

TIDY: Temporal Indexing for Die-Young Objects

PANAGIOTIS SIMATIS, Department of Computer Science and Engineering, University of Ioannina, Greece

PANAGIOTIS BOUROS, Institute of Computer Science, Johannes Gutenberg University Mainz, Germany

MANOS ATHANASSOULIS, Department of Computer Science, Boston University, USA

NIKOS MAMOULIS, Department of Computer Science and Engineering, University of Ioannina, Greece

Managing temporal intervals for processing time-travel queries has recently gained interest, with new indices proposed for memory-resident data. General-purpose interval data structures are not designed to handle high-ingestion workloads involving “die-young” temporal objects, i.e., short-lived temporal objects. In addition, they are not optimized for time intervals inserted in order of their end times. We show that this case of temporal data is quite common in real applications and propose a new temporal index, called TIDY, that achieves superior search performance compared to the state-of-the-art while supporting insertions in $O(1)$ amortized time, which is asymptotically lower than current interval indices. We model the intervals as points in a 2-d (end-time, duration) space and decompose the data into frequent short-lived intervals and less frequent long-lived ones. We manage each of these two data partitions in a dedicated index, called B-Rail, which is cache-friendly and append-only. We show that a time-range query can be decomposed to a 1-d range query in the end-time dimension and a triangular query in the 2-d (end-time, duration) space, both efficiently evaluated by the B-Rail modules of TIDY. Our experimental analysis on real-world datasets demonstrates that TIDY outperforms state-of-the-art in-memory methods for temporal objects.

CCS Concepts: • **Information systems** → **Data access methods**; **Temporal data**; **Unidimensional range search**.

Additional Key Words and Phrases: Main memory indexing, interval data, temporal data, access methods, query processing

ACM Reference Format:

Panagiotis Simatis, Panagiotis Bouros, Manos Athanassoulis, and Nikos Mamoulis. 2026. TIDY: Temporal Indexing for Die-Young Objects. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD '27)*. ACM, New York, NY, USA, 28 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 Introduction

Many applications manage large volumes of temporal data, where each object is associated with a time validity interval. Temporal databases [9, 10, 35, 42, 70] are extensions to relational databases, where records are associated with time intervals. PostgreSQL [58], Microsoft SQL Server [51] and MariaDB [50] offer support for temporal data and queries [1, 38, 45]. Due to the increase of memory

Authors' Contact Information: Panagiotis Simatis, Department of Computer Science and Engineering, University of Ioannina, Greece, p.simatis@uoi.gr; Panagiotis Bouros, Institute of Computer Science, Johannes Gutenberg University Mainz, Germany, bouros@uni-mainz.de; Manos Athanassoulis, Department of Computer Science, Boston University, Boston, MA, USA, mathan@bu.edu; Nikos Mamoulis, Department of Computer Science and Engineering, University of Ioannina, Greece, nikos@cs.uoi.gr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD '27, Huntington Beach, CA, USA

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-XXXX-X/2018/06

<https://doi.org/XXXXXXX.XXXXXXX>

capacities in commodity machines, there is much recent work on in-memory indexing of interval and temporal data [16, 21, 34, 39].

A temporal record (object, fact, or event) r is associated with a time interval $[r.tb, r.te]$, where $r.tb$ and $r.te$ model the beginning and end of the validity period of r . Quite naturally, temporal records are inserted into a historical database in order of their end timestamp $r.te$ and never deleted from it. For example, consider a temporal table T recording taxi trips, as those provided by the NYC Taxi and Limousine Commission (TLC) [54]. As soon as a taxi trip finishes, the corresponding record is added to T .

The most common query on a temporal relation is the *snapshot* or *point* query, which retrieves the objects whose validity period includes a query timestamp $q.t$ (e.g., the taxi trips which were active at time $q.t$). A *timerange* or simply *range* query is defined by a time interval $[q.tb, q.te]$ and retrieves the objects whose validity period overlaps with the query interval. Snapshot and timerange queries have been the main focus of research on temporal and interval data management [2, 16, 21, 23, 25, 31], while there has also been work on interval joins [14, 28] and temporal aggregation [42, 55].

Problem. We focus on high-performance, in-memory indexing of dynamic temporal data to efficiently process snapshot and timerange queries. New records arrive in order of their end-time $r.te$, while interleaving queries apply on the recorded temporal objects so far.

Why the State-of-the-Art is Not Enough. Temporal indexing has been well studied for both in-memory and disk-based data; however, existing methods have the following drawbacks:

- HINT [23], the state-of-the-art index for managing intervals in main memory, is optimized for static data. While it has been adapted for dynamic data in [25], it requires restructuring as the time domain evolves: new levels are introduced to the index, and controlling the number of levels optimally requires transferring intervals between partitions. In addition, HINT applies data replication and requires super-linear space; each insertion and query requires random accesses to multiple partitions. Other data structures and access methods, such as the interval tree [31] and RD-index [21], have similar drawbacks. None of these methods builds on the fact that temporal tables are append-only, and records are inserted in order of their $r.te$.
- Previous work ignores the fact that, in most applications, the temporal validity of most records is short compared to the time domain. For example, taxi trips last minutes to hours, whereas their entire history in TLC spans multiple years.

Design Goals. Every indexing structure exhibits a tradeoff between read, space, and write amplification [5]. Given the query performance focus of the state-of-the-art, we set three design goals: (G1) **exploit the order of inserted data by t_e** to allow for **more efficient ingestion**, and (G2) **minimize space** requirements, while (G3) **improving query** performance when the great majority of records have **short duration** compared to the time domain.

Our Solution. To this end, we propose a new Temporal Index for Die-Young objects (TIDY). TIDY has a totally different structure compared to the state-of-the-art HINT. TIDY models intervals as points in the t_e -duration space and decomposes each query q to a one-d $[q.tb, q.te]$ range query in the t_e space and to a triangular range query in the t_e -duration space. To address the query complexity in the transformed space, TIDY separates the indexing of the vast majority of records that have short durations from the indexing of rare, long-lived objects. As a core component of TIDY, we propose B-Rail, a novel index that decomposes and stores the records in three arrays (*ids*, *durations*, and *end times*) and uses a hierarchy of sparse indices to search them efficiently. Due to its array-based architecture, B-Rail supports insertions in $O(1)$ amortized time. This is an asymptotic improvement over existing interval indices under our target workload; tree-based structures require

an $O(\log n)$ root-to-leaf traversal per insertion, while HINT accesses up to two partitions per level of its hierarchy, whose height is logarithmic in the evolving time domain. None of them exploits the t_e -ordered arrival of temporal records, which B-Rail turns into plain appends. Although similar transformations for intervals have been applied in the past (e.g., t_s -duration mapping by RD-index [21], t_e -duration mapping in TIDE [73]), to our knowledge, TIDY is the first work that organizes the data primarily by t_e , supports updates in $O(1)$ time, and applies cache-friendly search, as it sequentially accesses query results from the dedicated *ids* arrays of its B-Rail structures.

Contributions. Our work offers the following contributions:

- We exploit two properties of temporal workloads: $[r.tb, r.te]$ records arrive ordered by $r.te$, and most are short compared to the domain, by transforming records into points in the $(t_e, duration)$ space. Under this transformation, a query is rewritten as a 1-d range query on t_e (Q1), and a triangular range query (Q2).
- We propose B-Rail, an append-only structure that stores the transformed records in three contiguous arrays (*ids*, *durations*, and *end times*), summarized by a hierarchy of layers holding $(t_e, maxd)$ metadata. B-Rail supports insertions in $O(1)$ time, answers Q1 optimally over sequentially stored results, and uses the *maxd* bounds to prune blocks during Q2.
- We introduce TIDY, which decouples the indexing of short-lived intervals from the rare long-lived ones using a duration threshold δ and a dedicated B-Rail for each class, thereby bounding the search area of Q2. As the duration distribution is unknown a priori and may drift, we propose adaptive variants of TIDY that maintain a target ratio of long intervals, including TIDY-Lrn, which self-tunes δ online.
- We evaluate TIDY on real and synthetic datasets, showing $1.8\times$ and $642\times$ faster updates than HINT [23] and R*-tree, respectively, and $2.4\times$ and $3.7\times$ faster queries on average. Even on adversarial data dominated by long intervals, where TIDY answers queries $1.9\times$ slower than HINT, it retains $3\times$ faster updates and a $4\times$ smaller memory footprint.

Outline. Section 2 presents background on temporal databases and indexing. Section 3 motivates the design of our method via an analysis of real-world datasets and existing solutions. Section 4 describes TIDY and B-Rail, our proposed indices for interval records. Section 5 presents our experimental analysis. Finally, Section 6 shows how to make TIDY an external-memory data structure, Section 7 discusses related work, and Section 8 concludes the paper.

2 Background and Motivation

We now provide the necessary background to motivate this work.

2.1 Problem Definition

A temporal relation T stores records (e.g., objects, facts, or events); each record r is associated with a validity interval $[r.tb, r.te]$, with $r.tb \leq r.te$; $r.dur = r.te - r.tb$ is the *duration* of r . Since time evolves, we denote by t_{now} the current timestamp and by D_{now} the size of the domain so far. T is populated by a stream of insertions and, as it models a growing history, we target the following update workload:

- (A1) records are inserted in order of their end time $r.te$, i.e., a record enters T at the time its validity ends;
- (A2) records are never deleted from T .

Insertions are interleaved with the following query types:

- a *snapshot query* $q.t$ finds all records $r \in T$ with $r.tb \leq q.t \leq r.te$;
- a *timerange query* $[q.tb, q.te]$, with $q.tb \leq q.te$, retrieves all records $r \in T$ whose validity interval overlaps the query interval, i.e., $q.tb \leq r.te \wedge r.tb \leq q.te$.

Table 1. Summary of notation

Symbol	Description
$[r.tb, r.te]$	validity interval of record r
$r.dur$	duration of record r
n	number of records indexed so far
t_{now}, D_{now}	current timestamp, domain size so far
$q.t, [q.tb, q.te]$	snapshot query, timerange query
Q1, Q2	the two parts of a transformed query (Section 4.1)
C	B-Rail block capacity
$maxd, maxDur$	max duration per block, global max duration
B_s, B_l	B-Rail for short/long intervals in TIDY
δ	duration threshold separating B_s from B_l
ρ	target fraction of long intervals
K, K_δ	#results of Q1, #records with $t_e \in [q.te, q.te + \delta]$
M, η	warmup size and learning rate of TIDY-Lrn

We make no assumption about the distribution or the frequency of the queries. Our goal is to design a main-memory index for T that supports the above mixed workload, while **(G1)** exploiting the arrival order of the data for efficient ingestion, **(G2)** minimizing the space requirements, and **(G3)** offering fast query processing when most records have a short duration relative to the domain. Table 1 summarizes the notation used throughout the paper.

2.2 Background and Limitations of Prior Art

Temporal Data Processing. SQL has been extended (e.g., in [38, 45]) to support the temporal selection queries defined in Section 2.1. Native range data types (e.g., temporal intervals) are part of PostgreSQL since v. 9.2, which also supports range search [58]. In this paper, we focus on temporal selection queries, due to their importance and popularity, and because they can be used as a core module of other queries, such as temporal aggregation [33, 36, 55].

Temporal Indexing in Memory. Although indexing record versions is a topic that dates back to the 1980's, recently there is a renewed interest in indexing temporal relations and interval datasets in main memory [2, 21, 23–25]. We distinguish between static intervals, where the indexed collection and its time domain are fixed, and evolving ones, where records arrive continuously and the time domain grows with t_{now} . For static data, the state-of-the-art index HINT [23] hierarchically partitions a fixed time domain into multiple levels, assigns each interval to at most two partitions per level, and answers timerange queries with mostly comparison-free partition accesses. HINT excels in this setting, outperforming alternative interval indices, including the classic interval tree [31].

Temporal databases, however, are evolving by nature. To handle evolving data, the LIT framework [25] (summarized in Figure 1) decouples the indexing of live (i.e., current) and dead (i.e., past) record versions, using a LiveIndex and a DeadIndex, respectively. When a record r is inserted at time t_{now} , it enters the LiveIndex with validity $[t_{now}, \infty)$; when it is deleted at t_{now} , LIT removes it from the LiveIndex, sets $r.te = t_{now}$, and inserts $[r.tb, r.te)$ to the DeadIndex. Queries probe both indices. The LiveIndex answers a 1-sided range query ($r.tb \leq q.te$) using a partitioned hashmap design that performs well for both queries and updates. The DeadIndex, which finds the dead versions overlapping the query range and accumulates the entire history, is a dynamic version of HINT, adapted to cope with the evolving time domain.

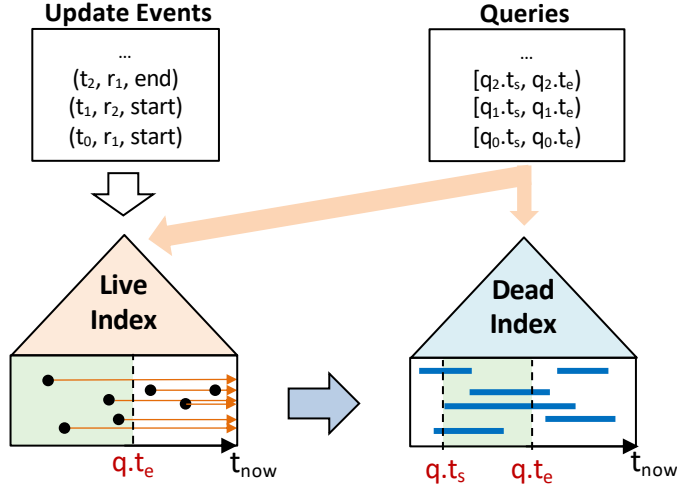


Fig. 1. The state-of-the-art temporal index, LIT [25], decouples indexing of *live* (current) and *dead* (past) intervals to (partially) separate updates from queries (image from [25]).

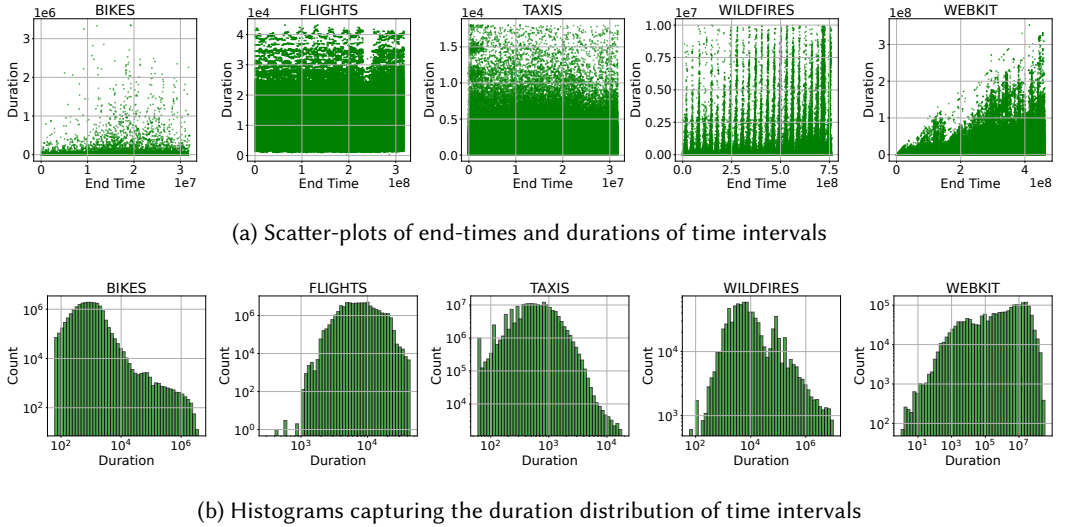


Fig. 2. With the exception of WEBKIT in the rightmost graphs, real-life temporal datasets often exhibit durations that are much shorter than the domain size. The duration is anywhere between one and four orders of magnitude smaller than the domain for the first four datasets (BIKES, FLIGHTS, TAXIS, WILDFIRES), and comparable for the fifth (WEBKIT).

HINT limitations on evolving data. Dead records are exactly the historical, end-time-ordered data that we target with TIDY. In this setup, the dynamic version of HINT has the following drawbacks:

- The beginning point $r.tb$ of a migrated interval from the LiveIndex to HINT can be arbitrary. Since HINT replicates intervals and assigns them to multiple partitions, the insertion of $[r.tb, r.te)$ randomly accesses different partitions, incurring cache misses.

Table 2. Dataset Statistics

Dataset	# Intervals	Domain Extent	Median Dur.	99th Perc. Dur.
BIKES	19.5M	31.9M	781 (0.002%)	5.4K (0.017%)
FLIGHTS	61.3M	315.5M	7.3K (0.002%)	22.5K (0.007%)
TAXIS	169.2M	31.5M	600 (0.002%)	2.6K (0.008%)
WILDFIRES	0.7M	757.3M	6.6K (0.0008%)	2.3M (0.3%)
WEBKIT	1.8M	461.8M	1.6M (0.3%)	107.6M (23%)

- To cope with the evolving time domain, HINT needs to double its domain whenever the current time t_{now} exceeds the current domain, such that the domain can include the end times $r.t_e$ of the dead records. These domain expansions, in addition to data replication, greatly increase HINT's storage requirements.
- The number of levels in HINT should be tuned to ensure good search performance. For static data, tuning can be easily done using interval statistics [23]. On the other hand, when the time domain expands it becomes challenging to maintain an optimal number of levels. Furthermore, dynamic changes to HINT levels require costly transfer of intervals between partitions [25].

Besides versioned data, there is also a need to index historical data, such as the taxi trip records discussed in the introduction. Such temporal objects are produced in order of their end-time t_e (e.g., a taxi trip is recorded at the time of its completion) in an evolving time domain, and they can be readily handled by the DeadIndex component of LIT; however, as discussed above, using a dynamic version of HINT has several shortcomings.

3 Designing *Die-Young* Indexing

In addition to addressing the shortcomings of the SOTA interval index [23] and leveraging the update order, we build on the observation that the temporal durations of objects are typically short.

Die-Young Objects are Frequent in Temporal Datasets. Table 2 presents statistics of five real datasets of temporal objects, which we use in our experiments (presented in detail in Section 5). Note that across all datasets, the median interval duration is very small compared to the domain size. In all datasets, except for WEBKIT, the 99th percentile of interval duration (last column) is negligible compared to the domain extent. Figure 2 shows the distribution of the duration for each dataset. The scatter plots in Figure 2(a) show that, across all datasets except WEBKIT, the durations are *one to four orders of magnitude smaller* compared to the maximum end-time (which represents the domain extent). For instance taxi trip durations (minutes to hours) are short compared to the entire recorded history (months to years). Figure 2(b) confirms that the great majority of intervals are very short compared to the domain extent. Once again, WEBKIT is an exceptional (pathological) case, where the percentage of long intervals is relatively large and the durations of intervals exhibit an increasing trend, as the domain evolves.

Temporal Index for Die-Young Objects (TIDY). We target the indexing of temporal relations, where the great majority of intervals are short as in four out of the five datasets above. We name our approach *Temporal Index for Die-Young objects (TIDY)*. To evaluate its robustness, our experimental analysis also includes WEBKIT, as a point of reference on how TIDY performs on adversarial datasets that include a large percentage of long-lived intervals.

LIT DeadIndex as the Bottleneck for Die-Young Objects. Table 3 compares the performance breakdown of managing live and dead intervals in LIT. On the update side, operations are categorized

Table 3. Performance breakdown: Live vs. Dead Index (LIT)

Dataset	Update Time (%)			Query Time (%)	
	Live Ins.	Live Rem.	Dead	Live	Dead
BIKES	25.41	37.42	37.17	0.10	99.90
FLIGHTS	25.70	37.48	36.83	0.03	99.97
TAXIS	26.18	37.04	36.78	0.02	99.98
WILDFIRES	21.83	28.21	49.96	0.50	99.50
WEBKIT	19.83	38.12	42.06	18.84	81.16

into three phases: insertion into the live index, removal from the live index upon expiration, and insertion into the dead index. The results validate our motivation, as maintaining the dead index accounts for more than one-third of the total update time. On the query side, the results highlight that dead index queries are the bottleneck. In most datasets, more than 99% of the query latency occurs within the dead index. A notable exception is WEBKIT, where 18.84% of the time is spent querying the live index. This occurs because WEBKIT contains numerous long intervals (as shown in Table 2), resulting in a larger number of records remaining alive for extended periods. TIDY does not target datasets such as WEBKIT, but rather applications in which record durations are very short relative to the domain. Intuitively, the larger the history of tracked data, the shorter the intervals are compared to the recorded time domain, so TIDY is appropriate for big historical data.

4 Temporal Index for *Die-Young* Objects

In this section, we present in detail the design of our new **temporal index for *die-young*** objects, which we term TIDY. We first describe a transformation from the original $[r.tb, r.te]$ intervals to $(end, duration)$ pairs that permeates our design. We then discuss how this allows us to build appropriate data structures for the transformed space and finally discuss various optimizations and how to adapt to shifting workloads.

4.1 Data Transformation and Query Rewriting

TIDY is based on an $(end, duration)$ transformation of intervals and a corresponding transformation of queries, as illustrated in Figure 3. The validity interval $[r.tb, r.te]$ of each record r becomes a 2-d point on the t_e -duration plane. A timerange query $[q.tb, q.te]$ is then rewritten as the union of two queries having disjoint results. The first query (Q1), illustrated by the dark-grey area in Figure 3, returns all records r such that $q.tb \leq r.te < q.te$. These records are guaranteed to be part of the query result. Note that this is a 1-d range query on the t_e domain, which is easy to process. The second query (Q2), illustrated by the light-grey area in Figure 3, is 2-d and has the shape of a 45-45-90 triangle including records satisfying $q.te \leq r.te \wedge r.te - q.te \leq r.dur$, which is equivalent to $r.tb \leq q.te \leq r.te$. Processing Q2 efficiently is challenging because (i) of the irregular query shape, and (ii) the triangle covers half of the data space after $q.te$. More importantly, snapshot queries have just the Q2 component, and for short timerange queries in general, Q2 is expected to return the majority of query results. Despite the challenges in processing Q2, the space transformation and query rewriting bring the following opportunities:

- Since the records are inserted in order of end-time $r.te$, they can be simply appended at the end of an index, which primarily organizes records by $r.te$. This makes insertions extremely efficient. Records are never deleted from a temporal database, so the overall handling of updates does not present challenges.

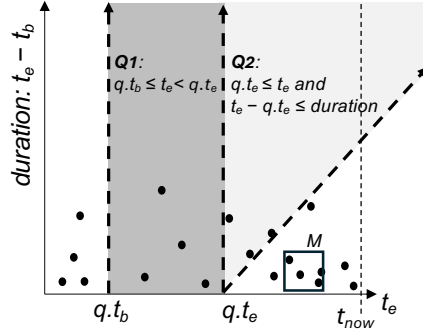


Fig. 3. TIDY transforms intervals from the $[start, end]$ domain to the $(end, duration)$ domain. This allows any timerange query to be decomposed into a full scan (Q1, dark gray area) and a scan that can be accelerated by discarding tail blocks (corresponding to the Q2, light gray triangle).

- Queries on short intervals become 1-d range queries on t_e (for Q1 and Q2) with post-filtering on the duration axis (for Q2).
- Short intervals can be separated from long ones. Q2 on long intervals covers a large part of the space, but for datasets with a small fraction of long vs. short intervals (as the ones we target), this does not have a big impact on performance.

Use of General-Purpose Indices. The transformed records can be indexed by any general-purpose 2-d index, such as the R-tree [37]. The leaf node entries of the R-tree are (t_e, dur) pairs and its non-leaf entries are $[(min(t_e), max(t_e)), (min(dur), max(dur))]$ minimum bounding rectangles (MBRs), covering all (t_e, dur) points in the pointed subtree. A depth-first search algorithm can be used for timerange queries, which, starting from the root, examines the entries at each node, visiting their pointed nodes and subtrees recursively. A non-leaf entry x is pruned if $x.max(t_e) < q.t_b$ or if its top-left corner is under the diagonal, i.e., if $x.min(t_e) - q.t_e > x.max(dur)$. If one of the above conditions applies, then there is no chance that any record in the subtree pointed by x is a query result. For example, assume that rectangle M in Figure 3 is the MBR of a non-leaf entry. Since the top-left corner of the MBR is under the diagonal, no point in it can be a result of Q2, hence, the entry (and the corresponding sub-tree) can be pruned. As we will show in Section 5, such an approach is inferior to our search- and update-optimized TIDY.

4.2 The B-Rail Index

The core data structure of TIDY is an append-only variant of the B^+ -tree designed for high-throughput memory-resident workloads, termed *B-Rail*. Unlike traditional structures that require complex rebalancing, B-Rail leverages the fact that records arrive sorted by their end-time ($r.t_e$), to achieve $O(1)$ amortized insertion costs.

B-Rail's structure, illustrated in Figure 4, resembles parallel railway tracks, where the bottom track contains the data (i.e., leaf nodes), and the upper tracks (i.e., internal nodes) store metadata to accelerate search. Specifically, the bottom level of B-Rail is three dedicated arrays of contiguous memory, storing the identifiers $r.id$, end-times $r.t_e$, and time-durations $r.dur$ of records aligned in order of $r.t_e$. For each *block* of C consecutive records, where C is a *capacity* parameter, the level above stores an entry having the first t_e in the block and the maximum duration $maxd$ in the block. For every C entries at the level above the record arrays, one entry is generated at the level above. Hence, the levels of B-Rail form a hierarchy of sparse indices on t_e , and each level is stored in a

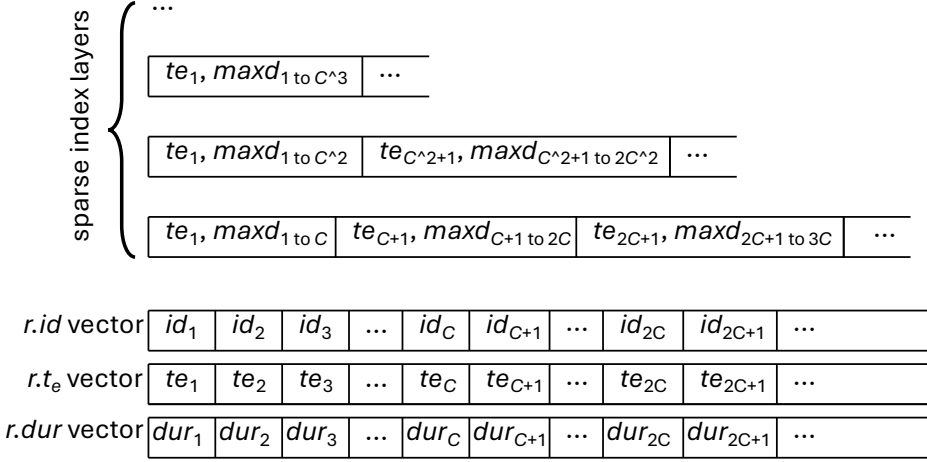


Fig. 4. B-Rail stores base data vertically decomposed in three arrays. Data is stored in order of end times (arrival order). Virtual blocks of fixed capacity C are summarized recursively by the minimum t_e and the maximum dur . The upper levels are equivalent to the inner levels of a bulk-loaded B⁺-Tree; and each parent block uses the minimum t_e and the maximum dur of its children as its metadata.

dedicated array. Note that the entries in each layer are stored in contiguous arrays (similar to the CSB⁺-tree [63] design), for fast serial accessing.

Insertions. Since the data arrives sorted by end-time, the insertion algorithm of B-Rail, appends each new interval at the end of the arrays in $O(1)$ time (design goal **G1**). At every C insertions, B-Rail appends a new entry at the sparse index above containing the block's first t_e and the maximum duration in the block. The maximum duration for the current block is updated incrementally as new intervals are added to it. For every *block* of C entries at the bottom level, a new entry is appended to the level above and so on.

Query Processing. B-Rail is very efficient for processing range queries on the t_e attribute. Consider, for instance, the Q1 query $q.tb \leq t_e < q.te$ in Figure 3. For this query, B-Rail can be used similarly to a B⁺-tree. One traversal locates the position of the first record r_f such that $r_f.te \geq q.tb$ and one more search is applied to find the last record r_l such that $r_l.te < q.te$. The sequentially stored record-ids in-between $r_f.id$ and $r_l.id$ are all reported as results.

To process the triangle query Q2, B-Rail employs the $maxd$ bounds stored together with the t_e -separators in the sparse index layers. Note that the $(t_e, maxd)$ pair stored at a B-Rail index entry x serves as a top-left corner of the MBR that bounds all points in the subtree pointed by x . Hence, if $x.min(t_e) - q.te > x.max(dur)$, we can prune the subtree pointed by x . Algorithm 1 illustrates the snapshot query algorithm on B-Rail, which is the same as the Q2 algorithm for timerange queries if we set $q.t = q.te$. The algorithm leverages both the sorted data and the hierarchical separators to optimize the search. It begins by locating the first interval whose end time is greater than or equal to $q.t$, establishing the initial leaf block for evaluation. To optimize performance, the search evaluates a global pruning condition. Specifically, if the elapsed time since $q.t$ exceeds the global maximum duration ($maxDur$), the search terminates as no subsequent intervals can satisfy the query. Furthermore, for a more granular optimization the algorithm utilizes *hierarchical skipping*. Let k be the current key (i.e., current t_e value) processed. By traversing the metadata from the highest level down, it evaluates whether the blocks in the subtree that contains k can be bypassed. If

ALGORITHM 1: Snapshot query with pruning**Input:** B-Rail index B , Query timestamp $q.t$ **Output:** Result set S

```

1   $idx \leftarrow \text{FindPosition}(B, q.t)$  // Locate entry point on the array
2   $b \leftarrow \lfloor idx / \text{CAPACITY} \rfloor$  // Identify starting leaf block
3   $S \leftarrow \emptyset$ 
4  while  $b < B.\text{numLeafBlocks}$  do
5       $start \leftarrow \max(b \times \text{CAPACITY}, idx)$ 
        // Global pruning check
6      if  $B.\text{keys}[start] - q.t > B.\text{maxDur}$  then
7          break
8       $skip \leftarrow 0$ 
        // Hierarchical skip check
9      for  $\ell \leftarrow B.\text{levels.size} - 1$  down to 0 do
10          $groupSize \leftarrow \text{CalculateGroupSize}(\ell)$ 
11         if  $b \% groupSize = 0$  then
12              $sepIdx \leftarrow b / groupSize$ 
13             if  $B.\text{keys}[start] - q.t > B.\text{levels}[\ell][sepIdx].\text{maxd}$  then
14                  $skip \leftarrow groupSize$ 
15                 break
16     if  $skip > 0$  then
17          $b \leftarrow b + skip$ 
18     else
19          $end \leftarrow \min(b \times \text{CAPACITY} + \text{CAPACITY}, B.\text{size})$ 
20         for  $i \leftarrow start$  to  $end - 1$  do
21             if  $B.\text{keys}[i] - q.t \leq B.\text{durations}[i]$  then
22                  $S \leftarrow S \cup \{B.\text{values}[i]\}$ 
23          $b \leftarrow b + 1$ 
24 return  $S$ 

```

the maximum duration (maxd) associated with a high-level separator x is smaller than the current key's k difference from $q.t$, no interval in the subtree rooted under x overlaps $q.t$, and the entire subtree is skipped. When pruning is impossible, the algorithm linearly scans the elements of the leaf block, collecting those where the temporal range encompasses $q.t$.

Complexity Analysis. The space complexity of B-Rail is $O(n)$, i.e., the same as that of a B^+ -tree of order C , but with a smaller constant due to the lack of need for empty slots (design goal **G2**). It is easy to show that the insertion cost is $O(1)$. Besides the constant time needed to append a new record to the vectors, we have to append a new entry at level ℓ with probability $1/C^\ell$; i.e., the total cost of adding new entries at levels $1, 2, \dots$ is $(1/C + 1/C^2 + \dots)$, which converges to $1/(C - 1)$. Hence, the amortized cost of an insertion is $O(1/(C - 1)) = O(1)$. This is an asymptotic improvement over existing interval indices, such as the interval tree [31], HINT [23], and an R-tree for the $(\text{end}, \text{duration})$ space, as these general-purpose methods do not exploit assumptions A1 and A2 in Section 2.1.

A Q1 query is evaluated optimally using B-Rail in $O(\log_C n + K)$ time, where K is the number of query results. A Q2 or snapshot query accesses, using the maxd bounds, all groups of C records such that the (t_e, maxd) bounds of the group are above the diagonal of the query. Assuming a uniform distribution of durations in these groups, half of the points are expected to be above the diagonal

and the expected cost of Q2 queries is also $O(\log_C n + K)$. However, in practice, the uniformity assumption does not hold in real datasets, where short durations are much more frequent than long ones, meaning that most records are under the diagonal of Q2. In this case, if $B.maxDur$ is large, $O(n)$ records may have to be scanned and verified, while K is near-zero, rendering the worst-case of Q2 $O(n)$. In the next subsections, we present TIDY, which decouples the storage of short and long intervals and achieves better performance than the simple use of B-Rail in practice, fulfilling design goal G3.

The same analysis bounds the cache misses of an insertion. Appending a record touches only the last block of each of the three vectors (ids , t_e , dur), incurring at most three cache misses. A separator is propagated to level ℓ with probability $1/C^\ell$, so the amortized number of cache misses per insertion is $O(1)$. In contrast, HINT updates up to two partitions at each level of its hierarchy per insertion, at scattered memory locations, and an R-tree of fanout f traverses a root-to-leaf path, incurring $O(\log_f n)$ misses. For queries, the above analysis directly translates to cache behavior. After $O(\log_C n)$ misses to locate $q.tb$ in the sparse levels, both Q1 and Q2 scan contiguous leaf blocks, so the accessed records are read sequentially at one miss per block, a hardware-prefetcher-friendly access pattern.

4.3 TIDY with a fixed δ threshold

We first present a version of our proposed TIDY index, where we impose a pre-determined duration threshold δ ; records r for which the duration $r.dur = r.te - r.tb$ is at most δ are considered to be *short*; the remaining records are considered to be *long*. The value of δ is set empirically, such that the great majority of records are classified as short. Setting this threshold is not easy, as the duration distribution of the records is not known a priori and it may drift over time. We return to the issue of setting δ in Section 4.4. For simplicity, we refer to the version of TIDY that uses a fixed δ threshold as TIDY- δ .

After determining δ , we use two B-Rail indices B_s and B_l , for the short-lived and the long-lived records, respectively, as illustrated in Figure 5a. In this small example, $C = 3$ and the blocks of the data vectors are illustrated, assuming that there is a long-lived object for every three short-lived ones. For every group (i.e., block) of C records, one separator x is appended to the sparse index layers having $x.te = r_f.te$, where r_f is the first record in the block and $x.maxd$ the maximum duration of the records in the block. In B_s , the $x.maxd$ of any entry cannot exceed δ .

Query Evaluation. A snapshot or timerange query is evaluated on both B_s and B_l indices for short and long intervals, respectively. Since the contents of the indices are disjoint, no duplicate results can be produced. For timerange queries, the Q1 part is straightforward to process; we simply treat each of B_s and B_l as a B⁺-tree and evaluate the range query $[q.tb, q.te]$. For a snapshot query q and the Q2 part of a timerange query, we apply Algorithm 1 on both B_s and B_l . For B_l , we use $q.te + maxDur$ as an upper bound for the t_e of the last record to be accessed. Since $maxDur$ can be arbitrarily large, the search range can be long; however, if the percentage of long intervals is limited, this is not expected to have a large effect on the overall search cost. On the other hand, the search range in the B-Rail index B_s of the short intervals is limited to $[q.te, q.te + \delta]$. In datasets where the vast majority of records are short-lived a small value of δ would be sufficient to separate these records from the rare long-lived ones and searching B_s is expected to be fast. Recall that, in snapshot queries and in the Q2 part of timerange queries, for each accessed record r , we have to verify whether $r.te - q.te \leq r.dur$, in order to include or not r in the result. Therefore, reducing the number of accessed records is crucial to TIDY's performance.

Figure 5b illustrates timerange query evaluation on TIDY- δ . The Q1 part of the query is a 1-d query on t_e , which is evaluated on B_s and B_l , ignoring the duration information and bounds. The Q2

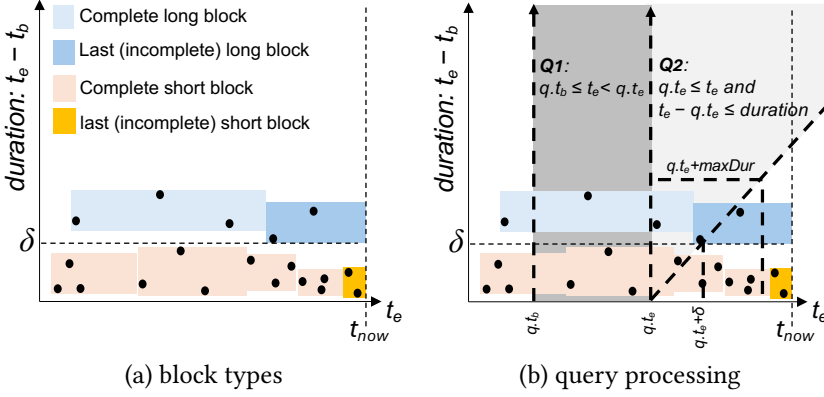


Fig. 5. TIDY with fixed δ routes incoming data to B_s (orange blocks) if $r.dur < \delta$, and to B_l (blue blocks) otherwise. This allows to physically separate the long-lived from the short-lived records and improve block skipping for snapshot queries.

part of the query on B_s applies Algorithm 1, by setting $q.t = q.t_e$ and using $q.t_e + \delta$ as a range bound. In this example, the two blocks of records that include $q.t_e$ and $q.t_e + \delta$, respectively will be accessed and for all points r in them having $q.t_e \leq r.t_e \leq q.t_e + \delta$, the predicate $r.t_e - q.t_e \leq r.dur$ will be checked. Note that the searched area corresponds to the small square that has side $[q.t_e, q.t_e + \delta]$ at its bottom. Regarding the long intervals B-Rail B_l , the two blocks that overlap the large square that has side $[q.t_e, q.t_e + maxDur]$ will be accessed and $r.t_e - q.t_e \leq r.dur$ will be verified for the records satisfying $q.t_e \leq r.t_e \leq q.t_e + maxDur$.

Data Ingestion and Complexity Analysis. Ingesting data in TIDY with a fixed δ threshold requires that for each new record r , we compare its duration to δ to determine if the record is short-lived or long-lived and append r to B-Rail B_s or B_l , respectively. Note that this separates the physical storage of records with short durations (stored in B_s) from records with long durations (in B_l). Hence, each insertion in TIDY costs $O(1)$ amortized time: a single duration comparison against δ , followed by an $O(1)$ amortized append to exactly one of the two B-Rail structures. The cache-miss analysis of Section 4.2 carries over unchanged: at most three cache misses per insertion, on the last blocks of the chosen B-Rail.

The query cost of Q2 is broken down to the query cost on B_s plus the query cost on B_l . The worst-case search cost of B_s for snapshot queries (Q2) is bounded by $O(\log_C n + K_\delta)$, where K_δ is the number of records whose t_e is in $[q.t_e, q.t_e + \delta]$ (which contains all entries below the diagonal). Assuming uniform distribution of t_e (e.g., as in discrete time semantics in versioned data [65, 69]), K_δ is expected to be $n \cdot \delta / D_{now}$, where D_{now} is the size of the time domain so far. Hence, a small δ guarantees high efficiency. Without the uniformity assumption, the bound $O(\log_C n + K_\delta)$ still holds, but K_δ can become $O(n)$ in a pathological case, where (i) the t_e of all records are immediately after $q.t_e$, and (ii) none of them overlaps the query; the search cost then becomes $O(n)$, while the result size K is near zero. We stress-test this case in Section 5.5. The cost of Q2 on B_l is bounded by $O(\log_C n + K_{maxDur})$, where K_{maxDur} is the number of records whose t_e is in $[q.t_e, q.t_e + maxDur]$. In the worst case, $maxDur = D_{now}$, and all long intervals may need to be examined. This is the reason why δ should be set such that the total number of long intervals $|B_l|$ is a small fraction of the database size n .

4.4 Adaptive TIDY Variants

A fixed threshold δ can be problematic for several reasons. First, a good δ value is hard to find without knowing the data beforehand. Second, the duration distribution may dynamically change over time, so a good initial estimate of δ may not be appropriate in the long run. Given this, we design two *adaptive* variants of TIDY, which separate short from long intervals locally, while constructing B_s and B_l , and use a variable value of δ .

Local Capacity-Driven Partitioning. The first adaptive variant of TIDY applies *local partitioning* of the incoming intervals to short and long ones before appending them to the corresponding B-Rail index (B_s or B_l). We refer to it as Local Capacity-Driven Partitioning TIDY (TIDY-Loc). The main idea is to set a global target ratio ρ of long intervals (e.g., $\rho = 0.01$) and maintain this ratio while generating the blocks of B_s and B_l , regardless of whether the local distribution of interval durations changes over the lifetime of the database. That is, at any time, if we have n records in total, $\rho \cdot n$ of these intervals will be classified as long-lived (and, hence, stored in B_l) and $(1 - \rho)n$ records will be short intervals and stored in B_s .

Given a target block capacity C and ratio ρ , the TIDY-Loc defines a *total buffer capacity* $C_{buf} = \lceil C / (1 - \rho) \rceil$. TIDY-Loc progressively collects batches of C_{buf} records in a buffer, before dividing them into long and short ones. Specifically, TIDY-Loc is updated as follows:

- (1) When a new interval arrives, it is appended to the buffer.
- (2) If the size of the buffer becomes C_{buf} , we select the $C_{buf} - C$ intervals that have the longest durations and we append them to B-Rail B_l in order of t_e . The remaining ones are appended to B_s in order of t_e . The buffer is then emptied.

To implement the above, while the buffer is populated, we keep track of the top- $(C_{buf} - C)$ durations and the corresponding buffer positions in a min-heap of size $(C_{buf} - C)$. When the buffer becomes full, we sort the buffer positions in the min-heap in t_e order and merge them with the buffer (à la *sort-merge-join*). That is, we access the records in the buffer; if a record r is in the min-heap, we move it to B_l ; otherwise we move r to B_s .

This capacity-driven logic ensures that the ratio of records in B_s to B_l is exactly $(1 - \rho)/\rho$ at the completion of every block. By calculating the outlier boundary per batch, TIDY-Loc adapts to local changes in the duration distribution as the database evolves.

Query Evaluation. Compared to TIDY- δ , the most important difference of TIDY-Loc is that it does not use a fixed δ threshold to separate short from long intervals. Another difference is that TIDY-Loc keeps the last $O(C_{buf} - C)$ intervals in a buffer, which does not distinguish between short and long intervals, until it becomes full. These differences affect the query evaluation algorithm as follows:

- To search B_s , we add the maximum duration $maxDurS$ for all short intervals encountered so far to $q.t_e$ to set the upper bound for Q2's search.
- Let t_e^{Buf} be the first t_e in the buffer. If point $(t_e^{Buf}, maxDur)$ is below the query diagonal, the buffer is ignored; otherwise, all contents of the buffer are checked for potential query results.

Learned Stochastic Threshold Adaptation. TIDY-Lrn is an adaptive TIDY variant that *learns* the distribution of interval durations as it shifts over time. TIDY-Lrn treats the duration threshold δ as a dynamic parameter that evolves through feedback-driven learning.

As in TIDY-Loc, assume that our target ratio of long intervals is ρ . To establish an initial baseline, TIDY-Lrn begins in a *warmup* phase. It buffers the first M incoming records and maintains a priority queue to track the top- k largest durations, where $k = \rho \times M$. Upon ingesting M records, the initial threshold δ that separates short from long intervals is set to the k -th largest duration (effectively the $1 - \rho$ percentile of the sample). The first M records are then flushed into the respective B-Rail structures, B_s and B_l .

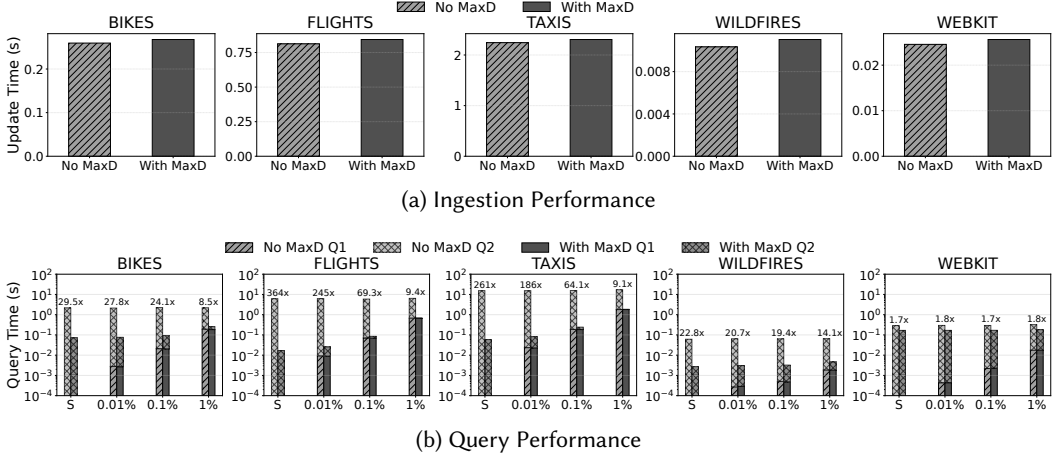


Fig. 6. While tracking *maxd* in B-Rail ingestion (top) costs around 5%, the benefit for queries (bottom) reaches up to 364x.

Once the warmup is complete, TIDY-Lrn enters the online adaptation phase. For every incoming record r , the algorithm updates δ using a stochastic gradient-based approach to maintain the target outlier ratio ρ . Specifically, let r be a new record that arrives at time t and assume that δ_t is used to label r as short or long; the threshold δ_{t+1} to be used for the next record is updated as follows:

$$\delta_{t+1} = \begin{cases} \delta_t + \eta(1 - \rho) & \text{if } r.dur > \delta_t \\ \delta_t - \eta \cdot \rho & \text{if } r.dur < \delta_t \\ \delta_t & \text{if } r.dur = \delta_t \end{cases} \quad (1)$$

This update rule, using *learning rate* η , ensures that if the update stream starts producing longer intervals, δ increases to absorb them into B_s and maintains the ρ ratio in B_l . Conversely, if intervals become shorter, δ decreases to move more data into B_s .

The update cost of TIDY-Lrn is the same as that of TIDY- δ , plus the one-time overhead of the warmup phase; after the warmup phase, it takes $O(1)$ time to append a new record to either B_s or B_l and update δ . Each record is compared against the current threshold δ_t and appended to the appropriate index.

Query Evaluation. Snapshot queries and Q2 in TIDY-Lrn are evaluated as in TIDY-Loc; the maximum duration of any short interval so far, $maxDurS$, is used to upper bound the range search in B_s . During the warmup phase, the records in M between $q.te$ and $q.te + maxDur$ are checked for potential results.

TIDY-Loc vs. TIDY-Lrn. In effect, TIDY-Lrn achieves the same goal as TIDY-Loc, without the overhead of buffering all records in C_{buf} -sized batches before deciding their assignment to either B_s or B_l . TIDY-Lrn reduces the ingestion overhead of TIDY-Loc by replacing priority queue operations with $O(1)$ arithmetic updates, making it more suitable for applications with high ingestion rates.

In addition, TIDY-Loc is more sensitive to local volatility. For example, if a spike of consecutive long intervals is divided between two buffers, the cutoff threshold could be too high for both buffers, increasing the maximum duration of intervals classified as short. TIDY-Lrn alleviates this problem by achieving smoother changes to δ , even when there are spikes in the duration distribution. Finally, TIDY-Loc is more sensitive to ties in record durations. If the $(C_{buf} - C)$ -th interval length in the buffer of TIDY-Loc is not unique, records of that length could be classified as either short or long.

5 Experimental Evaluation

All indices are in C++17, compiled with g++ using the -O3 and -mavx optimization flags.¹ Our tests ran on an Intel Core(TM) i7-14700K CPU at 5.60GHz with 128GB of RAM, running Ubuntu 24.04.3 LTS. All data and index structures reside entirely in main memory.

Experimental Setup. We evaluate TIDY using five real-world datasets with diverse characteristics, as summarized in Table 2:

- **BIKES:** Bike ride intervals in New York City from 2020, defined by the pick-up and drop-off timepoints of each trip [53].
- **FLIGHTS:** US Department of Transportation flight schedules from 2013 to 2022 capturing take-off to landing intervals [72].
- **TAXIS:** Pick-up and drop-off times of taxi trips in New York City from 2009 [54].
- **WILDFIRES:** US fire events, from 1992 to 2015, spanning discovery to containment [41].
- **WEBKIT:** Git file history between 2001 and 2016 tracking unchanged periods [14, 28, 74].

To evaluate query performance, for each dataset we generated range query workloads of 1,000 queries across extents of 0.01%, 0.1%, and 1% of the time domain. Range queries were constructed by drawing $q.t_e$ uniformly at random from the ingested time domain and setting $q.t_b = q.t_e - e$, where e is the specified percentage of the domain extent. The 1,000 queries of each workload are evenly interleaved with the update stream. We also included a workload of 1,000 snapshot (point) queries stabbing uniformly random timestamps of the domain ingested so far. Unless otherwise stated, reported results reflect total costs: update costs cover full dataset ingestion, while query times represent cumulative execution for all 1,000 queries.

5.1 B-Rail Sensitivity Analysis

B-Rail Pruning. We evaluate the benefit of storing hierarchical maximum duration (*maxd*) metadata in B-Rail. The *WithMaxD* B-Rail variant stores the maximum duration *maxd* in the internal levels. This allows the search algorithm to prune entire subtrees during Q2 if the maximum reach of the intervals in that branch cannot possibly satisfy the query's start-time constraint. In contrast, the *NoMaxD* variant operates similarly to a pure B⁺-Tree. While it remains efficient for 1-d end-time searches (i.e., Q1), it lacks duration-based pruning for Q2, necessitating long linear scans.

Figure 6a confirms that while *WithMaxD* incurs a small $O(1)$ amortized cost to propagate *maxd* values, the overhead is negligible. The search performance results in Figure 6b demonstrate the clear advantage of the pruning mechanism, achieving up to two orders of magnitude improvement over *NoMaxD*. Table 4 shows that the memory overhead introduced by *maxd* tracking is less than 0.04%; B-Rail remains highly compact while enabling significant pruning capabilities. For the remainder of this paper, we adopt *WithMaxD* in B-Rail and TIDY.

B-Rail Node Capacity. Table 5 presents the sensitivity of B-Rail to the node capacity C , by varying it from 256 to 4,096 on TAXIS. The update time shows negligible variance, hovering consistently around 2.65s to 2.69s. This stability is expected, as the $O(1)$ amortized cost of the append-only insertion is dominated by data ingestion rather than the hierarchical separator updates, which become less frequent as C increases. Similarly, query latency is resilient to changes in C , fluctuating narrowly between 0.221 to 0.225. While small C theoretically allows more fine-grained pruning, the overhead of navigating a deeper tree offsets these gains. Evidently, B-Rail is not sensitive to C , and any value in the tested range provides a robust balance between insertion efficiency and search speed. For the remainder of the paper we set B-Rail's capacity to 1024.

¹Code available in <https://github.com/sigmod27tidy/code>

Table 4. B-Rail variants memory consumption (MB).

Dataset	No MaxD	With MaxD
BIKES	233.77	233.85
FLIGHTS	736.18	736.42
TAXIS	2032.15	2032.81
WEBKIT	22.12	22.12
WILDFIRES	9.34	9.35

Table 5. B-Rail’s performance remains unaffected by the node capacity C as shown by the sensitivity analysis for TAXIS.

Capacity	256	512	1024	2048	4096
Update (s)	2.69	2.66	2.65	2.68	2.65
Query (s)	0.221	0.221	0.222	0.224	0.225

Table 6. B-Rail’s scalability on TAXIS.

Dataset %	20%	40%	60%	80%	100%
Avg. Ins. Latency (ns)	5.0	4.9	5.0	5.0	5.0
Memory (MB)	407	813	1220	1626	2033

B-Rail Scalability. We evaluate the scalability of B-Rail by measuring the average insertion time and memory consumption as the TAXIS dataset is ingested in incremental batches of 20%. As illustrated in Table 6, the average per-insertion cost remains constant. Unlike Table 5, insertions here are timed as one batch, excluding per-operation timer overhead. This empirically confirms the $O(1)$ update complexity. Furthermore, the memory footprint exhibits linear growth relative to the number of indexed intervals, validating the $O(N)$ space complexity. These results confirm that B-Rail scales gracefully with the dataset size.

5.2 B-Rail vs. B⁺-Tree Variants

We now compare B-Rail against a standard B⁺-Tree [71] and QuIT [60, 61], both of which are designed for 1-d indexing. To ensure a fair comparison, we focus on range searches over the end-time attribute t_e (i.e., finding all records where $t_e \in [a, b]$, as in Q1). This allows us to assess the raw indexing performance of B-Rail against tree structures on query types they were natively designed for.

Figure 7a presents the total ingestion cost for each dataset. B-Rail outperforms both the standard B⁺-Tree and QuIT. In the largest dataset, TAXIS, B-Rail achieves nearly 5 $\times$ ingestion throughput compared to B⁺-Tree and more than 2 $\times$ compared to QuIT. This trend is maintained across all datasets and attributed to the fact that B-Rail is optimized for append-only workloads of sorted keys. Figure 7b shows the total execution time across varying query extents. B-Rail delivers the lowest query latency in nearly all settings. Notably, the performance gap between B-Rail and the competitors widens as the query extent increases.

B-Rail achieves excellent performance due to the guaranteed locality of query results, stored sequentially in memory. Storing keys and record-ids in separate arrays is also important, as it

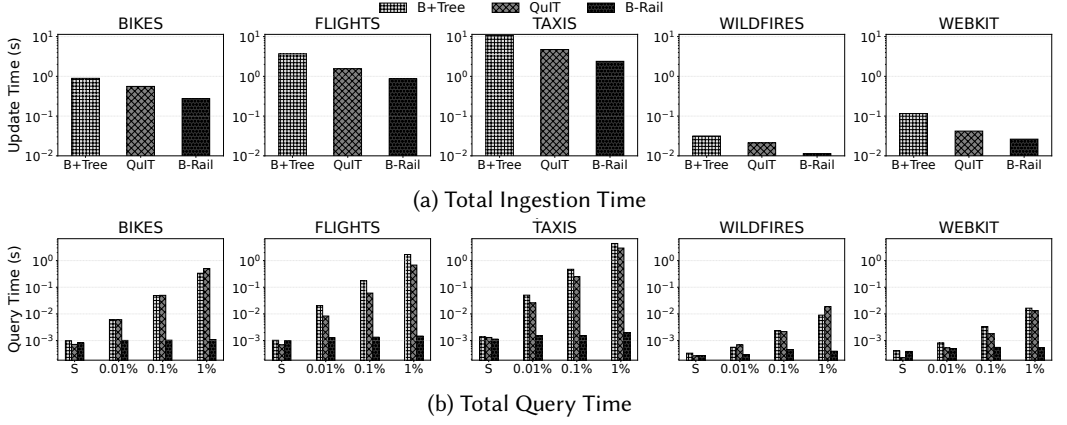


Fig. 7. B-Rail is up to $5\times$ faster on insertions (a) and orders of magnitude faster on range queries (b) than B⁺-Tree and QuIT.

avoids accessing keys in-between the first and the last key in the query range. All-in-all, by design, B-Rail greatly reduces cache misses during updates and searches, compared to general-purpose trees.

Table 7. Index footprint in MB.

Dataset	B ⁺ -Tree	QuIT	B-Rail
BIKES	313.41	304.74	233.85
FLIGHTS	987.00	550.31	736.42
TAXIS	2724.53	1683.83	2032.81
WEBKIT	29.04	16.88	22.12
WILDFIRES	12.53	13.00	9.35

Table 7 shows that B-Rail is consistently smaller than the B⁺-Tree, while remaining competitive with QuIT. B-Rail uses a contiguous array-based layout with a sparse index that stores metadata per block, and *implicit addressing* for interlinking between nodes, saving space and access cost compared to explicit pointers. In contrast, B⁺-Tree maintains pointers and keys within internal nodes to navigate the tree. We note that fully- and near-sorted data lead to worst-case space consumption for classical B⁺-Trees since most nodes are half-full, while QuIT exploits sortedness to pack the nodes [60].

5.3 TIDY Sensitivity Analysis

We now evaluate the relative behavior of TIDY variants, described in Section 4.4 and their tuning knobs.

TIDY Variants. We evaluate the effectiveness of the three partitioning strategies within the TIDY framework: TIDY- δ , TIDY-Loc, and TIDY-Lrn. TIDY- δ serves as a theoretical (i.e., *oracle*) baseline, utilizing global knowledge of the dataset (which is not known a priori) to pre-calculate a fixed duration threshold δ that perfectly isolates the top $\rho\%$ of intervals. TIDY-Loc determines the threshold locally by collecting intervals in batches of size $C/(1 - \rho)$ and using a max-priority queue to identify the K longest intervals to be routed to B_L . Finally, TIDY-Lrn treats δ as a dynamic

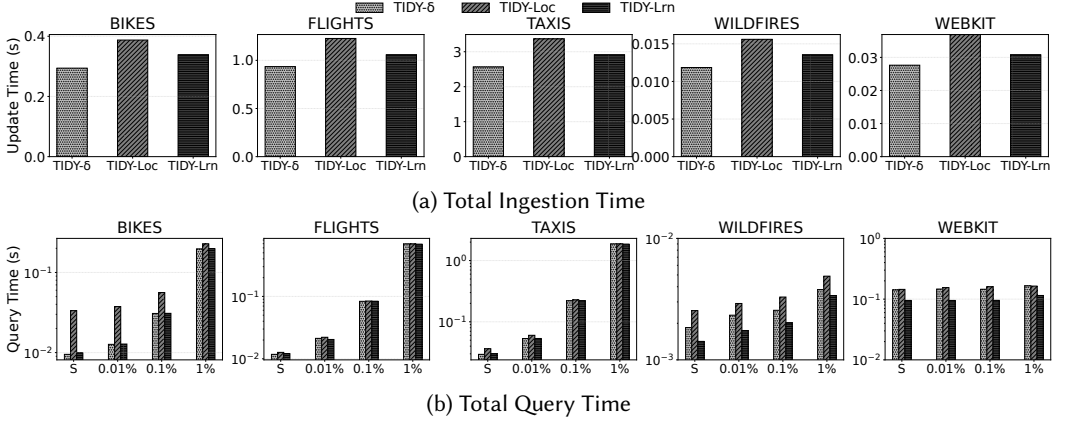


Fig. 8. TIDY-Lrn reduces TIDY-Loc’s ingestion cost (a) while matching the query performance of the TIDY- δ oracle (b) ($\rho = 1\%$).

Table 8. Sensitivity of TIDY to the outlier ratio ρ on TAXIS.

Long Intervals Ratio (ρ)	0.001	0.01	0.1
Total Ingestion Cost (s)	2.79	2.81	2.91
Total Query Cost (s)	0.078	0.072	0.068

parameter, adapting its value via a stochastic gradient-based update for every incoming record after an initial warmup phase. The learning rate η and M are set to 100 and 10,000, respectively, after experimentation (details omitted due to space limitations). For this experiment, the target ratio ρ of long intervals was set to 0.01.

Figure 8a shows the update performance of the three variants. The update costs are similar, since B-Rail treats updates as contiguous array appends followed by $O(1)$ amortized separator maintenance cost. Naturally, TIDY- δ is the fastest variant, as it does not involve any buffering or learning cost. TIDY-Lrn is faster than TIDY-Loc because updating δ at each insertion is less costly than buffering and splitting in TIDY-Loc. The query times in Figure 8b show that TIDY-Lrn achieves performance nearly identical to (and in some cases better than) the TIDY- δ oracle. The results demonstrate that automated partitioning strategies effectively optimize search speed with negligible runtime overhead. For the remainder of the paper we select TIDY-Lrn as the default TIDY.

Tuning ρ . We evaluate the impact of the ratio ρ on the performance of TIDY by varying its value across 0.001, 0.01, and 0.1; the ρ value dictates the threshold for classifying intervals as long.

As shown in Table 8, the update performance slightly drops with the increase in ρ . This is attributed to the increase in cache misses as more intervals are appended to B_l . Conversely, search performance improves as ρ increases, as $maxd$ bounds become more effective in B_s . In general, smaller (larger) ρ values should be selected in update-intensive (query-intensive) workloads. In the rest of the experiments, we use $\rho = 0.1$ by default.

5.4 TIDY vs. State-of-the-art Interval Indexing

We compare the following indices in our performance evaluation:

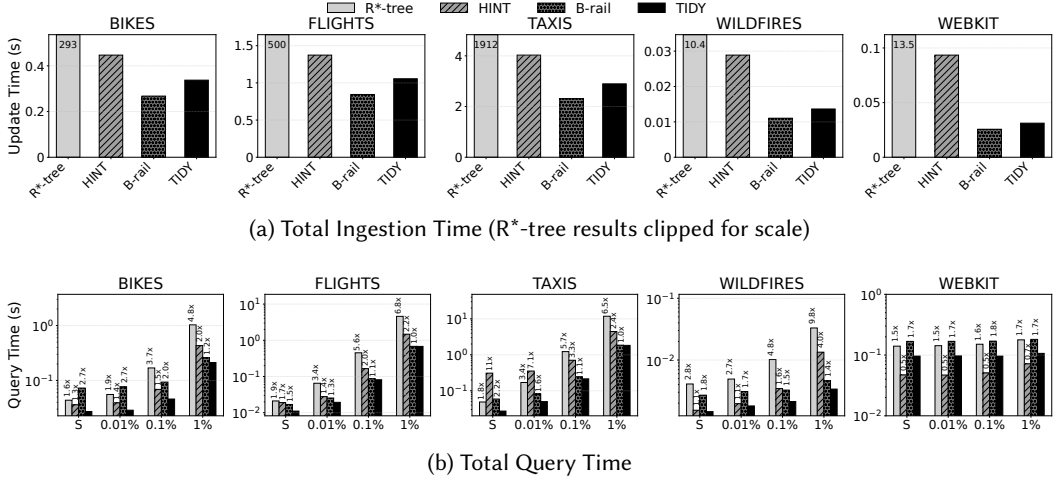


Fig. 9. TIDY offers up to 3× faster ingestion (a) and 11× faster queries (b) than HINT, beating R*-Tree and single B-Rail.

- **R*-tree**: A standard multi-dimensional index (via Boost Geometry [11]) that treats intervals as 2-d points in the (t_b, t_e) space and performs range searches via spatial intersection. A recent experimental study [47] shows that the boost R*-tree is one of the best-performing indices in range queries on 2-d points.
- **HINT** [23]: The state-of-the-art in-memory index for intervals, used as the DeadIndex in [25]. For timerange queries on memory-resident data, HINT is shown to outperform alternative indices, such as the interval tree [31], the Timeline Index [42], the period [6] and the RD-index [21], in different studies [20, 21, 23, 25].
- **B-Rail**: A single B-Rail storing all the intervals without short/long separation. We include such a competitor to evaluate the effectiveness of separating the indexing of short and long intervals in TIDY as opposed to an optimized unified index for both classes.
- **TIDY**: Our proposed framework utilizing the TIDY-Lrn variant to partition intervals to short and long and index each category by a separate B-Rail.

Figure 9a highlights the update-efficiency of TIDY, which ingests intervals 1.8× faster than HINT on average. The R*-tree is extremely slow, due to the expensive R*-tree insertions (recall that we cannot bulk-load the R-tree, as queries are interleaved with insertions in our workloads). Due to this disparity, the R*-tree costs are clipped to preserve the visual scale for the more competitive structures.

Figure 9b shows the query processing time for varying domain extents. TIDY consistently outperforms all competitors on datasets with short-lived records, demonstrating the superior pruning capabilities and cache-friendly structure of TIDY. Overall, TIDY outperforms HINT and R*-tree by 2.4× and 3.7×, respectively, on average. The stark performance advantage of TIDY is attributed mainly to the optimized, cache-friendly B-Rail structures employed in TIDY. The separation between short and long intervals employed by TIDY adds to the cost gap between TIDY and the state-of-the-art HINT, as TIDY achieves a moderate performance gain (1.6× on average) compared to using a single B-Rail.

Overall, TIDY's query performance is consistently better than HINT, except for WEBKIT where TIDY loses by 1.9× on average. WEBKIT represents an adversarial case for TIDY, as it contains a

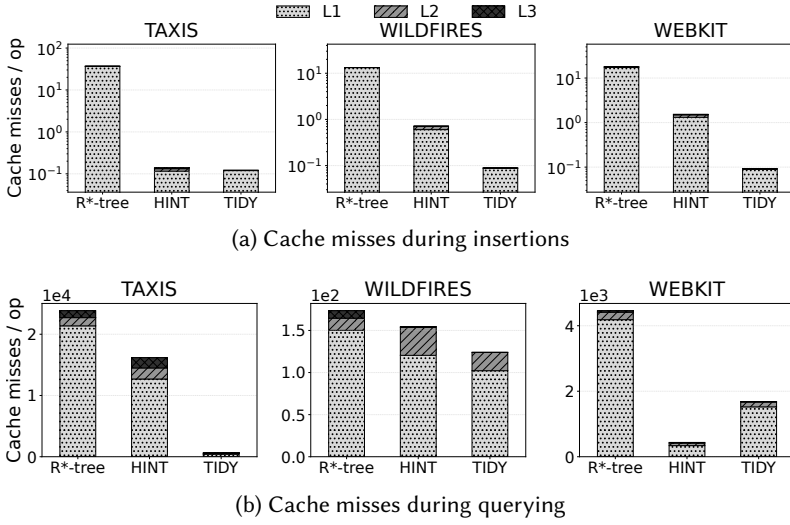


Fig. 10. Cache misses during insertions (a) and queries (b).

Table 9. Memory consumption of competing indexes (MB)

Dataset	R-tree	HINT	B-Rail	TIDY
BIKES	234.69	235.12	233.85	233.85
FLIGHTS	738.96	761.37	736.42	736.42
TAXIS	2040.50	2037.29	2032.81	2032.81
WILDFIRES	9.39	17.49	9.35	9.35
WEBKIT	22.21	95.04	22.12	22.12

large percentage of long intervals. However, on WEBKIT, TIDY has $3\times$ faster ingestion rate than HINT, which suffers from extensive replication and cache misses in insertions of long intervals.

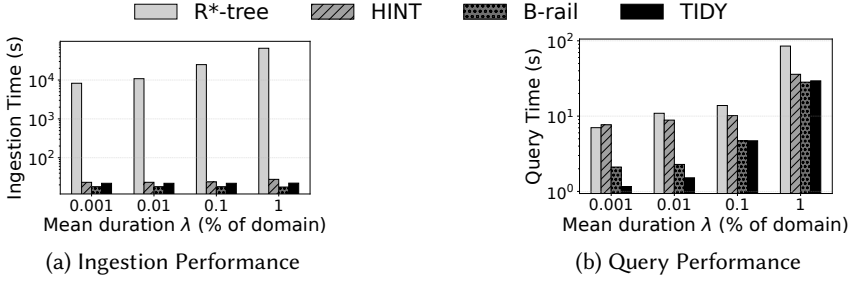
Figure 10 reports the average number of L1/L2/L3 cache misses per insertion and per query. TIDY incurs fewer than one miss per insertion on all datasets, confirming our analysis (Section 4.2). On TAXIS, short intervals are assigned to HINT's bottom levels and the t_e -ordered arrival keeps the target partitions cached, so HINT's misses remain low. However, as durations grow, intervals are replicated to more partitions across more levels, and HINT's misses rise to $6.9\times$ ($14.7\times$) those of TIDY on WILDFIRES (WEBKIT). The R*-tree incurs two orders of magnitude more misses than TIDY, as each insertion traverses a root-to-leaf path and may trigger node splits.

On the query side, TIDY's advantage stems from its sequential access pattern rather than from fewer computations alone. After locating $q.tb$, TIDY scans contiguous leaf blocks, whereas HINT and the R*-tree visit scattered partitions and nodes. On TAXIS, TIDY incurs $33\times$ and $55\times$ fewer misses than HINT and the R*-tree, respectively. On the adversarial WEBKIT, HINT incurs fewer misses than TIDY, as δ and $maxd$ bounds are large and TIDY performs long scans and numerous duration comparisons for Q2.

Table 9 shows the memory footprint of TIDY and the competitors. TIDY is the most space-efficient index, due to the contiguous array-based storage and minimal metadata overhead. HINT consumes significantly higher memory on WEBKIT and WILDFIRES, which have longer intervals and incur high replication in the partitions of HINT. The R*-tree has a slightly larger size than TIDY, due to

Table 10. 1B end-time skewed intervals (99% in a 1% hotspot).

	R-tree	HINT	B-Rail	TIDY
Total ingestion cost (s)	8,638	25.6	18.1	22.3
Total query cost (s), uniform	12.6	10.1	10.3	4.4
Total query cost (s), before hotspot	142.1	281.7	428.9	157.0

Fig. 11. Update and query times vs. Poisson mean λ .

the storage of internal node pointers and MBBs. The space difference between TIDY and B-Rail is negligible, as the two B-Rail indices of TIDY collectively store the same data as a single B-Rail.

5.5 Robustness to Non-uniform End Times

We stress test TIDY under skewed t_e distributions with a synthetic workload of one billion intervals, over a domain of 10^7 timestamps. 99% of $r.t_e$ are concentrated in a hotspot covering 1% of the domain. Interval durations follow a Zipf distribution, so that most intervals are short-lived. 1,000 timerange queries of extent 0.1% are interleaved with the updates, and we consider two query placements. In *uniform* placement, $q.t_e$ is drawn uniformly over the ingested domain. The *before hotspot* placement realizes a pathological case where every query ends within $0.43 \times \delta$ before the hotspot, so that no hotspot record ends inside the query range, but the Q2 scan on B_s always reaches into the hotspot. All indices use the same configuration as in Figure 9. As Table 10 shows, the total ingestion cost is consistent with the experiments with real data in Figure 9(a). Under *uniform* query placement, TIDY is about $2.3\times$ faster than HINT and B-Rail. Queries overlapping the hotspot access more records (i.e., larger K_δ), but since queries are placed uniformly they do not dominate the cost. TIDY remains competitive under the adversarial *before hotspot* placement. HINT is $1.8\times$ slower than TIDY, since most intervals fall in few leaf partitions, which are exhaustively scanned and verified against the query. The R*-tree keeps pace with TIDY, as it partitions the data on both endpoints simultaneously and prunes entire subtrees of intervals that start after $q.t_e$. Nevertheless, R*-tree's huge ingestion cost makes it prohibitive for streaming workloads. Note that this placement is highly contrived, since all queries must terminate right before an extremely narrow concentration of t_e .

5.6 Impact of the Duration Distribution

To evaluate the sensitivity of TIDY to the distribution of interval durations, we generated workloads in which the durations follow a Poisson distribution with a parameter λ controlling the skew. Each dataset contains one billion intervals over a time domain of 10^7 timestamps, with end times uniformly distributed over the domain. We vary λ in $\{10^2, 10^3, 10^4, 10^5\}$, such that the mean duration

ranges from 0.001% to 1% of the domain. Note that the Poisson distribution concentrates around its mean, so increasing λ shifts from die-young intervals towards uniformly long ones, without a heavy tail. Each workload interleaves 1,000 queries, uniformly distributed over the ingested domain and the query extent is fixed to 0.1%. All indices use the same configurations as in Section 5.4 (Figure 9).

Figure 11 reports the total query and ingestion times as a function of λ . TIDY's ingestion cost remains constant, while HINT's ingestion cost grows with λ , as longer intervals are replicated to more partitions. Hence, HINT's memory footprint inflates from 12.0GB to 16.5GB, whereas TIDY requires 12.0GB at all settings. The R*-tree is three orders of magnitude slower to update than TIDY at every λ .

The query cost of all indices grows with λ , as longer intervals yield more results. For die-young workloads ($\lambda \leq 10^3$, i.e., mean durations of up to 0.01% of the domain), TIDY is the fastest, outperforming a single B-Rail by up to 1.8 $\times$, HINT by 6.6 $\times$, and the R*-tree by 7.2 $\times$. As λ grows, TIDY and B-Rail converge, since the durations are concentrated around λ , there is no short/long separation to exploit, and at $\lambda = 10^5$ TIDY is 4% slower than B-Rail, while both outperform HINT and the R*-tree. Notably, TIDY calibrates its duration threshold to $\delta \approx \lambda$ at every setting, showing that the learned tuning tracks the duration distribution without prior knowledge.

5.7 Summary of Experimental Findings

Our experimental evaluation shows that (i) our novel B-Rail index, designed as a component of TIDY, strongly outperforms 1-d indices (B⁺-Tree and QuIT) that disregard the order of ingestions by t_e ; (ii) TIDY-Lrn is the most robust variant of TIDY, gracefully and efficiently adapting to distribution drifts in interval durations; and (iii) TIDY consistently outperforms the state-of-the-art in update cost over short-lived intervals by up to 3 $\times$ (fulfilling design goal G1), and up to one order of magnitude in terms of query cost (G3). For an adversarial dataset of long intervals, TIDY has 1.9 $\times$ higher query latency, however, even in this case, TIDY is 3 $\times$ faster in updates and has 4 $\times$ lower memory footprint than HINT (G2).

6 External-memory TIDY

Although our focus is memory-resident data, TIDY extends naturally to disk-based data. In the following, we sketch an external-memory (EM) version of TIDY and analyze its I/O cost.

In EM TIDY, the three arrays (ids , t_e , dur) of each of the B_s and B_l B-Rail structures are stored as three contiguous, paginated, append-only disk files, so the leaf level of each B-Rail maps directly to sequences of disk blocks. We choose the capacity parameter C of the disk-based B-Rail, such that each disk block fits C elements (e.g., ids , timestamps, or durations). The sparse separator levels are also files that grow by a factor of $1/C$ per level; hence, their total size is a small fraction of the data and they can be cached in memory, as can the tail (rightmost) blocks of B_s and B_l , which are the only blocks that insertions touch.

The complexity analysis of EM TIDY mirrors the cache-miss analyses of Sections 4.2 and 4.3, with the disk block taking the place of the cache line. An insertion appends to the pinned tail block of either B_s or B_l , incurring no I/Os. The current tail block is written out once every C insertions (i.e., $O(1/C)$ amortized I/Os per insertion). For queries, locating $q.tb$ costs $O(\log_C n)$ I/Os. From there, Q1 scans $O(K_{Q1}/C)$ consecutive blocks of the ids file, to retrieve K_{Q1} results. Q2 on B_s scans the blocks covering $[q.t_e, q.t_e + \delta]$ (i.e., $O(K_\delta/C)$ expected sequential I/Os), while on B_l the scan is bounded by the (small) size of B_l , as in the in-memory analysis.

In comparison, the External Memory Interval Tree of Arge and Vitter [4] answers queries optimally in $O(\log_C n + K/C)$ I/Os, but each insertion costs $O(\log_C n)$ I/Os. The RI-Tree [44] maps a virtual backbone tree of height h onto built-in relational B⁺-tree indexes and also pays $O(\log_C n)$

I/Os per insertion and $O(h \cdot \log_C n + K/C)$ I/Os per query, where h grows with the expansion of the data space. EM TIDY trades these per-insertion logarithmic costs for $O(1/C)$ amortized appends, at the price of query bounds that depend on K_δ rather than the total number of results K .

Three properties of TIDY become more valuable on disk. First, the sequential leaf scans of Q1 and Q2 translate to sequential reads, the access pattern on which both HDDs and SSDs achieve their highest bandwidth, whereas other external memory structures issue scattered reads. Second, the decomposition into three arrays acts as columnar storage. Q1 reads only the *ids* of the results and a couple of t_e blocks, avoiding I/O for durations. Third, the append-only ingestion performs no in-place updates, making EM TIDY directly compatible with log-structured storage.

7 Related Work

Interval Data Structures for Timerange Queries. Classic data structures for intervals, such as the interval tree [30] and segment tree [27], provide optimal $O(\log n + K)$ query time but often suffer from high space overhead or complex rebalancing requirements in dynamic settings. In the context of temporal databases, the Timeline Index [42], implemented in SAP HANA [32], manages intervals using a sorted list of endpoints, while the Period Index [6] utilizes a hierarchical grid that locally organizes the intervals based on their durations, and is suitable for range-duration queries.

State-of-the-art main-memory performance is achieved by HINT [23, 24], which employs a hierarchical partitioning of the time domain aiming at minimizing comparisons. Other recent works include TIDE [73], a disk-based index, which partitions the duration-endpoint space, and the RD-index [20, 21], which targets range-duration predicates. TIDE primarily partitions intervals by duration and builds one append-only B^+ -tree for each duration range. Queries perform one search per B^+ -tree. TIDE's goal is to minimize I/O cost at search, but disregards the CPU overhead of comparisons and cache-friendly data accessing. The RD-index is designed for range queries with duration constraints (e.g., find intervals of duration between d_{low} and d_{high} that overlap a query interval). It maps records to points in a $(start, duration)$ space, so it is not update-efficient. Finally, it is not cache-friendly. These data structures are generally outperformed by HINT in standard range query scenarios on memory-resident data. Recent extensions of HINT have addressed batch query processing [12, 18], relevance-based ranking [16], and keyword search predicates [64].

Other Queries on Temporal Data. Temporal joins [13–15, 17, 19, 28, 29, 56, 57] find record pairs which intersect temporally. In temporal aggregation [33, 36, 55], a measure is aggregated along all versions of a relation within a time period. Recently, research has expanded into independent range sampling on interval data [2, 3].

Data Structures for Near-Sorted Data. The workloads we target in this paper ingest data that is always fully sorted on one end of the interval and near-sorted on the other end. Recent work on data structures for near-sorted data quantifies sortedness [48] using the K-L sortedness metric [7, 59] and uses a smart buffering design with opportunistic bulk loading to exploit near-sortedness for efficient ingestion [62]. To further improve query performance, QuIT simplifies the design by replacing the buffer with a *fast path* that allows insertions that are in-order to skip traversing the tree [60]. While the workloads we consider exhibit near-sortedness, this line of work cannot fully exploit their two-dimensional nature with full in-order append in one dimension, as we show experimentally.

Append-only Storage and Block Summaries. Augmented tree structures [26] maintain per-subtree metadata to support interval queries. They require $O(\log n)$ time per insertion and their pointer-based layout is not cache-friendly. ChronicleDB [67] is an event store built on the Temporal Aggregated B^+ -tree, which appends events in arrival order and stores small materialized aggregates (SMA) [52] for filtering. SMA are lightweight per-block summaries, originally proposed for data

warehousing, and are most effective when the data arrive in order. B-Rail’s append-only design and *maxd* separators are reminiscent of per-block SMA. However, B-Rail manages intervals rather than instantaneous events, organizing them by end time; its *maxd* bounds enable duration-based pruning.

Cache-Friendly Indexing. Prior work on cache-friendly data structures has led to various designs [8, 22, 43, 46, 49, 63]. Our work, and specifically the B-Rail data structure, is inspired by the CSB⁺-tree [63], which packs groups of sibling nodes at the same level into a contiguous array. This allows for high locality and succinct representation of the overall tree using a few pointers per level. Since we only have appends in our workloads, we take this design to the extreme and employ only one contiguous array per level.

Learned Tuning for Data Structures. While tuning data structures is a classical research question, recent work also focuses on learning-based techniques to find optimal values for tuning knobs. For example, Adaptive Indexing [66] tunes the partitioning strategy using gradient descent. AXE [40] tunes an LSM-tree using a learned cost model, treating tuning as training a neural network with the learned cost model for its loss function. In our work, we formulate the decision of the δ knob as a simple learning problem that uses past data to calibrate using a stochastic gradient-based approach.

8 Conclusion

Our work is grounded in the observation that temporal workloads are predominantly composed of die-young intervals. Specifically, the vast majority of records exhibit short lifespans relative to the system’s total history. Driven by this insight, we developed TIDY, a framework designed to exploit these inherent temporal distributions through its core indexing structure, B-Rail. By combining a cache-conscious memory layout with duration-aware partitioning and pruning, we achieve $O(1)$ amortized insertion cost and superior query efficiency. Our evaluation on diverse real-world datasets demonstrates the superior performance of TIDY. In datasets with short-lived intervals, it consistently outperforms state-of-the-art indexes with significant gains in both update efficiency and query throughput while maintaining a compact memory footprint.

As future work, we plan to implement EM TIDY, as described in Section 6, and compare it to other disk-based approaches [73] and to a disk-based extension of LIT [68]. TIDY will also be implemented in a multi-threaded setting. TIDY’s architecture favors parallelism, as ingestion is append-only on a contiguous memory layout. To maintain high throughput, while ensuring data consistency, a Single-Writer Multi-Reader model can be adopted. Concurrent query threads can safely traverse the index up to the last committed record in a latch-free manner.

Acknowledgments

Manos Athanassoulis is partially supported by the National Science Foundation under Grant No. IIS-2144547, a Facebook Faculty Research Award, and a Meta Gift. This research has been co-financed by the European Union and Greek national funds through the Program COMPETITIVENESS under the call RESEARCH - INNOVATE (project code: EKTIAPP02-0074880).

References

- [1] Mohammed Al-Kateb, Ahmad Ghazal, Alain Crolotte, Ramesh Bhashyam, Jaiprakash Chimanchode, and Sai Pavan Pakala. 2013. Temporal query processing in Teradata. In *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*. ACM, 573–578. doi:10.1145/2452376.2452443
- [2] Daichi Amagata. 2024. Independent Range Sampling on Interval Data. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 449–461. doi:10.1109/ICDE60146.2024.00041

- [3] Daichi Amagata, Panagiotis Simatis, Panagiotis Bouros, and Nikos Mamoulis. 2026. FIRAS: A Framework for Interval Range Search and Sampling. *Proc. ACM Manag. Data* 4 (2026).
- [4] Lars Arge and Jeffrey Scott Vitter. 2003. Optimal External Memory Interval Management. *SIAM J. Comput.* 32, 6 (2003), 1488–1508. doi:10.1137/S009753970240481X
- [5] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 461–466. <http://dx.doi.org/10.5441/002/edbt.2016.42>
- [6] 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, SSTD 2019, Vienna, Austria, August 19-21, 2019*. ACM, 100–109. doi:10.1145/3340964.3340965
- [7] Sagi Ben-Moshe, Yaron Kanza, Eldar Fischer, Arie Matsliah, Mani Fischer, and Carl Staelin. 2011. Detecting and Exploiting Near-Sortedness for Efficient Relational Query Evaluation. In *Proceedings of the International Conference on Database Theory (ICDT)*. 256–267. <http://doi.acm.org/10.1145/1938551.1938584>
- [8] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 521–534. doi:10.1145/3183713.3196896
- [9] Michael H. Böhlen, Anton Dignös, Johann Gamper, and Christian S. Jensen. 2017. Temporal Data Management - An Overview. In *Business Intelligence and Big Data - 7th European Summer School, eBISS 2017, Bruxelles, Belgium, July 2-7, 2017, Tutorial Lectures (Lecture Notes in Business Information Processing, Vol. 324)*. Springer, 51–83. doi:10.1007/978-3-319-96655-7_3
- [10] Michael H. Böhlen, Richard T. Snodgrass, and Michael D. Soo. 1996. Coalescing in Temporal Databases. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*. Morgan Kaufmann, 180–191. <http://www.vldb.org/conf/1996/P180.PDF>
- [11] Boost. 2026. Boost Geometry Library. <https://www.boost.org/doc/libs/latest/libs/geometry/doc/html/index.html>. (Accessed: February 24, 2026).
- [12] Panagiotis Bouros, George Christodoulou, Christian Rauch, Artur Titkov, and Nikos Mamoulis. 2025. Querying Interval Data on Steroids. *IEEE Trans. Knowl. Data Eng.* 37, 10 (2025), 6120–6134. doi:10.1109/TKDE.2025.3597399
- [13] Panagiotis Bouros, Konstantinos Lampropoulos, Dimitrios Tsitsigkos, Nikos Mamoulis, and Manolis Terrovitis. 2020. Band Joins for Interval Data. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*. OpenProceedings.org, 443–446. doi:10.5441/002/EDBT.2020.53
- [14] Panagiotis Bouros and Nikos Mamoulis. 2017. A Forward Scan based Plane Sweep Algorithm for Parallel Interval Joins. *Proc. VLDB Endow.* 10, 11 (2017), 1346–1357. doi:10.14778/3137628.3137644
- [15] Panagiotis Bouros and Nikos Mamoulis. 2018. Interval Count Semi-Joins. In *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*. OpenProceedings.org, 425–428. doi:10.5441/002/EDBT.2018.38
- [16] Panagiotis Bouros and Nikos Mamoulis. 2025. Relevance Queries for Interval Data. *Proc. ACM Manag. Data* 3, 3 (2025), 206:1–206:26. doi:10.1145/3725343
- [17] Panagiotis Bouros, Nikos Mamoulis, Dimitrios Tsitsigkos, and Manolis Terrovitis. 2021. In-Memory Interval Joins. *VLDB J.* 30, 4 (2021), 667–691. doi:10.1007/S00778-020-00639-0
- [18] Panagiotis Bouros, Artur Titkov, George Christodoulou, Christian Rauch, and Nikos Mamoulis. 2024. HINT on Steroids: Batch Query Processing for Interval Data. In *Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28*. OpenProceedings.org, 440–446. doi:10.48786/EDBT.2024.38
- [19] Francesco Cafagna and Michael H. Böhlen. 2017. Disjoint interval partitioning. *VLDB J.* 26, 3 (2017), 447–466. doi:10.1007/S00778-017-0456-7
- [20] 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, SSDBM 2023, Los Angeles, CA, USA, July 10-12, 2023*. ACM, 3:1–3:12. doi:10.1145/3603719.3603732
- [21] Matteo Ceccarello, Anton Dignös, Johann Gamper, and Christina Khnaisser. 2025. Indexing temporal relations for range-duration queries. *Distributed Parallel Databases* 43, 1 (2025), 7. doi:10.1007/S10619-024-07452-6
- [22] Shimin Chen, Phillip B Gibbons, Todd C Mowry, and Gary Valentin. 2002. Fractal Prefetching B+-Trees: Optimizing Both Cache and Disk Performance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 157–168. <http://dl.acm.org/citation.cfm?id=564691.564710>
- [23] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2022. HINT: A Hierarchical Index for Intervals in Main Memory. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 1257–1270. doi:10.1145/3514221.3517873

- [24] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2024. HINT: a hierarchical interval index for Allen relationships. *VLDB J.* 33, 1 (2024), 73–100. doi:10.1007/S00778-023-00798-W
- [25] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2024. LIT: Lightning-fast In-memory Temporal Indexing. *Proc. ACM Manag. Data* 2, 1 (2024), 20:1–20:27. doi:10.1145/3639275
- [26] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.
- [27] Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. 2008. *Computational geometry: algorithms and applications, 3rd Edition*. Springer. doi:10.1007/978-3-540-77974-2
- [28] Anton Dignös, Michael H. Böhlen, and Johann Gamper. 2014. Overlap interval partition join. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. ACM, 1459–1470. doi:10.1145/2588555.2612175
- [29] Anton Dignös, Michael H. Böhlen, Johann Gamper, Christian S. Jensen, and Peter Moser. 2022. Leveraging range joins for the computation of overlap joins. *VLDB J.* 31, 1 (2022), 75–99. doi:10.1007/S00778-021-00692-3
- [30] Herbert Edelsbrunner. 1980. *Dynamic Rectangle Intersection Searching*.
- [31] Herbert Edelsbrunner. 1983. A New Approach to Rectangle Intersections, Part I. *Intern. J. Computer Math.* 13 (1983), 209–219.
- [32] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2011. SAP HANA database: data management for modern business applications. *SIGMOD Rec.* 40, 4 (2011), 45–51. doi:10.1145/2094114.2094126
- [33] Johann Gamper, Michael H. Böhlen, and Christian S. Jensen. 2018. Temporal Aggregation. In *Encyclopedia of Database Systems, Second Edition*, Ling Liu and M. Tamer Özsu (Eds.). Springer. doi:10.1007/978-1-4614-8265-9_386
- [34] Johann Gamper, Matteo Ceccarello, and Anton Dignös. 2022. What’s New in Temporal Databases?. In *Advances in Databases and Information Systems - 26th European Conference, ADBIS 2022, Turin, Italy, September 5-8, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13389)*. Springer, 45–58. doi:10.1007/978-3-031-15740-0_5
- [35] Dengfeng Gao, Christian S. Jensen, Richard T. Snodgrass, and Michael D. Soo. 2005. Join operations in temporal databases. *VLDB J.* 14, 1 (2005), 2–29. doi:10.1007/S00778-003-0111-3
- [36] Juozas Gordevicius, Johann Gamper, and Michael H. Böhlen. 2012. Parsimonious temporal aggregation. *VLDB J.* 21, 3 (2012), 309–332. doi:10.1007/S00778-011-0243-9
- [37] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD’84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, Beatrice Yormark (Ed.). ACM Press, 47–57. doi:10.1145/602259.602266
- [38] Suchen H. Hsu, Christian S. Jensen, and Richard T. Snodgrass. 1995. Valid-Time Selection and Projection. In *The TSQL2 Temporal Query Language*, Richard T. Snodgrass (Ed.). Kluwer, 249–296.
- [39] Xiao Hu, Stavros Sintos, Junyang Gao, Pankaj K. Agarwal, and Jun Yang. 2022. Computing Complex Temporal Join Queries Efficiently. In *SIGMOD ’22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 2076–2090. doi:10.1145/3514221.3517893
- [40] Andy Huynh, Anwesha Saha, Harshal A Chaudhari, and Manos Athanassoulis. 2025. AXE: A Task Decomposition Approach to Learned LSM Tuning. *Proceedings of the VLDB Endowment* 18, 13 (2025), 5582–5595. doi:10.14778/3773731.3773735
- [41] Kaggle. 2026. 1.88 Million US Wildfires. <https://www.kaggle.com/datasets/rtatman/188-million-us-wildfires>. (Accessed: February 24, 2026).
- [42] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter M. Fischer, Donald Kossmann, Franz Färber, and Norman May. 2013. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. ACM, 1173–1184. doi:10.1145/2463676.2465293
- [43] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott A Brandt, and Pradeep Dubey. 2010. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 339–350. doi:10.1145/1807167.1807206
- [44] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. 2000. Managing Intervals Efficiently in Object-Relational Databases. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*. Morgan Kaufmann, 407–418. <http://www.vldb.org/conf/2000/P407.pdf>
- [45] Krishna G. Kulkarni and Jan-Eike Michels. 2012. Temporal features in SQL: 2011. *SIGMOD Rec.* 41, 3 (2012), 34–43. doi:10.1145/2380776.2380786
- [46] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 38–49. doi:10.1109/ICDE.2013.6544812

- [47] Qiyu Liu, Maocheng Li, Yuxiang Zeng, Yanyan Shen, and Lei Chen. 2025. How good are multi-dimensional learned indexes? An experimental survey. *VLDB J.* 34, 2 (2025), 17. doi:10.1007/S00778-024-00893-6
- [48] Heikki Mannila. 1985. Measures of Presortedness and Optimal Sorting Algorithms. *IEEE Transactions on Computers (TC)* 34, 4 (1985), 318–325. <https://doi.org/10.1109/TC.1985.5009382>
- [49] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. 183–196. doi:10.1145/2168836.2168855
- [50] MariaDB. 2026. System Versioned Tables. <http://mariadb.com/kb/en/system-versioned-tables>. (Accessed: February 24, 2026).
- [51] Microsoft. 2026. Temporal Tables. <http://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables>. (Accessed: February 24, 2026).
- [52] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 476–487. <http://dl.acm.org/citation.cfm?id=645924.671173>
- [53] NYC. 2026. Citi Bike Trip Histories. <https://citibikenyc.com/system-data>. (Accessed: February 24, 2026).
- [54] NYC. 2026. TLC Trip Record Data. <https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page>. (Accessed: February 24, 2026).
- [55] Danila Piatov and Sven Helmer. 2017. Sweeping-Based Temporal Aggregation. In *Advances in Spatial and Temporal Databases - 15th International Symposium, SSTD*. doi:10.1007/978-3-319-64367-0_7
- [56] Danila Piatov, Sven Helmer, and Anton Dignös. 2016. An interval join optimized for modern hardware. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. IEEE Computer Society, 1098–1109. doi:10.1109/ICDE.2016.7498316
- [57] Danila Piatov, Sven Helmer, Anton Dignös, and Fabio Persia. 2021. Cache-efficient sweeping-based interval joins for extended Allen relation predicates. *VLDB J.* 30, 3 (2021), 379–402. doi:10.1007/S00778-020-00650-5
- [58] PostgreSQL. 2026. Temporal Extensions. http://wiki.postgresql.org/wiki/Temporal_Extensions. (Accessed: February 24, 2026).
- [59] Aneesh Raman, Konstantinos Karatsenidis, Subhadeep Sarkar, Matthaios Olma, and Manos Athanassoulis. 2022. BoDS: A Benchmark on Data Sortedness. In *Performance Evaluation and Benchmarking - TPC Technology Conference (TPCTC)*. 17–32. doi:10.1007/978-3-031-29576-8_2
- [60] Aneesh Raman, Konstantinos Karatsenidis, Shaolin Xie, Matthaios Olma, Subhadeep Sarkar, and Manos Athanassoulis. 2025. QuIT your B+-tree for the Quick Insertion Tree. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 451–463. <https://doi.org/10.48786/edbt.2025.36>
- [61] Aneesh Raman, Konstantinos Karatsenidis, Shaolin Xie, Matthaios Olma, Subhadeep Sarkar, and Manos Athanassoulis. 2026. QuIT Codebase. <https://github.com/BU-DiSC/quick-insertion-tree>. (Accessed: February 24, 2026).
- [62] Aneesh Raman, Subhadeep Sarkar, Matthaios Olma, and Manos Athanassoulis. 2023. Indexing for Near-Sorted Data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 1475–1488.
- [63] Jun Rao and Kenneth A. Ross. 2000. Making B+-trees Cache Conscious in Main Memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 475–486. doi:10.1145/342009.335449
- [64] Christian Rauch and Panagiotis Bouros. 2025. Fast Indexing for Temporal Information Retrieval. *Proc. ACM Manag. Data* 3, 4 (2025), 246:1–246:28. doi:10.1145/3749164
- [65] Betty Salzberg and Vassilis J. Tsotras. 1999. Comparison of Access Methods for Time-Evolving Data. *ACM Comput. Surv.* 31, 2 (1999), 158–221. doi:10.1145/319806.319816
- [66] Felix Martin Schuhknecht, Jens Dittrich, and Laurent Linden. 2018. Adaptive Adaptive Indexing. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 665–676.
- [67] Marc Seidemann, Nikolaus Glombiewski, Michael Körber, and Bernhard Seeger. 2019. Chronicledb: A high-performance event store. *ACM Transactions on Database Systems (TODS)* 44, 4 (2019), 1–45.
- [68] Panagiotis Simatis, George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2026. Scalable lightning-fast temporal indexing. *VLDB J.* 35, 3 (2026), 17. doi:10.1007/S00778-026-00968-6
- [69] Richard T. Snodgrass. 1992. Temporal Databases. In *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space, International Conference GIS - From Space to Territory: Theories and Methods of Spatio-Temporal Reasoning, Pisa, Italy, September 21-23, 1992, Proceedings (Lecture Notes in Computer Science, Vol. 639)*. Springer, 22–64. doi:10.1007/3-540-55966-3_2
- [70] Richard T. Snodgrass and Ilsoo Ahn. 1986. Temporal Databases. *Computer* 19, 9 (1986), 35–42. doi:10.1109/MC.1986.1663327
- [71] TLX. 2026. B+-Tree. <https://github.com/tlx/tlx>. (Accessed: February 24, 2026).
- [72] US Government. 2026. Bureau of Transportation Statistics. <https://www.bts.gov/>. (Accessed: February 24, 2026).
- [73] Kai Wang, Moin Hussain Moti, and Dimitris Papadias. 2025. TIDE: Indexing Time Intervals by Duration and Endpoint. In *Proceedings of the 19th International Symposium on Spatial and Temporal Data, SSTD 2025, Osaka, Japan, August*

25-27, 2025. ACM, 207–217. doi:10.1145/3748777.3748785

[74] Webkit. 2026. Reference. <https://webkit.org/>. (Accessed: February 24, 2026).

Received 17 April 2026; revised 19 August 2026; accepted 12 September 2026