AITTI: Augmented Interval Tree for Temporal Indexing

Panagiotis Simatis
University of Ioannina
Ioannina, Greece
p.simatis@uoi.gr

Panagiotis Bouros
Johannes Gutenberg University Mainz
Mainz, Germany
bouros@uni-mainz.de

Daichi Amagata*
The University of Osaka
Suita, Osaka, Japan
amagata.daichi@ist.osaka-u.ac.jp

Konstantinos Tsakalidis
University of Liverpool &
Archimedes, Athena RC
Liverpool, United Kingdom
K.Tsakalidis@liverpool.ac.uk

Nikos Mamoulis
University of Ioannina &
Archimedes, Athena RC
Ioannina, Greece
nikos@cs.uoi.gr

ABSTRACT

Querying temporal and versioned data is a classic problem that has been studied for more than four decades. In a temporal database, a record version is characterized by a temporal validity interval. Queries on temporal databases retrieve record versions that were valid at a time point or sometime within a time interval and satisfy a selection predicate on some object attribute. There is little recent work on indexing temporal data for queries with range filters on a given attribute, especially considering modern commodity hardware. We revisit this classic problem by proposing AITTI, an in-memory index, which uses an interval tree as a backbone to index the time dimension. The record versions in each interval tree node are organized by a new data structure tailored for open-ended range queries in the time-attribute domain. We theoretically analyze the search and update performance of AITTI and its component structures. We experimentally show the superiority of our index over previous work on real data.

CCS CONCEPTS

• **Information systems** → **Query operators**; **Main memory engines**; **Data access methods**.

KEYWORDS

Main memory indexing, temporal data, access methods, query processing

ACM Reference Format:
Panagiotis Simatis, Panagiotis Bouros, Daichi Amagata, Konstantinos Tsakalidis, and Nikos Mamoulis. 2026. AITTI: Augmented Interval Tree for Temporal Indexing. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD '27).* ACM, New York, NY, USA, 16 pages. https://doi.org/XXXXXXX.XXXXXXX

*Daichi Amagata also belongs to Nagoya University.

SIGMOD '27, June 13–19, 2027, Huntington Beach, CA, USA

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

Name	Salary	From	To
Smith	30K	2020/11	2022/06
Jones	60K	2020/07	2025/04
Smith	50K	2022/06	2023/11
Young	42K	2023/11	2025/09

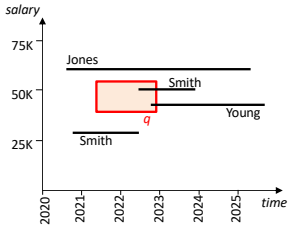

Figure 1: A Temporal employee records (left); their geometric representation and a time-travel query (right)

1 INTRODUCTION

Research in temporal databases spans several decades [21, 22, 40, 48, 63]. Temporal DBMSs track the evolution of a database to support *time-travel* queries, which retrieve records that are valid at a specified snapshot or time interval. Accordingly, database systems such as PostgreSQL[1], Microsoft SQL Server[2] and MariaDB[3] provide temporal features [2, 43, 52]. Interest in this area has surged [39], as large main memories in commodity systems make it practical to retain extensive history. Recent work [23, 26, 45] concentrates on improving the performance of time-travel search, yet comparatively little attention has been paid to indices that combine temporal predicates with range filters over additional non-temporal attributes.

Formally, we consider a *temporal relation* R, with a schema $R(A_1, A_2, \ldots, T_l, T_h)$ which includes two attributes T_l and T_h that capture the validity of the records, with $T_l \leq T_h$. Specifically, the validity of a record $r \in R$ is $r.T_l$ to $r.T_h$. The table in Figure 1 shows an example of a temporal relation R, where each record $r \in R$ is a version of an employee record, and $[r.T_l, r.T_h)$ is the employment period of r in R. The temporal relation stores record versions where some attributes (e.g., Name) are time-invariant and some (e.g., Salary) may change from version to version.

Traditional database indices (e.g., B$^+$-trees) support selection (equality or range) queries on a designated attribute A. When applied to a temporal relation, such queries additionally include temporal predicates. Listed as Query (II) in [59], such a time-travel query q is defined by a filter (range) predicate $[q.A_l, q.A_h]$ on an

[1] http://wiki.postgresql.org/wiki/Temporal_Extensions
[2] http://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables
[3] http://mariadb.com/kb/en/system-versioned-tables

attribute A of the relation and a time period $[q.T_l, q.T_h]$. The objective is to retrieve all records r in the queried temporal relation R, such that (1) the time period of the query overlaps with r's time period, i.e., $r.T_h \geq q.T_l \wedge r.T_l \leq q.T_h$, and (2) $r.A$ is inside the range $[q.A_l, q.A_h]$, i.e., $q.A_l \leq r.A \leq q.A_h$. As an example, a time-travel query q on the relation in Figure 1 seeks employees who were employed between May 2021 and December 2023 and their salary then was between 40K and 55K. Figure 1 (right) illustrates the records of the temporal table as horizontal line segments in the time-salary space and the exemplary query q as a rectangle in this space. All segments that intersect with the query rectangle are query results; i.e., the second employment record of Smith and the employment record of Young. Besides temporal databases [59] and OLAP temporal analytics [1], time-travel composite queries find applications in other domains that analyze temporal data. For example, an urban planner looking to improve the transportation infrastructure in NYC, can request all taxi trips on 2009-03-01 that took place during the rush hours from 07:00 until 09:00, carrying more than 2 passengers.

Motivation. A classic access method for temporal relations is the multi-version B-tree (MVB-tree) [13], which has optimal space and time complexity. However, the MVB-tree is designed for external memory and underperforms when the temporal database fits in memory [32]. Using the Relational Interval Tree (RI-tree) [50, 51], the authors in [37] assign intervals to a virtual tree and store them in B$^+$-trees along with the non-temporal attribute with different key orderings (including Hilbert-curve encoded keys). While it achieves optimal I/O, it requires replication, which incurs significant space overhead and high construction cost. Another indexing approach is to transform each record r to a 3-d $(r.T_l, r.T_h, r.A)$ point and model the query as a 3-d $([0, q.T_h], [q.T_l, \infty], [q.A_l, q.A_h])$ rectangle. The records can be organized by any 3-d data structure for points (e.g., an R-tree [42] or a kd-tree [16]). The drawback of such an approach is that the 3-d query extent is large (since it contains an open-ended $[q.T_l, \infty)$ range) and good performance cannot be guaranteed. For instance, a kd-tree uses $O(n)$ space to store n points, and answers 3-d range queries in $O(n^{1-1/3} + K)$ time, where K is the number of results, while the time complexity of a time travel with fractional cascading is $O(\log^2 n + K)$, however, requiring $O(n \log^2 n)$ space [33]. In a recent paper [32], a data structure that divides the domain of A in bands and defines a pure time-travel index for each band (i.e., HINT [30, 31]) was proposed. The drawback of this approach is that it may access and search a large number of data structures, if the A-filter range $[q.A_l, q.A_h]$ of the query is not selective.

Contributions. In this paper, we revisit the problem of time-travel queries on a temporal relation having a range filter predicate on an attribute A of the relation. We consider this problem in modern commodity machines, which typically have enough memory to accommodate the entire temporal relation. We propose the *Augmented Interval Tree for Temporal Indexing* (AITTI), a novel temporal index, which extends the interval tree [33], a theoretically optimal data structure for interval data, to support composite range search on both the time domain and the attribute domain.

AITTI comprises a backbone data structure, which is an interval tree that indexes the time validity periods $[r.T_l, r.T_h]$ of the records r in the temporal relation R. However, unlike the classic interval tree, each node v of the backbone structure organizes its records in appropriate data structures, which facilitate search for attribute A. Our first novel contribution is *augmenting* each node of the interval tree by three data structures, which ensure theoretically optimal search at each accessed node. During search, for each node v in the backbone structure which may contain query results, we probe exactly one of the three data structures. In particular, we show that we need to apply either just the range filter on A, or one of the possible two 3-sided queries that includes the A-filter and a one-sided range in the time domain. Depending on the case, we search the data structure that yields the query results in logarithmic time. Overall, this design of AITTI stores n records using $O(n)$ space, has query time complexity $O(\log n + K)$ *at each accessed node* of the backbone structure and supports updates in $O(\log^2 n)$ time.

While this design is theoretically sound, it is not attractive in practice for two reasons. First, it replicates information to three different data structures. Second, the three data structures used at each interval tree node are binary search trees. Although theoretically optimal, these binary trees have a high traversal cost, due to branch mispredictions and the fact that they are not cache-friendly.

Given the above, our second major contribution is a novel, unified indexing structure for one-dimensional and 3-sided 2-d range queries on horizontal segments, called RISH. Our data structure organizes the records in each node to enable fast searches and updates, while being efficient for all three query types it has to support. RISH is a cache-friendly, multi-way search tree that keeps the (A, T_l, T_h) records sorted in its leaves of maximum capacity C (as a B$^+$Tree would), while inner nodes have as keys bounds for the A, T_l, and T_h values of all records in their subtrees. The difference between RISH and an off-the-shelf multidimensional index is that RISH is used for queries that involve a closed range $[q.A_l, q.A_h]$ on attribute A and a potential open range (either $(-\infty, q.T_h]$ on T_l or $[q.T_l, \infty)$ on T_h). By optimizing RISH to consider this, we achieve excellent performance when searching the nodes of AITTI's backbone interval tree. Moreover, RISH is update-friendly, efficiently supporting arbitrary insertions and deletions to the temporal relation R.

We compare our AITTI against representative methods, which we implemented for in-memory query processing. These include HINT [30, 31], a time-only index that post-filters its results using $[q.A_l, q.A_h]$; an approach that maps records to points in a 3-d (T_l, T_h, A) space and puts them in an R-tree [42]; aHINT, an extension of HINT that handles attribute filters as suggested in [32]; and the index setting proposed in [37] which builds upon RI-tree [50, 51]. AITTI outperforms all competitors in most setups in terms of search, while being competitive with them in storage requirements and update cost.

Outline. The rest of the text is organized as follows. Section 2 defines the problem of time-travel queries and discusses the necessary background. Section 3 proposes our AITTI structure for temporal indexing and time-travel queries, and conducts a complexity analysis of the structure. Section 4 presents our RISH design for AITTI's nodes. Section 5 reports our experimental analysis. Finally, Section 6 reviews related work and Section 7 concludes our study.

Table 1: Notation and terminology summary

Notation	Description
Dataset and Queries	
R	A temporal relation containing records r
n	Number of records in R
$r.T_l, r.T_h$	The start/end timestamps of record r
$r.A$	The value of attribute A for record r
q	A time-travel query
$[q.T_l, q.T_h]$	The time interval predicate of q
$[q.A_l, q.A_h]$	The range predicate of q on attribute A
AITTI (Backbone Interval Tree)	
v	A node in the backbone interval tree
$v.c$	The center of node v
$v.A_{min}$ / $v.A_{max}$	Min/max attribute A values in v's subtree
$v.B(A)$	Data structure in v indexing records by attribute A
$v.PST(T_{l/h}, A)$	Priority Search Tree in v for 3-sided queries on $(T_{l/h}, A)$
RISH (Node Index Structure)	
u	A node in the RISH structure
C	Maximum capacity of u
$u_{A_{min}}, u_{A_{max}}$	Min/max attribute A values in u's subtree
$u_{Tl/h_{min/max}}$	Min/max T_l/T_h value for all records in u's subtree

2 PRELIMINARIES

2.1 Problem Definition and Notation

We assume a temporal relation R, whose schema contains two attributes T_l and T_h to capture the validity period of each record in R.[4] In addition, R contains a scalar attribute A on which we are interested in applying range filters.

A time-travel query q on R is defined by a time interval $[q.T_l, q.T_h]$ and a value range $[q.A_l, q.A_h]$. The objective is to retrieve all records $r \in R$, such that $r.T_h \geq q.T_l \wedge r.T_l \leq q.T_h$ and $q.A_l \leq r.A \leq q.A_h$. If $q.T_l = q.T_h$, the query is called *stabbing* time-travel query; if $q.A_l = q.A_h$, the query is an *equality* time-travel query; if $q.T_l = q.T_h$ and $q.A_l = q.A_h$, the query is an *equality-stabbing* time-travel query. Table 1 summarizes the notation and terminology used throughout the paper.

2.2 Background

Section 2.2.1 reviews the interval tree, a classic data structure for intervals which provides the basis for AITTI's temporal indexing. Section 2.2.2 recaps on the priority search tree for 3-sided range queries on 2-d points.

2.2.1 Interval tree. An interval $s = [s.T_l, s.T_h]$ is defined by a low $s.T_l$ and a high endpoint $s.T_h$. As fundamental search operations, a *range* query retrieves all intervals from an input set S that intersect a query interval $q = \langle [q.T_l, q.T_h] \rangle$ while a *stabbing* query, the intervals cut by a query point $q.T$. The interval tree [33, 35] is a worst-case optimal binary search tree for n intervals. The index is constructed in $O(n \log n)$ time, occupies $O(n)$ space, and can answer stabbing and range queries with K results in $O(\log n + K)$ time.

Each node v of the interval tree has a *center* value $v.c$ and up to two children. To construct the tree, we use the median of the T_l and T_h endpoints of all intervals as the center value of the root node v; this median can be computed in linear time (e.g., using the *median of medians* algorithm [20]). All intervals $s \in S$ containing value $v.c$ (i.e., $s.T_l \leq v.c \leq s.T_h$) are stored inside the root v in two sorted lists; $v.L$ contains the intervals $s \in v$ sorted by $s.T_l$ and $v.R$ contains the same intervals sorted by $s.T_h$. The data intervals $s \in S$, for which $s.T_h < v.c$ are assigned to the left subtree of v, and those having $s.T_l > v.c$ are assigned to the right subtree. The subtrees are constructed recursively. Given that each subtree gets at most $n/2$ intervals, the height of the tree is $O(\log n)$. Figure 2 shows an example of an interval tree (bottom) for a set of 14 intervals (top).

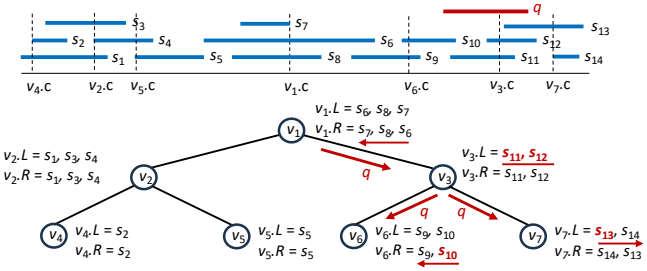

Figure 2: Example of an interval tree

To process *stabbing* queries, i.e., find all intervals that include a query value $q.T$, only one path of the tree needs to be traversed. Starting from v being the root of the tree, if $v.c \geq q.T$, then list $v.L$ is scanned from the beginning and intervals s are reported until $s.T_l > q.T$; the left child of v then becomes v and the process is repeated. If $v.c < q.T$, then list $v.R$ is scanned from the end to its beginning and the accessed intervals s are reported until $s.T_h < q.T$; search is then recursively applied to the right child of v.

For *range* queries, i.e., find the intervals that intersect query interval $[q.T_l, q.T_h]$, Algorithm 1 generalizes the stabbing search. Let v be the current node. If we have $v.c > q.T_h$ then list $v.L$ is traversed forward to report intervals s until $s.T_l > q.T_h$, and search is recursively applied to the left child of v. If we have $v.c < q.T_l$, list $v.R$ is traversed backward from its end, to report intervals s until $s.T_h < q.T_l$ and search is recursively applied to the right child of v. Finally, if $q.T_l \leq v.c \leq q.T_h$, all intervals in v are reported as results (without comparisons), and search is recursively applied to both children. Figure 2 highlights the interval tree nodes and intervals accessed for the range query q shown at the top.

2.2.2 Priority search trees. Given a set P of 2-d points, a *3-sided range* query $q = \langle [q.x_l, q.x_h], [q.y_l, \infty) \rangle$ retrieves all data points $p \in P$ with $q.x_l \leq p.x \leq q.x_h$ and $q.y_l \leq p.y$. The priority search tree (PST) [55, 68, 69] is a data structure that answers 3-sided range queries in logarithmic time. The PST is defined as follows. The data point $p_{y_{max}} = (x, y_{max})$ in P with the highest y value becomes the root of the PST. Then, the remaining points are divided into two equi-sized groups based on their median x-value x_m; P_L contains all points whose x value is at most x_m and P_R contains those having their x value greater than x_m. Then, for each of the two groups a PST is built recursively and the roots of the two trees become the children of the root of the PST. Figure 3 illustrates the PST for a set of 13 points.

Algorithm 2 illustrates the 3-sided range search on PST. Starting from the root u, we first compare $u.y$ with the lower y-bound of the query $q.y_l$. Due to the priority property of the PST with respect to y, if $u.y < q.y_l$, the subtree rooted at u cannot contain any results, as all its contents have values smaller than $u.y$. Hence, the subtree rooted at u is searched only if $u.y \geq q.y_l$. In this case, we examine

[4] For simplicity, we define this validity period to be closed at both ends. All methods discussed in this paper, can be trivially extended to support periods $[T_l, T_h)$ that are open at $s.T_h$; i.e., as typical in transaction-time temporal databases [59].

ALGORITHM 1: Range query on the interval tree

```
Input  : Interval tree I, query interval q = [q.T_l, q.T_h]
Output : set of all intervals that intersect q
1  Function ITSearch(node v, query interval q):
2      if v.c > q.T_h then
3          s ← v.L.first();
4          while s.T_l ≤ q.T_h do
5              output s;
6              s ← s.L.next();
7          ITSearch(v.leftchild, q);
8      else if v.c < q.T_l then
9          s ← v.R.last();
10         while s.T_h ≥ q.T_l do
11             output s;
12             s ← v.R.prev();
13         ITSearch(v.rightchild, q);
14     else                                   ▷ q.T_l ≤ v.c ≤ q.T_h
15         output all s ∈ v.L;
16         ITSearch(v.leftchild, q);
17         ITSearch(v.rightchild, q);
18 ITSearch(I.root, q);                ▷ Traverse the tree, depth-first
```

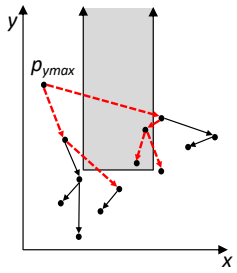

Figure 3: Example of a priority search tree and 3-sided query

ALGORITHM 2: 3-sided query on PST

```
Input  : PST I, 3-sided query q
Output : set of all points in q
1  Function PSTSearch(node u, query range q):
2      if u.y ≥ q.y_l then
3          if u.x > q.x_r then
4              PSTSearch(u.leftchild, q);
5          else if u.x < q.x_l then
6              PSTSearch(u.rightchild, q);
7          else                               ▷ q.x_l ≤ u.x ≤ q.x_r
8              output u;
9              PSTSearch(u.leftchild, q);
10             PSTSearch(u.rightchild, q);
11 PSTSearch(I.root, q);               ▷ Traverse the tree, depth-first
```

$u.x$ w.r.t. the x-range of q; if $u.x > q.x_r$, only the left subtree may contain results; if $u.x < q.x_l$, only the right subtree may contain results; finally, if $q.x_l \leq u.x \leq q.x_r$, u itself is a result and both children are searched recursively. A PST storing n points occupies $O(n)$ space and has $O(\log n + K)$ search cost for 3-sided queries.

Similar to the bottom-up construction algorithm for heaps, there is a bottom-up construction algorithm for PST which takes $O(n \log n)$ time [69]. Insertions and deletions take $O(\log n)$ time [69]. In addition, numerous variants of the PST have been proposed for the internal-memory word-RAM model of computation (e.g., [27, 38, 73]) and have been optimized for data management on secondary storage (e.g., [10, 19, 38, 47, 56, 62, 67]).

3 THE AITTI STRUCTURE

We now present AITTI, our augmented interval tree that supports time-travel queries on attribute A as defined in Section 2.1.

The backbone structure of AITTI is an interval tree [33] (see Section 2.2.1). We assign the records r of the input relation R to nodes of AITTI based on their validity intervals $[r.T_l, r.T_h]$ exactly as in the interval tree. However, unlike the interval tree, each AITTI node no longer keeps two sorted lists with the time endpoints of its records. Instead, we define three data structures for every node v. The first one is a sorted array (or a binary balanced search tree), on attribute $r.A$. We denote this data structure by $B(A)$. The second data structure is a priority search tree (PST) [55] (see Section 2.2.2), which supports the following 3-sided search: $q.A_l \leq r.A \leq q.A_h \wedge r.T_l \leq q.T_h$. We denote this data structure by $PST(T_l, A)$. Note that this data structure contains one 2-d point $(r.A, r.T_l)$ for each record r in node v. The third data structure is another priority search tree which supports the $q.A_l \leq r.A \leq q.A_h \wedge r.T_h \geq q.T_l$ 3-sided search. We denote this data structure by $PST(T_h, A)$. This data structure contains one 2-d point $(r.A, r.T_h)$ for each record r in node v.

From Section 2.1, recall that a time-travel query q is defined by a time range $[q.T_l, q.T_h]$ and a range filter $[q.A_l, q.A_h]$ on attribute A. The objective is to find all records r, such that $[r.T_l, r.T_h]$ overlaps $[q.T_l, q.T_h]$ (i.e., $\neg(r.T_h < q.T_l \vee q.T_h < r.T_l)$) and $q.A_l \leq r.A \leq q.A_h$. To evaluate the query, we conduct a range query search on AITTI, using the following recursive algorithm. Starting from the root node v, with center point $v.c$, we distinguish three cases:

(1) If $v.c > q.T_h$, then only records r in v having $r.T_l \leq q.T_h$ satisfy the temporal predicate of the query. These objects must also satisfy range filter $q.A_l \leq r.A \leq q.A_h$ to be query results. Hence, we apply a 3-sided query, which is solved optimally using $v.PST(T_l, A)$.
(2) If $v.c < q.T_l$, then only records r in v having $r.T_h \geq q.T_l$ satisfy the temporal predicate of the query. These records also have to satisfy range filter $q.A_l \leq r.A \leq q.A_h$ to be query results. Hence, we apply a 3-sided query, which is solved optimally using $v.PST(T_h, A)$.
(3) If $q.T_l \leq v.c \leq q.T_h$, then all records in v satisfy the temporal predicate of the query. To find those which also satisfy the range filter on attribute A, we apply a range search on $v.B(A)$; this is done by a applying a successor query on $v.B(A)$ to find the first result and continue the depth-first search in $v.B(A)$, until the first non-result.

Algorithm 3 illustrates time-travel search on AITTI. As an example, Figure 4 (top) shows a set of temporal records, each capturing a time interval and a value on attribute A. The figure (bottom) shows the structure of AITTI which assigns the records to nodes based on median T-endpoints as in Figure 2. The figure also shows the local data structure accessed at each of the four visited nodes (v_1, v_3, v_6, and v_7). Specifically, since for the root node v_1 we have $v_1.c < q.T_l$, we apply a 3-sided range query on the (T_h, A) points in $v_1.PST(T_h, A)$, to retrieve r_6 as a result, because point $(r_6.T_h, r_6.A)$ is inside the 3-sided query range. Then, the right child of v_1 (i.e.,

ALGORITHM 3: Time-travel query on AITTI

Input : AITTI $\mathcal{I}$, query $q = \langle [q.T_l, q.T_h], [q.A_l, q.A_h] \rangle$
Output : set of all records r, such that $[r.T_l, r.T_h]$ intersects $[q.T_l, q.T_h]$ and $q.A_l \le r.A \le q.A_h$

```
1  Function AITTISearch(node v, query q):
2      if v.c > q.T_h then                                   ▷ case 1
3          output all r ∈ v, satisfying 3-sided query
             q.A_l ≤ r.A ≤ q.A_h ∧ r.T_l ≤ q.T_h;
4          AITTISearch(v.leftchild, q);
5      else if v.c < q.T_l then                              ▷ case 2
6          output all r ∈ v, satisfying 3-sided query
             q.A_l ≤ r.A ≤ q.A_h ∧ r.T_h ≥ q.T_l;
7          AITTISearch(v.rightchild, q);
8      else                                                  ▷ case 3
9          output all r ∈ v, satisfying 2-sided query
             q.A_l ≤ r.A ≤ q.A_h;
10         AITTISearch(v.leftchild, q);
11         AITTISearch(v.rightchild, q);
12 AITTISearch(I.root, q);   ▷ Traverse the tree, depth-first
```

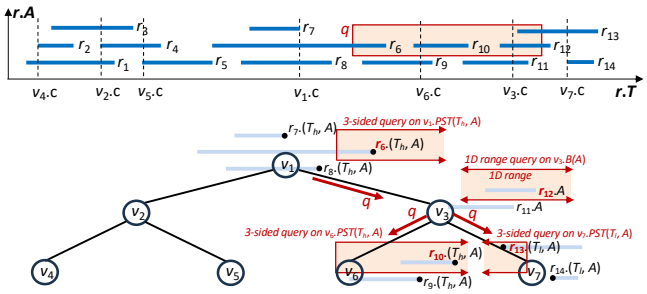

Figure 4: An AITTI for temporal records and an exemplary query q, shown as a rectangle that models a combined temporal range with an attribute range filter. At each interval tree node accessed by the query, only the searched local data structures are shown.

v_3) is searched recursively, and since $q.T_l \le v_3.c \le q.T_h$, we apply a 1-d range query on $v_3.B(A)$, looking for records in $r \in v_3$ with $q.A_l \le r.A \le q.A_h$; this finds r_{12} as a result. Last, we search v_6 and v_7, children of v_3, applying 3-sided queries on $v_6.PST(T_h, A)$ and $v_7.PST(T_l, A)$ to report results r_{10} and r_{13}, respectively.

Space and search complexity analysis. The space occupied by an AITTI storing n records is $O(n)$, as each record appears once in its data structures, the number of nodes in the backbone interval tree is $O(n)$ and each of the $B(A)$, $PST(T_l, A)$, $PST(T_h, A)$ data structures also occupies linear space to the number of records in it.

A time-travel query $q = \langle [q.T_l, q.T_h], [q.A_l, q.A_h] \rangle$ traverses two paths of the backbone interval tree and the nodes in between, so it visits $O(\log n + K_T)$ nodes, where K_T is the number of records that satisfy the $[q.T_l, q.T_h]$ temporal predicate of the query. The worst-case occurs when $[q.T_l, q.T_h]$ is not selective (e.g., $q.T_l = -\infty$ and $q.T_h = \infty$). In this case, $O(n)$ nodes of the backbone interval tree are visited and $K_T = O(n)$; if $K_T \ll K$ (i.e., the $[q.T_l, q.T_h]$ predicate is not selective), the query has $O(n)$ cost. However, typical queries in temporal databases have a selective temporal predicate and $O(\log n)$ nodes are visited from the backbone interval tree structure. For example, in stabbing queries, where $q.T_l = q.T_h$, only one path of the interval tree is searched. The worst case is that the $O(\log n)$ visited nodes collectively hold $O(n)$ records. For a node v holding m records, search using one of the $B(A)$, $PST(T_l, A)$, $PST(T_h, A)$ data structures costs $O(\log m + K_v)$, where K_v is the number of query results in v. Therefore, if the $O(\log n)$ nodes hold $O(n)$ records altogether, the total cost for the query q is $O(\log^2 n + K)$.[5] On the average case, where each of the $O(\log n)$ visited nodes holds $m = O(n)/n = O(1)$ records, the cost drops to $O(\log n + K)$.

Construction. To construct AITTI, as for the interval tree, we can assign the records in a top-down manner to the backbone interval tree structure in $O(n \log n)$ time, e.g., using *median of medians* algorithm at each node to partition its contents to its subtrees recursively (see Section 2.2.1). For each node v of the backbone structure, we can construct each of $B(A)$, $PST(T_l, A)$, and $PST(T_h, A)$ data structures in $O(n_v \log n_v)$ time, where n_v is the number of intervals assigned to node v, i.e., the construction cost of binary-balanced trees and PSTs (see Section 2.2.2). Hence, the total construction cost of AITTI is $O(n \log n) + \sum_n O(n_v \log n_v) = O(n \log n)$.

Insertions. We now explain how AITTI handles insertions. First, each node v of the backbone interval tree maintains the number of records in the sub-tree rooted at v, denoted by $S(v)$. Given a new record r, we find the first node v such that $r.T_l \le v.c \le r.T_h$. For every node v' in this search path, we update $S(v')$. If such a node exists, we insert r into its $B(A)$, $PST(T_l, A)$, and $PST(T_h, A)$. Otherwise, we create a new leaf node and initialize $B(A)$, $PST(T_l, A)$, and $PST(T_h, A)$ to include just r. If there exists a node v in the insertion path where $\max\{S(v.leftchild), S(v.rightchild)\} > \lambda S(v)$ ($\lambda \in (0.5, 1)$ is a hyper-parameter), we rebuild the sub-tree rooted at v, using the AITTI construction algorithm. Since $\max\{S(v.leftchild), S(v.rightchild)\} \le \lambda S(v)$ for every node in the interval tree, the height H of the tree satisfies $1 \le \lambda^H n$, so $H = O(\log n)$.

Next, let us analyze the insertion cost. If we find an existing node v to insert r, the cost is bounded by $O(\log n)$, because we traverse a single path with $H = O(\log n)$ and updating $B(A)$, $PST(T_l, A)$, and $PST(T_h, A)$ needs $O(\log n)$ time. The case where there exists no node v such that $r.T_l \le v.c \le r.T_h$ costs $O(\log n)$ time to traverse the interval tree. Furthermore, when rebuilding the sub-tree rooted at v, we need $O(S(v) \log S(v))$ time. Notice that, at first, we have $S(v.leftchild) \approx S(v.rightchild) \approx S(v)/2$. For $\max\{S(v.leftchild), S(v.rightchild)\} > \lambda S(v)$ to hold, we should receive $\lambda S(v) - S(v)/2 = \Omega(S(v))$ insertions. These insertions can affect each level $l \in [1, \log n]$ of the sub-tree. At level l, this sub-tree has $O(2^l)$ records, hence its rebuilding cost is $O(2^l \log 2^l) = O(l2^l)$. This means that the amortized cost per new record at this level is $O(l2^l)/2^l = O(l)$. Summing up this cost for all levels, the amortized cost is $\sum_{l=1}^{\log n} O(l) = O(\log^2 n)$. This analysis does not have any assumption; even in an extreme case, where numerous skewed insertions fall into the same node, they have the above bounded

[5] We either have $O(1)$ nodes with $O(n)$ records each, or $O(\log n)$ nodes with $O(n/\log n)$ records each. In both cases the per-node complexities sum up to $O(\log^2 n + K)$, as $\sum K_v = K$.

cost. A practical λ value that achieves a tradeoff between frequent rebuilds and balancing would be 0.75.

Deletions. To delete a record r, we first find the node v that contains r in $O(\log n)$ time. Then, we remove the record from the $v.B(A)$, $v.PST(T_l, A)$ and $v.PST(T_h, A)$ structures, which also costs $O(\log n)$ time. The sub-tree size $S(v')$ is updated accordingly for each node v' along the path to v. After the deletion, if v becomes empty, we keep it in the backbone interval tree as an empty node. Let n' be the number of nodes in the interval tree. If the current number of records n drops below $n'/2$, we rebuild AITTI. The amortized cost for rebuilding AITTI after a deletion is $O(n \log n)/0.5n = O(\log n)$.

***A*-pruning optimization**, To improve the performance of AITTI in practice, we introduce an A-pruning optimization. For each node v of the backbone interval tree, we maintain a range $[A_{min}, A_{max}]$ with the minimum and maximum A-values of all records stored in the subtree rooted at v, similar to [5, 54]. This range is kept at v's parent. At Lines 4, 7, 10, and 11 of Algorithm 3, we recursively search the corresponding child node only if its $[A_{min}, A_{max}]$ range intersects $[q.A_l, q.A_h]$. The $[A_{min}, A_{max}]$ range for each node of the backbone structure is updated after insertions and deletions.

For insertions, when traversing AITTI to insert a record r, we update the bounds of each visited node v along the path with $v.A_{min} = \min(v.A_{min}, r.A)$ and $v.A_{max} = \max(v.A_{max}, r.A)$. This adds a negligible constant overhead, preserving the $O(\log n)$ complexity. If a record r is deleted from a node v and $r.A = v.A_{min}$, we may have to update A_{min} for v and the nodes along the deletion path. For this, we first check whether the deleted $r.A$ equals the A_{min} of one of v's children; if so, we do not have to do any updates to v or any other node. However, if the deleted $r.A$ is smaller than the A_{min} bounds of v's children, we update $v.A_{min}$ and the corresponding bounds along the deletion path by the minimum of the smallest A value in v and the A_{min} bounds of v's two children. The smallest A value in v can be found in $O(\log n)$ time by traversing the leftmost path of $B(A)$. A symmetric procedure is applied if the deleted $r.A$ equals the current $v.A_{max}$. Overall, the deletion complexity is unaffected by the maintenance of A_{min} and A_{max} bounds.

Points of criticism. Although the three-data structures design of $B(A)$, $PST(T_l, A)$, and $PST(T_h, A)$ at each AITTI node is theoretically appealing, it has significant drawbacks in practice. The most important issue is the processing of queries at each node v, where one of these data structures must be used. Since all of them are binary trees, their access during queries will be slow. Specifically, for $B(A)$ we require a tree search to locate the first query result and from thereon, we need to traverse the tree, by a *successor* function, to obtain the next nodes in order and output them to the query result one-by-one until we find a node that is outside the search range $[q.A_l, q.A_h]$. Besides being computationally slow, such a process incurs random memory accesses and therefore *cache misses*. For the PST structures, we also have to visit the nodes of the tree one-by-one, starting from the root, collecting query results, each requiring random accesses and comparisons (see Section 2.2.2).

Another drawback of these data structures is that they are slow to update, as insertions and deletions require tree traversals and potentially rebuilding. In addition, each update to the base data requires changing three data structures at the affected interval tree node, rather than one. Finally, we need three times the space that would be required if a single data structure per node v of the backbone interval tree was used.

4 RISH: RANGE INDEX FOR 1-D AND 3-SIDED QUERIES ON HORIZONTAL SEGMENTS

In this section, we propose a practical alternative to the three-data structure design of $B(A)$, $PST(T_l, A)$, and $PST(T_h, A)$, which takes less space and has lower search and update costs in practice. For this purpose, we devise a novel in-memory tree index, called RISH, which unifies all three $B(A)$, $PST(T_l, A)$, and $PST(T_h, A)$ into a single data structure at each AITTI node. For efficiency and scalability, RISH is a *multi-way* tree, where each node holds up to C entries. RISH organizes the records r based on their $r.A$ attribute values, and inner nodes keep bounds for the A, T_l, and T_h in their subtrees.

4.1 Motivation and Index Design

Recall from Section 3 that AITTI should support three possible querying operations at each node of its backbone interval tree. All three operations include closed and possibly small range $[q.A_l, q.A_h]$ predicate on the A attribute. If the center timestamp $v.c$ for the accessed AITTI node v is included in the query timerange $[q.T_l, q.T_h]$ no other predicates apply on v's contents; otherwise, one open-ended predicate (either $r.T_l \leq q.T_h$ or $r.T_h \geq q.T_l$) is added to the query. Therefore, in all cases, our new RISH index should support range search on A, but the potential $r.T_l \leq q.T_h$ and $r.T_h \geq q.T_l$ are not always applied and when they are, they (1) are open-ended on one side (i.e., not selective) and (2) apply on different attributes of r, i.e., either $r.T_l$ or $r.T_h$.

This motivates us to primarily organize the records r in RISH by $r.A$. To build an RISH from a static set of records, we sort the records by $r.A$ and pack them into the leaf nodes of the RISH with capacity C. The inner node levels are constructed recursively. For each RISH leaf u, we compute the minimum $u_{A_{min}}$ and the maximum $u_{A_{max}}$ values of A from the node records, the minimum T_l value $u_{Tl_{min}}$, and the maximum T_h value $u_{Th_{max}}$. These four bounds form the key of the node u, which is stored in its parent node together with a pointer u_{ptr} to u. For any inner node u, the $[u_{A_{min}}, u_{A_{max}}]$ range is the minimum and maximum A value of all records in its subtree, and the $u_{Tl_{min}}$ and $u_{Th_{max}}$ are the minimum and maximum T_l and T_h, respectively, of all records in its subtree. The entries in an inner node are ordered by $[u_{A_{min}}, u_{A_{max}}]$ and, for two consecutive entries e_i, e_{i+1} in an inner node, it is guaranteed that $e_i.u_{A_{max}} \leq e_{i+1}.u_{A_{min}}$.

Now, let us revisit the three enumerated cases in Section 3 concerning the center value $v.c$ of the current AITTI node v with respect to the query time-range:

(1) If $v.c > q.T_h$, then we compute the 3-sided query $q.A_l \leq r.A \leq q.A_h \wedge r.T_l \leq q.T_h$, using bound $u_{Tl_{min}}$ at each visited node u of v's RISH to prune u if $u_{Tl_{min}} > q.T_h$; bounds $u_{Th_{max}}$ are not used by this query. After solving the problem for the current node v, we recursively search $v.leftchild$. If, for a RISH node u, the range $[u_{A_{min}}, u_{A_{max}}]$ is *covered by* the query A-range $[q.A_l, q.A_h]$, the A-bounds or A-values are ignored while searching the subtree rooted at u.

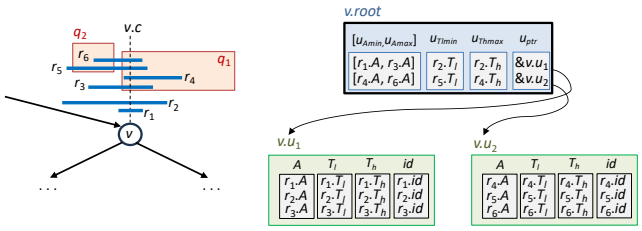

Figure 5: The RISH index (right) of an AITTI node v (left) and two time-travel queries

(2) If $v.c < q.T_l$, then we compute the 3-sided query $q.A_l \leq r.A \leq q.A_h \wedge r.T_h \geq q.T_l$ on v's RISH, pruning node u and its (and subtree) if $u_{Th_{max}} < q.T_l$; bounds $u_{Tl_{min}}$ are not used by this query. After solving the problem for the current AITTI node v, we recursively search $v.rightchild$. If, for a RISH node u, the range $[u_{A_{min}}, u_{A_{max}}]$ is covered by the query A-range $[q.A_l, q.A_h]$, the A-bounds or A-values are ignored while searching the subtree rooted at u.

(3) If $q.T_l \leq v.c \leq q.T_h$, then we traverse current AITTI node's v RISH to identify all records r with $q.A_l \leq r.A \leq q.A_h$; for this query, the bounds $u_{Tl_{min}}$ and $u_{Th_{max}}$ at each RISH node u are *ignored*. We recursively search *both* children of AITTI node v. If, for a RISH node u, the range $[u_{A_{min}}, u_{A_{max}}]$ is covered by the query A-range $[q.A_l, q.A_h]$, all records in the subtree rooted at u are reported without any comparisons.

Storage optimization. To reduce cache misses during search, the layout of RISH nodes is optimized as follows.

- Within each internal RISH node, the $[u_{A_{min}}, u_{A_{max}}]$, $u_{Tl_{min}}$, and $u_{Th_{max}}$ bounds of all entries are stored in three separate (dedicated) arrays and the corresponding pointers to lower-level nodes are also in a separate array. Hence, depending on the query and the overlap type between the query and the bounds, we access and use only the necessary information from each RISH node during search. Specifically, for queries of case (3), we do not access and use the $u_{Tl_{min}}, u_{Th_{max}}$ arrays from RISH nodes; whereas for queries of cases (1) and (2), we ignore the $u_{Tl_{max}}$ and $u_{Th_{min}}$ arrays respectively. In addition, when the query's A-range covers the $[u_{A_{min}}, u_{A_{max}}]$ range of a node u, we avoid accessing the $(u_{A_{min}}, u_{A_{max}})$ arrays when traversing the subtree rooted at u.
- Within each leaf node, the values $r.A$, $r.T_l$, and $r.T_h$ of every record are also stored in dedicated arrays, aligned with an array with the record identifiers. Queries only access the necessary arrays required for comparisons.

Example. Figure 5 illustrates the structure of RISH for a set of intervals that are stored in an AITTI node v; for simplicity, assume node capacity is $C = 3$ records. Hence, the RISH index has one root and two leaf nodes. The root node $v.root$ has two entries, which are vertically decomposed into four tables. The leaf nodes $v.u_1$ and $v.u_2$ are also decomposed into one array for each record attribute.

Consider query q_1 shown on the right of the figure. Since the temporal range $[q_1.T_l, q_1.T_h]$ of q_1 includes center $v.c$, all records in v are guaranteed to temporally overlap with q_1, so columns $u_{Tl_{min}}$ and $u_{Th_{max}}$ in non-leaf RISH nodes and columns T_l and T_h in leaf nodes are ignored. Hence, the query only accesses the $[u_{A_{min}}, u_{A_{max}}]$ table of the root to find that $[r_1.A, r_3.A]$ overlaps with the query range $[q_1.A_l, q_1.A_h]$. Then, node $v.u_1$ is accessed and from its column A, we report r_3 as q_1 result, since $q_1.A_l \leq r_3.A \leq q_1.A_h$. From the $[u_{A_{min}}, u_{A_{max}}]$ table of the root, we also find that attribute range $[r_4.A, r_6.A]$ is *covered by* $[q_1.A_l, q_1.A_h]$. Therefore, all records in the subtree rooted at $v.u_2$ are query results and no comparisons are necessary to retrieve and report them.

For query q_2 shown on the left of the figure, observe that $q_2.T_h < v.c$. This means that a record $r \in v$ temporally overlaps with the query if $r.T_l \leq q_2.T_h$. Hence, the query accesses $[u_{A_{min}}, u_{A_{max}}]$ and $u_{Tl_{min}}$ at the root of the tree (but $u_{Th_{max}}$ is ignored), to find that $[r_1.A, r_3.A]$ does not overlap with $[q_2.A_l, q_2.A_h]$; i.e., $v.u_1$ (and its subtree) is pruned. We also find that $[r_4.A, r_6.A]$ overlaps with $[q_2.A_l, q_2.A_h]$ and $r_5.T_l \leq q_2.T_h$; due to this, we access $v.u_2$ and columns A and T_l in it, to identify query results r_5 and r_6.

Complexity analysis. The space complexity of an RISH at an AITTI node v is $O(n_v)$, where n_v is the number of records assigned to v. This is due to the fact that the number of inner RISH nodes is smaller than the number of leaf nodes that occupy $O(n_v)$ space. Hence, the space complexity of AITTI when using RISH at its nodes is $O(n)$, as the total number of nodes in the backbone interval tree is $O(n)$ and the total number of records in these nodes is also $O(n)$.

The time complexity to search a single AITTI node v indexed by RISH when $q.T_l \leq v.c \leq q.T_h$ (case 3) is $O(\log n_v + K_v)$, where K_v is the number of results at node v, as RISH follows the total order of the records based on attribute A. For AITTI nodes v, where either $q.T_h < v.c$ or $q.T_l > v.c$ (cases 1 or 2, respectively), this search cost is $O(\log n_v + K_v^A)$, where K_v^A is the number of records that satisfy the query's A-range predicate; in the worst case, $K_v^A = O(n_v)$. Overall, the total search cost becomes $O(n)$ since $\sum O(\log n_v + K_v^A) = \sum O(n_v) = O(n)$. However, the A-range query predicate is more selective than the open-ended $r.T_l \leq q.T_h$ and $r.T_h \geq q.T_l$ predicates, resulting in a strong performance of RISH on average.

4.2 Updates

RISH's update process resembles the one in B$^+$-tree but adapted to maintain the bounds required for the 2- and 3-sided pruning described in Section 4.1.

Insertions. To insert a new record r, we traverse RISH from the root to a leaf. At each internal node u, we identify the appropriate child to descend with a binary search on the children's attribute values. As we descend, we update the bounds of every visited node u such that $u_{A_{min}} = \min(u_{A_{min}}, r.A)$, $u_{A_{max}} = \max(u_{A_{max}}, r.A)$, $u_{Tl_{min}} = \min(u_{Tl_{min}}, r.T_l)$, and $u_{Th_{max}} = \max(u_{Th_{max}}, r.T_h)$. This maintains the pruning capability of the index after insertion.

Upon reaching a leaf, we insert $r.id$, $r.A$, $r.T_l$, and $r.T_h$ into their respective arrays maintaining the sorted order primarily by $r.A$, and secondarily by $r.T_l$. If the insertion causes the leaf to exceed the capacity constraint C, we trigger a split. First, the leaf's content is divided into two halves, creating a new sibling node. Second, the bounds are recomputed for the original and the new sibling node. Finally, the new sibling is inserted into the parent internal node. If the parent also reaches capacity, the split propagates upwards, potentially creating a new root.

Table 2: Characteristics of tested datasets

		BIKES	BOOKS	FLIGHTS	TAXIS	WILDFIRES
	Cardinality	101472950	2050707	61328124	169290307	778410
Temporal	**Domain**	8 years	1 year	10 years	1 year	23 years
	Min duration	1 min	1 hour	5 min	1 min	1 min
	Max duration	7.5 months	1 year	12 hours	5 hours	4 months
	Avg. duration	16 mins	67 days	2.5 hours	12 mins	28 hours
		0.0004%	18.6%	0.0028%	0.0024%	0.0135%
Attribute A	**Description**	rider's birth year	number of books lent	departure delay (secs)	trip fare (USD)	fire extent (acres)
	Type	integer	integer	integer	real	real
	Value range	[1940, 2005]	[1, 38]	[0, 233400]	[2.5, 235.5]	[0.0001, 606945]
	Distribution	normal	zipfian	zipfian	normal	zipfian

Table 3: Impact of storage optimization on AITTI's indexing and maintenance costs; RISH node capacity set to $C = 128$

	Storage optimization	BIKES	BOOKS	FLIGHTS	TAXIS	WILDFIRES
Build time (secs)	no	2.81	0.31	9.89	29.61	0.14
	yes	2.90	0.31	10.07	30.28	0.15
Memory (MBs)	no	495.92	34.07	1355.61	4299.75	27.71
	yes	542.37	36.00	1597.51	5946.01	27.72
Insertions time (secs)	no	2.99	1.26	13.04	41.43	0.07
	yes	2.88	0.09	11.28	35.75	0.07
Deletions time (secs)	no	0.84	2.00	4.66	11.90	0.02
	yes	0.81	1.58	4.06	10.06	0.02

Deletions. To remove a record r from RISH, we first locate the leaf containing it. Unlike a standard B$^+$-tree deletion which would rely solely on the search key $r.A$, RISH exploits the stored temporal bounds to prune the search space. Starting from the root, we traverse the RISH recursively. At any node u, if the node's temporal bounds do not encompass the record's interval, we discard the subtree, as r cannot exist within it. Otherwise, we search the children (or the records within a leaf) using binary search on $r.A$. Once the record is located in a leaf, it is removed.

Complexity analysis. For both insertions and deletions, a single path is traversed from the root to a leaf. Since RISH is a balanced multi-way tree, the height is logarithmic to the number of records in the AITTI node. At each level, we perform binary search over maximum C entries. Thus, the worst time complexity for both operations is $O(\log_C n)$.

5 EXPERIMENTAL EVALUATION

Finally, we report our experimental analysis. We implemented all indices in C++, compiled with gcc using flags -O3, -mavx and -march=native.[6] 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, i.e., inputs and indices, reside in main memory.

5.1 Setup

We experimented with five real-world temporal datasets from [32], which additionally include a scalar attribute A. The datasets were provided by the authors. Table 2 summarizes their characteristics. BIKES contains bike rides in NYC from 2014 to 2021; pick-up and drop-off timepoints define the time range, while attribute A is the birth year of the rider. BOOKS is constructed from the lending records of the public libraries in Aarhus, Denmark; records are time periods when books were lent out, and A is the number of books during each period. FLIGHTS contains flight schedules from the US Transportation Department from 2013 to 2022; each record has the take-off and landing timepoints of the flight, while attribute A is the occurred departure delay. TAXIS contains taxis trips in NYC from 2009; pick-up and drop-off timepoints define the time range, while A is the trip fare. Last, WILDFIRES records fire events in the USA from 1992 to 2015, i.e., the time period since each event was discovered or declared and until it was contained or controlled and an estimate of the area burnt as the attribute A. The datasets cover diverse input types: (1) BOOKS and WILDFIRES represent inputs that include long time ranges, while TAXIS and BIKES have relatively short ranges, while ranges in FLIGHTS are in the middle; (2) for the scalar attribute A, we consider real or integer values drawn from a small domain (BIKES and BOOKS) or a large one (WILDFIRES). A's values follow either a normal or a Zipfian distribution.

Table 4: Impact of A-pruning on AITTI's Indexing and maintenance costs; RISH node capacity set to $C = 1024$

	A-pruning	BIKES	BOOKS	FLIGHTS	TAXIS	WILDFIRES
Build time (secs)	no	2.47	0.31	9.00	26.64	0.13
	yes	2.89	0.31	10.05	30.75	0.15
Memory (MBs)	no	500.59	32.53	1590.29	4873.70	26.53
	yes	502.58	32.71	1599.98	4879.20	27.72
Insertions time (secs)	no	3.02	0.08	12.74	42.87	0.06
	yes	3.05	0.08	12.70	45.71	0.05
Deletions time (secs)	no	0.86	0.37	3.86	11.85	0.02
	yes	0.84	0.36	3.78	12.49	0.01

To assess the query performance of the indices, we measure their total running time on 1000 time-travel queries, which includes the cost of retrieving the result record ids.[7] We directly control the selectivity of each query (i.e., the fraction of returned records) inside the $\{0.01\%, 0.1\%, 1\%, 10\%\}$ value range[8]; we achieve the desired number of results by appropriately setting the extent of the $[q.T_l, q.T_h]$ temporal query range and of the $[q.A_l, q.A_h]$ query range for attribute A. In addition, we study the individual effect of these query extents by varying one of them while fixing the other to a single default value. Such queries correspond to equality time-travel queries (when the A query value is fixed with $q.A_l = q.A_h$) and to stabbing time-travel queries (when query timepoint T is fixed with $q.T_l = q.T_h$), as defined in Section 2.1. Specifically, we vary the temporal query extent inside $\{1, 6, \mathbf{12}, 18, 24\}$ hours for BIKES and TAXIS, inside $\{1, 7, \mathbf{14}, 21, 30\}$ days for BOOKS and WILDFIRES, and inside $\{1, 2, \mathbf{3}, 4, 5\}$ days for FLIGHTS. We vary attribute A inside $\{10, 20, \mathbf{30}, 40, 50\}$ years for BIKES, $\{5, 10, \mathbf{15}, 20, 25\}$ lent books for BOOKS, $\{5, 10, \mathbf{30}, 60, 120\}$ minutes for FLIGHTS, $\{3, 5, \mathbf{10}, 30, 50\}$ USD for TAXIS, and $\{5, 10, \mathbf{15}, 20, 25\}$ for WILDFIRES. Default values are marked in bold, in all cases. Finally, for completeness purposes, we also tested queries by fixing both the temporal and the A query ranges to their default values, to test the performance in equality-stabbing time-travel queries.

[6] Source code available in https://github.com/psimatis/AITTI.

[7] Although search cost is low in absolute numbers, achieving high throughput is critical for modern applications where data systems receive thousands or million queries per second from real users or AI agents, https://aws.amazon.com/blogs/aws/amazon-s3-two-trillion-objects-11-million-requests-second/.

[8] We also experimented with higher values, e.g., 20% and 30% with similar findings.

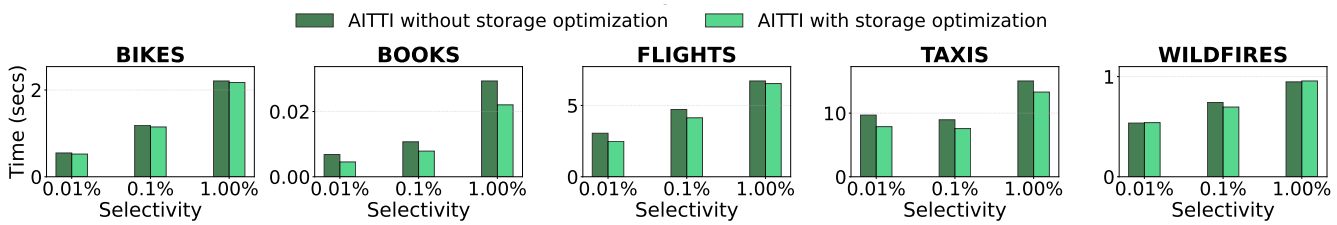

Figure 6: Impact of storage optimization on AITTI's total time while varying the query selectivity; RISH node capacity $C = 128$

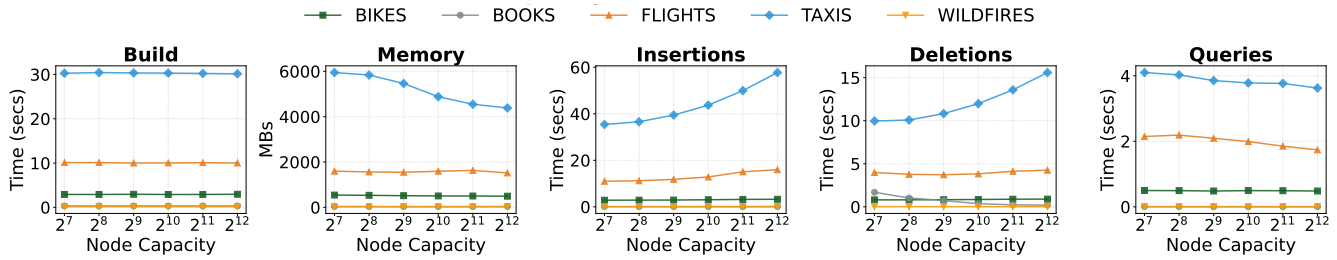

Figure 7: Impact of RISH's node capacity on AITTI; time-travel queries with a fixed query selectivity of 0.1%

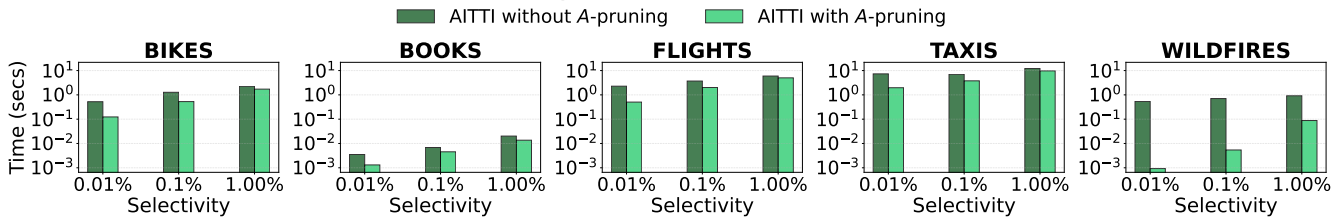

Figure 8: Impact of A-attribute pruning on AITTI's total time while varying the query selectivity

5.2 AITTI: Ablation Study

We first evaluate the impact of our design choices in Sections 3 and 4.1; (1) the cache-conscious, columnar layout of RISH nodes (storage optimization), (2) the impact of the node capacity C on RISH, and (3) the effectiveness of the A-pruning mechanism in AITTI.

Storage optimization. Recall from Section 4.1 that RISH uses separate (dedicated arrays) to store the contents of a node; for an internal node, arrays $[u_{Amin}, u_{Amax}]$, $u_{Tl_{min}}$, and $u_{Th_{max}}$ are defined, while for leaf nodes, we have the four arrays of $r.T_l$, $r.T_h$, $r.A$ and $r.d$ (refer to Figure 5 for an example). We compare this layout against a single-array alternative. Table 3 and Figure 6 report the results of this comparison; for these tests we set the capacity C for each RISH node to 128; we study the impact of this capacity in the next set of experiments. As Table 3 shows, the optimized layout incurs a slight increase on indexing costs, but reduces maintenance costs. This is due to improved cache locality, as each field is stored in a compact, contiguous array. Figure 6 shows that the storage optimization improves query performance. Hence,, we enable storage optimization for AITTI by default for the remainder of our analysis.

RISH node capacity. We now determine the optimal capacity C for RISH nodes. Recall that RISH is a multi-way tree, hence C dictates its fan-out. Figure 7 plots how the value of C affects the indexing costs, maintenance costs, and query performance of RISH, and ultimately of AITTI. As the key takeaway, we observe a clear trade-off between maintenance costs and query performance, making RISH and therefore AITTI, adaptable to different workloads. A low capacity value (e.g., $C = 128$) favors write-heavy scenarios resulting in lower update times as the number of memory operations required to shift elements during insertions and deletions is reduced. Conversely, a high capacity (e.g., $C = 2048$) favors read-heavy scenarios due to constructing a shallower tree which improves cache locality and query performance. In view of the above, we fix $C = 1024$ for the remainder of our experiments, which strikes a good balance between maintenance and query performance.

Attribute A-pruning. Last, we assess the attribute A-pruning discussed in Section 3. Table 4 reports on AITTI's indexing and maintenance costs of utilizing this optimization, while Figure 8 unveils the impact on time-travel query evaluation. As expected, maintaining the necessary A bounds for the pruning incurs slightly more memory and marginally increases the build time due to the recursive computation of these bounds per AITTI node. In contrast, the update times for insertions and deletions remain almost unchanged. On the other hand, we observe that A-pruning significantly lowers the query time, especially for highly selective queries. For instance, for queries on FLIGHTS of 0.01% selectivity, A-pruning improves performance by 4× (reducing time from 2.26 secs to 0.56 secs). The key reason is that AITTI avoids visiting entire subtrees even if they satisfy the temporal predicate. Accordingly, AITTI uses A-pruning for the rest of our experiments.

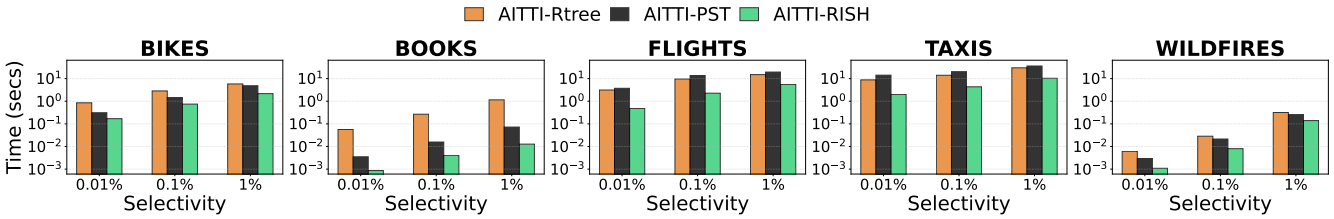

Figure 9: Total time of AITTI variants for time-travel queries while varying the query selectivity

Table 5: Indexing and maintenance costs of AITTI variants

	Variant	BIKES	BOOKS	FLIGHTS	TAXIS	WILDFIRES
Build time (secs)	AITTI-Rtree	3.19	0.32	10.4	35.1	0.33
	AITTI-PST	3.24	0.45	11.4	38.3	0.18
	AITTI-RISH	2.84	0.38	11.2	37.2	0.19
Memory (MBs)	AITTI-Rtree	441.70	31.87	1106.97	2926.80	30.43
	AITTI-PST	651.63	107.94	2775.60	8048.43	29.03
	AITTI-RISH	516.72	35.24	1695.15	5440.09	29.03
Insertions time (secs)	AITTI-Rtree	3.35	0.92	14.50	48.11	0.055
	AITTI-PST	4.08	0.10	21.03	61.37	0.067
	AITTI-RISH	4.03	0.08	17.40	58.60	0.065

5.3 AITTI: Component Selection

We next investigate the best data structure for the nodes of AITTI's backbone interval tree. We tested the following three alternatives leading to three AITTI variants. The first variant denoted by AITTI-PST, adopts the three-data structure design in Section 3 for which $B(A)$ is a red-black tree from the C++ std library and two priority search trees are defined according to [55, 69]. The second variant denoted by AITTI-RISH, implements the single-data structure design of RISH proposed in Section 5.3. Lastly, we also consider an alternative single-structure design denoted by AITTI-Rtree, where RISH is replaced by a 2-d R-tree with a 1024 node capacity similar to RISH in AITTI-RISH.[9] For this purpose, every record r of an interval tree node v is modeled as a horizontal line segment in the time dimension and indexed inside an MBR of the R-tree. These MBRs are defined by an A-range $[r.A, r.A]$ and a time-range $[r.T_l, r.T_h]$. Given a $q = \langle [q.T_l, q.T_h], [q.A_l, q.A_h] \rangle$ time-travel query, we consider the three cases for processing a node v, covered in Section 3:

(1) If $v.c > q.T_h$, then we evaluate the 3-sided range predicate $q.A_l \leq r.A \leq q.A_h \wedge\ r.T_l \leq q.T_h$.
(2) If $v.c < q.T_l$, then we evaluate the 3-sided range predicate $q.A_l \leq r.A \leq q.A_h \wedge r.T_h \geq q.T_l$.
(3) If $q.T_l \leq v.c \leq q.T_h$, then we evaluate the 1-d (stripe) range predicate $q.A_l \leq r.A \leq q.A_h$.

We compare the three variants in terms of their indexing and search performance, in Table 5 and Figure 9, respectively. Our tests clearly show the advantage of AITTI-RISH in querying over both alternatives, capitalizing on the single-data structure design and the efficiency of our RISH index. At the same time, the AITTI-RISH always exhibits competitive building times and lower memory requirements to AITTI-PST. As expected AITTI-Rtree has the lower maintenance cost but at the expense of typically the higher query processing time. In view of the above, we will consider only the AITTI-RISH variant for the rest of our experimental analysis.

5.4 AITTI-RISH against the Competition

We compare AITTI-RISH against the following solutions:

[9] In our code, we used the R-tree implementation found in the Boost.Geometry library.

Table 6: AITTI-RISH against competition, indexing costs

	Index	BIKES	BOOKS	FLIGHTS	TAXIS	WILDFIRES
Build time (secs)	HINT+PF	2.21	0.33	6.99	20.28	0.11
	3-d R-tree	0.99	0.17	3.99	17.38	0.04
	aHINT	2.21	0.36	7.65	20.88	0.13
	RI-trees	9.96	0.89	27.04	128.13	0.32
	AITTI-RISH	2.58	0.39	10.49	35.66	0.17
Memory (MBs)	HINT+PF	224.03	91.92	724.59	1942.71	10.67
	3-d R-tree	312.20	32.88	983.17	2713.94	12.48
	aHINT	406.03	179.24	1288.77	3580.66	55.23
	RI-trees	2239.81	236.00	7053.18	19469.27	89.63
	AITTI-RISH	319.73	31.90	966.17	2664.04	23.79

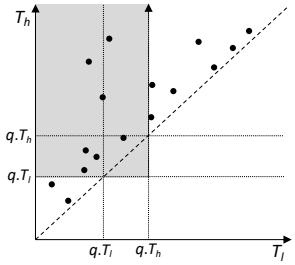

Figure 10: Space transformation for the 3-d R-tree baseline

Table 7: AITTI-RISH against competition, total query time (secs) for equality-stabbing time-travel queries

Index	BIKES	BOOKS	FLIGHTS	TAXIS	WILDFIRES
HINT+PF	0.034	0.80	0.02	0.99	0.0092
3-d R-tree	0.02	0.30	0.008	0.02	0.0028
aHINT	0.03	0.15	0.02	1.03	0.0007
RI-trees	0.003	0.002	0.005	0.004	0.002
AITTI-RISH	0.001	0.001	0.003	0.001	0.0004

- **HINT+PF**: A baseline that probes the state-of-the-art for interval indexing HINT [30, 31] to find all r records whose $[r.T_l, r.T_h]$ overlaps $[q.T_l, q.T_h]$, and then checks $q.A_l \leq r.A \leq q.A_h$ as a post-filter.
- **3-d R-tree**: We consider a data transformation in the 3-d (T_l, T_h, A) space as a geometric solution. The records are modeled as 3-d points using $r.T_l, r.T_h, r.A$ as coordinates. Figure 10 explains the transformation of the record intervals only (for simplicity) where a $[q.T_l, q.T_h]$ range query becomes a 2-sided range query $(-\infty, q.T_h, q.T_l, \infty)$, as shown in the gray area. This query is then extended to a 3-sided after including $q.A_l \leq r.A \leq q.A_h$. To index the record points, we use an 3-d R-tree.
- **aHINT**: We adapt HINT to support attributes by leveraging the partitioning design of a-LIT [32]. This approach divides the A-domain into bands. Each band has a dedicated HINT built over the $[r.T_l, r.T_h]$ time interval of the records r whose $r.A$ falls within the band.

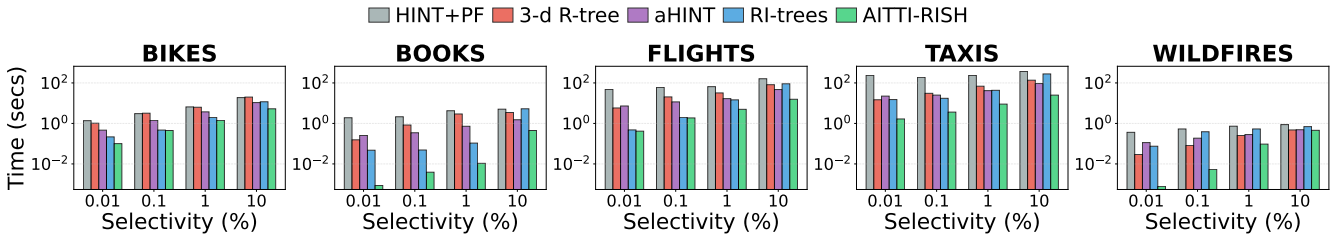

Figure 11: AITTI-RISH against competition, total time for time-travel queries while varying query selectivity

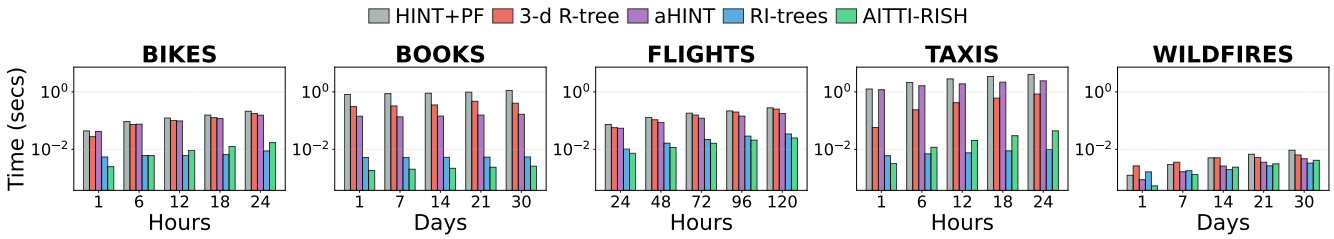

Figure 12: AITTI-RISH against competition, total time for equality time-travel queries while varying time range query extent

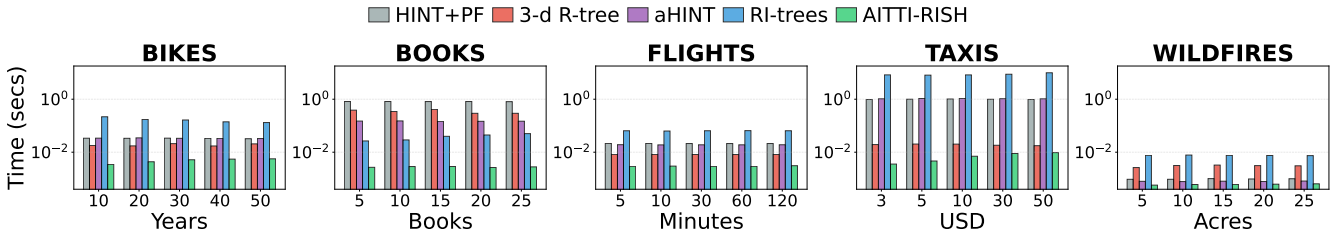

Figure 13: AITTI-RISH against competition, total time for stabbing time-travel queries while varying A-range query extent

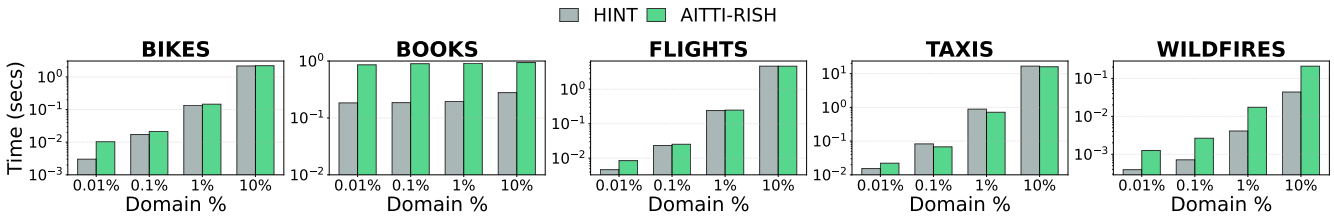

Figure 14: AITTI-RISH against competition, total time for time-only queries while varying query extent

- **RI-trees**: We implemented the index setting from [37] (Figure 24) which achieves the best performance across all time-travel query variants. This setting constructs four RI-trees [50], each defining a virtual (not materialized) interval tree. The solution maintains six relational indices as B$^+$-trees[10] for the following record orderings: $(h\{v.n, r.A\})$ exclusively for *general* time-travel queries, $(v.n, r.A, r.T_l)$ exclusively for *stabbing* time-travel queries, $(v.n, h\{r.A, r.T_l\})$, $(v.n, h\{r.A, r.T_h\})$ for *general* time-travel and *stabbing* queries, and $(r.A, v.n, r.T_l)$, $(r.A, v.n, r.T_h)$ for *equality* and *equality-stabbing* time-travel queries. Entry $v.n$ denotes a node from one of the virtual interval trees and h denotes a mapping to the 1-d space using the Hilbert curve.

Indexing costs. We first compare the three solutions with respect to their indexing costs. Table 6 reports the build times and the space requirements of each index for the tested datasets. Thanks to its single-structure and straightforward design, the 3-d R-tree solution always exhibits the lowest construction time, followed by HINT+PF and aHINT, which are comparable as both are built around HINT. Nevertheless, AITTI-RISH times are competitive. RI-trees incurs the highest build cost, since it builds six B$^+$-trees, including two with Hilbert-encoded keys, which on TAXIS amounts to 128s, roughly 7× the next slowest method. In terms of space requirements, HINT+PF is the most frugal across most datasets, as it indexes only T_l, T_h. The only exception is BOOKS, where the long intervals create multiple replicas. Our tests also show the advantage of AITTI-RISH, confirming in practice its $O(n)$ space complexity discussed in Sections 3 and 4.1. The size of the 3-d R-tree is always comparable to AITTI-RISH, while aHINT consistently requires more memory due to its partitioning strategy that maintains multiple HINT indices. RI-trees, however, carries the highest memory overhead, as every record is replicated across six B$^+$-trees.

Query performance. We consider all four types of time-travel queries discussed in Section 2.1. Figures 11, 12, and 13 report on

[10] We used the implementation from the TLX library [17].

time-travel, equality time-travel and stabbing travel queries, respectively, while Table 7 reports on equality-stabbing time-travel queries. The key observation in all experiments is that AITTI-RISH outperforms the HINT+PF, 3-d R-tree, aHINT and RI-trees. In fact, the performance gap can be up to one order of magnitude. An exception occurs for equality time-travel queries, where RI-trees occasionally surpasses AITTI-RISH. In general, the second fastest solution varies based on the characteristics of the dataset and the type of queries, which further proves the robustness of AITTI-RISH in outperforming the competition across the spectrum.

For general $q = \langle [q.T_l, q.T_h], [q.A_l, q.A_h] \rangle$, i.e., time-travel queries (see Figure 11), the performance of all indices increases with the query selectivity as the number of query results goes up. HINT+PF consistently exhibits the worst performance as post-filtering temporal results is costly. The 3-d R-tree also suffers due to the open-ended nature of the queries in the transformed space. Often the large defined 3-d query areas access a large fraction of the 3-d R-tree. aHINT and the RI-trees compete for the second place, with the RI-trees winning at low selectivities. Overall, AITTI-RISH outperforms its competitors across all settings, by effectively pruning the search space both temporally and on the non-temporal attribute.

For $q = \langle [q.T_l, q.T_h], q.A \rangle$, i.e., equality time-travel queries (see Figure 12), we observe again that the indices slow down as we increase the temporal query extent $[q.T_l, q.T_h]$. HINT+PF is the slowest often by orders of magnitude comparing to the best indices. The 3-d R-tree and aHINT follow closely. Note that while aHINT needs to search a single HINT index corresponding to the band containing $q.A$, the number of scanned partitions within HINT increases with the temporal extent. While the RI-trees occasionally gains an edge at larger temporal extents by leveraging its specialized multi-tree structure, AITTI-RISH maintains an overall superiority. This consistent performance stems from its ability to efficiently couple the temporal filtering of its backbone interval tree with the precise attribute search of RISH.

For $q = \langle q.T, [q.A_l, q.A_h] \rangle$, i.e., stabbing time-travel queries (see Figure 13), RI-trees underperforms, with BOOKS being the sole exception where it ranks second. Similarly, HINT+PF and aHINT struggle, with the latter hampered by the need to examine multiple bands to answer a query. In contrast, the 3-d R-tree is competitive in most datasets, since it benefits from the fact that wider attribute queries resemble traditional spatial range searches. Finally, AITTI-RISH handles this gracefully as the temporal stabbing restricts the search to a logarithmic number of nodes in the backbone interval tree regardless of the attribute range.

Last, for $q = \langle q.T, q.A \rangle$, i.e., equality-stabbing time travel queries (Table 7), HINT+PF is the slowest, degrading severely as excessive post-filtering degrades performance. aHINT shows inconsistent performance, but it generally sits between HINT+PF and the 3-d R-tree. The RI-trees ranks second overall surpassing the 3-d R-tree. AITTI-RISH outperforms all competitors due to its ability to effectively prune the search both temporally and on the A-attribute.

Figure 14 compares AITTI-RISH against the state-of-the-art HINT for time-only queries $q = \langle [q.T_l, q.T_h] \rangle$, which retrieve all records whose only time range $[r.T_l, r.T_h]$ overlaps $[q.T_l, q.T_h]$ while no condition exists on the non-temporal attribute A. Although time-only search falls outside AITTI's focus of composite time-travel queries, AITTI-RISH has similar performance as HINT in BIKES, FLIGHTS, and TAXIS. It trails HINT on BOOKS and WILDFIRES which include long intervals and under extreme selectivities (i.e., 0.01%). For datasets with long durations, AITTI's interval tree backbone becomes top-heavy, as many long intervals overlap the center points of the upper nodes. The time-only queries that access these nodes cannot exploit the A-based RISH in them. Furthermore, under highly selective queries, HINT's hierarchical partitioning excels by mapping narrow time windows directly to localized, lower-level partitions, avoiding unnecessary comparisons entirely, whereas AITTI must always traverse its tree from the root.

Table 8: AITTI-RISH against competition, processing mixed workload of 15% insertions, 5% deletions, and 1000 time-travel queries of selectivity 0.1%

	Index	BIKES	BOOKS	FLIGHTS	TAXIS	WILDFIRES
Insertions time (secs)	HINT+PF	0.49	0.08	2.29	7.53	0.02
	3-d R-tree	57.83	1.74	95.20	364.62	1.58
	aHINT	0.15	0.03	1.39	3.15	0.01
	RI-trees	17.55	1.47	65.71	168.97	0.51
	AITTI-RISH	3.10	0.08	11.87	43.98	0.05
Deletions time (secs)	HINT+PF	6.74	1.12	6.94	449.16	0.01
	3-d R-tree	2.12	1.50	787.81	20.27	1.28
	aHINT	6.98	0.96	7.02	501.13	0.01
	RI-trees	10.99	8.64	1007.27	79.58	0.83
	AITTI-RISH	0.87	0.36	3.95	12.10	0.01
Query time (secs)	HINT+PF	4.62	1.84	51.68	207.10	0.32
	3-d R-tree	2.15	0.46	10.33	13.49	0.05
	aHINT	0.99	0.22	9.69	17.41	0.13
	RI-trees	0.67	0.06	2.24	17.39	0.88
	AITTI-RISH	0.51	0.01	2.07	3.82	0.01
Total time (secs)	HINT+PF	11.86	3.03	60.92	663.79	0.34
	3-d R-tree	62.10	3.70	893.34	398.37	2.90
	aHINT	8.13	1.22	18.10	521.69	0.14
	RI-trees	29.21	10.18	1075.22	265.94	2.22
	AITTI-RISH	4.48	0.45	17.89	59.90	0.06

Maintenance costs. Finally, we study how effectively the three indices handle updates, i.e., insertions and deletions. For this purpose, we defined a mixed workflow for each dataset, where we use 85% of the dataset to construct an initial index, and then add the remaining 15% of records as insertion updates. We also use 5% of the indexed records to define deletion updates and executed 1000 time-travel queries with 0.1% selectivity. The three types of operations are interleaved to showcase not only how fast each index can handle updates but also whether the relative query performance we observed so far is affected by the updates.

Table 8 reports the results of our tests. RI-trees updates are costly since records are indexed across multiple B$^+$-trees. The 3-d R-tree also struggles with updates due to the high cost of node splitting and re-balancing of the R*-tree algorithm [14]. aHINT offers fast insertions since it simply appends records at the end of the relevant partitions; recall aHINT does not employ any sorting. However, this design fails during deletions, where deleting a record requires linear time to locate it. This results in a deletion time multiple times greater than AITTI-RISH. Furthermore, aHINT outperforms HINT+PF in insertions due to the smaller HINT structure deployed per band. Ultimately, AITTI-RISH prevails offering the best balance between maintenance and querying; this is reflected by the total cost of processing the workflow.

6 RELATED WORK

We last discuss related work in indexing interval and temporal data, beyond Section 2.2 and the competitor solutions in Section 5.4.

6.1 Indexing Interval Data

Indexing intervals has received significant attention in the literature. Intervals can be considered as 1-d boxes indexed by multi-dimensional data structures, e.g., the R-tree [42], or natively by a 1-d grid. Duplicated results due to data replication for the latter, are eliminated via hashing, sorting or the reference value technique [34]. Following up on [35], a number of works adapted the interval tree (see Section 2.2) for external memory [11, 12], mainly building upon the B$^+$-tree. The *interval B-tree* [9] adopts a multi-structure design; a B$^+$-tree is used as the primary structure to organize the $s.T_h$ high endpoints of the intervals, accompanied by 1-d range trees and an on-top binary tree. The *interval B$^+$-tree* [24] utilizes a B$^+$-tree on the $s.T_l$ low endpoints of the intervals, augmented with maximum $s.T_h$ high points of the intervals in the internal tree nodes. The *relation interval tree* (RI-tree) [50, 51] defines a backbone interval tree which is is never materialized. Instead, its nodes are organized in two composite $(node, T_l)$ and $(node, T_h)$ structures, indexed by a dedicated B$^+$-tree each. In Section 5, the RI-trees competitor builds upon the RI-tree to additionally index the non-temporal attribute A, as proposed in [37]. Interval indexing have also been extended to support probabilistic queries over uncertain data [29].

Another classic data structure for intervals is the *segment tree* [33], later extended for the external memory in [18]. The segment tree is constructed in $O(n \log n)$ time and can answer stabbing queries in $O(\log n + K)$, which is employed for weighted intervals [7, 8]. However, it occupies $O(n \log n)$ space and it does not support range search, so it is considered inferior to the interval tree. AIT [3, 4] and FIRAS [6] which were proposed for independent range sampling on interval data, can process range queries in $O(\log n + K)$ time, consuming $O(n \log n)$ and $O(n)$ space, respectively. The state-of-the-art in-memory index for intervals called HINT was proposed in [30, 31]. HINT hierarchically divides the time domain, such that each level has double the number of partitions compared to the previous one. Each interval is assigned to 0-2 partitions in each level, whose timespan collectively covers the interval. A query is processed by accessing the partitions which overlap with the query. By ensuring that comparisons are conducted at a limited number of partitions, HINT computes the query result very fast. HINT was adopted in [32] to manage dynamic intervals. In our experiments, we compared our AITTI against the HINT+PF solution which first determines overlapping time intervals and then checks the condition on the non-temporal attribute A as a post-filter.

Last, geometric solutions which build upon a data *transformation* (similar to our 3-d R-tree competitor in Section 5.4) have been extensively studied, especially for indexing intervals in external memory. Among the first works applying the typical mapping to a (T_l, T_h) 2-d space (see Figure 10) was [46], which proposed the *diagonal corner structure*. The semantics of corner structures were theoretically studied also in [12, 59, 70]. Although the 2-d space could be directly indexed by an off-the-shelf multi-dimensional index such as the R-tree [42] and the kd-tree [16] (or the kdB-tree [58]), the *Start/End timestamp B-tree* (SEB-tree) [64] first slices the space in columns which are indexed by a top-level B$^+$-tree; each column is then indexed by a dedicated B$^+$-tree. Under the same premise, the *triangular decomposition tree* (TD-tree) [65, 66] is a binary tree built upon a triangle-based partitioning of the (T_l, T_h) space. As alternative mappings, the authors in [41, 53, 61] consider a (T_l, d) 2-d space while TIDE [71] uses a (d, T_h) one, where in both cases $d = s.T_h - s.T_l$ is the duration for each interval s.

6.2 Indexing Temporal Data

Temporal databases need to support a search predicate on an attribute A, in addition to time search [59]. The MVB-tree [13] supports such composite queries, by optimizing partially persistent B-trees, whose versions are being recorded as the B-tree is updated. Modeling geometrically temporal records as horizontal line segments in 2-d (see Figure 1), enables the usage of geometric point location queries on the orthogonal subdivision induced by the input horizontal segments. Although, existing offline planar orthogonal point location algorithms for internal [60] and external memory [44] (which are also based on partially persistent search trees) use only logarithmic query time and linear space, they cannot be updated and do not support attribute range filters. To support stabbing time-travel queries with attribute range filters in logarithmic time, dynamic data structures for orthogonal line segment intersection queries [28] have to be employed that support updates of horizontal segments in very fast sub-logarithmic $O\left(\log^{1/2+\varepsilon} n\right)$ time, but at the expense of superlinear $O\left(n \log^{1/2+\varepsilon} n\right)$ space, for some $\varepsilon > 0$.

The *Timeline index* [48] is a more recent and practical solution was proposed for SAP-HANA. It builds upon the Time index [36, 49] to support very fast updates. The index keeps in a time-ordered table the $r.T_l$ and $r.T_h$, time-endpoints of all record versions. In addition, at certain timestamps called checkpoints, the entire set of valid records are materialized, i.e., those whose $[r.T_l, r.T_h]$ range contains the checkpoint. To evaluate a time-travel query $q = [q.T_l, q.T_h]$, the latest checkpoint before $q.T_l$ is first determined to report the contained records as results and then the table is scanned from thereon until $q.T_h$ to identify the rest of the records active during the query. While being extremely fast in updates, Timeline is shown to be slow in queries [32] and is tailored for time-only queries, simply using predicates on other attributes as post-filters.

More recently, the LIT framework [32] decouples the indexing of current (live) and past (dead) versions of records. It adopts a two-structure design comprising a highly dynamic LiveIndex to ingest new live records and a DeadIndex on dead records for which both the start and end-time are known. For the latter, LIT uses a time-evolving variant of HINT [30, 31] as the in-memory interval index. In practice, DeadIndex is the most demanding component, as the great majority of record versions in a temporal database are expected to be dead. The authors in [32] also devised a-LIT, an extension to LIT that evaluates time-travel queries as defined in Section 2.1. The key idea is to split the domain of attribute A into a number of bands and separately index each of them using a dedicated LIT. To answer query $q = \langle [q.T_l, q.T_h], [q.A_l, q.A_h] \rangle$, LIT first identifies the bands whose A-range overlaps with the query A-range and then, probes each of LiveIndex and DeadIndex against $[q.T_l, q.T_h]$. In our experiments, we compare our AITTI against the DeadIndex of a-LIT, which we call aHINT.

Finally, the recent *period index* [15] and *RD-Index* [25, 26] were designed to answer *time-duration* queries, where the objective is

to retrieve intervals that overlap a query period and satisfy a predicate on their duration at the same time. These two indices cannot effectively support our time-travel queries. The period index defines a 2-d structure which hierarchically assigns objects to a level according their interval length. RD-Index builds upon a (T_l, d) transformation, where d is the interval duration. Replacing d with a non-temporal A would result in losing the T_h information, necessary for the time-travel part of our query.

7 CONCLUSIONS

We proposed a new index for temporal relations that fit in memory. The goal is to efficiently retrieve records which were valid sometime within a query time interval and their attribute A satisfies a range selection predicate. Our AITTI index organizes the records primarily by the time dimension (which is typically more selective in queries) using a variant of the interval tree [35]. Each node of the backbone structure is indexed by a novel data structure, RISH, which supports range queries on A and 3-sided queries in the time-A space. We theoretically and experimentally analyze the performance of AITTI against previous work on a number of real datasets with different characteristics and show that AITTI answers queries several times faster, while being competitive in updates.

There are several directions for future work. First, we plan to generalize AITTI for handling equality queries on low-cardinality attributes. For this, we need to replace RISH by appropriate data structures that pair low-cardinality attribute access methods (e.g., bitmaps) with open-ended search in the time dimension. In the same direction, AITTI could be extended to support time-travel keyword search [57], or time-travel vector similarity search [72] (e.g., for temporal Retrieval-Augmented Generation). Another interesting direction would be to integrate AITTI in the LIT framework [32] as the index for the dead records.

ACKNOWLEDGMENTS

Daichi Amagata is supported by JSPS KAKENHI Grant Number 24K14961 and Broadening Opportunities for Outstanding young researchers and doctoral students in STrategic areas JST (JPMJBY24A3).

REFERENCES

[1] W. Ahmed, E. Zimányi, A. A. Vaisman, and R. Wrembel. 2020. A Temporal Multidimensional Model and OLAP Operators. *Int. J. Data Warehous. Min.* 16, 4 (2020), 112–143. https://doi.org/10.4018/IJDWM.2020100107

[2] 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. https://doi.org/10.1145/2452376.2452443

[3] 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. https://doi.org/10.1109/ICDE60146.2024.00041

[4] Daichi Amagata. 2024. Independent Range Sampling on Interval Data (Longer Version). *arXiv preprint arXiv:2405.08315* (2024).

[5] Daichi Amagata and Jimin Lee. 2025. Top-k Range Search on Weighted Interval Data. In *International Symposium on Spatial and Temporal Data, SSTD*. ACM, 218–228. https://doi.org/10.1145/3748777.3748778

[6] Daichi Amagata, Panagiotis Simatis, Panagiotis Bouros, and Nikos Mamoulis. 2026. FIRAS: A Framework for Interval Range Search and Sampling. *Proceedings of the ACM on Management of Data* 4, 3 (SIGMOD (2026), 1–26. https://doi.org/10.1145/3802062

[7] Daichi Amagata, Junya Yamada, Yuchen Ji, and Takahiro Hara. 2024. Efficient Algorithms for Top-k Stabbing Queries on Weighted Interval Data. In *DEXA (Lecture Notes in Computer Science, Vol. 14910)*. Springer, 146–152. https://doi.org/10.1007/978-3-031-68309-1_12

[8] Daichi Amagata, Junya Yamada, Yuchen Ji, and Takahiro Hara. 2024. Efficient Algorithms for Top-k Stabbing Queries on Weighted Interval Data (Full Version). *CoRR* abs/2405.05601 (2024). https://doi.org/10.48550/ARXIV.2405.05601 arXiv:2405.05601

[9] Chuan-Heng Ang and Kok-Phuang Tan. 1995. The Interval B-Tree. *Inf. Process. Lett.* 53, 2 (1995), 85–89. https://doi.org/10.1016/0020-0190(94)00176-Y

[10] Lars Arge, Vasilis Samoladas, and Jeffrey Scott Vitter. 1999. On Two-Dimensional Indexability and Optimal Range Search Indexing. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania, USA*. ACM Press, 346–357. https://doi.org/10.1145/303976.304010

[11] Lars Arge and Jeffrey Scott Vitter. 1996. Optimal Dynamic Interval Management in External Memory (extended abstract). In *37th Annual Symposium on Foundations of Computer Science, FOCS 1996, Burlington, Vermont, USA, 14-16 October, 1996*. IEEE Computer Society, 560–569. https://doi.org/10.1109/SFCS.1996.548515

[12] Lars Arge and Jeffrey Scott Vitter. 2003. Optimal External Memory Interval Management. *SIAM J. Comput.* 32, 6 (2003), 1488–1508. https://doi.org/10.1137/S009753970240481X

[13] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. 1996. An Asymptotically Optimal Multiversion B-Tree. *VLDB J.* 5, 4 (1996), 264–275. https://doi.org/10.1007/S007780050028

[14] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*. ACM Press, 322–331. https://doi.org/10.1145/93597.98741

[15] 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. https://doi.org/10.1145/3340964.3340965

[16] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517. https://doi.org/10.1145/361002.361007

[17] Timo Bingmann. 2018. TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers. https://github.com/tlx/tlx.

[18] Gabriele Blankenagel and Ralf Hartmut Güting. 1994. External Segment Trees. *Algorithmica* 12, 6 (1994), 498–532. https://doi.org/10.1007/BF01188717

[19] G. Blankenagel and R. H. Güting. 1990. *XP-trees — External Priority Search Trees*. Technical Report Informatik-Bericht Nr. 92. FernUniversität Hagen, Fachbereich Informatik, Hagen, Germany.

[20] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. 1973. Time Bounds for Selection. *J. Comput. Syst. Sci.* 7, 4 (1973), 448–461. https://doi.org/10.1016/S0022-0000(73)80033-9

[21] 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. https://doi.org/10.1007/978-3-319-96655-7_3

[22] 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

[23] Panagiotis Bouros and Nikos Mamoulis. 2025. Relevance Queries for Interval Data. *Proc. ACM Manag. Data* 3, 3 (2025), 206:1–206:26. https://doi.org/10.1145/3725343

[24] Tolga Bozkaya and Z. Meral Özsoyoglu. 1998. Indexing Valid Time Intervals. In *Database and Expert Systems Applications, 9th International Conference, DEXA '98, Vienna, Austria, August 24-28, 1998, Proceedings (Lecture Notes in Computer Science)*. Springer, 541–550. https://doi.org/10.1007/BFB0054512

[25] 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. https://doi.org/10.1145/3603719.3603732

[26] Matteo Ceccarello, Anton Dignös, Johann Gamper, and Christina Khnaisser. 2025. Indexing temporal relations for range-duration queries. *Distributed Parallel Databases* 43, 1 (2025), 7. https://doi.org/10.1007/S10619-024-07452-6

[27] Timothy M. Chan and Konstantinos Tsakalidis. 2018. Dynamic Orthogonal Range Searching on the RAM, Revisited. *J. Comput. Geom.* 9, 2 (2018), 45–66. https://doi.org/10.20382/JOCG.V9I2A5

[28] Timothy M. Chan and Konstantinos Tsakalidis. 2018. Dynamic Planar Orthogonal Point Location in Sublogarithmic Time. In *34th International Symposium on Computational Geometry, SoCG 2018, Budapest, Hungary, June 11-14, 2018 (LIPIcs, Vol. 99)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 25:1–25:15. https://doi.org/10.4230/LIPICS.SOCG.2018.25

[29] Reynold Cheng, Yuni Xia, Sunil Prabhakar, Rahul Shah, and Jeffrey Scott Vitter. 2004. Efficient Indexing Methods for Probabilistic Threshold Queries over Uncertain Data. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3*

2004, Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer (Eds.). Morgan Kaufmann, 876–887. https://doi.org/10.1016/B978-012088469-8.50077-2

[30] 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. https://doi.org/10.1145/3514221.3517873

[31] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2024. HINT: a hierarchical interval index for Allen relationships. *VLDB J.* 33, 1 (2024), 73–100. https://doi.org/10.1007/S00778-023-00798-W

[32] 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. https://doi.org/10.1145/3639275

[33] Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. 2008. *Computational geometry: algorithms and applications, 3rd Edition*. Springer. https://doi.org/10.1007/978-3-540-77974-2

[34] Jens-Peter Dittrich and Bernhard Seeger. 2000. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*. IEEE Computer Society, 535–546. https://doi.org/10.1109/ICDE.2000.839452

[35] Herbert Edelsbrunner. 1983. A New Approach to Rectangle Intersections, Part I. *Intern. J. Computer Math.* 13 (1983), 209–219.

[36] Ramez Elmasri, Gene T. J. Wuu, and Yeong-Joon Kim. 1990. The Time Index: An Access Structure for Temporal Data. In *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*. Morgan Kaufmann, 1–12. http://www.vldb.org/conf/1990/P001.PDF

[37] Jost Enderle, Nicole Schneider, and Thomas Seidl. 2005. Efficiently Processing Queries on Interval-and-Value Tuples in Relational Databases. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. ACM, 385–396. http://www.vldb.org/archives/website/2005/program/paper/wed/p385-enderle.pdf

[38] Michael L. Fredman and Dan E. Willard. 1990. BLASTING through the Information Theoretic Barrier with FUSION TREES. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*. ACM, 1–7. https://doi.org/10.1145/100216.100217

[39] 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. https://doi.org/10.1007/978-3-031-15740-0_5

[40] 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. https://doi.org/10.1007/S00778-003-0111-3

[41] Cheng Hian Goh, Hongjun Lu, Beng Chin Ooi, and Kian-Lee Tan. 1996. Indexing Temporal Data Using Existing B+-Trees. *Data Knowl. Eng.* 18, 2 (1996), 147–165. https://doi.org/10.1016/0169-023X(95)00034-P

[42] 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*. ACM Press, 47–57. https://doi.org/10.1145/602259.602266

[43] 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.

[44] Xiaocheng Hu, Cheng Sheng, and Yufei Tao. 2019. Building an Optimal Point-Location Structure in O(sort (n)) I/Os. *Algorithmica* 81, 5 (2019), 1921–1937. https://doi.org/10.1007/S00453-018-0518-2

[45] 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. https://doi.org/10.1145/3514221.3517893

[46] Paris C. Kanellakis, Sridhar Ramaswamy, Darren Erik Vengroff, and Jeffrey Scott Vitter. 1996. Indexing for Data Models with Constraints and Classes. *J. Comput. Syst. Sci.* 52, 3 (1996), 589–612. https://doi.org/10.1006/JCSS.1996.0043

[47] Alexis C. Kaporis, Apostolos N. Papadopoulos, Spyros Sioutas, Konstantinos Tsakalidis, and Kostas Tsichlas. 2010. Efficient processing of 3-sided range queries with probabilistic guarantees. In *Database Theory - ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, Proceedings (ACM International Conference Proceeding Series)*. ACM, 34–43. https://doi.org/10.1145/1804669.1804676

[48] 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. https://doi.org/10.1145/2463676.2465293

[49] Vram Kouramajian, Ibrahim Kamel, Ramez Elmasri, and Syed Waheed. 1994. The Time Index+: An Incremental Access Structure for Temporal Databases. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94), Gaithersburg, Maryland, USA, November 29 - December 2, 1994*. ACM, 296–303. https://doi.org/10.1145/191246.191298

[50] 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

[51] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. 2001. Interval Sequences: An Object-Relational Approach to Manage Spatial Data. In *Advances in Spatial and Temporal Databases, 7th International Symposium, SSTD 2001, Redondo Beach, CA, USA, July 12-15, 2001, Proceedings (Lecture Notes in Computer Science)*. Springer, 481–501. https://doi.org/10.1007/3-540-47724-1_25

[52] Krishna G. Kulkarni and Jan-Eike Michels. 2012. Temporal features in SQL: 2011. *SIGMOD Rec.* 41, 3 (2012), 34–43. https://doi.org/10.1145/2380776.2380786

[53] Chiang Lee and Te-Ming Tseng. 1998. Temporal Grid File: A File Structure for Interval Data. *Data Knowl. Eng.* 26, 1 (1998), 71–97. https://doi.org/10.1016/S0169-023X(97)00027-X

[54] Jimin Lee and Daichi Amagata. 2026. Efficient algorithms for top-k range search on weighted interval data. *GeoInformatica* 30, 1 (2026), 18. https://doi.org/10.1007/S10707-026-00576-0

[55] Edward M. McCreight. 1985. Priority Search Trees. *SIAM J. Comput.* 14, 2 (1985), 257–276. https://doi.org/10.1137/0214021

[56] Sridhar Ramaswamy and Sairam Subramanian. 1994. Path Caching: A Technique for Optimal External Searching. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 24-26, 1994, Minneapolis, Minnesota, USA*. ACM Press, 25–35. https://doi.org/10.1145/182591.182595

[57] Christian Rauch and Panagiotis Bouros. 2025. Fast Indexing for Temporal Information Retrieval. *Proc. ACM Manag. Data* 3, 4 (2025), 246:1–246:28. https://doi.org/10.1145/3749164

[58] John T. Robinson. 1981. The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indexes. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, USA, April 29 - May 1, 1981*. ACM Press, 10–18. https://doi.org/10.1145/582318.582321

[59] Betty Salzberg and Vassilis J. Tsotras. 1999. Comparison of Access Methods for Time-Evolving Data. *ACM Comput. Surv.* 31, 2 (1999), 158–221. https://doi.org/10.1145/319806.319816

[60] Neil Sarnak and Robert Endre Tarjan. 1986. Planar Point Location Using Persistent Search Trees. *Commun. ACM* 29, 7 (1986), 669–679. https://doi.org/10.1145/6138.6151

[61] Han Shen, Beng Chin Ooi, and Hongjun Lu. 1994. The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases. In *Proceedings of the Tenth International Conference on Data Engineering, February 14-18, 1994, Houston, Texas, USA*. IEEE Computer Society, 274–281. https://doi.org/10.1109/ICDE.1994.283041

[62] Spyros Sioutas, Christos Makris, Nectarios Kitsios, George Lagogiannis, John Tsaknakis, Kostas Tsichlas, and Bill Vassiliadis. 2004. Geometric Retrieval for Grid Points in the RAM Model. *J. Univers. Comput. Sci.* 10, 9 (2004), 1325–1353. https://doi.org/10.3217/JUCS-010-09-1325

[63] Richard T. Snodgrass and Ilsoo Ahn. 1986. Temporal Databases. *Computer* 19, 9 (1986), 35–42. https://doi.org/10.1109/MC.1986.1663327

[64] Zhexuan Song and Nick Roussopoulos. 2003. SEB-tree: An Approach to Index Continuously Moving Objects. In *Mobile Data Management, 4th International Conference, MDM 2003, Melbourne, Australia, January 21-24, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2574)*. Springer, 340–344. https://doi.org/10.1007/3-540-36389-0_25

[65] Bela Stantic, Justin Terry, Rodney W. Topor, and Abdul Sattar. 2010. Indexing Temporal Data with Virtual Structure. In *Advances in Databases and Information Systems - 14th East European Conference, ADBIS 2010, Novi Sad, Serbia, September 20-24, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6295)*. Springer, 591–594. https://doi.org/10.1007/978-3-642-15576-5_53

[66] Bela Stantic, Rodney W. Topor, Justin Terry, and Abdul Sattar. 2011. A Triangular Decomposition Access Method for Temporal Data - TD-tree. In *Twenty-Second Australasian Database Conference, ADC 2011, Perth, Australia, January 2011 (CRPIT, Vol. 115)*. Australian Computer Society, 113–122. http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV115Stantic.html

[67] Sairam Subramanian and Sridhar Ramaswamy. 1995. The P-range Tree: A New Data Structure for Range Searching in Secondary Memory. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1995. San Francisco, California, USA*. ACM/SIAM, 378–387. http://dl.acm.org/citation.cfm?id=313651.313769

[68] Roberto Tamassia. 1993. Priority Search Trees - Part I, Computational Geometry Lecture Notes. https://cs.brown.edu/courses/cs252/misc/resources/lectures/pdf/notes07.pdf

[69] Roberto Tamassia. 1993. Priority Search Trees - Part II, Computational Geometry Lecture Notes. https://cs.brown.edu/courses/cs252/misc/resources/lectures/pdf/notes18.pdf

[70] Jeffrey Scott Vitter. 2001. External memory algorithms and data structures. *ACM Comput. Surv.* 33, 2 (2001), 209–271. https://doi.org/10.1145/384192.384193

[71] 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. https://doi.org/10.1145/3748777.3748785

[72] Yuxiang Wang, Ziyuan He, Yongxin Tong, Zimu Zhou, and Yiman Zhong. 2025. Timestamp Approximate Nearest Neighbor Search Over High-Dimensional Vector Data. In *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong, May 19-23, 2025*. IEEE, 3043–3055. https://doi.org/10.1109/ICDE65448.2025.00228

[73] Bryan T. Wilkinson. 2014. Amortized Bounds for Dynamic Orthogonal Range Reporting. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8737)*. Springer, 842–856. https://doi.org/10.1007/978-3-662-44777-2_69