Skip Pages in the Unsorted Section.* When a BF probe returns positive, all pages of the unsorted section are marked for scanning. However, we skip unnecessary page accesses using the Zonemaps that are already part of the SWARE-buffer (used to identify the `last_sorted_zone`).

*Using Per-Page BF.* While the BF and Zonemaps help avoid many unnecessary accesses, they are insufficient in case of lower data sortedness. Hence, we also maintain a BF for every buffer page. These are updated as data is appended to the buffer. Overall, a query starts searching in the unsorted section by first visiting the *global* BF. In Fig. 7, we show with an example that for a point query on key 1400, if the BF returns a positive result, we access the Zonemaps to find which pages may contain the target key. Subsequently, for the Zonemaps that may contain the key, we probe the corresponding per-page BF and we access only the qualifying pages. We discuss further details about the filter configuration in §V.

**Interpolation Search to Search Sorted Section.** After searching the unsorted section, if the query has not yet terminated, it will search in the sorted section. Regardless of whether the newly ingested data overlaps with the sorted section (and thus, have moved the `last_sorted_zone`), the data retained after the previous flush is kept sorted, and we maintain the position marking this sorted section, as `previous_boundary`. While the `last_sorted_zone` may move to the left, as new entries are inserted into the buffer, the `previous_boundary` may only move rightward as long as entries are inserted in fully sorted order and until the first out-of-order entry is inserted. We employ interpolation search [43, 52] on the sorted section, that finishes in $O(log(log(N)))$ steps. This is a notable upgrade from the binary search and is highly efficient unless there is a very high data skew, in which case we can opt for a simple binary search or a variation of exponential search [9].

**An Optimized Read Path.** Putting it all together, we have now an optimized read path that avoids the vast majority of unnecessary data accesses. As Fig. 6 shows, we maintain two more Zonemaps: one for the SWARE-buffer and one for the tree. If the desired key is not in the range of the buffer, we skip the buffer entirely. On the other hand, in the worst case, we may have to access the unsorted section of the buffer. However, due to the per-page BFs, even if the data is completely scrambled, we will only access a very small number of pages from the unsorted section. We discuss more optimizations and range queries in §IV-C.

### C. Fine-tuning a Sortedness-Aware Index

The SWARE meta-design introduces new components and tuning knobs that can be carefully calibrated to further improve performance of a sortedness-aware index.

**Choice of Sorting Algorithm.** To reduce the cost of reads, we sort the buffer after every flush. Ideally, we want the sorting cost to be minimal to attain the maximum benefits of the SWARE paradigm. While any sorting algorithm that leverages data sortedness (e.g., TimSort [44], Replacement Selection Sort [34]) can be used, here we consider three sorting algorithms: (i) *quicksort*, as it is common and has minimal space requirements, (ii) $(K, L)$-*adaptive sorting* [7], as it aggressively takes into account pre-existing data sortedness with $O(K + L)$ space usage, and (iii) *mergesort* (specifically, the C++ standard library `std::stable_sort`), as it maintains relative order of duplicate values with $O(n)$ space usage. Because we need to maintain the relative order of duplicates, we are constrained between mergesort and $(K, L)$-adaptive sorting. Our experimental analysis shows that for low data-sortedness, mergesort outperforms $(K, L)$-adaptive sorting (in fact, $(K, L)$-adaptive sorting fails for significantly high values of $K$ or $L$). However, for $K < 20\%$ or $L < 5\%$, their performance is similar, and we opt for $(K, L)$-adaptive sorting due to its **smaller space requirements** $(K + L < n)$ [7]. So, when the estimated (meta-data) values are $K < 20\%$ or $L < 5\%$ of the buffer size we employ $(K, L)$-adaptive sorting while using `std::stable_sort`), otherwise.

**Query-Driven Sorted Components.** We further employ a new read optimization technique to adaptively add structure to the unsorted part of the buffer through incoming queries, called *query-driven partial sorting* (inspired by Cracking [31, 32] and adaptive merging [27]). This technique increases the number of sorted sub-components within the buffer (called *query-sorted blocks*) that can be probed quickly using the faster interpolation search. We set a threshold (`query_sorting_threshold`) to restrict the size of the unsorted part of the buffer. If this threshold is exceeded, the next read query will sort this portion and create a new sorted component, before continuing to buffer incoming inserts. Similar to progressive indexing [29] that allocates a small indexing budget for every query, we allocate a small sorting budget for every query as long as we have enough entries in the unsorted component. In general, the SWARE-buffer may contain the main sorted section, multiple sorted sub-components (query-sorted blocks) of size equal to `unsorted_threshold`, and a small unsorted section. For example, if the `query_sorting_threshold` is set to $10\%$ of the buffer size, the buffer can contain up to four query-sorted blocks in addition to the remaining unsorted section and the main sorted section, as shown in Fig. 8. Note that the remaining unsorted section still uses all the metadata discussed

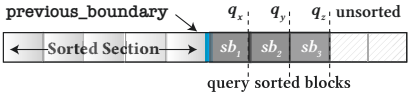


Fig. 8: As data x in the unsorted buffer above a threshold, an incoming query sorts this part as a standalone sorted component, to benefit future queries.

in §IV-B. Once the buffer is full and needs to be sorted (either before or after flushing qualifying entries), we sort the small unsorted component, while the remaining sorted blocks are merged. This approach further amortizes the ingestion cost.

**Supporting Range Queries.** Every range query first checks the SWARE-buffer Zonemap and if there is no overlap, the query executes by only accessing the tree. Otherwise, we begin by collecting the qualifying entries from the unsorted section. To do so, we first sort the (previously) unsorted section. We avoid re-sorting this section via a dedicated flag, which is reset upon receiving a new out-of-order insert. After retrieving entries from the unsorted section, we merge them with the qualifying entries from the sorted section and the query-driven segments (if any). The collected entries are then sort-merged with qualifying entries from the tree and returned.

**SWARE-buffer Size.** The size of the in-memory buffer is crucial to both ingestion and query performance of a sortedness-aware index. Our goal is to have a large enough buffer that can capture sortedness (specifically focusing on $L$) to maximize opportunistic bulk loading. However, a large buffer implies costlier reads, since the cost of scanning the unsorted part will be significant. Thus, it is crucial to strike the right balance to obtain the ideal performance. In §V, we vary the buffer size and relative values of $K$ and $L$, and show that even with a buffer significantly smaller than $L$, we can absorb sortedness to a large degree, without hurting queries.

**Adjusting Split & Fill Factors.** The textbook bulk loading algorithm used in §IV-A fills every node with the bulk loaded data to maximize node utilization (and thus, minimizes space amplification). Since we anticipate several top-inserts (the fraction of which depends on the sortedness of the workload), we also leave in every bulk loaded node a few empty slots (5%) to facilitate top-inserts without expensive cascading splits.

Similarly, in textbook insertion and bulk loading, an internal node when full splits at $50\%$ to generate two half-full internal nodes. Since during bulk loading, we anticipate that most of the future inserts will be of larger values, we also adjust the split factor to more than $50\%$, as shown earlier in Fig. 3c. This maintains most of the internal nodes of the underlying tree nearly full, even when the data is coming fully sorted, and most importantly, allows us to avoid the worst-case space amplification of B$^+$-trees for sorted data. In addition to reducing space amplification, we also reduce the number of node splits, lowering the overall insertion cost.

### D. Discussion

**Deletes in the SWARE-buffer.** Every delete in a *SA B$^+$-tree* is inserted as a tombstone in the SWARE-buffer, if within the buffer's range of indexed keys. Deletes outside the buffer's range and within that of the tree are directly applied to the tree. Here, we inherit a *delete debt* that is paid off during a flush. During data reorganization (e.g., due to query-driven sorting or during a flush), tombstones discard any invalidated keys. When flushing the buffer, tombstones are propagated to the tree as classical deletes through the root node.

**Concurrency Control in the SWARE-buffer.** Concurrency control for B$^+$-tree has been extensively studied in the past [6, 10, 24], so here we focus on the updates needed to the SWARE-buffer. Inserts to *SA B$^+$-tree* are appends, with the exception of an insert that causes a flush. Every insert instantaneously takes a lock on the entire buffer to check if it will cause a flush. If no flush is triggered, the buffer-wise lock is released, and the worker locks only the page to append (similar to lock-crabbing [24]) and the corresponding metadata (Global BF, Zonemaps of the updated page, and `last_sorted_zone`). Note that the page-wise lock protects all page-level metadata. If a flush is triggered, the buffer-wise *exclusive* lock will be held until the flush is complete. All retained locks are released post-insertion. Queries by default acquire *read* locks. However, when query-driven sorting (inspired by Cracking [31]) is triggered, we upgrade the read lock to an exclusive lock (in a similar manner that concurrent read queries in adaptive indexing require concurrency control [26]). These mechanisms allow multi-threaded execution where inserts and queries on *SA B$^+$-tree* are issued concurrently.

## V. Experimental Evaluation

We illustrate the benefits of the SWARE paradigm by applying it to a B$^+$-tree and B$^\epsilon$-tree to form their sortedness-aware counterparts: *SA B$^+$-tree* and *SA B$^\epsilon$-tree*. We present the performance evaluation of the *SA B$^+$-tree* in §V-A-§V-F. We then demonstrate the benefits of *SA B$^\epsilon$-tree* in §V-G. Lastly, we experiment with TPC-H [51] data to compare the performance of a B$^+$-tree and a *SA B$^+$-tree*.

**Experimental Setup.** We run the experiments in our in-house server equipped with two sockets each with an Intel Xeon Gold 5230 2.1GHz processor with 20 cores and virtualization enabled. The server has 384GB of RDIMM main memory at 2933 MHz with a 240GB SSD and runs CentOS 8.

**Index Design.** We use an in-house B$^+$-tree (inspired by the state-of-the-art implementation [11]) and B$^\epsilon$-tree [8] implementations that support opportunistic bulk loading and variable fill/split factors. Note that the B$^\epsilon$-trees internal node buffer is unaffected by the variable fill/split factors. For the B$^\epsilon$-tree, we use $\epsilon$=1/2 [8]. Both the B$^+$-tree and B$^\epsilon$-tree implementations are equipped with a bufferpool of 300GB, so most experiments are purely in-memory. The bufferpool is orthogonal to the SWARE-buffer, which is always in main memory. The default entry size is 8B (4B key), and we use a 4KB index page size. Our code is available at https://github.com/BU-DiSC/sware.

**Default Setup.** By default, the size of the SWARE-buffer is 40MB which can hold up to 5M entires. The buffer is implemented as a dense array accompanied by the Zonemaps and BFs. We maintain filters at two levels: (i) global BF and (ii) per-page BFs. For the global BF we pre-allocate 10 bits-per-

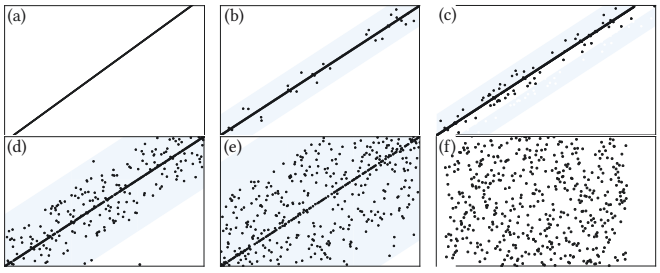


Fig. 9: Workloads with varying *sortedness*: (a) sorted, (b) K=10%, L=10%, (c) K=20%, L=10%, (d) K=50%, L=25%, (e) K=100%, L=50%, and (f) scrambled (uni-rand). x-axis: position of entry in data, y-axis: entry-value. The scattered points represent $K$ whereas the gray-band denotes the $L$.

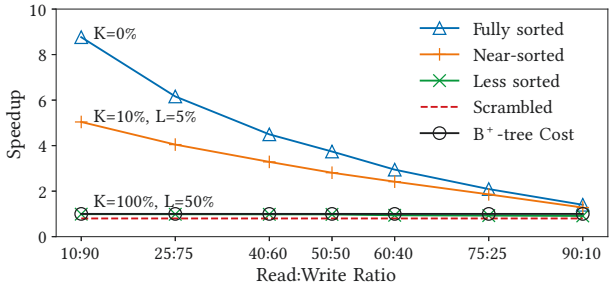


Fig. 10: *SA B$^+$-tree* (Buffer size=5M) is efficient with reasonable data sortedness for any read-write ratio.

entry (giving $\sim 0.8\%$ false-positive rate) for the entire buffer's capacity, while for the per-page BFs, we pre-allocate 10 bits-per-entry for a page's capacity. We use *MurmurHash* [3] combined with hash sharing and bit rotation [53] for hashing.

**Data Sortedness Benchmark & Workload.** We use the Benchmark on Data Sortedness [46] for testing indexes against varying sortedness. The benchmark uses the $(K, L)$-near sorted metric and creates a family of differently sorted collections that vary in both $K$ and $L$ (as a fraction of the total data size). Fig. 9 shows a sample set of differently sorted collections, represented by position and value. For performance evaluation, the benchmark measures: (i) raw ingestion latency, (ii) overall operational latency of a mixed workload with variable read/write ratio.

We use the benchmark's workload generator to create ingestion workloads with varying sortedness. Unless otherwise mentioned, the ingestion workload consists of 500M key-value entries with a total size of 4GB, and the query workload has a variable number of uniform random non-empty point lookups, interleaved with inserts after 80% of the ingestion is complete.

**SWARE Tuning.** By default, *SA B$^+$-tree* and *SA B$^\epsilon$-tree* are tuned as follows. The SWARE-buffer flushes 50% of the entries when saturated. The nodes split at an 80:20 ratio, and the opportunistic bulk loading fills every leaf up to 95%. By default, we set the query-based sorting threshold to 10%.

### A. Mixed Workload

We first compare the performance (using the offered speedup) of *SA B$^+$-tree* with the B$^+$-tree by executing a set of mixed workloads with interleaved inserts and queries. We vary the read-write ratio, constructing a continuum between a write-heavy and a read-heavy workload. For each workload, we also vary the data sortedness as: (i) *fully sorted* (K=0%), (ii) *near-sorted* (K=10%, L=5%), (iii) *less sorted* (K=100%, L=50%), and (iv) *scrambled* (uniformly random).

***SA B$^+$-tree* Outperforms B$^+$-Tree.** Fig. 10 shows that *SA B$^+$-tree* significantly outperforms B$^+$-tree if the data is fully sorted or near-sorted. For an ingestion-heavy workload, *SA B$^+$-tree* leads to 8.8× speedup for fully sorted data and 5× better for near-sorted data in ingestion-heavy workloads. *SA B$^+$-tree* achieves this by buffering entries in-memory to add structure to the data, thus, reducing the number of top-inserts. Fig. 11 shows that *SA B$^+$-tree* performs significantly fewer top-inserts for workloads with a high degree of sortedness. With fully sorted data, entries are ingested only through the bulk insertion, while for near-sorted data, only ∼4% are top-inserts.

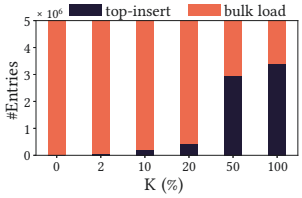


Fig. 11: For higher $K$%, *SA B$^+$-tree* performs more top-inserts and bulk loads fewer entries.

As data becomes less sorted, *SA B$^+$-tree* mimics the behavior of a B$^+$-tree. Now, *SA B$^+$-tree*'s ability to capture the unordered entries using a relatively small buffer (1% of data size) diminishes, and top-inserts are comparable to a B$^+$-tree. Regardless of the sortedness, *SA B$^+$-tree*'s benefit is more pronounced for write-intensive workloads. Conversely, in Fig. 10 we observe that for a lookup-heavy workload (90% lookups), *SA B$^+$-tree* offers a speedup of 1.4× and 1.3× for fully sorted and nearly sorted data, respectively, as the significant performance benefits of *SA B$^+$-tree* during ingestion are countered by the lookup overhead incurred.

**Ingesting Scrambled Data Does Not Benefit from *SA B$^+$-tree*.** When the ingestion is scrambled, *SA B$^+$-tree* does not offer performance benefits. Specifically, when the data is generated uniformly random, using a B$^+$-tree is about 20% faster than *SA B$^+$-tree*, regardless the read-write proportion in the workload. This is a result of the finite buffer being unable to capture the (minimal) sortedness of the incoming data, that in turn, forces *SA B$^+$-tree* to always perform top-inserts. Consequently, the SWARE-buffer management cost (sorting the buffer, managing metadata, and probing BFs during lookups) does not pay off, however, it keeps the penalty to a modest 20%. This observation is consistent with our goals and our expectation. While *SA B$^+$-tree* is very useful for a varying degree of sortedness, for fully scrambled data, the worst-case guarantees of a classical B$^+$-tree are sufficient.

### B. Raw Performance

Next, we compare the *SA B$^+$-tree* and B$^+$-tree in terms of ingestion and query performance separately. In the ingestion workload, we vary $K$ while keeping $L$ constant, as varying $L$ would entail changing the buffer size as a proportionally to $L$.

*Setup.* We insert 500M entries (4GB) and subsequently perform 50M non-empty point lookups. To report the worst-

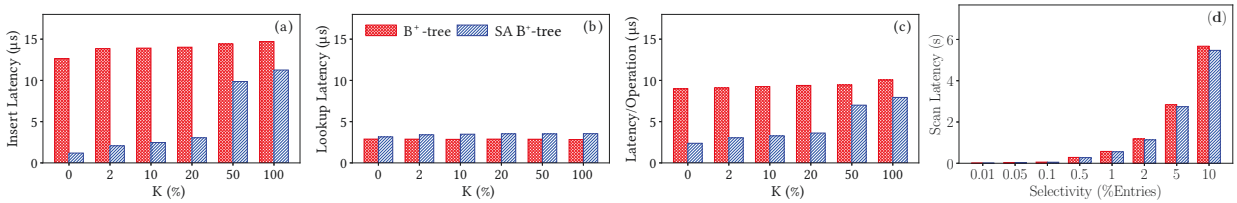


Fig. 12: Performance of *SA B$^+$-tree* (buffer size=40MB) for $L$=5%: (a) *SA B$^+$-tree* offers better ingestion performance with workloads with some degree of data sortedness. (b) *SA B$^+$-tree* incurs a small overhead for point queries. (c) For mixed workloads with equal proportions of reads/writes, the ingestion-benefits outweigh the lookup-overhead and offers better overall performance. (d) *SA B$^+$-tree* offers competitive performance for both short and long range scans.

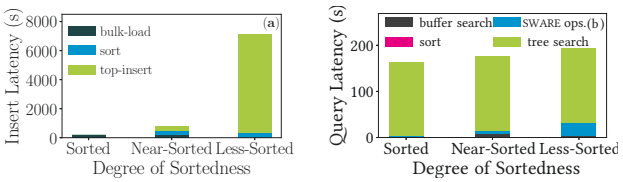


Fig. 13: Latency breakdown of operations in *SA B$^+$-tree*. (a) Time spent by *SA B$^+$-tree* for top-insert escalates when decreasing sortedness. (b) Tree search dominates query latency, while time spent managing metadata and maintaining Zonemaps + BFs (SWARE ops.) increases for lower data sortedness.

case lookup performance we ensure the buffer is full before executing any query. We execute 100 range queries generated randomly from the key domain as well as 100 range queries targeting the recently inserted data for various selectivities.

***SA B$^+$-tree* Dominates the Ingestion Performance.** Fig. 12(a) shows that *SA B$^+$-tree* performs significantly better than B$^+$-tree for inserts if there exists any data sortedness. While *SA B$^+$-tree* shows up to ∼90% improvement in ingestion latency for fully sorted data, it still manages an impressive 23% improvement over the B$^+$-tree's ingestion latency even as sortedness decreases. Fig. 13a shows the breakdown of the ingestion costs in *SA B$^+$-tree* for (i) fully sorted ($K$=0%), (ii) nearly sorted ($K$= 10%, $L$=5%), and less sorted ($K$=100%, $L$=50%) workloads. We observe that for fully sorted workloads, the *SA B$^+$-tree* is able to bulk load the entire data set without any additional processing. For near-sorted data, *SA B$^+$-tree* sorts the buffer periodically which accounts for 38% of the workload execution latency. However, the additional effort in establishing structure leads to significantly fewer top-inserts, which in turn, reduces the overall latency. Finally, for less sorted data, *SA B$^+$-tree* ingests fewer entries via bulk loading. Instead, a significant amount of data is ingested through top-inserts, resembling a B$^+$-tree. However, *SA B$^+$-tree* still outperforms the state of the art by a significant margin (Fig. 10). Note that *SA B$^+$-tree* achieves this performance with a buffer that is smaller in size when compared to $L$ (1% vs. 5%). This implies that even with a considerably small buffer that *does not capture all of the out-of-order entries*, *SA B$^+$-tree* performs significantly fewer top-inserts and is able to bulk load a large fraction of the data.

**Fast Ingestion Comes at a Small Cost for Queries.** Fig. 12b compares the point lookup performance of *SA B$^+$-tree* to that of the B$^+$-tree. We observe that *SA B$^+$-tree* incurs an overhead between ∼5% and ∼26% for point lookups. This is due to the additional time spent searching for the target key in the buffer. Fig. 13b shows the breakdown of the point query latency in *SA B$^+$-tree* for the same (i) fully sorted, (ii) nearly sorted and less sorted, workloads as before. Regardless of data sortedness, ∼80%-99% of the query latency is attributed to searching for the target key in the tree. With a full buffer, *SA B$^+$-tree* suffers an overhead due to (i) probing the SWARE-buffer using interpolation search and sequential scans and (ii) performing SWARE-operations like sort-merging the entries in the buffer and updating metadata. This overhead depends on the number of entries in the buffer and the data sortedness. For a fair comparison, we maintain a full buffer before executing the query workload; however, in practice, the buffer is expected to be 50% saturated on average, potentially cutting down the query overhead by half. Further, querying the buffer and the tree in parallel can potentially reduce the lookup cost.

**The Benefits Outweigh the Overhead almost Always.** Fig. 12c shows the mean latency per operation for a mixed workload. We observe that for a workload with equal reads and writes, *SA B$^+$-tree* improves the mean latency by ∼70% for fully and nearly sorted data. Even for workloads with lower sortedness, *SA B$^+$-tree* offers 1.25× improved overall performance. To summarize, for read-only workloads, the performance of *SA B$^+$-tree* is similar to that of B$^+$-trees, as the buffer remains empty, and thus, adds no overhead. However, if a mixed workload is read-dominated (writes $<$ 1%), the incurred read overhead outweighs the benefits of ingestion.

***SA B$^+$-tree* Offers Competitive Scan Performance.** Fig. 12d shows that *SA B$^+$-tree* performs similarly to B$^+$-trees for random range queries with different selectivity, varying from 0.01% (50K entries) to 10% (50M entries). Overall, *SA B$^+$-tree* offers an improvement of $3\% - 12\%$ in mean latency. For larger selectivities, the performance of *SA B$^+$-tree* remains comparable to the baseline even when considering P95 and P99 latencies. We only observe a maximum overhead of 1% at the P99 latency. For range queries that target the most recently inserted data, *SA B$^+$-tree* is $0.4\% - 8\%$ faster on average, and leads up to 7% lower P95 latency, and $0.1\% - 1.4\%$ lower P99 latency (graph omitted for brevity). With the higher fill factor in *SA B$^+$-tree*, we reduce leaf-scan costs that compensate for any overhead due to probing multiple components of the buffer, and thus, maintain a comparable range query performance to the baseline B$^+$-tree.

### C. Workload Influence

*Setup.* To measure the speedup offered by *SA B$^+$-tree* over the state-of-the-art B$^+$-tree for mixed workloads, we vary

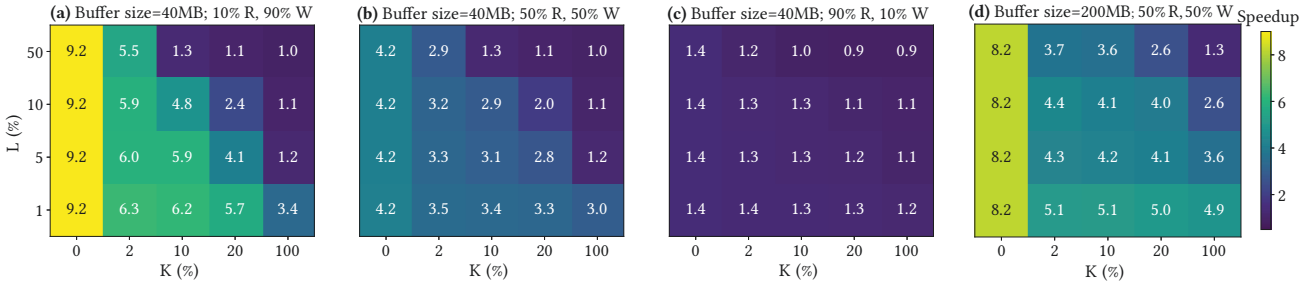


Fig. 14: Performance of *SA B$^+$-tree* with varying degrees of sortedness. (a) In a write-heavy workload, *SA B$^+$-tree* exploits data sortedness to offer maximum benefit in overall performance. (b) For more reads the benefit reduces. (c) *SA B$^+$-tree* performs similar to B$^+$-tree for read-heavy workloads with minimal performance benefits due to data sortedness. (d) A larger buffer in *SA B$^+$-tree* is better at capturing even higher sortedness, to improve overall performance.

both $K(\%)$=$(0, 2, 10, 20, 100)$ and $L(\%)$=$(1, 5, 10, 50)$ and experiment with 20 different degrees of data sortedness.

**Varying the Workload Composition.** We observe in Fig. 14a-c that as reads increase in the workload, the read-overhead incurred by *SA B$^+$-tree* counterbalances the ingestion benefits. Even for fully sorted data, increasing the reads from $10\%$ to $90\%$ reduces the speedup from $9.2\times$ to $1.4\times$. Fig. 14 thus serves as a guideline for applicability of the *SA B$^+$-tree* design.

**Varying the Degree of Sortedness.** Analyzing the speedup for $K$=$2\%$ (second column) and $L$=$1\%$ (fourth row) in Fig. 14a, we observe that $K$ influences the performance of *SA B$^+$-tree* to a greater extent compared to $L$. This is because if the buffer size is comparable to the $L$-value, $K$ drives the cost of data reordering, and relative overlaps between buffer cycles (causing top-inserts) are minimal. However, as $L$ gets larger, it impacts the overall speedup more significantly. With an increase in both $K$ and $L$, the *SA B$^+$-tree* operates similar to B$^+$-trees, and the speedup approaches 1.

### D. SWARE-buffer Tuning

Setup. Here, we first increase the buffer size to $5\%$ of the data size and compare the results with those discussed in §V-B. Next, we run a workload with 500M inserts followed by 50M lookups, and vary the buffer between $0.5\%$-$5\%$ of the data size. For a mixed workload, we pre-load the index (to $80\%$) and interleaved equal reads and writes with varying sortedness.

**Increasing the Buffer Size Improves Performance.** Increasing the buffer size allows us to opportunistically bulk load a larger fraction of the data. Fig. 14d shows the speedup offered by *SA B$^+$-tree* when SWARE-buffer size is increased to $5\%$ of the data size (200MB) for a mixed workload ($50\%$W-$50\%$R). Comparing with Fig. 14b (buffer size of $1\%$), we observe that the $5\times$ increase in the buffer size further increases the overall speedup to $8.2\times$ (a $95.2\%$ increase) for a fully sorted data; between $27.6\%$ and $176.9\%$ for nearly sorted data; and improves the speedup to $1.3\times$ for lower data sortedness.

**Buffer Size Affects the Ingestion Performance Significantly.** Fig. 15 shows that the buffer size affects the ingestion and lookup performance for *SA B$^+$-tree*. We vary the buffer size for a fixed sortedness ($K$= $10\%$, $L$=$5\%$), and observe that even with a small buffer equivalent to $0.5\%$ of the data size, *SA B$^+$-tree* offers a $5.7\times$ speedup during ingestion. As the buffer size increases to $5\%$, the ingestion speedup increases to $7\times$ due to increased opportunistic bulk loading. However, query performance in *SA B$^+$-tree* is marginally affected by the buffer size. A $10\times$ increase in buffer size increases query latency by $11\%$. This validates our observations from Fig. 12b and 14c regarding the applicability of the SWARE paradigm; however, the benefits of *SA B$^+$-tree* (Fig. 10) outweigh the read-overheads even for a small fraction of writes ($\geq$$5\%$).

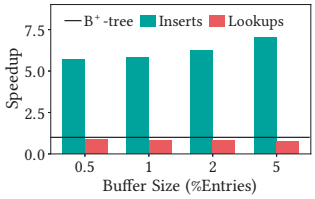


Fig. 15: The ingestion performance of *SA B$^+$-tree* increases with buffer size.

**Tuning the Buffer Flush Threshold.** Adjusting the flush threshold of the SWARE-buffer affects the overall performance of *SA B$^+$-tree*. We now vary the proportion of entries flushed from the buffer at a given cycle between $25\%, 50\%$, and $75\%$, and run mixed workloads. In the interest of space, we omit the figure and focus on key observations. When the buffer flush threshold is set to $25\%$, *SA B$^+$-tree* offers a speedup between $1.0\times$ and $4.0\times$. For flush threshold $50\%$, the speedup of *SA B$^+$-tree* ranges between $1.0\times$ and $4.3\times$, and for a threshold of $75\%$, between $0.91\times$ and $4.2\times$. Hence, *SA B$^+$-tree* performs best for $50\%$ flush threshold, which we default to.

**Tuning Query-Based Sorting.** Fig. 16 shows the implications of query-based sorting on *SA B$^+$-tree*'s overall performance. We vary the query-based sorting threshold between $1\%$-$100\%$ (i.e., query-based sorting is disabled for $100\%$) and run mixed workloads to compare the speedup against the baseline.

Employing query-based sorting offers a performance improvement between $7\%$ (for $1\%$ threshold) and $25\%$ (for $10\%$ threshold). As expected, gradually sorting the buffered data significantly accelerates query performance on average since the unsorted section is kept small. Moreover, we observe that query-based sorting has diminishing returns if applied too frequently. Specifically, a threshold of $10\%$ offers the

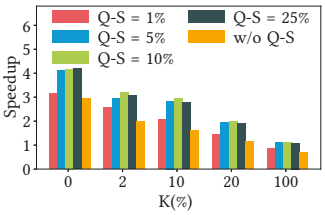


Fig. 16: Query-based sorting threshold set to $10\%$ offers the highest speedup.

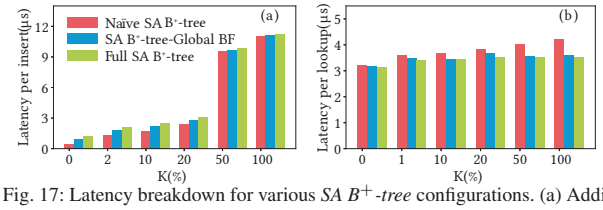


Fig. 17: Latency breakdown for various *SA B$^+$-tree* configurations. (a) Adding BFs to the SWARE-buffer slightly increases the insert latency. (b) The use of BFs in the buffer for lookups is more pronounced as data sortedness decreases.

maximum speedup for any data sortedness, while other values affect performance adversely (if at all). Reducing the threshold (to 1% or 5%) leads to too frequent sorting while increasing the threshold (to 25%) results in fewer sorted blocks so scanning the unsorted section remains expensive. Thus, we empirically tune *SA B$^+$-tree* to perform query-based sorting with the threshold at 10%.

**Benefits from the Global and Per-page BFs.** Here, we compare *SA B$^+$-tree* with two variations: one without any BF (*Naïve* SA B$^+$-tree) and one with only the global BF (SA B$^+$-tree-*Global BF*) to demonstrate the benefits of BFs. Updating the BFs for every insert marginally increases the ingestion latency (Fig. 17a). The added cost is a small fraction of the total insert time and is settled by the significant performance improvement during lookups. Fig. 17b shows that adding the global BF speeds up queries by up to 14%, while the per-page BFs boost performance further to 16%. The positive impact of per-page BFs is limited by query-based sorting that also helps avoid scanning unnecessary data (restricting scanning to <10%). Note that ingestion in *SA B$^+$-tree* is always significantly faster than B$^+$-tree regardless the BF cost. Overall, the benefit of BFs is pronounced with more reads.

**Tuning Zonemaps.** The SWARE-buffer uses Zonemaps during ingestion to approximate sortedness (§IV-A), thus, are integral to the overall design. While we opt to always use them at query time since they are always available, we observed that skipping Zonemaps for lookups reduces performance by 35%.

**Tuning Split Factor.** Table I shows the normalized number of leaf splits compared to the textbook split ratio of 50:50 in *SA B$^+$-tree*. Here, we vary the split ratios of the underlying tree index (B$^+$-tree) to split at 50%, 60%, 70%, 80%, and 90% when ingesting data with varied degrees of sortedness. While splitting the leaf node at 90% offers up to a 22% reduction in the total number of splits during ingestion for near-sorted data, this suffers a $\sim 1.8\times$ overhead when inserting data with lower sortedness. In fact, the textbook splitting at 50% works best with low data sortedness as it leaves enough space for unordered entries to be inserted without splitting. Overall, we observe *SA B$^+$-tree* offers the best performance across any data sortedness by splitting at 80%, which we use by default.

| Split Ratio | K= 2%, L= 1% | K= 20%, L= 10% | K= 100%, L= 50% |
|---|---|---|---|
| 50 : 50 | 1.00 | 1.00 | 1.00 |
| 60 : 40 | 0.90 | 0.97 | 0.94 |
| 70 : 30 | 0.82 | 0.96 | 1.04 |
| 80 : 20 | 0.79 | 0.96 | 1.27 |
| 90 : 10 | 0.78 | 0.98 | 1.82 |

TABLE I: Splitting at 80% best reduces leaf splits overall (lower value for the normalized number of splits implies better memory utilization).

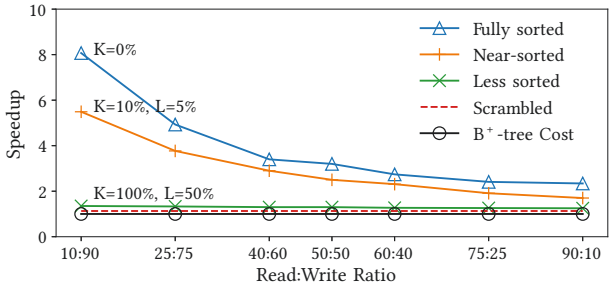


Fig. 18: *SA B$^+$-tree* always outperforms B$^+$-tree on disk (1% bufferpool).

### E. On-Disk Performance

We now experiment with an *SA B$^+$-tree* setup that accesses *disk-resident data*. For this, we configure the bufferpool to fit only the internal tree nodes ($\sim$1% of data size). The SWARE-buffer is set to 1% of the data size. We repeat the experiment for variable data sortedness and variable read/write ratios, and present the speedup of *SA B$^+$-tree* over B$^+$-tree as shown in Fig. 18. From the disk-based experiments, we draw similar conclusions to the in-memory ones (Fig. 10), but with a notable difference. With the data on disk, *SA B$^+$-tree always* outperforms B$^+$-tree, even for read-intensive workloads and fully scrambled data. This is because, regardless of sortedness, we increase locality through our sorting procedures in the buffer. Though this is applicable for both in-memory and disk-based experiments with *SA B$^+$-tree*, the overhead of managing the buffer is negligible compared to accessing tree nodes on disk. Overall, when spilling to disk, *SA B$^+$-tree* offers up to $8\times$ performance benefits for write-intensive workloads with high data sortedness, while always outperforming B$^+$-tree.

### F. Scalability

To analyze the scalability of *SA B$^+$-tree*, we increase the number of entries ingested from 31.25M to 1B while varying $K$ and $L$ proportional (5%) to the workload. We also scale SWARE-buffer by keeping it equal to 1% of the dataset size. Here, we run mixed workloads with equal reads and writes.

***SA B$^+$-tree* Scales Better than the State of the Art.** In Fig. 19a, we observe that the B$^+$-tree performance remains flat as the data size increases, which is attributed to the tree having the same height. Further, we observe a marginal increase in latency for 16GB, which is the point that the B$^+$-tree height increases by one. *SA B$^+$-tree* has a similar trend (it remains flat with one step increase at 4GB) and it offers a speedup between $2.32\times$ and $3.14\times$. The speedup comes from a lower tree height due to the higher factor when compared with the baseline. Overall, maintaining the buffer size proportional to $L$ allows *SA B$^+$-tree* to absorb out-of-order elements at a similar pace as we increase the data size.

***SA B$^+$-tree* Scales Better for Fixed $L$ and Buffer Size.** In the next experiment, shown in Fig. 19b, we maintain $L$ and the buffer size constant as we vary the data size. We

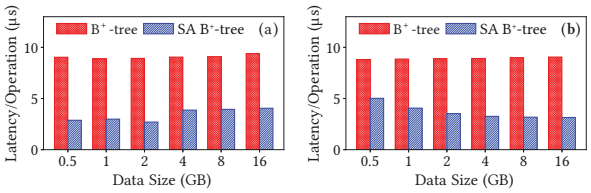


Fig. 19: *SA B$^+$-tree* scales well with data size and outperforms B$^+$-tree when: (a) we vary both $K$ and $L$ as a fraction (5%) of data size; (b) we vary $K$ as a fraction (5%) of data size but keep $L$ fixed ($L$=12.5M).

| Datasize | 0.5GB | 1GB | 2GB | 4GB | 8GB | 16GB |
|---|---|---|---|---|---|---|
| #Entries | 31.25M | 62.5M | 125M | 250M | 500M | 1B |
| %Ent. in Buff. | 8% | 4% | 2% | 1% | 0.5% | 0.25% |
| #.Pages scanned per query | 0.094 | 0.0468 | 0.0227 | 0.0110 | 0.0052 | 0.0024 |

TABLE II: For a fixed $L$ and buffer size, the fraction of entries as well as the pages scanned in the buffer reduce with increasing workload data size.

set $L$ to 12.5M entries and the buffer size to 40MB (holding 2.5M entries). We then run a mixed workload with an equal number of reads and writes on a preloaded index. As expected, the behavior of the B$^+$-tree is the same as in the previous experiment. On the other hand, we observe that the average latency per operation for *SA B$^+$-tree reduces* as the data size increases, offering a 43% to 65% improvement compared to the B$^+$-tree. This result may initially seem counter-intuitive, however, it is explained by the third row of Table II which reports the number of buffer pages scanned per query for the experiment in Fig. 19b. Since both $L$ and the buffer have fixed size, the fraction of data retained in the buffer reduces with data size. For example, 8% entries are contained in the buffer for a 0.5GB dataset, while it is only 0.25% for a 16GB dataset. Since our queries are uniformly random in the entire domain, a smaller fraction of data kept in the buffer means that fewer queries will access the buffer, hence on average fewer unsorted buffer pages will be scanned per query, which is the most expensive part of the query in *SA B$^+$-tree*. Overall, this leads to a 22% reduction of latency per operation for *SA B$^+$-tree* as we increase the data size from 0.5GB to 16GB.

### G. Sortedness-Aware B$^\epsilon$-tree

We now evaluate the *SA B$^\epsilon$-tree* against a B$^\epsilon$-tree. Here, we compute the normalized speedup of both the indexes with varying sortedness against the performance of a B$^\epsilon$-tree with scrambled data, using mixed workloads.

***SA B$^\epsilon$-tree* further boosts performance.** By applying the SWARE paradigm to the B$^\epsilon$-tree, we further amplify its performance with increasing data sortedness. Fig. 20 shows that *SA B$^\epsilon$-tree* significantly outperforms a B$^\epsilon$-tree for any read-write ratio or data sortednes. The B$^\epsilon$-tree by itself improves its performance with increasing data sortedness due to having a *buffer in every internal node*. This internal node buffer makes the B$^\epsilon$-tree ingestion friendly, compared to the B$^+$-tree, and is able to benefit from data sortedness to some extent. Meanwhile, *SA B$^\epsilon$-tree* fully exploits data sortedness, offering up to 7.8× relative speedup to the B$^\epsilon$-tree. The *SA B$^\epsilon$-tree* opportunistically bulk loads when possible, leaving internal node buffers empty. Top inserts (if any) occupy this empty space without having to flush entries to lower levels, hence, improving overall performance. The internal node buffer of the B$^\epsilon$-tree, however, induces an overhead during lookups [8]. For this reason, we see a noticeable drop in the performance of both indexes with increasing reads, though the *SA B$^\epsilon$-tree* still offers at least a 1.1× relative speedup compared to the B$^\epsilon$-tree. Thus, write-optimized tree indexes like the B$^\epsilon$-tree **further benefit** by using the SWARE paradigm to improve ingestion performance by exploiting data sortedness.

| Read : Writes | Buffer Size (%data size ) | | | | |
|---|---|---|---|---|---|
| | 0.05% | 0.1% | 0.25% | 0.5% | 1.0% |
| 10% : 90% | 1.63× | 2.60× | 2.88× | 4.46× | 5.28× |
| 25% : 75% | 1.54× | 2.40× | 2.49× | 3.62× | 4.57× |
| 50% : 50% | 1.56× | 2.07× | 2.82× | 3.21× | 3.40× |
| 75% : 25% | 1.25× | 1.65× | 1.72× | 2.09× | 2.01× |
| 90% : 10% | 1.14× | 1.24× | 1.28× | 1.42× | 1.41× |

TABLE III: When querying TPC-H data, *SA B$^+$-tree* always outperforms B$^+$-tree offering speedups between 1.14× and 5.3×.

### H. Experimenting with TPC-H

*Setup.* For this experiment, we quantify sortedness of data from the `lineitem` table of TPC-H [51] data. We sort the tuples based on the `shipdate` attribute which, in turn, creates a nearly sorted data set with respect to the `receiptdate` attribute. We attribute this degree of sortedness on `receiptdate` as: $K$=96.67% and $L$=0.1% of the total 6M tuples. Using these values of $K$ and $L$, we use our custom workload generator to obtain a $(K, L)$-sorted data collection for ingestion. For both *SA B$^+$-tree* and the B$^+$-tree, we preload the index with 4.8M entries and then execute mixed workloads. We also vary the buffer size between 0.05% and 1% of the data size and report the speedup in overall latency.

***SA B$^+$-tree* Offers Superior Performance.** Table III shows that *SA B$^+$-tree* performs significantly better than B$^+$-trees across all buffer sizes and workload compositions. Even with a buffer that is 0.05% of the data size, *SA B$^+$-tree* offers between 1.14× and 1.63× speedup. As the buffer size increases, it is able to cache more entries before flushing, which reduces the number of top-inserts performed, improving ingestion performance. We also observe that the benefits of *SA B$^+$-tree* diminish as the proportion of reads increase in the workload; however, even for a workload with 90% reads, *SA B$^+$-tree* offers a speedup of 1.3× on average. Interestingly, for larger proportions of reads (≥75%), a larger buffer size (≥1%) causes reads to probe more data in the buffer for every lookup, which, in turn, causes a slight drop in *SA B$^+$-tree*'s speedup. Overall, this experiment highlights that the SWARE meta-design is able to offer significant performance benefits compared to the state of the art with a very small buffer size (0.05%) even for workloads with higher reads (90%).

***SA B$^+$-tree* also outperforms B$^+$-tree for high L and low K.** Here, we tweak our workload generator to obtain a data collection of 6M tuples with $K = 5\%$ and $L = 95\%$, i.e., another extreme of sortedness, while repeating the same experiment as TPC-H data. We observe that *SA B$^+$-tree* offers at least 13% improvement in overall performance against the

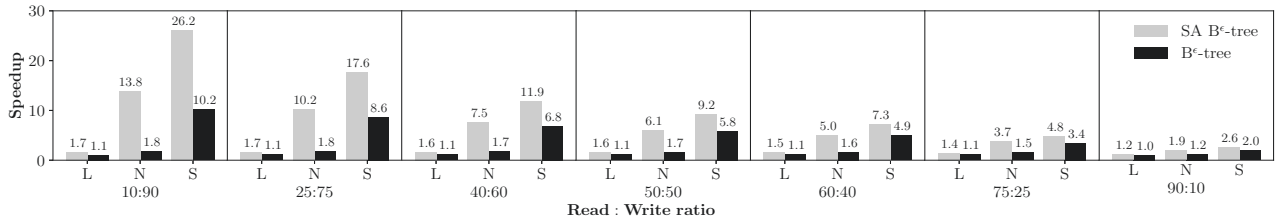


Fig. 20: *SA $B^{\epsilon}$-tree* always outperforms a $B^{\epsilon}$-tree for any degree of data sortedness (less sorted, near-sorted & fully sorted)

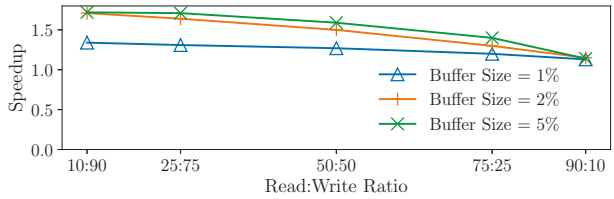


Fig. 21: *SA $B^{+}$-tree* always outperforms $B^{+}$-tree even with data that has a high $L$ (95%) and low $K$ (5%)

$B^{+}$-tree (Fig. 21) with a buffer equivalent to only 1% of the total data size. If we increase the buffer size to 2% (or 5%), we gain up to 71% in overall performance, as a larger buffer is able to better capture the overlapping entries during ingestion.

## VI. Related Work

To the best of our knowledge, this is the first work on designing sortedness-aware indexes. Here, we discuss the literature on ingestion-optimized index structures.

**Optimizing for Tree Ingestion.** $B^{+}$-trees are widely used indexes in commercial data systems due to their balanced ingestion and read performance [17]. Several variants have also been proposed that optimize ingestion via batching. For example, T-tree [36] improves insertion and lookup performance by storing pointers to data in the nodes. To reduce ingestion latency, $CSB^{+}$-tree [47] and PLI-tree [50] maximize cache line utilization by using arithmetic operations (instead of pointer chasing) to locate the child nodes. YATS-tree [33] is a hierarchical index structure that maximizes bulk insertion by pushing new inserts into separate blocks based on a total order. Partitioned $B^{+}$-trees [23] optimize bulk insertion by using an artificial leading column to always append, which leads to creating multiple indexes on overlapping data.

While the aforementioned $B^{+}$-tree-variants improve ingestion performance, the SWARE paradigm allows them to **further optimize** ingestion in the presence of *data sortedness*.

**LSM-trees.** LSM-trees [38, 41] optimize data ingestion by buffering entries and flushing them to disk as sorted runs; however, this comes at a high write amplification cost. The entries are repeatedly re-written to disk as they are periodically sort-merged to create larger sorted collections of data through *compactions* [48]. While LSM-trees aim to maximize ingestion throughput, they are not designed to exploit sortedness. In fact, most LSM-designs are completely agnostic to data sortedness and perform the same amount of merging and (re-)writing of the data on disk even when the data arrive fully sorted. For LSM-trees employing partial compactions with *least overlap* data movement policy [48], it can accelerate ingestion of sorted data; however, these benefits do not apply for nearly sorted data. LSM can benefit from the SWARE meta-design to better exploit variable sortedness. Further, the LSM design *per se* can be optimized to better handle near-sorted data ingestion.

**Data Series and Data Streaming.** Data series store data with a monotonically increasing component, typically a timestamp [42]. Data series indexing assumes that data ingestion follows the expected order [35, 54]–[56]. The ingested data is converted to shapes using specialized representations like iSAX [13], in order to allow similarity comparisons between data series. Data streaming applications operate on windows of data (typically time-based) to calculate state on the fly, and then, discard the incoming entries [14, 21]. Hence, streaming systems inspect whether data arrives out of the expected order and often use a buffer to capture this arrival skew [49]. They do not build an index for the entire dataset, rather, the default expectation is again that data arrives in the expected order.

Contrary to data series and data streaming, in relational systems, the arrival of data is, in general, scrambled; however, indexes are not designed to exploit data arriving with some order. In this work, we *treat sortedness as a resource*, and we build a framework that allows indexes to substantially outperform their classical counterparts if data arrives with some order, while otherwise falling back to baseline performance.

## VII. Conclusion

Inserting data to an index can be perceived as the process of adding structure to an otherwise unsorted data collection. We identify inherent *data sortedness as a resource* that should be harnessed when ingesting data. State-of-the-art index designs like $B^{+}$-trees support faster ingestion through bulk loading when data arrives fully sorted, however, they fail to benefit from sortedness when data is near-sorted.

To address this, we propose an index meta-design that allows for progressively faster ingestion for higher data sortedness. Our proposed SWARE paradigm, combines opportunistic bulk loading, index appends, variable split/fill factor, and an intelligent buffering scheme to amortize the index insertion cost. To ensure competitive lookup performance, we augment the design with Bloom filters, Zonemaps, and query-based sorting that alleviate read overheads. By applying the SWARE paradigm to a $B^{+}$-tree and a $B^{\epsilon}$-tree, we demonstrate that their sortedness-aware counterparts, *SA $B^{+}$-tree* and *SA $B^{\epsilon}$-tree*, outperform their baselines by up to 8.8× and 7.84×.

## VIII. Acknowledgements

This work was funded by NSF grants IIS-1850202 & IIS-2144547, a Facebook Faculty Research Award & a Meta gift.

## REFERENCES


[1] D. Achakeev and B. Seeger, "Efficient Bulk Updates on Multiversion B-trees," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1834–1845, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol6/p1834-achakeev.pdf

[2] M. Ajtai, T. S. Jayram, R. Kumar, and D. Sivakumar, "Approximate counting of inversions in a data stream," in *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC)*, 2002, pp. 370–379. [Online]. Available: https://doi.org/10.1145/509907.509964

[3] A. Appleby, "MurmurHash," *https://sites.google.com/site/murmurhash/*, 2011.

[4] M. Athanassoulis and A. Ailamaki, "BF-Tree: Approximate Tree Indexing," *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1881–1892, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol7/p1881-athanassoulis.pdf

[5] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan, "Designing Access Methods: The RUM Conjecture," in *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2016, pp. 461–466. [Online]. Available: http://dx.doi.org/10.5441/002/edbt.2016.42

[6] R. Bayer and M. Schkolnick, "Concurrency of Operations on B-Trees," *Acta Informatica*, vol. 9, pp. 1–21, 1977. [Online]. Available: http://dx.doi.org/10.1007/BF00263762

[7] S. Ben-Moshe, Y. Kanza, E. Fischer, A. Matsliah, M. Fischer, and C. Staelin, "Detecting and Exploiting Near-Sortedness for Efficient Relational Query Evaluation," in *Proceedings of the International Conference on Database Theory (ICDT)*, 2011, pp. 256–267. [Online]. Available: http://doi.acm.org/10.1145/1938551.1938584

[8] M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan, and Y. Zhan, "An Introduction to Bε-trees and Write-Optimization," *White Paper*, 2015. [Online]. Available: http://supertech.csail.mit.edu/papers/BenderFaJa15.pdf

[9] J. L. Bentley and A. C.-C. Yao, "An Almost Optimal Algorithm for Unbounded Searching," *Information Processing Letters*, vol. 5, no. 3, pp. 82–87, 1976. [Online]. Available: https://doi.org/10.1016/0020-0190(76)90071-5

[10] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[11] T. Bingmann, "STX B+ Tree," *https://github.com/bingmann/stx-btree*, 2007.

[12] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970. [Online]. Available: http://dl.acm.org/citation.cfm?id=362686.362692

[13] A. Camerra, T. Palpanas, J. Shieh, and E. J. Keogh, "iSAX 2.0: Indexing and Mining One Billion Time Series," in *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, 2010, pp. 58–67. [Online]. Available: https://doi.org/10.1109/ICDM.2010.124

[14] P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos, "Beyond Analytics: The Evolution of Stream Processing Systems," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2651–2658. [Online]. Available: https://doi.org/10.1145/3318464.3383131

[15] S. Carlsson and J. Chen, "An Optimal Parallel Adaptive Sorting Algorithm," *Information Processessing Letters*, vol. 39, no. 4, pp. 195–200, 1991. [Online]. Available: https://doi.org/10.1016/0020-0190(91)90179-L

[16] S. Carlsson and J. Chen, "On Partitions and Presortedness of Sequences," in *Acta Informatica*, vol. 29, no. 3, 1992, pp. 267–280. [Online]. Available: https://doi.org/10.1007/BF01185681

[17] D. Comer, "The Ubiquitous B-Tree," *ACM Computing Surveys*, vol. 11, no. 2, pp. 121–137, 1979. [Online]. Available: http://doi.acm.org/10.1145/356770.356776

[18] G. Cormode, S. Muthukrishnan, and S. C. Sahinalp, "Permutation Editing and Matching via Embeddings," in *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, vol. 2076, 2001, pp. 481–492. [Online]. Available: https://doi.org/10.1007/3-540-48224-5{_}40

[19] J. V. den Bercken and B. Seeger, "An Evaluation of Generic Bulk Loading Techniques," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2001, pp. 461–470. [Online]. Available: http://www.vldb.org/conf/2001/P461.pdf

[20] E. W. Dijkstra, "Smoothsort, an Alternative for Sorting In Situ," *Science of Computer Programming*, vol. 1, no. 3, pp. 223–233, 1982. [Online]. Available: https://doi.org/10.1016/0167-6423(82)90016-8

[21] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, "A Survey on the Evolution of Stream Processing Systems," 2020. [Online]. Available: https://arxiv.org/abs/2008.00842

[22] P. Gopalan, T. S. Jayram, R. Krauthgamer, and R. Kumar, "Estimating the Sortedness of a Data Stream," in *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2007, pp. 318–327. [Online]. Available: http://dl.acm.org/citation.cfm?id=1283383.1283417

[23] G. Graefe, "Sorting And Indexing With Partitioned B-Trees," in *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003. [Online]. Available: http://www-db.cs.wisc.edu/cidr/cidr2003/program/p1.pdf

[24] G. Graefe, "A survey of B-tree locking techniques," *ACM Transactions on Database Systems (TODS)*, vol. 35, no. 3, 2010. [Online]. Available: http://doi.acm.org/10.1145/1806907.1806908

[25] G. Graefe, "Modern B-Tree Techniques," *Foundations and Trends in Databases*, vol. 3, no. 4, pp. 203–402, 2011. [Online]. Available: http://dx.doi.org/10.1561/1900000028

[26] G. Graefe, F. Halim, S. Idreos, H. Kuno, and S. Manegold, "Concurrency control for adaptive indexing," *Proceedings of the VLDB Endowment*, vol. 5, no. 7, pp. 656–667, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2180918

[27] G. Graefe and H. A. Kuno, "Adaptive indexing for relational keys," in *Proceedings of the IEEE International Conference on Data Engineering Workshops (ICDEW)*, 2010, pp. 69–74.

[28] A. Gupta and F. Zane, "Counting inversions in lists," in *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2003, pp. 253–254. [Online]. Available: http://dl.acm.org/citation.cfm?id=644108.644150

[29] P. Holanda, S. Manegold, H. Mühleisen, and M. Raasveldt, "Progressive Indexes: Indexing for Interactive Data Analysis," *Proceedings of the VLDB Endowment*, vol. 12, no. 13, pp. 2366–2378, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p2366-holanda.pdf

[30] S. Idreos and M. Callaghan, "Key-Value Storage Engines," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2667–2672.

[31] S. Idreos, M. L. Kersten, and S. Manegold, "Database Cracking," in *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2007.

[32] S. Idreos, S. Manegold, H. Kuno, and G. Graefe, "Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores," *Proceedings of the VLDB Endowment*, vol. 4, no. 9, pp. 586–597, 2011. [Online]. Available: https://www.vldb.org/pvldb/vol4/p586-idreos.pdf

[33] C. Jermaine, A. Datta, and E. Omiecinski, "A Novel Index Supporting High Volume Data Warehouse Insertion," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1999, pp. 235–246.

[34] D. E. Knuth, *The art of computer programming, Volume III: Sorting and Searching (2nd Edition)*. Addison-Wesley, 1998. [Online]. Available: http://www.worldcat.org/oclc/312994415

[35] H. Kondylakis, N. Dayan, K. Zoumpatianos, and T. Palpanas, "Coconut: A Scalable Bottom-Up Approach for Building Data Series Indexes," *Proceedings of the VLDB Endowment*, vol. 11, no. 6, pp. 677–690, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol11/p677-kondylakis.pdf

[36] T. J. Lehman and M. J. Carey, "A Study of Index Structures for Main Memory Database Management Systems," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1986, pp. 294–303. [Online]. Available: http://www.vldb.org/conf/1986/P294.PDF

[37] C. Levcopoulos and O. Petersson, "Splitsort - An Adaptive Sorting Algorithm," *Information Processessing Letters*, vol. 39, no. 4, pp. 205–211, 1991. [Online]. Available: https://doi.org/10.1016/0020-0190(91)90181-G

[38] C. Luo and M. J. Carey, "LSM-based Storage Techniques: A Survey," *CoRR*, vol. abs/1812.0, 2018. [Online]. Available: https://arxiv.org/abs/1812.07527

[39] H. Mannila, "Measures of Presortedness and Optimal Sorting Algorithms," *IEEE Transactions on Computers (TC)*, vol. 34, no. 4, pp. 318–325, 1985. [Online]. Available: https://doi.org/10.1109/TC.1985.5009382

[40] G. Moerkotte, "Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1998, pp. 476–487. [Online]. Available: http://dl.acm.org/citation.cfm?id=645924.671173

[41] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996. [Online]. Available: http://dl.acm.org/citation.cfm?id=230823.230826

[42] T. Palpanas, "Data Series Management: The Road to Big Sequence Analytics," *ACM SIGMOD Record*, vol. 44, no. 2, pp. 47–52, 2015. [Online]. Available: https://doi.org/10.1145/2814710.2814719

[43] Y. Perl, A. Itai, and H. Avni, "Interpolation Search—A log logN Search," *Communications of the ACM*, vol. 21, no. 7, pp. 550–553, 1978. [Online]. Available: http://dl.acm.org/citation.cfm?id=359545.359557

[44] T. Peters, "Timsort description," *https://svn.python.org/projects/python/trunk/Objects/listsort.txt*, 2022.

[45] R. Ramakrishnan and J. Gehrke, *Database Management Systems*. McGraw-Hill Higher Education, 3rd edition, 2002.

[46] A. Raman, K. Karatsenidis, S. Sarkar, M. Olma, and M. Athanassoulis, "BoDS: A Benchmark on Data Sortedness," in *Proceedings of the TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC)*, 2022.

[47] J. Rao and K. A. Ross, "Making B+-trees Cache Conscious in Main Memory," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2000, pp. 475–486. [Online]. Available: http://dl.acm.org/citation.cfm?id=342009.335449

[48] S. Sarkar, D. Staratzis, Z. Zhu, and M. Athanassoulis, "Constructing and Analyzing the LSM Compaction Design Space," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2216–2229, 2021. [Online]. Available: http://vldb.org/pvldb/vol14/p2216-sarkar.pdf

[49] U. Srivastava and J. Widom, "Flexible Time Management in Data Stream Systems," in *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, 2004, pp. 263–274. [Online]. Available: https://doi.org/10.1145/1055558.1055596

[50] K. Torp, L. Mark, and C. S. Jensen, "Efficient Differential Timeslice Computation," *IEEE Trans. Knowl. Data Eng.*, vol. 10, no. 4, pp. 599–611, 1998. [Online]. Available: https://doi.org/10.1109/69.706059

[51] TPC, "TPC-H benchmark," *http://www.tpc.org/tpch/*, 2021.

[52] P. Van Sandt, Y. Chronis, and J. M. Patel, "Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?" in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2019, pp. 36–53.

[53] Z. Zhu, J. H. Mun, A. Raman, and M. Athanassoulis, "Reducing Bloom Filter CPU Overhead in LSM-Trees on Modern Storage Devices," in *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, 2021, pp. 1:1–1:10.

[54] K. Zoumpatianos, S. Idreos, and T. Palpanas, "Indexing for interactive exploration of big data series," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2014, pp. 1555–1566. [Online]. Available: http://dl.acm.org/citation.cfm?id=2588555.2610498

[55] K. Zoumpatianos, S. Idreos, and T. Palpanas, "ADS: the adaptive data series index," *The VLDB Journal*, vol. 25, no. 6, pp. 843–866, 2016. [Online]. Available: http://dx.doi.org/10.1007/s00778-016-0442-5

[56] K. Zoumpatianos and T. Palpanas, "Data Series Management: Fulfilling the Need for Big Sequence Analytics," in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2018.