TIDE: Indexing Time Intervals by Duration and Endpoint

Kai Wang
HKUST
Clearwater Bay, Hong Kong
kwangbn@connect.ust.hk

Moin Hussain Moti
HKUST
Clearwater Bay, Hong Kong
mhmoti@connect.ust.hk

Dimitris Papadias
HKUST
Clearwater Bay, Hong Kong
dimitris@cs.ust.hk

Abstract

Indexes for large collections of intervals are common in temporal databases, where each record has a lifespan, or validity interval. We propose a universal representation that encapsulates various interval indexes using diagonal corner structures, providing valuable insights about their effectiveness. Moreover, we exploit our findings to develop TIDE, a disk-based index for historical intervals. TIDE adopts a two-level architecture. A top tree organizes intervals by their duration. The leaf nodes of the top tree correspond to the root nodes of bottom trees, ordering intervals by their endpoints. Both top and bottom trees are append-only B+-trees to facilitate fast insertions. An experimental evaluation with real data sets shows that TIDE achieves impressive performance gains with respect to its direct competitor, on insertion (up to x100) and query processing (up to x7000) speed.

CCS Concepts

• **Information systems → Data structures**.

Keywords

Intervals, Time-series, Temporal Indexes

ACM Reference Format:
Kai Wang, Moin Hussain Moti, and Dimitris Papadias. 2025. TIDE: Indexing Time Intervals by Duration and Endpoint. In *19th International Symposium on Spatial and Temporal Data (SSTD '25), August 25–27, 2025, Osaka, Japan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3748777.3748785

1 Introduction

Intervals are fundamental to representing and analyzing a wide range of real-world data in applications such as resource scheduling (e.g., bookings), financial platforms (e.g., transactions), and IoT monitoring. For instance, a soil sensor may report moisture status every 10 minutes, while a wearable device tracks high heart rates during hiking. The efficient management of large volumes of interval data is critical in spatial analysis, temporal reasoning, and complex event processing. Traditional database indexes, optimized for discrete scalar values, cannot effectively handle interval relationships (e.g., overlaps in Allen Algebra [1]). As datasets grow in size and complexity, the demand for scalable, high-performance interval indexes persists today, despite decades of research.

SSTD '25, Osaka, Japan

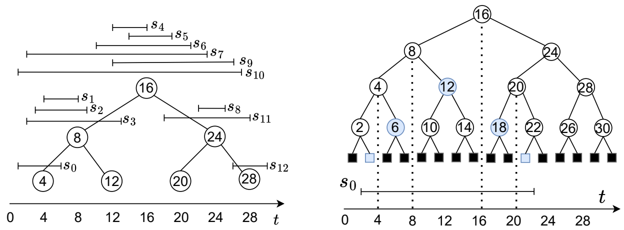
ACM ISBN 979-8-4007-2094-9/25/08
https://doi.org/10.1145/3748777.3748785

In temporal databases each record has a lifespan, or validity interval $[t_s, t_e)$, where $t_s < t_e$. A record is considered alive if t_e equals the current time; otherwise, it is dead. Most temporal interval indexes are *partially persistent* [14, 27], i.e., they only allow updates at the current time. Consequently, nodes storing dead intervals are *immutable*. On the other hand, *fully persistent* [7, 23] structures also allow updates on dead intervals, and all their nodes are *mutable*. Moreover, indexes can be classified as disk or main-memory based. Different structures are evaluated on their performance for *stabbing* and *range* queries, which retrieve all intervals intersecting a timestamp or period in history. These query types form the basic building blocks of more complex tasks.

Despite their conceptual differences, we demonstrate that interval indexes can be captured by some corner structure in a 2D space, defined by selecting two out of three possible dimensions: namely starting time t_s, ending time t_e, or duration d. This unified representation facilitates the optimization of query processing by identifying nodes that *must* contain query results (i.e., all their intervals can be directly reported) versus nodes that *may* contain results (i.e., their intervals must be examined individually). Moreover, the representation provides useful insight into the advantages and shortcomings of each index. Specifically, some structures have highly unbalanced nodes, while others involve redundancy, which necessitates duplicate elimination during query processing.

These shortcomings led to the development of the proposed TIDE, a partially persistent, disk-based index for intervals that satisfy the *increasing ending time* (*IET*) assumption, i.e., intervals arrive in increasing order of t_e. TIDE has balanced nodes (in terms of both cardinality and duration variance), without redundancy and the associated problems (extra space, need for duplicate elimination). It involves a two-level architecture. The *top tree* is an append-only B+-tree on interval duration. The leaf nodes of the top tree correspond to the root nodes of append-only *bottom* B+-trees that index the ending time. Finally, the leaf nodes of the bottom trees are *data nodes* that store the records (interval + payload). Given IET, only the last data node of every bottom B+-tree is mutable.

We evaluated TIDE against SEB [31], its direct competitor also based on the IET assumption, using real datasets with diverse characteristics. TIDE outperforms SEB, often by orders of magnitude, on insertion and query speed. Moreover, it achieves better space efficiency because its insertion strategy generates full nodes. The rest of the paper is organized as follows. Section 2 reviews existing interval indexes. Section 3 contains a representation that enables optimization of query processing and conceptual evaluation of different indexes under a unifying framework. Section 4 presents the insertion and query processing algorithms of TIDE. Section 5 compares TIDE against SEB, and Section 6 concludes the paper.



(a) interval-tree (b) segment-tree

Figure 1: Classical structures

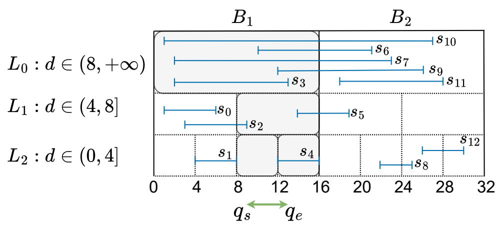


Figure 2: Period-index

2 Related work

Section 2.1 focuses on main-memory, and Section 2.2 on disk-based interval indexes.

2.1 Main-memory structures

The first main-memory indexes for intervals use a binary search tree (BST) as the *base structure* T. Each node of T corresponds to a time instant, and has a *secondary structure* (i.e., a list) with unlimited capacity storing all intervals intersecting with that instant. In the *interval-tree* [15], each interval is stored in the secondary structure of the highest overlapping node of T. Figure 1a shows an interval-tree managing thirteen intervals $s_0, ..., s_{12}$. For example, $s_5 = [14, 19)$ is assigned to root node n_{16}. The interval-tree consumes $O(N)$ space, where N is the number of data intervals. Nodes of the interval-tree may be very unbalanced. For instance, most intervals could be assigned to the top node, while the rest of the nodes could be almost empty. Moreover, intervals in the same node may have large length variance.

To avoid these shortcomings, the *segment-tree* [25] partitions and stores each interval in multiple nodes. Specifically, an interval is first stored at all leaf nodes that intersect it. Then, consecutive partitions are merged at the upper level recursively, if they cover their parent node. For example, in Figure 1b, interval $s_0 = [2, 22)$ is stored in nodes n_2^R, n_6, n_{12}, n_{18} and n_{22}^L, where L (R) denotes left (right) leaf node. Due to replication, the segment-tree consumes $O(N \log N)$ space. Range query processing requires duplicate elimination since the same interval may exist in several nodes, possibly at different levels.

A learned *period-index* [5] splits the time domain into coarse buckets and divides each bucket hierarchically. Figure 2 shows thirteen intervals in two buckets B_1 and B_2. Each bucket has length $l = 16$ and is partitioned into $m = 3$ levels. Both l and m are learned parameters. An interval is stored at the top level such that its duration is more than half of the extent of that level. For example, $s_2 = [3, 9)$ is assigned to L_1 because its length (6) exceeds half the length of level 1. Since s_2 intersects two partitions of L_1, it is stored in both. A range query searches all levels intersecting its range. Compared to the segment-tree, the period-index incurs less redundancy (because all copies of an interval are at the same level), but duplicate elimination is still necessary for range queries.

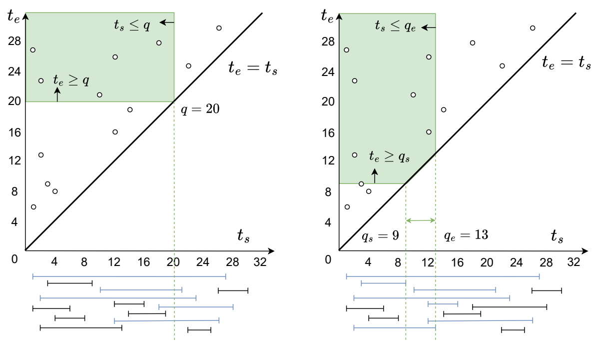
HINT [9] also applies a hierarchical decomposition of buckets. Each interval is partitioned and assigned to different levels, similarly to segment trees. The first occurrence of an interval is marked as *original* and the rest are *replicas*. During query processing, for all partitions intersecting the start of the query range, all data intervals are examined. For the remaining partitions, only originals may constitute results (replicas correspond to duplicates). *LIT* [10] extends HINT for dynamic intervals. The *RD-tree* [8] is a recent structure that has two variants: RD-tree-td, sorts intervals by t_s and duration d; RD-tree-dt, sorts intervals by d and t_s. Finally, interval indexes have been applied for specialized tasks including temporal aggregation [20] and interval joins [6, 18].

2.2 Disk-resident structures

The *external interval-tree* (*EI-tree*) and *external segment-tree* (*ES-tree*) [2] are disk-based extensions of their main-memory counterparts that replace the BST with a B+-tree as the base structure T. They both use a B+-tree with fanout $\sqrt{B}$, where B is the page capacity. Their secondary structures are lists with unlimited capacity, leading to expensive updates. For example, a split would force numerous intervals to move between nodes. To decrease the update cost, the EI-tree and ES-tree use complicated *buffer tree* and *weight balancing* techniques, which are theoretical in nature.

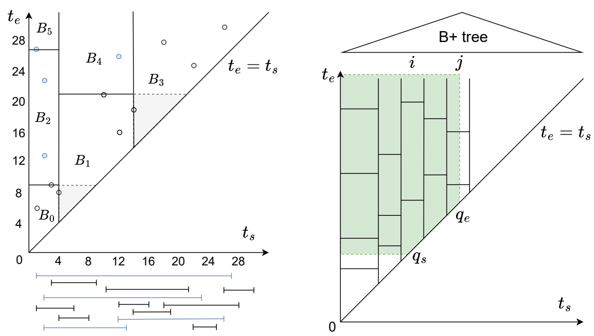
The *time-index* [16] stores alive records at the first timestamp of each node, and incremental updates at the following timestamps. The *append only-tree* (*AP-tree*) [28] orders intervals by their starting time t_s using an append-only B+-tree. The AP-tree has optimal insertion speed, but is slow to answer stabbing queries. The *relational interval-tree* (*RI-tree*) [22] is an external version of the interval-tree for relational databases. Another trend extends R-tree and its variants [4, 29] to manage intervals. However, R-trees are not effective for long intervals and high overlaps [22]. To deal with long intervals, the *segment R-tree* (*SR-tree*) [21] combines the main-memory segment-tree with the disk-based R-tree. Similar to the segment-tree, intervals in SR-tree are stored in both leaf and internal nodes, leading to redundancy. Therefore, it consumes $O(\frac{N}{B} \log_B N)$ space [27].

The *diagonal corner structure* [19] is a disk-based index, managing dynamic intervals in a 2-dimensional (2D) space, where t_s is the horizontal and t_e is the vertical axis. Intervals are mapped to points above line $t_e = t_s$, because $t_e > t_s$, forming a diagonal corner space. Figure 3 shows the 2D representation for a set of intervals. A stabbing query for intervals containing timestamp q returns points with $t_s \le q \le t_e$. For example, in Figure 3a, a query at time 20 will retrieve all points (intervals) in the shaded area. In Figure 3b, a query with range $[9, 13]$ returns points with $t_s \le 13$ and $t_e \ge 9$. Early work on corner structures has been of theoretical nature, assuming static data [3, 27, 32],



(a) stabbing query (b) range query

Figure 3: Diagonal corner queries

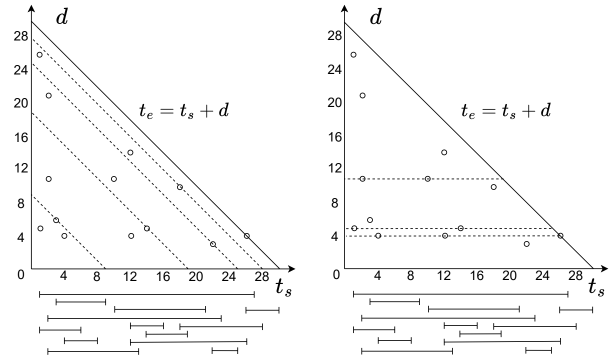


(a) the structure (b) range query

Figure 4: SEB

The *start/end timestamp B-tree* (SEB) [31] applies corner structures for indexing intervals following the IET assumption: intervals are inserted in increasing order of t_e. Figure 4a shows SEB for thirteen intervals in six leaf nodes with capacity 3. Initially, there is a single data node B_0 that covers the entire diagonal corner space, $t_s \in [0, +\infty)$ and $t_e \in (0, +\infty)$. When B_0 overflows at time 9, it generates two nodes B_1 and B_2. B_1 is the *first* node in its column with a new t_s range $(4, +\infty)$. B_2 is a node with the same t_s range as B_0 $[0, 4]$, and a different t_e range $[9, +\infty)$. B_0 is full and becomes immutable. B_1 and B_2 are non-full and ready to accept insertions. A new point always falls into B_1 (if its $t_s > 4$) or B_2 (if $t_s \le 4$). When a *non-first* node overflows, it only requires horizontal partitioning. For example, at time 27, the overflow of B_2 creates node B_5 with the same t_s range $[0, 4]$ as B_2. Points in B_2 fulfill $t_e \in [9, 27]$, while points in B_5 have $t_e \in [27, +\infty)$.

SEB uses a top-layer B+-tree to index the columns, which correspond to nodes with the same t_s range, as shown in Figure 4b. Each column is indexed by a separate B+-tree. A new insertion of interval $[t_s, t_e)$ first locates the column according to t_s. Given the IET assumption, point (t_s, t_e) is appended to the last node of that column. A range query $[q_s, q_e]$ identifies column i (and j) covering q_s (and q_e). All nodes in columns up to j are examined for results, provided that their t_e exceeds q_s. SEB has been used mostly for trajectory management [12, 13, 26]. The *compressed start end-tree* [33] applies similar concepts, but switches the order from $< t_s, t_e >$ to $< t_e, t_s >$. This however introduces high update cost because insertions may occur at any node (as opposed to the last node of some column as in SEB).



(a) D-ordering by t_e, t_s (b) H-ordering by d, t_s

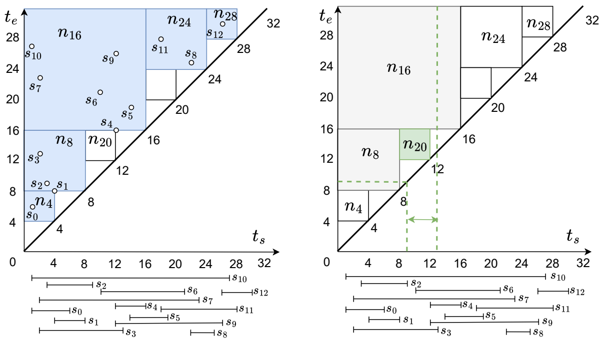
Figure 5: Interval spatial transformation

The *interval spatial transformation* (*IST*) [17, 24, 30] maps intervals into a 2D space of t_s (x-axis) and d (y-axis), where d is the duration. IST forms a triangle starting from point $(0, 0)$ and bounded by line $t_e = t_s + d \le now$. There are three variants: D(iagonal)-ordering (sort by t_e, t_s), V(ertical)-ordering (sort by t_s, t_e), and H(orizontal)-ordering (sort by d, t_s). The sorted points are indexed by a single B+-tree. Figure 5a shows D-ordering, assuming a B+-tree node capacity of 3. The first node contains the three intervals closest to $(0, 0)$ with the smallest t_e. Figure 5b shows an example of H-ordering, where the first node of B+-tree contains the three intervals with the shortest duration.

3 A Unified Representation for Interval Indexes

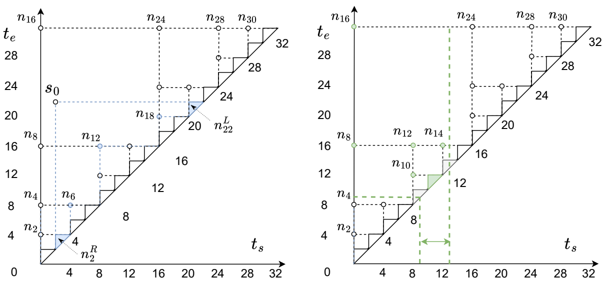
Interval indexes can be captured by some corner structure in a 2D space, defined by the endpoints t_s, t_e, or duration d. This representation enables the identification of nodes that *must* contain query results (i.e., all their intervals can be directly reported) versus nodes that *may* contain results (i.e., their intervals must be individually examined). In addition to reducing the computation cost of regular queries, this may also decrease the I/O cost of *aggregate* queries. For instance, when we wish to compute the *count* of intervals intersecting a range, the number of intervals within each node inside the range can be aggregated directly, without visiting the node. This is particularly beneficial for large ranges that contain multiple nodes, possibly at high levels.

First, we focus on the interval-tree. Figure 6a maps the nodes and thirteen intervals of Figure 1a into a corner structure, where t_s is the x-axis and t_e the y-axis. Each node corresponds to a square-shaped area in the mapped space. For example, node n_{16} stores intervals intersecting with time 16, i.e., its mapped area is $t_s \le 16 \le t_e$. Similarly, n_8 is mapped to the space of $t_s \le 8 \le t_e < 16$, and n_4 corresponds to the space of $t_s \le 4 \le t_e < 8$. Figure 6b shows the processing of a range query $[9, 13]$. Points in nodes (n_8, n_{16}) partially intersecting the range must be examined because they may constitute results (for these intervals $t_s \le 13$ and $t_e \ge 9$). On the



(a) the structure (b) query with range $[9, 13]$

Figure 6: Corner structure of the interval-tree



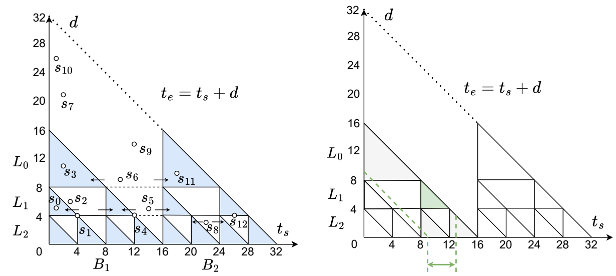
(a) the structure (b) query with range $[9, 13]$

Figure 7: Corner structure of the segment-tree

other hand, those in nodes covered by the range (n_{20}) are directly reported, because they are query results (for these intervals $t_s \geq 9$ and $t_e \leq 13$).

Similar to the interval-tree, the corner structure for the segment-tree organizes intervals in nodes based on t_s and t_e, without explicitly considering the duration d. Figure 7a maps the nodes of Figure 1b into a 2D corner structure that has a bottom layer of triangle-shaped leaf nodes. Recall that an interval may be partitioned and stored in multiple nodes, possibly at different levels. For instance, $s_0 = [2, 22)$ is first assigned to leaf nodes, which are merged recursively, if the parent node is fully covered by s_0. The first (last) partition containing t_s (t_e) of s_0 is stored in leaf node n_2^R (n_{22}^L). The remaining partitions are merged in internal nodes. Specifically, copies of s_0 are stored in nodes $n_6 = [4, 8)$, $n_{12} = [8, 16)$ and $n_{18} = [16, 20)$. Internal nodes are mapped to points because they only store intervals covering their full range. A query with range $[9, 13]$ in Figure 7b reports directly the intervals of green-shaded nodes (i.e., n_8, n_{10}, n_{10}^R, n_{12}, n_{14},n_{16}) in the area defined by $t_s \geq 9$ and $t_e \leq 13$. Intervals in grey nodes (i.e., n_{10}^L and n_{14}^L) require inspection. Duplicate elimination is necessary. HINT [9] translates to a similar corner structure and query processing mechanism.

Since the period-index assigns intervals to buckets based on their duration, it is better represented by a 2D space, with t_s as the x-axis and d as the y-axis. Figure 8a maps the buckets of Figure 2 to triangles in a corner structure. B_1 (B_2) corresponds to the triangular space with $t_s, d \in [0, 16)$ ($t_s, d \in [16, 32)$). Horizontal lines for



(a) the structure (b) query with range $[9, 13]$

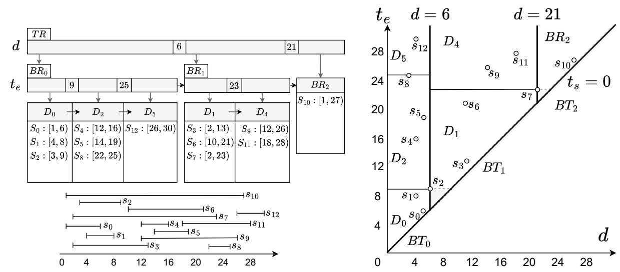
Figure 8: Corner structure of the period-index

duration 8 and 4 subdivide B_1 and B_2 into three levels L_0, L_1 and L_2. Intervals in L_0 must have duration $d > 8$. Accordingly, the valid space of B_1 (B_2) in L_0 is restricted to the upper triangle $t_s \in [0, 8)$ ($[16, 24)$) and $d > 8$, shaded in blue. Similarly, intervals in L_1 must have duration in the range $(4, 8]$, restricting the valid space of the corresponding buckets to the blue triangles. Intervals falling in a valid space (e.g., s_0, s_3) are stored directly in the corresponding bucket, whereas the rest (e.g., s_2, s_5, s_6) generate duplicates in adjacent buckets at the same level. For example, s_2 and s_5 are duplicated at L_1, while s_6 is duplicated at L_0, and stored at both B_1 and B_2. Figure 8b shows a query with range $[9, 13]$. Intervals in the green partition ($[8, 16)$) are directly reported, while those in grey buckets ($[8, 12)$) require inspection. It is worth mentioning that the original period-index [5] does not differentiate between the two result types, missing an optimization opportunity.

None of the above indexes includes a maximum (or minimum) node/bucket capacity constraint. Thus, some nodes/buckets may contain numerous intervals, whereas the rest may be (almost) empty. This is particularly true for the interval-tree, where top level nodes (e.g., n_{16} in Figure 6) may include most intervals. In addition, each node may contain intervals with large duration variance. The other structures alleviate these problems at the expense of redundancy. The segment-tree and HINT may store copies at multiple levels, while the period-index generates duplicates at a single level based on duration. In addition to its negative effect on index size, redundancy necessitates duplicate elimination, increasing the cost and complexity of query processing. Motivated by the above observations, we aim at a novel index that combines the best characteristics of existing work, namely balanced nodes containing intervals with similar duration, and no redundancy.

4 TIDE

Similar to SEB, we assume that intervals arrive in increasing order of ending time t_e (IET), i.e., they are inserted in the database when they die. The proposed **TIDE** (**t**ime **i**ntervals by **d**uration and **e**ndpoint) consists of two layers. The *top tree* is an append-only B+-tree organizing intervals by duration d. The leaf nodes of the top tree correspond to the root nodes of append-only B+-trees, called *bottom trees*, ordering intervals by t_e. Finally, the leaves of the bottom trees are *data nodes* that store the records (interval + payload). Each leaf node of the top and bottom trees stores a pointer to its next sibling. Under the IET assumption, each interval is inserted



(a) nodes **(b) corner structure**

Figure 9: TIDE

into the latest leaf node of the bottom tree that corresponds to its duration. The rest of the nodes are full and immutable. Grouping intervals with similar durations at the top tree, leads to a small number of bottom trees, especially for datasets that involve low duration variance. This decreases the non-full nodes (each bottom tree has a single non-full node), facilitating high compactness and cache locality (i.e., most non-full nodes are cached).

Figure 9a shows an example of TIDE assuming that the data node capacity is 3. The root node of the top tree TR contains three leaf nodes, corresponding to three bottom tree roots BR_0, BR_1 and BR_2, separated by duration keys 6 and 21. BR_0 (BR_1) separates its data nodes with ending time keys 9 and 25 (23). BR_2 is both a root and a data node. A new interval (with $t_e \geq 28$) may only be inserted at D_5, D_4 or BR_2. Figure 9b shows the corresponding corner structure in the duration and end-time 2D space. All intervals appear on the upper half of the diagonal $t_e = d$. The three bottom trees correspond to three adjacent columns on the d-axis, separated by 6 and 21. Their respective data nodes are rectangular partitions of those columns on the t_e-axis. Each interval maps to a point into a data node based on its d and t_e. For instance, intervals with $d \in (6, 21]$ are mapped to points in D_1 or D_4 (of BT_1), with D_1 storing older, and D_4 more recent intervals.

4.1 Insertions

We adopt the AP-tree [28] to facilitate *append-only* insertions for top tree and bottom trees. Each root (TR or BR) stores a pointer to its last leaf node, where insertions may occur. Algorithm 1 describes insertion of interval I and its payload P. First, TIDE searches the top tree TT for the bottom tree root BR containing the duration key_d of I (line 5). Then BR returns the last data node DN of its bottom tree BT. If DN is not full, TIDE appends I and P into DN (line 8) and the insertion terminates. Otherwise, DN requires processing the overflow, generating a new data node (lines 12-24). There are three cases of splits:

- Case 1: horizontal split only, in lines 13-16.
- Case 2: vertical split only, in lines 17-21.
- Case 3: vertical and horizontal split, in lines 17-24.

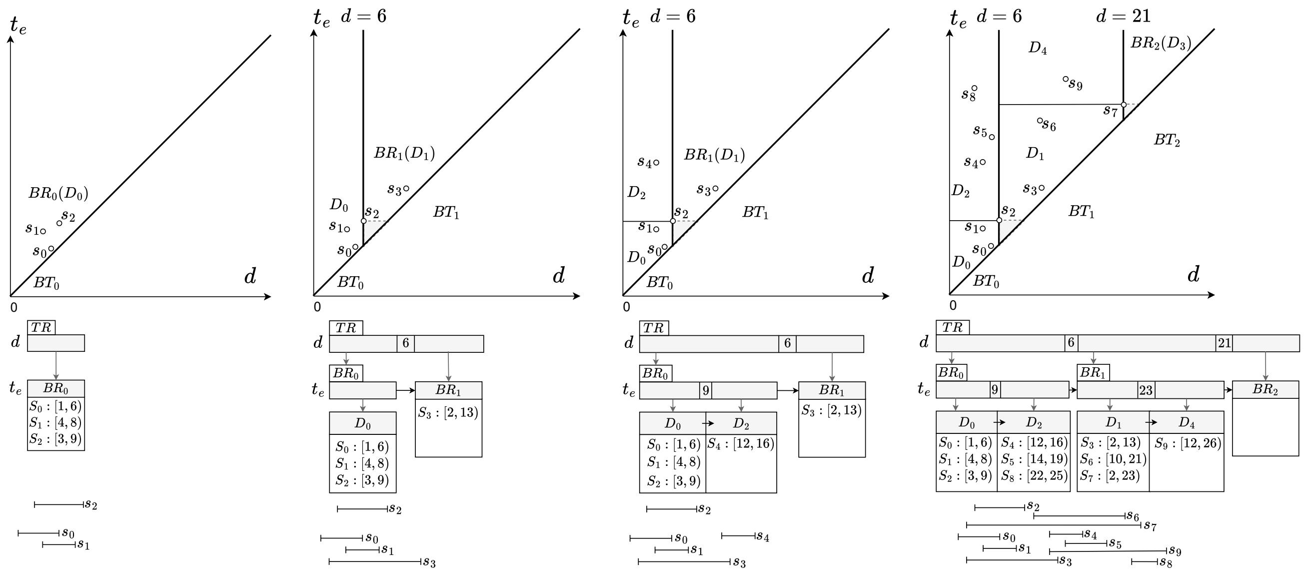
If the overflowed node DN is not a bottom tree root BR, TIDE splits horizontally at the largest ending time key_{te}^{new}, generating data node DN^{new} for the insertion (Case 1). If the overflowed node DN is a bottom tree root BR (i.e., the bottom tree BT has a single data node DN), TIDE splits vertically at the maximum duration key_d^{new},

Algorithm 1 Inserting a new record (I: Interval, P: Payload)

1: **procedure** INSERT_RECORD(I, P)
2:   **if** TT is empty **then**
3:     Initialize the top tree TT
4:   $key_d \leftarrow I.t_e - I.t_s$ ▷ *Compute the duration key*
5:   $BR \leftarrow$ find the bottom root corresponding to key_d in TT
6:   $DN \leftarrow$ latest data node in the bottom tree of BR
7:   **if** DN is not full **then**
8:     Insert (I, P) to DN
9:   **else** ▷ *DN is full*
10:     $DN^{new} \leftarrow$ PROCESS_OVERFLOW(BR, DN, key_d)
11:     Insert (I, P) to DN^{new}
12: **procedure** PROCESS_OVERFLOW(BR, DN, key_d)
13:   **if** $DN \neq BR$ **then** ▷ *Case 1*
14:     $key_{te}^{new} \leftarrow$ ending time of the last interval in BR
15:     $DN^{new} \leftarrow$ add new data node to BR's tree with key_{te}^{new}
16:     **return** DN^{new}
▷ *If DN is also the bottom tree root*
17:   $DN' \leftarrow$ transfer the data in DN to another page
18:   Upgrade BR to a branch node pointing to DN'
19:   $key_d^{new} \leftarrow$ maximum duration in DN
20:   $BR^{new} \leftarrow$ add new leaf node to TT with key_d^{new}
21:   **if** $key_d^{new} < key_d$ **then return** BR^{new} ▷ *Case 2*
▷ *Case 3*
22:   $key_{te}^{new} \leftarrow$ ending time of the last interval in BR
23:   $DN^{new} \leftarrow$ add new data node to BR's tree with key_{te}^{new}
24:   **return** DN^{new}

creating a bottom tree BT^{new} with root BR^{new} (lines 19-20). BR^{new} is also a data node, storing intervals with durations in $d \geq key_d^{new}$. Due to the constraint that only the latest bottom tree root can be a data node, BR is upgraded from a leaf to a branch node (lines 17-18), which has a single child. If $I.d > key_d^{new}$, the insertion falls into BR^{new} of BT_{new} (Case 2). Otherwise (Case 3), TIDE requires another horizontal split at the current largest ending time key_{te}^{new}, creating data node DN_{new} in BT (lines 22-24) for insertion.

Figure 10 illustrates the insertion of the ten intervals $s_0, ..., s_9$ of Figure 9 into TIDE (assuming data node capacity 3). Initially (Figure 10a), the top tree root TR has a single child BR_0, which is both a bottom tree root and a data node D_0, containing s_0, s_1 and s_2. In Figure 10b, the next insertion s_3 forces D_0 to overflow at current maximum duration 6 (i.e., duration of s_2), generating bottom tree BT_1 with a single data node D_1, which is also a bottom tree root BR_1 (Case 2). Interval s_3 is inserted into D_1, since it has duration $d > 6$. BR_0 is upgraded from leaf to branch node, pointing to D_0. In Figure 10c, s_4 has duration below 6 and should be inserted into the latest data node under BR_0, forcing D_0 to overflow at its maximum $t_e = 9$ (Case 1). A new data node D_2 accommodates s_4, and D_0 becomes immutable. Future insertions either fall into D_1 (e.g., s_6 and s_7) or D_2 (e.g., s_5 and s_8) according to their duration. In Figure 10d, inserting s_9 causes D_1 to overflow at its current largest $d = 21$, generating BT_2 with a single data node D_3. Meanwhile, BR_1 is upgraded to a branch node, pointing to immutable D_1. Since s_9 has $d \leq 21$ and should be inserted into BT_1, D_1 overflows at its



(a) 3-rd insertion (b) 4-th insertion (c) 5-th insertion (d) 10-th insertion

Figure 10: Insertions in TIDE

current largest $t_e = 23$ (Case 3), generating data node D_4 to insert s_9. The following Lemma 4.1 analyzes the insertion cost of TIDE.

LEMMA 4.1. *When there are N intervals in m bottom trees, an insertion has I/O cost $O(\log_B m + \frac{1}{B} \cdot \log_B N + \frac{m}{N} \cdot \log_B m)$, where B is the capacity of a disk page.*

PROOF. Each insertion always incurs a search cost C_{Search} to find the proper leaf node. Locating the proper bottom tree given the duration, is bounded by the depth of the top tree $O(\log_B m)$. Since each bottom tree is append-only, fetching its last node costs only $O(1)$. Therefore, the cost of insertions in the absence of overflows is dominated by the search cost $C_{Search} = O(\log_B m)$. In addition, overflows require a horizontal split (Case 1) with cost C_{HS}, or vertical split (Case 2) with cost C_{VS} or both (Case 3) with cost $C_{HS} + C_{VS}$. A horizontal split inserts a new data node into a bottom tree[1], and may propagate all the way up to its root with cost bounded by the depth of the largest bottom tree $O(\log_B N)$. In the worst case (Case 1&3), a horizontal split occurs after every $O(B)$ insertions; thus, the amortized $C_{HS} = O(\frac{1}{B} \cdot \log_B N)$. Moreover, an insertion may incur a vertical split with a probability $\frac{m}{N}$, generating a single-node bottom tree, which requires an insertion into the top tree with $O(\log_B m)$ cost. Therefore, the amortized cost $C_{VS} = O(\frac{m}{N} \cdot \log_B m)$. Adding the three terms together: $C_{Search} + C_{HS} + C_{VS} = O(\log_B m + \frac{1}{B} \cdot \log_B N + \frac{m}{N} \cdot \log_B m)$. □

In practice, C_{Search} and C_{VS} are negligible since $m << N$, and C_{HS} dominates the insertion cost of TIDE. Observe that the duration range of the bottom trees is determined by the initial points (i.e., intervals with the smallest t_e), and may not be optimal if the duration distribution changes over time. Similar issues exist for all corner structures, including SEB, and can be solved if the interval distribution is known in advance.

4.2 Range Queries

Since the top tree organizes intervals by duration, it is not useful for range queries. Instead, given a range query $[q_s, q_e]$, where $q_s \leq q_e$, TIDE searches *all* bottom trees and returns intervals fulfilling $t_e \geq q_s$ (horizontal boundary) and $t_s \leq q_e$ (diagonal boundary). A stabbing query can be considered as a special case of a range with $q_s = q_e$. Figure 11 shows an example range, where the result area is shaded in green. For instance, nodes in BT_0 with possible results (i.e., D_2, D_5 and D_6) intersect $[q_s, q_e + d_1]$. Nodes below the horizontal boundary $t_e < q_s$ contain intervals that end before q_s, whereas nodes above the diagonal boundary $t_s > q_e$ have intervals that start after q_e. Similarly, in BT_1 nodes possibly containing results are D_1, D_7 and D_9. Intervals in nodes, such as D_4, covered by the range are directly reported.

Algorithm 2 shows the pseudocode for range query processing $[q_s, q_e]$. Each bottom tree BT stores intervals with durations ranging from key_d^{min} to key_d^{max}, shaped as a column. Searching a bottom tree starts from the data node containing q_s (line 7), and stops when reaching the data node containing $q_e + key_d^{max}$ (lines 11-12) or the last data node (line 8). Under the unified representation, TIDE identifies nodes that can be directly reported (i.e., all its intervals are results), if $q_s \leq key_{te}^{min}$ and $key_{te}^{max} \leq q_e + key_d^{min}$ (lines 13-14). The intervals of the remaining searched nodes may constitute results, and must be individually examined (lines 15-16). TIDE can easily be extended to *count* queries, returning only the number of intersected intervals, as opposed to their IDs. In this case, immutable nodes fully covered by the range (e.g., D_3, D_4 of Figure 11) no longer require disk accesses, but their intervals are directly aggregated to the total count. Since such nodes are fully

[1] A horizontal split has no cost for the top tree.

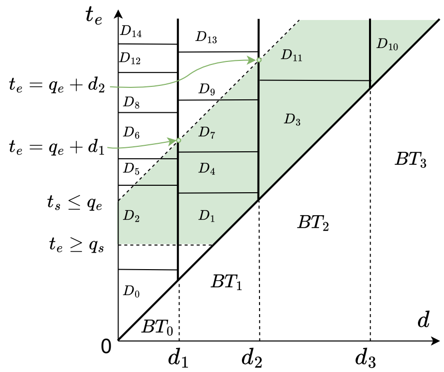


Figure 11: Range query $[q_s, q_e]$ in TIDE

Algorithm 2 Searching a range ($[q_s, q_e]$) in TIDE

```
procedure RANGE_QUERY([q_s, q_e])
    R ← {}                                            ▷ Result Intervals
    BT ← earliest bottom tree
    key_d^min ← 0
    while BT ≠ null do
        key_d^max ← maximum duration in BT
        DN ← find the data node for q_s in BT
        while DN ≠ null do
            key_te^min ← first key of DN
            key_te^max ← last key of DN
            if q_e + key_d^max < key_te^min then
                break
            if q_s ≤ key_te^min and key_te^max ≤ q_e + key_d^min then
                Append all records of DN to R         ▷ Report
            else
                Append qualifying records of DN to R
            DN ← next sibling of DN
        key_d^min ← key_d^max
        BT ← next sibling of BT
    return R
```

packed, their number of intervals is fixed. Only mutable covered nodes, or those (mutable or immutable) partially intersecting the range, necessitate disk accesses.

Lemma 4.2. *When there are N intervals in m bottom trees, a stabbing/range query has I/O cost $O(m \log_B N + \frac{k}{B})$, where k is the number of query results.*

Proof. Given a range query $[q_s, q_e]$, TIDE searches (in each bottom tree BT_i) for the data node containing q_s with cost $O(\log_B N)$. Then it scans its siblings until finding the data node containing $q_e + key_d^{max}$ (i.e., the maximum duration in a bottom tree) or the last data node. The total number of data nodes containing k results is $\frac{k}{B}$. Combining the search and scan terms, we obtain $O(m \log_B N + \frac{k}{B})$. □

TIDE can efficiently process queries with length constraints, e.g., find all intervals in $[q_s, q_e]$ with duration in the range $[d_s, d_e]$. In this case, the top tree is used to identify bottom trees with intervals satisfying $[d_s, d_e]$. Even for conventional ranges (without duration constraints), where all m bottom trees are accessed, TIDE's query performance is outstanding because m is very low in real-world datasets. We experimentally evaluate our claims in the next section.

	BIKE	NFT
Object Interval #Intervals	NYC bicycle riding trips 100507903	non-fungible token unchanged price 28581957
Min. Duration	60 seconds	1 second
Avg. Duration	984 secs (0.0004%) (16.4 minutes)	2724947 secs (3.2%) (1 month)
Max. Duration	19513649 secs (8.8%) (7.4 months)	83743716 secs (99.6%) (2.7 years)

Table 1: Properties of interval datasets

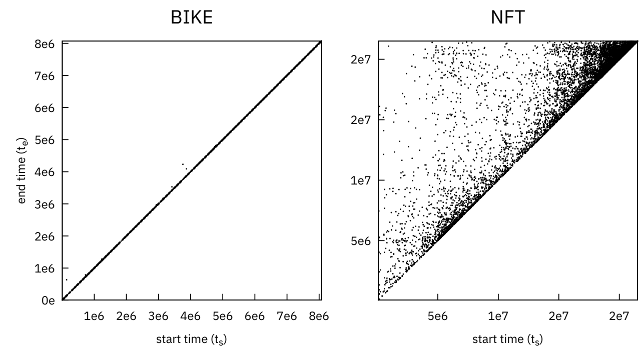


Figure 12: 10k data points sampled from BIKE and NFT

5 Experimental Evaluation

The evaluation was conducted on Ubuntu Linux with an AMD Ryzen Threadripper 3960X 3.8GH CPU and 64GiB RAM. We developed a generic disk-based framework for historical indexes in Rust [2], and implemented both TIDE and SEB using the same append-only B+-tree structures [28]. The disk-page and cache sizes are set to 4KiB and 4MiB, respectively, for all experiments. We use the following real datasets, summarized in Table 1:

- **BIKE** [3]: Start and end timestamps (in seconds) 100M bicycle trips, during 2014-2020 in New York City.
- **NFT** [4] [11]: 28M intervals denoting the stable price period (in seconds) of non-fungible tokens from OpenSea transactions, during 2021-2023.

Figure 12 plots a sample of 10k data points from the two datasets. Almost all the data points of BIKE lie close to the $t_s = t_e$ diagonal line, implying that most of its intervals are short-lived, whereas in NFT they have high variance. This is also evident from the minimum, average, and maximum duration for the full datasets

[2] https://codeberg.org/mhm/indie-histree
[3] https://citibikenyc.com/system-data
[4] https://huggingface.co/datasets/MLNTeam-Unical/NFT-70M_transactions

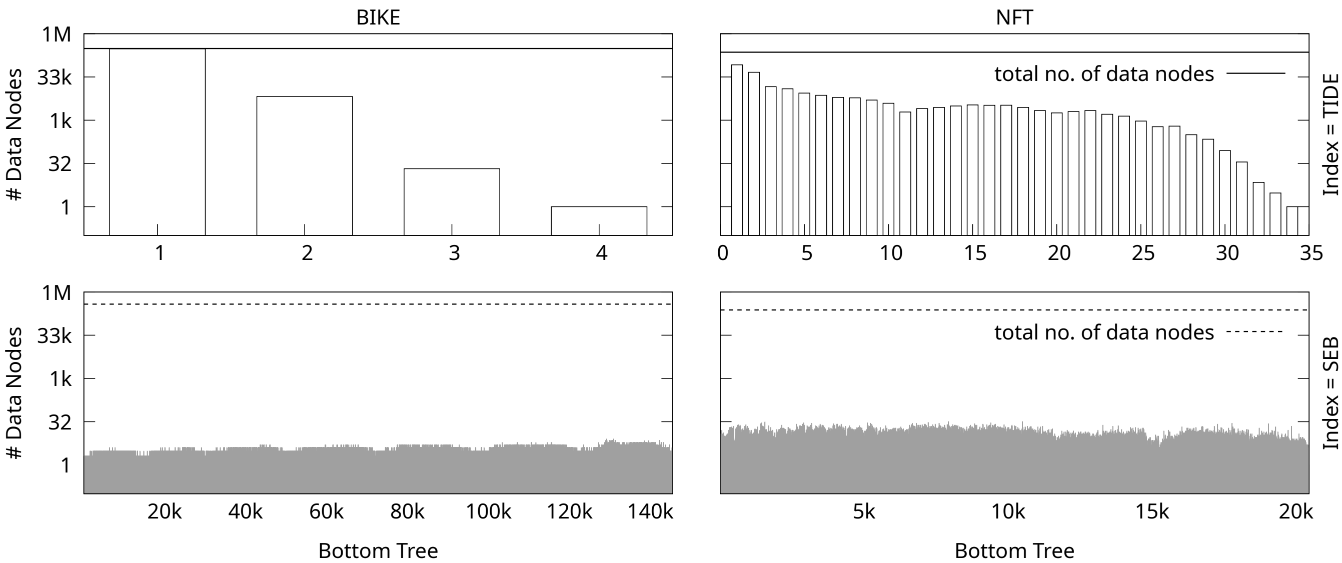


Figure 13: Number of data nodes per bottom tree

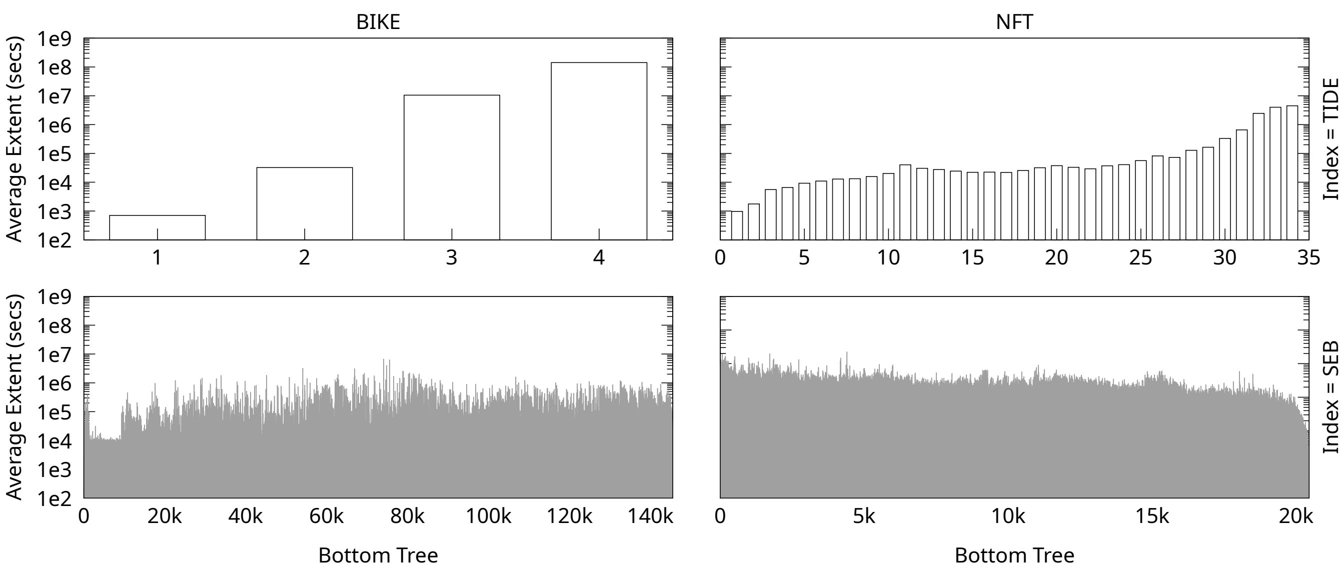


Figure 14: Average extent of data nodes per bottom tree

in Table 1. The average and maximum duration for BIKE is only 0.0004% and 8.8% of the whole extent of the dataset, in comparison to NFT's 3.2% and 99.6%. Section 5.1 investigates the size and other index characteristics, while Sections 5.2 and 5.3 evaluate insertion and query performance, respectively.

5.1 Index Characteristics

We build TIDE and SEB by sequentially inserting the BIKE and the NFT datasets. Figure 13 contains four plots, each measuring the number of data nodes (y-axis) stored in the corresponding bottom tree (x-axis) for TIDE (first row) and SEB (second row). The first (second) column shows these statistics for BIKE (NFT). Most intervals in BIKE have short durations as shown in Figure 12. Consequently, TIDE generates only four bottom trees, the first of which contains 98% of the intervals. On the other hand, SEB uses the start time t_s to index the top tree, which requires it to frequently add bottom trees to accommodate the latest intervals. This leads to numerous (145523) bottom trees, each containing only a few data nodes (below 30). The intervals in NFT are less regular (i.e., their durations have higher variance) than BIKE, yielding 34 bottom trees for TIDE. Notably, the first two bottom trees of TIDE still contain the majority (56%) of the intervals. In case of SEB, since the latest intervals in NFT are less likely to have started as recently as

Dataset	BIKE		NFT	
	TIDE	SEB	TIDE	SEB
#Data Nodes	**320091**	395187	**238201**	248306
#All Nodes	**320801**	541031	**238748**	268800
Size (GiB)	**1.22**	2.06	**0.91**	1.03
#Horizontal Splits	**320087**	249664	**238167**	227857
#Vertical Splits	**3**	145522	**33**	20448

Table 2: Number of nodes and splits

in BIKE, they are more prone to fall into past bottom trees. This, in addition to NFT being one-third of BIKE's size, leads to fewer (20449) bottom trees.

Figure 14 shows the average extent of the data nodes in every bottom tree of TIDE (first row) and SEB (second row), for BIKE (first column), and NFT (second column). The extent of a data node is the difference between the maximum end time of its intervals, and that of the previous node's. For BIKE, the first bottom tree of TIDE contains the shortest 98% of all intervals, and the remaining trees contain fewer data nodes with increasing interval durations. Similarly, the average data node extent of TIDE for NFT increases inversely to the number of data nodes shown in Figure 13. In SEB, all data nodes have a similar extent, which is significantly higher than the average node extent in TIDE. Long nodes negatively affect performance as they are expected to intersect more queries.

Table 2 lists various properties of the indexes generated by TIDE and SEB. The index size of TIDE is dominated by the data nodes in both datasets. On the other hand, for BIKE, the SEB data nodes amount to less than 60% of the total, and the bottom tree roots make up more than a quarter of all the nodes. Furthermore, each of the bottom tree roots contains only a handful of data node entries, the last of which is only partially filled. Consequently, for BIKE, SEB consumes 1.69x more space than TIDE, which is fully packed, except for the last data node of its four bottom trees. The difference in size (1.13x) is less pronounced for NFT, where the bottom tree roots constitute 7.6% of the total nodes in SEB. Table 2 also contains the number of horizontal (HS) and vertical splits (VS). Although HS is comparable in TIDE and SEB for both datasets, VS is much higher in SEB. This has a negative effect on its insertion performance, because the top tree in SEB is large, and a vertical split may be expensive.

5.2 Insertion Performance

Figure 15 shows the I/O cost (page read and write operations) of insertions versus the percentage of the dataset inserted. TIDE and SEB are based on similar append-only frameworks, which cache recently accessed nodes using an LRU buffer with 4MiB. Specifically, TIDE (SEB) caches the last/mutable data node of the most recent bottom trees based on duration (t_s values). Given the low number of bottom trees (see Figure 13), TIDE can keep in the buffer the last (mutable) data node of every B+-tree, as well as the entire top tree. This is not possible for SEB because of the numerous bottom trees. Thus, searching for the proper data node to accommodate an insertion incurs I/O cost. Moreover, SEB suffers from the high number of vertical splits (on a large top tree), as discussed in the context of Table 2. Consequently, SEB is about twice more expensive than TIDE on BIKE, and about two orders of magnitude on NFT. The very poor performance on NFT is due to its high duration variance, which impacts cache locality based on t_s, i.e., the insertion of long intervals necessitates fetching from the disk nodes with low t_s, which have not been accessed recently.

5.3 Query Performance

Figure 16 shows the cost of stabbing queries and ranges covering 0.0001% to 0.1% of the total history. Each reported result is the average of 1000 uniformly distributed queries. The output cardinality is above the diagrams. The number on top of each bar shows the ratio of SEB cost over TIDE. For stabbing queries and short ranges on BIKE, SEB is about 7000 times slower than TIDE. There are multiple reasons for the superiority of TIDE. First, TIDE has only four bottom trees, prioritizing their high level nodes in the LRU buffer, reducing disk accesses. In contrast, SEB searches numerous bottom trees with $t_s \le q_s$ because any interval starting before q_s may die after q_e. For instance, a stabbing query in the middle of the data space is expected to visit at least half of the 145523 bottom trees, which cannot be cached, leading to frequent disk accesses.

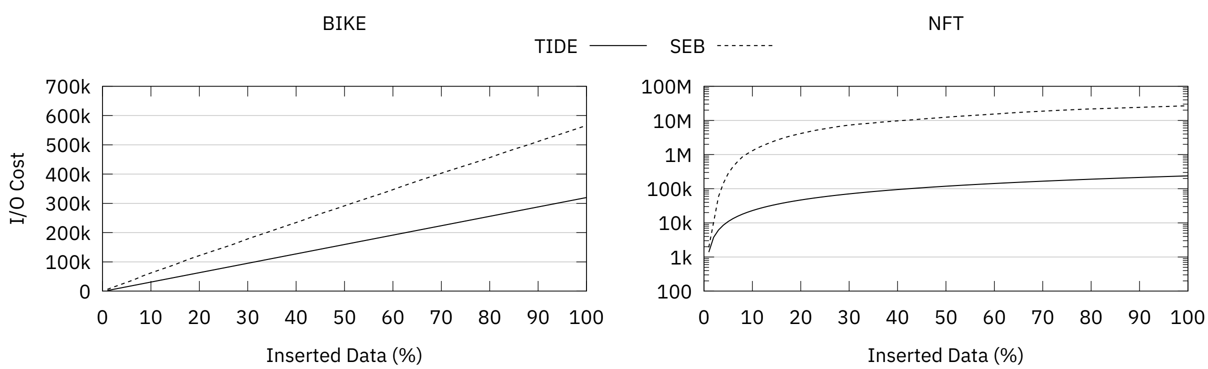


Figure 15: I/O cost of sequential insertions

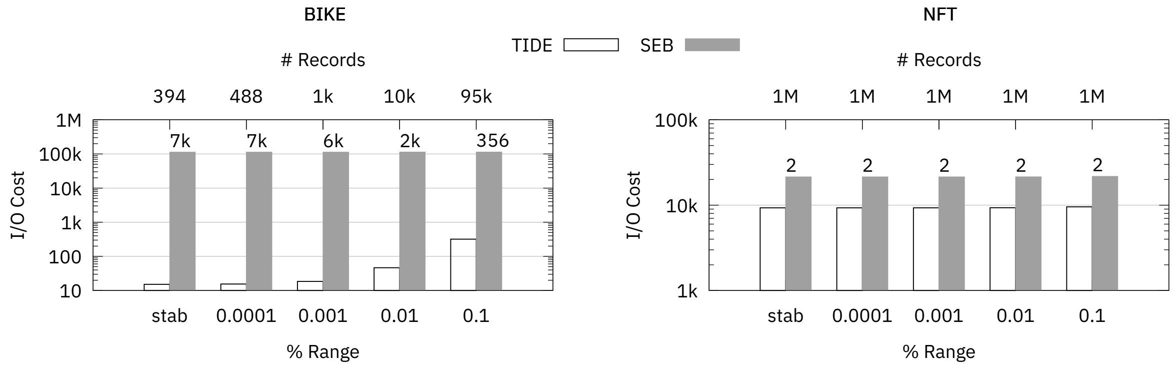


Figure 16: I/O cost of range queries

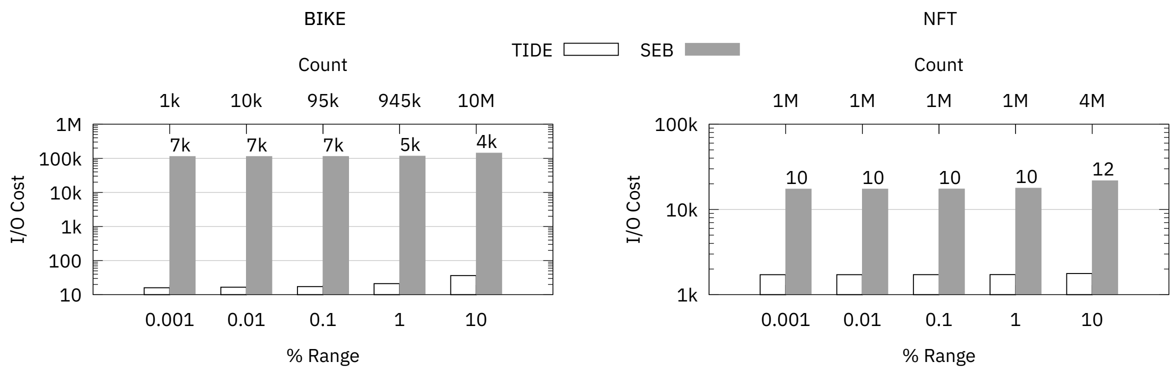


Figure 17: I/O cost of count queries

Moreover, SEB has another serious weakness: bottom trees with small t_s rarely contain results, although their mutable nodes intersect q_s. Such nodes can be very long (they extend to the current time), but are not likely to receive insertions because most BIKE intervals are short. The issue (i.e., visiting irrelevant data nodes) also exists in TIDE (e.g., D_{11} of Figure 11), but has low overhead because TIDE searches a small number of bottom trees, and only the last few may suffer. As the range increases, the difference between TIDE and SEB gradually drops (down to x356 for %0.1 ranges in BIKE) since they both have to access multiple data nodes that contain results. Although the cost of BIKE naturally increases visibly with the range length, that of SEB is rather stable, indicating that it is dominated by visits to irrelevant nodes. On NFT, TIDE is only two times faster than SEB because of the larger output cardinality. Observe that even a stabbing query retrieves 1 million intervals, and this number remains almost the same for all ranges. This is due to the average interval duration in NFT, which is 3.2% of the entire history (see Table 1), significantly higher than that of BIKE (0.0004%). Besides, NFT has diverse durations, causing TIDE to traverse more bottom trees (34 instead of 4 in BIKE). Nevertheless, compared to SEB, it still exhibits better cache locality.

Figure 17 shows the I/O cost of *count* queries for ranges covering 0.001% to 10% of history. A count query only returns the number of qualifying records, shown on top of each plot, instead of retrieving their IDs. Compared to conventional ranges, they incur less I/O cost in both TIDE and SEB because they aggregate directly the results of full nodes covered by the query. Only partially intersecting nodes need to be visited. The benefits of TIDE are even more substantial in this setting, outperforming SEB 4000-7000 times on BIKE and 10 times on NFT. This is because its bottom trees are larger than those of SEB and contain shorter nodes that may be covered by the query range. In addition, SEB examines numerous irrelevant mutable nodes, which cannot be covered since they are open-ended.

6 Conclusion

We first propose a unified representation that treats intervals as points in a two-dimensional corner space. This representation enables optimization opportunities for query processing and reveals strengths and weaknesses of different interval indexes. Then, we describe TIDE, a novel index that has desirable properties, including small size, short nodes and cache locality. We compare TIDE against its main competitor SEB, which is also based on the same IET assumption (i.e., intervals arrive in increasing order of the endpoint) with two real datasets. TIDE achieves considerable gains on both insertion and query performance under all settings.

Acknowledgments

This work was supported by GRF grant 16208623 from Hong Kong RGC.

References

[1] James F Allen. 1983. Maintaining knowledge about temporal intervals. *Commun. ACM* 26, 11 (1983), 832–843.

[2] Lars Arge and Jeffrey Scott Vitter. 1996. Optimal dynamic interval management in external memory. *Annual Symposium on Foundations of Computer Science - Proceedings* May (1996), 560–569. doi:10.1109/sfcs.1996.548515

[3] Lars Arge and Jeffrey Scott Vitter. 2003. Optimal external memory interval management. *SIAM J. Comput.* 32, 6 (2003), 1488–1508.

[4] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An efficient and robust access method for points and rectangles. *SIGMOD* (1990), 322–331.

[5] Andreas Behrend, Anton Dignös, Johann Gamper, Philip Schmiegelt, Hannes Voigt, Matthias Rottmann, and Karsten Kahl. 2019. Period Index: A Learned 2D Hash Index for Range and Duration Queries. In *Proceedings of the 16th International Symposium on Spatial and Temporal Databases* (Vienna, Austria) *(SSTD '19)*. Association for Computing Machinery, New York, NY, USA, 100–109. doi:10.1145/3340964.3340965

[6] Panagiotis Bouros, Nikos Mamoulis, Dimitrios Tsitsigkos, and Manolis Terrovitis. 2021. In-Memory Interval Joins. *The VLDB Journal* 30, 4 (April 2021), 667–691. doi:10.1007/s00778-020-00639-0

[7] Gerth Stølting Brodal, Casper Moldrup Rysgaard, and Rolf Svenning. 2023. External Memory Fully Persistent Search Trees. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing* (Orlando, FL, USA) *(STOC 2023)*. Association for Computing Machinery, New York, NY, USA, 1410–1423. doi:10.1145/3564246.3585140

[8] Matteo Ceccarello, Anton Dignös, Johann Gamper, and Christina Khnaisser. 2023. Indexing Temporal Relations for Range-Duration Queries. In *Proceedings of the 35th International Conference on Scientific and Statistical Database Management* (Los Angeles, CA, USA) *(SSDBM '23)*. Association for Computing Machinery, New York, NY, USA, Article 3, 12 pages. doi:10.1145/3603719.3603732

[9] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2022. HINT: A Hierarchical Index for Intervals in Main Memory. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1257–1270. doi:10.1145/3514221.3517873

[10] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2024. LIT: Lightning-fast In-memory Temporal Indexing. *Proc. ACM Manag. Data* 2, 1, Article 20 (mar 2024), 27 pages. doi:10.1145/3639275

[11] Davide Costa, Lucio La Cava, and Andrea Tagarelli. 2023. Unraveling the NFT economy: A comprehensive collection of Non-Fungible Token transactions and metadata. *Data in Brief* 51 (2023), 109749.

[12] Philippe Cudre-Mauroux, Eugene Wu, and Samuel Madden. 2010. Trajstore: An adaptive storage system for very large trajectory data sets. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 109–120.

[13] Victor Teixeira De Almeida and Ralf Hartmut Güting. 2005. Indexing the trajectories of moving objects in networks. *GeoInformatica* 9, 1 (2005), 33–60.

[14] J R Driscoll, N Sarnak, D D Sleator, and R E Tarjan. 1986. Making data structures persistent. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing* (Berkeley, California, USA) *(STOC '86)*. Association for Computing Machinery, New York, NY, USA, 109–121. doi:10.1145/12130.12142

[15] H Edelsbrunner. 1980. Dynamic rectangle intersection searching. *Technical Report* (1980), 47.

[16] Ramez Elmasri, Gene T. J. Wuu, and Yeong-Joon Kim. 1990. The time index—an access structure for temporal data. In *Proceedings of the Sixteenth International Conference on Very Large Databases* (Brisbane, Australia). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1–12.

[17] Cheng Hian Goh, Hongjun Lu, Beng-Chin Ooi, and Kian-Lee Tan. 1996. Indexing temporal data using existing B+-trees. *Data Knowl. Eng.* 18, 2 (March 1996), 147–165. doi:10.1016/0169-023X(95)00034-P

[18] Xiao Hu, Stavros Sintos, Junyang Gao, Pankaj K. Agarwal, and Jun Yang. 2022. Computing Complex Temporal Join Queries Efficiently. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2076–2090. doi:10.1145/3514221.3517893

[19] Paris C. Kanellakis, Sridhar Ramaswamy, Darren E. Vengroff, and Jeffrey S. Vitter. 1993. Indexing for data models with constraints and classes (extended abstract). In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Washington, D.C., USA) *(PODS '93)*. Association for Computing Machinery, New York, NY, USA, 233–243. doi:10.1145/153850.153884

[20] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter Michael Fischer, Donald Kossmann, Franz Färber, and Norman May. 2013. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. *SIGMOD* (2013), 1173–1184.

[21] Curtis P. Kolovson and Michael Stonebraker. 1991. Segment indexes: dynamic indexing techniques for multi-dimensional interval data. *SIGMOD Rec.* 20, 2 (April 1991), 138–147. doi:10.1145/119995.115807

[22] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. 2000. Managing intervals efficiently in object-relational databases. *VLDB* (2000), 407–418.

[23] Sitaram Lanka and Eric Mays. 1991. Fully persistent B+-trees. *ACM SIGMOD Record* 20, 2 (1991), 426–435.

[24] Chiang Lee and Te-Ming Tseng. 1998. Temporal Grid File: A file structure for interval data. *Data & knowledge engineering* 26, 1 (1998), 71–97.

[25] de Berg Mark, Cheong Otfried, van Kreveld Marc, and Overmars Mark. 2008. *Computational geometry algorithms and applications* (3 ed.). Spinger.

[26] Jignesh M Patel, Yun Chen, and V Prasad Chakka. 2004. STRIPES: an efficient index for predicted trajectories. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 635–646.

[27] Betty Salzberg and Vassilis J. Tsotras. 1999. Comparison of access methods for time-evolving data. *ACM Comput. Surv.* 31, 2 (jun 1999), 158–221. doi:10.1145/319806.319816

[28] A. Segev and H. Gunadhi. 1993. Efficient Indexing Methods for Temporal Relations. *IEEE Trans. on Knowl. and Data Eng.* 5, 3 (June 1993), 496–509. doi:10.1109/69.224200

[29] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB '87)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 507–518.

[30] Han Shen, Beng Chin Ooi, and Hongjun Lu. 1994. The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases. In *Proceedings of the Tenth International Conference on Data Engineering*. IEEE Computer Society, USA, 274–281.

[31] Zhexuan Song and Nick Roussopoulos. 2003. SEB-tree: An Approach to Index Continuously Moving Objects. In *Proceedings of the 4th International Conference on Mobile Data Management (MDM '03)*. Springer-Verlag, Berlin, Heidelberg, 340–344.

[32] Jeffrey Scott Vitter. 2001. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv.* 33, 2 (June 2001), 209–271. doi:10.1145/384192.384193

[33] Longhao Wang, Yu Zheng, Xing Xie, and Wei-Ying Ma. 2008. A Flexible Spatio-Temporal Indexing Scheme for Large-Scale GPS Track Retrieval. In *Proceedings of the The Ninth International Conference on Mobile Data Management (MDM '08)*. IEEE Computer Society, USA, 1–8. doi:10.1109/MDM.2008.24