LIT: Lightning-fast In-memory Temporal Indexing

GEORGE CHRISTODOULOU*, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Netherlands
PANAGIOTIS BOUROS, Institute of Computer Science, Johannes Gutenberg University Mainz, Germany
NIKOS MAMOULIS, Department of Computer Science and Engineering, University of Ioannina, Greece

We study the problem of temporal database indexing, i.e., indexing versions of a database table in an evolving database. With the larger and cheaper memory chips nowadays, we can afford to keep track of all versions of an evolving table in memory. This raises the question of how to index such a table effectively. We depart from the classic indexing approach, where both current (i.e., live) and past (i.e., dead) data versions are indexed in the same data structure, and propose LIT, a hybrid index, which decouples the management of the current and past states of the indexed column. LIT includes optimized indexing modules for dead and live records, which support efficient queries and updates, and gracefully combines them. We experimentally show that LIT is orders of magnitude faster than the state-of-the-art temporal indices. Furthermore, we demonstrate that LIT uses linear space to the number of record indexed versions, making it suitable for main-memory temporal data management.

CCS Concepts: • **Information systems** → **Database query processing**; **Data access methods**; *Temporal data.*

Additional Key Words and Phrases: Temporal data, Query processing, Indexing

ACM Reference Format:
George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2024. LIT: Lightning-fast In-memory Temporal Indexing. *Proc. ACM Manag. Data* 2, 1, Article 111 (February 2024), 27 pages. https://doi.org/10.1145/3639275

1 INTRODUCTION

Temporal data management has been studied extensively for at least four decades [6, 7, 20, 24, 38]. Temporal databases track the database evolution for the support of *time-travel* queries: given a database query and a past time moment (or time interval), process the query on the database instance(s) that was (were) *valid* then. Temporal and multi-version data management re-gained interest recently [4, 5, 8, 11, 15, 21, 23, 28, 33, 43], due to the increase of cheap storage that makes it possible to track the versions of a database even in the main memory of a commodity machine.

As an example, consider a database table T, storing information about company employees. The table has three attributes: ID, Name, and Salary. As the database evolves over time, there are changes in the table, where records are inserted or deleted, or attribute values of existing records are updated. Figure 1 shows some versions of T, where, at time t_0, T is initialized to include two

*Work was done while the author was with the University of Ioannina, Greece

Authors' addresses: George Christodoulou, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Netherlands, g.c.christodoulou@tudelft.nl; Panagiotis Bouros, Institute of Computer Science, Johannes Gutenberg University Mainz, Germany, bouros@uni-mainz.de; Nikos Mamoulis, Department of Computer Science and Engineering, University of Ioannina, Greece, nikos@cse.uoi.gr.

2836-6573/2024/2-ART111 $15.00
https://doi.org/10.1145/3639275

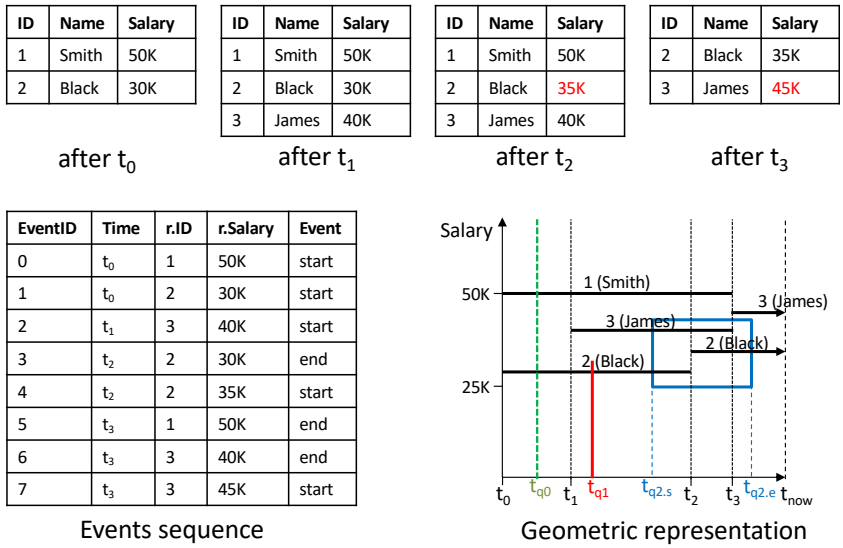


Fig. 1. Example of a time-evolving table

records; at time t_1, a new record (with ID=3) is inserted to T; at time t_2, the Salary value of record 2 is updated; and at time t_3, record 1 is deleted and record 2 is updated. The evolution of T can be seen as a stream (time-sequence) of events, also shown in the figure (bottom-left). Insertions (deletions) are modeled by events of type *start* (*end*); each update (i.e., value changes) is modeled by a deletion immediately followed by an insertion. Finally, the figure (bottom-right) shows the *validity* intervals of the records and their values in the Salary attribute, as flat line segments. The current time is denoted by t_{now}.

We study the problem of indexing a database table T to support time-travel queries. We first focus on indexing for *pure* time travel queries, where the objective is to retrieve the record versions that were valid at a given *timepoint* or *timerange* in the past. In our running example (Figure 1), such a pure timepoint query q_0 is "find all records in T, which were valid at time t_{q0}" and the answer records are (1, Smith, 50K) and (2, Black, 30K). Then, we study how our indexing scheme can be extended to temporally index T with respect to a specific attribute $T.A$, for *range* time travel queries, that retrieve record versions r in T which were valid at a given timepoint/timerange and their $r.A$ satisfies a range query predicate. Such a range-timepoint query q_1 is: "find all records in T, which were valid at time t_{q1} and have Salary at most 32K." Query q_1 is geometrically represented by the vertical line segment starting at time t_{q1} and retrieves the records, corresponding to the line segments intersected by vertical segment starting at t_{q1}, i.e., record (2, Black, 30K). Another example is range-timerange query q_2: "find all records in T, which were valid anytime between $t_{q2.s}$ and $t_{q2.e}$ and have Salary between 25K and 43K," modeled by the rectangle in Figure 1. Again, the query results are the segments that intersect the rectangle, namely (2, Black, 30K) valid in $[t_0, t_2)$, (2, Black, 35K) valid in $[t_2, t_{now})$, and (3, James, 40K) valid in $[t_1, t_3)$. Note that it is important to find the records *and* their validity intervals in order to be able to distinguish between results corresponding to different versions of the same record/entity (e.g., Black in the results of q_2).

Previously proposed temporal indices can be classified to (1) methods for transaction-time and multi-versioned databases (e.g., MVB-tree [1], Timeline index [24]), (2) data structures for (time) intervals [3, 12, 14, 17, 26]. Our work belongs to the first category, where the goal is to support not only the aforementioned queries, but also data version updates in real-time, in a continuously

evolving database. Indices in the second category offer fast search times, but (1) they do not support effectively *live* data versions, i.e., records which are valid now, but we do not know up to when in the future and (2) are mainly designed for static intervals in a static domain.

Contribution We aim at the efficient support of updates in a continuously evolving database, and target a much better performance in queries compared to the state-of-the-art access methods for time-evolving data. Our proposal is LIT, a *hybrid* index, which indexes *live* records (i.e., those which valid at t_{now}), like (2, Black, 35K), by a different data structure compared to *dead* records (i.e., those not currently valid), like (2, Black, 30K). Specifically, LIT includes a LiveIndex for the live records; LiveIndex only needs to index the begin time of the validity of each live record. For dead records we use a DeadIndex, which includes their validity intervals with both starting and ending timepoints. When a temporal record is created, it is added to LiveIndex; when the record dies (i.e., deleted from the temporal table T, or updated), it is deleted from LiveIndex and added to the DeadIndex. Given these operations, LiveIndex supports fast *temporal appends* (i.e., add a new live record at the "temporal" end of the index) and deletions, whereas DeadIndex needs only to support insertions (anywhere in the time domain up to t_{now}), but no deletions (since past data versions are never deleted from a temporal DB). Both LiveIndex and DeadIndex gracefully adapt to the ever-evolving time domain. We tuned, implemented, and tested the best implementations of LiveIndex and DeadIndex and compared LIT with in-memory versions of the state-of-the-art temporal and multi-version indices [1, 24] on mixed workloads of queries and version updates, showing that LIT is orders of magnitude faster.

Outline The rest of the paper is organized as follows. Section 2 reviews related work on managing intervals and database record versions. In Section 3, we define time-travel queries and the record version data whereon they apply. Section 4 proposes an extension to the state-of-the-art interval index [12] to manage live and dead record versions in an ever-growing time domain. In Section 5, we present LIT, the main proposal of this paper for pure time-travel queries. Section 6 discusses how LIT can be extended to index an attribute A of the records besides their temporal validity intervals, in order to support range time-travel queries. Section 7 discusses the integration of our main-memory LIT in a DMBS that should support persistence and fault-tolerance (recovery). Section 8 includes our experimental analysis and, finally, Section 9 concludes the paper.

2 RELATED WORK

In this section, we review related work on (1) indexing intervals and (2) indexing data versions in a time-evolving database; we also briefly present other recent work on temporal data management.

2.1 Indexing Intervals

Valid-time temporal databases store record versions which are valid during a well-defined time interval [32]. This interval could refer to the past, the future, or may start at some time in the past and finish in the future (for example, an activated credit card which expires at some time in the future). The order by which records in a valid-time database are inserted, deleted, or updated is not necessarily related to the validity time of the records.

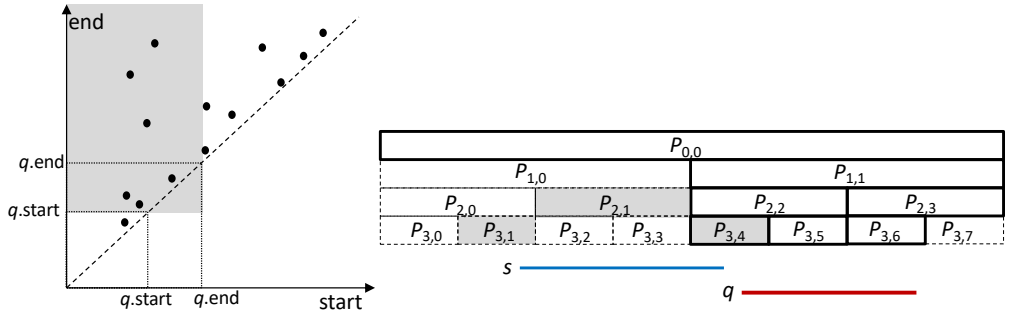
Managing valid-time records for the evaluation of time-travel queries can then be considered as a case of indexing intervals (i.e., one-dimensional ranges), which is a well-studied problem with lots of previous work [3, 12, 14, 17, 26]. Classic data structures for intervals include the segment tree [14] and the interval tree [17]. They are both binary search trees, built from a *static* set of intervals and designed to answer point queries (i.e., find the intervals that contain a given value) in $O(\log n + K)$ time, where n is the number of data intervals and K is the number of query results. Their space complexity is $O(n \log n)$ and $O(n)$, respectively. The interval tree can be used to also answer *range* queries, i.e., find intervals that overlap with a query interval (value range) in $O(\log n + K)$.

Data structures for multi-dimensional boxes, such as the R-tree [2, 22], can also be used for 1D intervals. For example, a simple and dynamic data structure for intervals is the 1D-grid, which divides the space into a number of partitions, either uniformly or adaptively to the interval distribution. Each interval is then assigned to all partitions that overlap with it. A point (or range) query q is evaluated by accessing the partition(s) intersecting q and reporting the intervals there after conducting comparisons as necessary. Duplicate results can be avoided using the reference point technique [16] or after dividing the data in each partition to classes based on whether they begin inside or before the partition [12, 29, 40]. A data structure which considers both the values and the durations of the intervals is the period index [3]. The domain is primarily divided by a 1D-grid. For each grid partition, a hierarchical partitioning of its sub-domain is applied and each interval in the partition is assigned to a sub-partition according to its position and duration.

An alternative approach is to map intervals to 2D points and then index them by an off-the-shelf spatial data structure [14, 19]. Specifically, each data interval $s = [s.start, s.end)$ is mapped to point $(s.start, s.end)$ in the $D \times D$ space, where D is the domain of the interval endpoints. Figure 2(a) shows a number of intervals as points in this 2D space. Since $s.start < s.end$ for each interval s, the points are all above the diagonal connecting points $(0, 0)$ and (D, D). Each point or range query becomes a rectangular range query in the 2D space, having x- and y-projections $[0, q.end]$ and $[q.start, D]$, respectively, as shown by the shaded rectangle in Figure 2(a). This approach has been used in previous work on managing text document versions [41] and temporal data [28].

HINT [12] is the state-of-the-art in-memory index for intervals. HINT defines a hierarchy of $m+1$ levels, such that level ℓ, $0 \leq \ell \leq m$ uniformly divides the domain into 2^m partitions, as shown in Figure 2(b) for $m = 3$. Each data interval s is then normalized and discretized in the $[0, 2^m - 1]$ domain, and assigned to the smallest set of partitions from all levels that cover s. So, s is assigned to at most 2 partitions per level. The intervals in each partition are divided into two classes: those that start before the partition (*replicas*) and those that start inside the partition (*originals*). For instance, in Figure 2(b), interval s is added to the shaded partitions; in $P_{3,1}$, s is added to the sub-division $P^O_{3,1}$, storing *original* intervals in $P_{3,1}$, while in $P_{2,1}$ and $P_{3,4}$, s is stored to the corresponding replica sub-divisions ($P^R_{2,1}$ and $P^R_{3,4}$, respectively). Given a point or a range query q, at every level ℓ of HINT only the sequence of partitions that intersect q are accessed. For the query q in Figure 2(b), the partitions with a solid/bold outline will be accessed. In addition, only for the first such partition in ℓ both originals and replicas in it are considered, while for the remaining partitions only originals are considered. Lastly, the number of partitions in the entire index for which comparisons between data interval endpoints and query endpoints are required is expected to be at most 4 [12]. This means that the great majority of query results are reported without needing to perform comparisons, which gives HINT a big performance advantage over other methods.

Deficiencies of interval indices. While HINT [12] is the best performing index, it shares a weakness of most other interval indexing methods: the domain of the interval endpoints should be known apriori. If the data domain grows (i.e., as in a temporal database), the partitions have to potentially be updated to cover the new part of the domain and it might be necessary to change the assignments of data intervals to partitions to maintain the good properties of the index. On the other hand, the 2D point transformation approach [41] does not have this problem as a 2D spatial index such as the R-tree can adapt to a growing domain. Still, the query regions are relatively large and touch a large part of the 2D space, most of which is sparsely populated, so this approach is not as efficient as [12]. More importantly, all methods discussed in this section are not appropriate for indexing *live* data versions in temporal databases, whose end is unknown (i.e., equal to the ever-changing t_{now}). Finally, data structures for intervals are not designed for indexing another attribute at the same time; i.e., they are not appropriate for the *range* time-travel queries discussed



(a) 2D mapping [28, 41] (b) HINT [12]

Fig. 2. Interval indices

in the Introduction. In Section 4, we show how we can adapt HINT to adapt to a growing time domain and to accommodate live data versions. However, this adaptation cannot cope with the high overhead of updates; hence, it is inferior compared to our main proposal (LIT).

2.2 Indexing Data Versions

Transaction-time databases [31] manage the evolution history of a database. In Section 1, we gave an example of such a database containing a table T with employees records. Indexing transaction-time DBs is more challenging than valid-time DBs, since there are *live* records which are valid now, but we do not know their end-time. These records comprise the current database state and may be changed or deleted in the future, but we are not aware of the exact time for this. On the other hand, *dead* records belong to past states for which we do know their end-time. Records (2, Black, 30K) and (2, Black, 35K) in Figure 1 are examples of dead and live records, having validity $[t_0, t_2)$ and $[t_2, t_{now})$, respectively. In fact, these two records correspond to versions of the same record (employee Black). Versions of the same record cannot temporally overlap.

Previous work on temporally indexing an evolving DB table extend current-state indices to support search on all table versions. A representative access method in this category is the Multi-version B-tree (MVB-tree) [1], which is a forest of B-trees that succinctly capture the values of the indexed attribute in all versions of records. A comprehensive survey of early indexing methods for time-evolving databases see [37]. These indices do not only support pure time travel queries, but also *range* time travel queries based on a search-key attribute A (i.e., from all records r which were valid at some timestamp or period in the past retrieve those for which $v_1 \leq A \leq v_2$). To support such queries, they index simultaneously the temporal versions of the records and their values on the search key attribute A. These methods focus on minimizing disk I/O during search; their main-memory versions are relatively slow in search and updates compared to the interval indices reviewed in Sec. 2.1.

A more recent index for transaction-time DBs implemented in SAP HANA is the *Timeline* index [24], which builds upon the Time index [18] and supports very fast updates. In a nutshell, the Timeline index is an Events Sequence Table (see Fig. 1) paired with a set of *Checkpoint Tables* (CT). A CT at timestamp t_i materializes the entire set of *active* record-ids at t_i. To evaluate a point or range query, the latest checkpoint before the query is accessed to *activate* the records in it, and the Events Sequence Table (EST) is scanned from thereon until the end time of the query to identify the records that are active at or during the query. The update cost of the Timeline index is minimal as a database change simply appends an event at the end of the EST; still, the rare CT construction events have significant cost. Query evaluation using the Timeline index is quite expensive due to

the overhead of scanning the events and updating the set of active records until the entire query result is retrieved.

In this paper, we revisit the indexing of transaction-time DB tables (i.e., version data), for pure time-travel and range time-travel queries, in main memory, focusing on minimizing the CPU cost in queries and updates. Our approach is a major departure from previous work which indexes dead and live versions in the *same* data structure. Instead, we define two separate data structures for live and dead versions; in principle, versions which die are transfered from the first data structure to the second. By decoupling indexing for live and dead versions, we can optimize both data structures.

2.3 Other related work

Recent work in temporal databases studies the efficient evaluation of other queries, besides time-travel selections. *Temporal aggregation* [25, 30, 34, 39, 42] computes aggregates of valid record versions (e.g., total project funding) during a query time period (e.g., from 3-23-2021 to 5-15-2023); the output is one value for each time interval in the query period where the aggregate does not change. *Temporal top-k* queries [21, 41] are a special case of temporal aggregation. A *temporal join* [9, 10, 23, 33, 36] finds pairs of record versions (in two different tables) whose validities temporally overlap and they agree on the join key attribute. Historical *what-if* queries compute the effect that a change in a historical record value would have to the evolution of the database [11]. Other recent related work includes the definition of new temporal semantics [15], system optimizations in the implementation of temporal and multi-version databases [5, 28, 43], temporal database benchmarking [4], and novel temporal integrity constraints [8].

3 PROBLEM DEFINITION

We consider a database table T, updated over time, by inserting, deleting or updating records. In this work, we focus on developing indexing for the following types of time-travel queries [37].

Query 1 (pure timeslice/timerange query). *Given a query time point $q.t$ or query time interval $[q.tstart, q.tend]$, retrieve the records in all versions of T which were valid at $q.t$ or some time during $[q.tstart, q.tend]$, respectively, together with their validity intervals.*

Query 2 (range timeslice/timerange query). *Given a query time point $q.t$ or query time interval $[q.tstart, q.tend]$, an attribute A of T, and a range $[q.Astart, q.Aend]$, retrieve the records r in all versions of T which (1) were valid at $q.time$ or some time during $[q.start, q.end]$, respectively, and (2) satisfy $q.Astart \leq r.A \leq q.Aend$ together with their validity intervals.*

Without loss of generality, we assume query intervals closed at both ends. In addition, for each change in T an update event is generated, which may trigger updates in the indices of T. These update events include: (1) the insertion of a record to T, (2) the deletion of a record from T, and (3) the change of one or more attribute values of a record in T. An event of type (3) can be modeled as an event of type (1) immediately followed by an event of type (2).

In a *pure-time index* that supports Query 1, each of the above event types affects one or more index entries. Specifically, the insertion of a record r at time point t inserts a new index entry for $r.id$ having as validity interval $[r.start = t, r.end = t_{now})$, where t_{now} models the current time point, which is ever-changing. The deletion of record r at time t updates the last index entry for $r.id$ from $[r.start, t_{now})$ to $[r.start, t)$. The update of a record r at time point t triggers a deletion of r at point t, followed by an insertion of the new version of r with $r.tstart = t$.

In an index that supports Query 2, the changes affect the index as described above with the exception that record updates on attributes other than the indexed attribute $r.A$ have no effect on the index. In other words, we consider two or more consecutive versions of r having the same value in $r.A$ as the same version.

As discussed in Section 2.2, there is ample previous work on temporal indexing for time-evolving database tables. However, these indices have poor search times compared to the interval indexing approaches of Section 2.1. In Section 4, we propose an extension of the state-of-the-art interval index to support Query 1 over time-evolving databases. LIT, the main proposal of this paper is first described in Section 5 for pure time queries (Query 1) and then extended in Section 6 for range time queries (Query 2).

4 TIME-EVOLVING HINT

A first attempt to define an efficient in-memory index for time-evolving tables is to convert HINT [12, 13], the state-of-the-art interval index, to a single data structure that can handle both live and dead intervals (records). We call this data structure *time-evolving* HINT (*te*-HINT). A *te*-HINT for pure time-travel queries (Query 1) extends HINT in two directions. First, it includes both live and dead records, whereas HINT indexes only intervals for which the *end* endpoint is immutable. Second, it supports an evolving domain for the interval endpoints (i.e., an evolving time domain); the original HINT requires a pre-defined domain. These differences require some structural changes and new update operations in *te*-HINT, compared to HINT [12], which are described next.

4.1 Live and dead sub-partitions

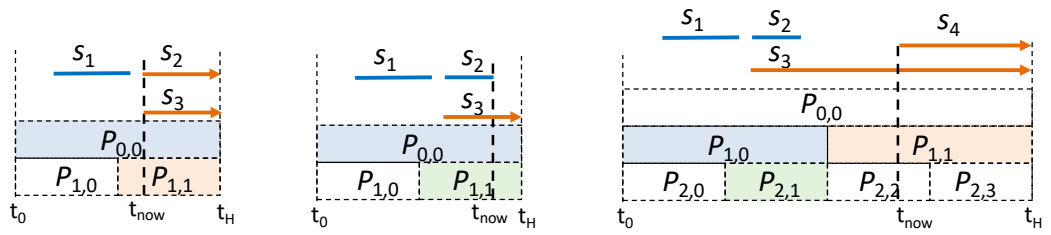
The first difference between *te*-HINT and HINT is the introduction of *live* partitions in the former. Recall from Section 2.1 that in each partition $P_{\ell,i}$ at level ℓ of HINT, the intervals are divided into two classes: the set of *originals* $P^O_{\ell,i}$ which start inside the domain range of $P_{\ell,i}$ and the set of *replicas* $P^R_{\ell,i}$, which start before the domain of $P_{\ell,i}$. In *te*-HINT, we further classify each interval $s \in P^O_{\ell,i}$ as *live original* or *dead original*, depending on whether its end time point is known; we denote the sub-partitions that hold live and dead originals by $P^{O_L}_{\ell,i}$ and $P^{O_D}_{\ell,i}$, respectively. Similarly, we maintain sub-partitions $P^{R_L}_{\ell,i}$ and $P^{R_D}_{\ell,i}$ for the replicas of $P_{\ell,i}$. Dead intervals in $P^{O_D}_{\ell,i}$ or $P^{R_D}_{\ell,i}$ are immutable, which means that they persist in the partition and cannot move to other partitions, whereas live intervals can be deleted or moved to other partitions.

4.2 Handling updates

There are two types of update events over time: either the creation of a new live interval (as a result of an insertion/modification to the database), or the finalization of an existing live interval (as a result of a deletion/modification to the database).

Insertion events. In the case where an insertion event arrives, i.e., a new *live* interval s begins corresponding to a version of a record r, we insert s to *te*-HINT (in live sub-partitions) using the insertion algorithm of HINT [12], assuming that the end time point of s is the end of the current domain of *te*-HINT (i.e., a timepoint in the future), called the *horizon* of *te*-HINT and denoted by t_H. At the same time we insert an entry $\langle r.id, s.start \rangle$ in an auxiliary key-value data structure $\mathcal{H}_{r.id \rightarrow start}$ that facilitates finding a live interval in *te*-HINT given the corresponding record id. Figure 3(a) shows a simple example of a 2-level *te*-HINT, holding interval s_1, which corresponds to a dead record, in partition $P_{0,0}$ (sub-partition $P^{O_D}_{0,0}$). Two new live intervals s_2 and s_3 are created at t_{now} and they are inserted to partition $P_{1,0}$ (sub-partition $P^{O_L}_{1,0}$).

Deletion events. When a deletion event arrives for record r carrying an $s.end$, i.e., an existing live interval s is terminated and becomes dead, we need to remove s from the live sub-partitions of *te*-HINT and add it to the appropriate dead partitions. For this, we use $\mathcal{H}_{r.id \rightarrow start}$ to retrieve $s.start$, using $r.id$, and we run the insertion algorithm of HINT for $s' = [s.start, t_H)$ to identify the partitions wherein s' appears and remove s' from the corresponding live sub-partitions. Subsequently, we use



(a) insert s_2 & s_3 (b) delete s_2 (c) insert s_4

Fig. 3. Example of *te*-HINT

the insertion algorithm again to add $s = [s.start, s.end)$ to the relevant dead sub-partitions. Note that some of the partitions identified by the deletion algorithm may differ from those found by the insertion algorithm, because $s \neq s'$. As an example, assume that at time t_{now} shown in Figure 3(b), a deletion event for live interval s_2 arrives, i.e., the record version corresponding to s_2 is deleted from the indexed table T. After finding $s_2.start$ using $\mathcal{H}_{r.id \rightarrow start}$, the partitions ($P^{O_L}_{1,0}$) where s_2 is stored as live are identified using interval $[s.start, t_H)$ and s_2 is removed from them, and, finally, s_2 becomes $[s_2.start, t_{now})$ and is re-inserted to *te*-HINT as dead (i.e., to partition $P^{O_D}_{1,0}$).

Domain Extension. *te*-HINT is initialized to have a single level (0) which includes a single partition $P_{0,0}$. The timespan $[0, t_H)$ of the partition is small (e.g., one hour) and depends on the application. In both insert and delete events, it may happen that the current time point t_{now} when the update takes place is beyond the current horizon t_H of *te*-HINT. Such an update triggers the *extension* of the (time) domain that *te*-HINT covers. The easiest way to accommodate this extension is to double the domain (and the horizon t_H), by adding one more level to *te*-HINT (and repeat as necessary). Specifically, we add a new level 0 to the index and add 1 to the identifiers of existing levels (i.e., previous level 0 becomes level 1, level 1 becomes level 2, etc.). This does not affect the identifiers and contents of existing partitions at each level ℓ, but doubles the number of possible partitions at ℓ. Subsequently, we add all live intervals from all partitions as *live replicas* to partition $P_{1,1}$, except from those in old partition $P_{0,0}$ which are moved to the new $P_{0,0}$. By this, we minimize the replication of live intervals and also minimize the necessary updates when new events arrive. Essentially, live intervals are moved *only* when there is a domain extension. Continuing the previous example, assume that a new live interval s_4 is created at t_{now} of Figure 3(c). Since t_{now} is greater than or equal to t_H, as per the previous state of *te*-HINT (Figure 3(b)), t_H is doubled, one more level is added to *te*-HINT, and the current partitions are renamed (i.e., previous $P_{0,0}$ now becomes $P_{1,0}$, etc.), without any change in their contents. Existing live interval s_3 is added to the new partition $P^{R_L}_{1,1}$. The new interval s_4 is added using the insertion algorithm to $P^{O_L}_{1,1}$.

5 THE LIT HYBRID INDEX

Capitalizing on the original HINT, *te*-HINT will deliver excellent performance on pure time-travel queries, as shown in [12, 13]. But, *te*-HINT will suffer from slow updates, mainly due to the insertion (and transfer) of intervals to (and beween) multiple partitions when record versions are initiated (terminated). In view of this shortcoming, we design a *hybrid* index, termed LIT, which decouples the indexing of live and dead versions. For now, we describe LIT for pure time-travel queries (Query 1 in Section 3). Its extension for range time-travel queries (Query 2) will be discussed in Section 6.

Overview of LIT Figure 4 shows an overview of LIT, which comprises two components; a LiveIndex denoted by $\mathcal{I}_L$, storing all current record versions (indexed by their *start* timepoint) and a DeadIndex,

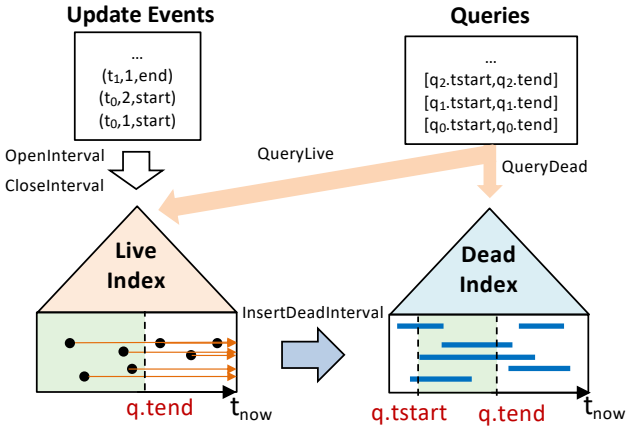


Fig. 4. Overview of LIT

denoted by $\mathcal{I}_D$, for the dead (i.e., past) record versions (indexed by their validity intervals). Both components are dynamic, albeit handling different updates. The stream of updates to the indexed table T is consumed by the LiveIndex $\mathcal{I}_L$. Specifically, when a new record version is created (i.e., an insertion to T), the start point $s.start = t_{now}$ of its validity interval is inserted to $\mathcal{I}_L$; this event type has no impact on the DeadIndex $\mathcal{I}_D$. On the other hand, when a record version "dies" (i.e., a deletion from T), the corresponding entry is removed from $\mathcal{I}_L$ and an entry is inserted to $\mathcal{I}_D$ for the dead record version. As already mentioned, record updates are treated by terminating (i.e., "deleting") the current (live) version of the record and inserting a new version.

To evaluate a pure time-travel query $q = [q.tstart, q.tend]$ both $\mathcal{I}_L$, $\mathcal{I}_D$ need to be probed. As the two components index disjoint sets of record versions, these probing tasks are completely independent. Specifically, we probe the LiveIndex $\mathcal{I}_L$ using only $q.tend$; every live record that started before $q.tend$ is guaranteed to be part of the query result. In contrast, the DeadIndex evaluates a typical interval range query to find all dead record versions with a validity interval that overlaps q. In what follows, we elaborate on the internals of the LIT components $\mathcal{I}_L$ and $\mathcal{I}_D$, and describe their key operations.

5.1 The LiveIndex Component

The LiveIndex $\mathcal{I}_L$ offers three key operations. Specifically, $\mathcal{I}_L$ is updated to index a new live record (Function OpenInterval) or updated to un-index a record version that just died (Function CloseInterval). $\mathcal{I}_L$ also evaluates pure time-travel queries (Function QueryLive). To efficiently implement these functions, $\mathcal{I}_L$ defines an internal identifier $r.num$ for each live record version r in it. The num identifier is a serial number that captures the order in which the version $start$ timepoints were read from the input stream of updates; num is used to (1) locate a live version to be deleted from $\mathcal{I}_L$ when a delete event arrives for it, and (2) define an implicit order of the live versions based on their $start$ points, used to index them in $\mathcal{I}_L$. LiveIndex also maintains an auxiliary hash table $\mathcal{H}_{r.id \rightarrow num}$, which returns the internal num id, for the live version of a given record id.

5.1.1 Data structures. We discuss three alternative data structures for LiveIndex, aiming at both fast updates and efficient time-travel queries. We experimentally compare them in Sec. 8.2.1.

Array. The first alternative is to use an *append-only array* to index live records in sequential fashion. Updates can be efficiently handled in constant time, as follows. Function OpenInterval simply appends an entry at the end of the array for a new live record version, while CloseInterval, drops a tombstone on the existing entry for a newly closed record version. This entry can be

directly accessed using the *num* of the record, which is obtained by probing the record *id* against $\mathcal{H}_{r.id \rightarrow num}$. To answer queries, the QueryLive function scans the dynamic array from its first entry, comparing the *start* of every live record to $q.tend$ while ignoring the tombstones. By construction, the dynamic array stores the live records sorted by their *num*, which means that the records are also implicitly sorted by their *start*, in increasing order. Hence, QueryLive terminates the scan when the first record that started after $q.tend$ is accessed.

Search tree. A second alternative data structure for the LiveIndex $\mathcal{I}_L$ is a search tree (e.g., a B$^+$-tree), using *num* as the search key. With such a search tree in place, we no longer need to lazy-update $\mathcal{I}_L$ when a record version dies. Instead, CloseInterval probes the tree using the *num* identifier of the record (obtained from $\mathcal{H}_{r.id \rightarrow num}$), and then directly removes the corresponding entry. As a search tree typically supports scanning its entries in the search key order, to answer a $[q.tstart, q.tend]$ query, we scan and report from the first entry until we find the first that has its *start* after $q.tend$.

Enhanced hashmap. In terms of updating (Functions OpenInterval and CloseInterval), we generally expect the sorted array to outperform the search tree, due to its simplicity. Querying efficiency depends on the characteristics of the input stream; update-heavy workloads create a large amount of tombstones to the array, rendering it slower than the search tree. In view of the above, we should consider a data structure, which will exhibit competitive update time to the array and have lower query time. To this end, we suggest using an *enhanced* hash table, similar to the Gapless hashmap proposed in [35] or the java.util.LinkedHashMap in Java. Such structures can handle insertions and deletions using *num* in constant time (typical for hash tables), but also offer scan time linear to the number of contained entries, which facilitates fast query processing. In particular, the Gapless hashmap uses a contiguous memory area to store the elements. Insertions append new elements at the end of this area, while deletions are handled by swapping the deleted element with the last one and reducing the array size by one. Scanning is fast as it steps through the contiguous storage area sequentially. Different to both the array and the search tree, the hashmap does not maintain the entries sorted by their *num*, and therefore, a full scan is required to answer time-travel queries.

5.1.2 Temporal partitioning of LiveIndex. Given a query, a LiveIndex implemented by any of the data structures in Section 5.1.1 would need to conduct comparisons for a large number of live versions (independently of the underlying data structure), since there is no way to directly output versions guaranteed to start before $q.tend$. In view of this, we propose a *temporal partitioning* of the LiveIndex to boost time-travel queries. The key idea is to maintain $\mathcal{I}_L$ as a *chain of temporal partitions* or simply *buffers*, instead of a single one, such that all *num*'s in a buffer are smaller than all *num*'s in the next buffer. Hence, the *start* points of live record versions in a buffer are smaller than or equal to the *start* points of live versions in the next buffer. For each query, only the buffers that may contain results are accessed and even more importantly, comparisons are conducted only for the last buffer. This partitioning of the LiveIndex $\mathcal{I}_L$ is orthogonal to the data structure used for each buffer.

Duration-based partitioning. An intuitive partitioning approach for $\mathcal{I}_L$ is to consider a *duration constraint* D_L. Under this, $\mathcal{I}_L$ essentially resembles a uniform 1D-grid of equi-sized partitions, one for each buffer. A buffer B_i contains the live entries that started inside the $[i \cdot D_L, (i+1) \cdot D_L)$ range of time units. Given a $[q.tstart, q.tend]$ time-travel query, we first determine the bucket B_{end} that contains the $q.tend$ timestamp; this can be done in constant time by a simple $\lfloor q.tend / D_L \rfloor$ division. The records inside the buffers before B_{end} can be directly reported as results; by construction of the LiveIndex, these records started before $q.tend$. In contrast, comparisons against $q.tend$ are required for the live records inside the last B_{end}, i.e., QueryLive handles B_{end} as if the LiveIndex comprised a single buffer. Regarding updates, inserting a new live record version to $\mathcal{I}_L$ (Function OpenInterval)

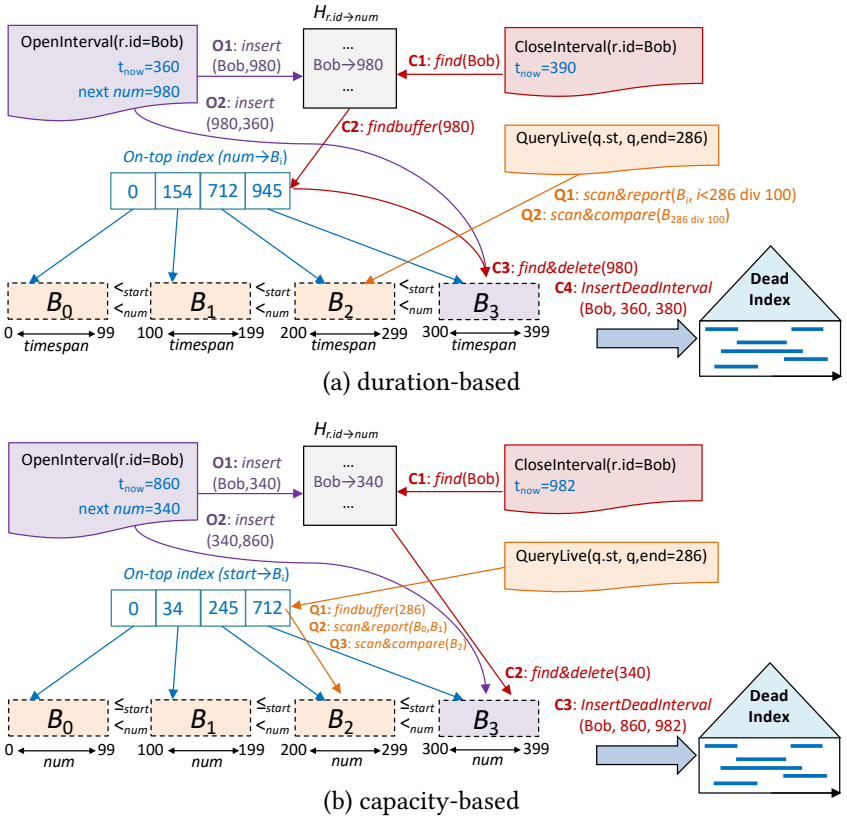


Fig. 5. LiveIndex: partitioning

is not significantly affected by the above partitioning, as the new entry will be added to the last buffer, i.e., the one containing the most fresh records; extra action is required when D_L time units have already past and a new buffer needs to be created first. CloseInterval is more challenging, as we need to fast determine the buffer which contains the *start* of the dying record version. For this purpose, we define an auxiliary, lightweight structure on top of the buffers. This structure stores a $\langle num, ptr \rangle$ entry for each buffer B of $\mathcal{I}_L$, where *num* is the lowest internal identifier of a live record version inside B and *ptr* is a pointer to directly access B in the chain. Recall at this point, that LiveIndex is organized by *num* and so is its on-top structure, by construction. When a version of record $r.id$ dies, CloseInterval finds its *num* using $\mathcal{H}_{r.id \rightarrow num}$, then binary-searches the on-top structure using $r.num$ and, lastly, follows the buffer pointer to locate the entry for *num* inside the corresponding buffer B. After deleting the entry from $\mathcal{I}_L$, CloseInterval, forwards the dead version for insertion to $\mathcal{I}_D$. OpenInterval may update the on-top structure when the last buffer is full and a new is created. Figure 5(a) exemplifies a *duration-based* partitioned LiveIndex, with the necessary steps taken for each of the OpenInterval, CloseInterval, and QueryLive operations.

Capacity-based partitioning. Duration-based partitioning may define unbalanced buffers with respect to the number of contained entries, rendering unbalanced query costs. An alternative partitioning approach that results in balanced partitions is to use a capacity constraint C_L, allowing each buffer to hold at most C_L entries. [1] Different to the duration-based partitioning discussed

[1]For array structure, tombstones are *not* excluded when counting the contained records.

above, capacity-based partitioning can directly access the needed buffers during both types of updates. For OpenInterval, we simply append the new live record version at the last buffer, while for CloseInterval, a simple num/C_L division exactly determines which buffer B contains the recently deceased version. Note that if the last buffer is already full, OpenInterval will create a new buffer B_{new} after the last one and simply append the new live version in B_{new}.

On the other hand, it is no longer possible to directly determine the B_{end} buffer for a $[q.tstart, q.tend]$ query. In view of this, we define an on-top structure, which stores a $\langle st, ptr \rangle$ entry for each buffer B of the LiveIndex, where st is the lowest $start$ timepoint of a record version inside B and ptr is a pointer to directly access B. Note that the on-top search structure is by construction sorted by version $start$ and that it may contain multiple entries for the same $start$. Hence, given a $[q.tstart, q.tend]$ query, QueryLive first binary-searches the on-top structure to identify the *first* buffer that could contain $q.tend$ and sets this as B_{end}. With B_{end}, the function proceeds similarly to the duration-based LiveIndex, by directly reporting records inside every buffer before B_{end} and conducting comparisons against $q.tend$ for B_{end}. Lastly, besides updating buffers, OpenInterval and CloseInterval also update accordingly the on-top structure. Figure 5(b) illustrates a detailed example of the *capacity-based* partitioning of LiveIndex and operations on it.

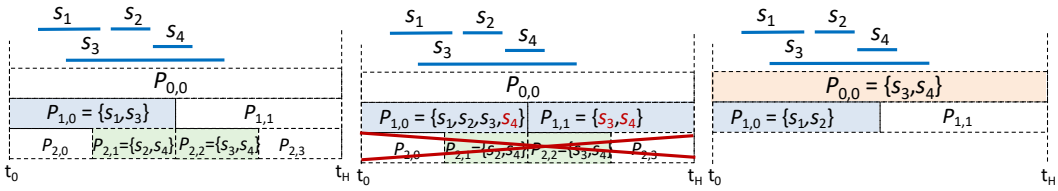
5.1.3 Optimizations. As the timeline evolves and live records die, buffers may become under-utilized or even completely empty. To deal with this issue, reorganization can be employed for both types of partitioning. For the duration-based LiveIndex, such a sparsity issue is expected to especially occur in the first (early) buffers. Hence, we could merge adjacent sparse buffers into one and accordingly update also the on-top structure.[2] To answer time-travel queries, an extra auxiliary structure is now needed to capture the time-range covered by this new buffer, as the $q.tend/D_L$ division can only work for un-merged buffers. Intuitively, a second on-top structure maintaining the lowest $start$ inside a buffer will allow us to deal with several rounds of buffer merging. For the capacity-based LiveIndex, one solution would be to define a lower-bound for the capacity of a buffer. When the capacity of a buffer drops below e.g., 50% of C_L, we mark the buffer and merge it with either its predecessor or its follower (if one of them is also marked), and then update accordingly the on-top structure. Finally, similar to the duration-based LiveIndex, a new on-top structure is again required, as the num/C_L division no longer works. This new structure will hold the lowest num inside a buffer, and will be binary searched by CloseInterval.

5.2 The DeadIndex Component

We now turn our focus on indexing dead record versions. Recall that these versions were evicted from the LiveIndex $\mathcal{I}_L$ by the CloseInterval function, after their *end* was read from the input stream. The DeadIndex $\mathcal{I}_D$ offers two key operations. Specifically, (1) $\mathcal{I}_D$ is updated to index a new dead record version (Function InsertDeadInterval) and (2) it evaluates time-travel queries (Function QueryDead). As the timeline evolves and new dead versions are added to $\mathcal{I}_D$, its domain grows. Under this, a straightforward solution for indexing dead record versions is the 2D point transformation approach from [41] as discussed in Section 2.1, where a 2D spatial index such as the R-tree, can adapt to the growing domain.

An alternative solution is to modify the state-of-the-art interval index HINT [12, 13] to adapt to a growing domain. Section 4.2 already discusses this for *te*-HINT. Implementing domain extension for a HINT DeadIndex is simpler, because we do not have to deal with transfers of live intervals between buckets as in *te*-HINT. Instead, we only have to add one more level and double the horizon t_H, as soon as we cannot accommodate a newly inserted interval s having at least one of its endpoints

[2]The number of buffers to be merged can be seen as a tunable system parameter.



(a) before level drop (b) level m drop (c) propagation and final HINT

Fig. 6. Steps of dropping last level m of HINT ($m = 2$)

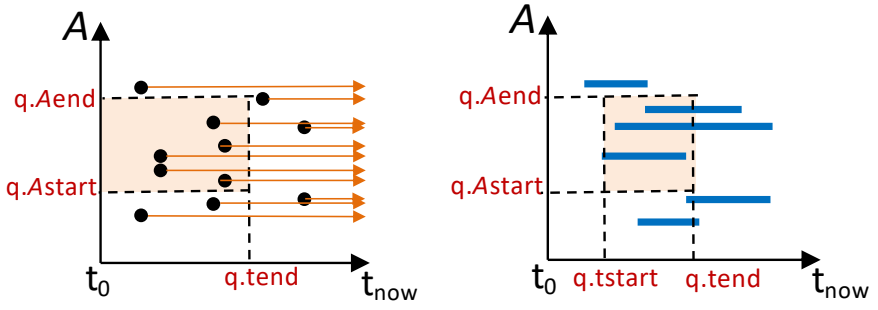
after t_H. As in *te*-HINT, after the expansion operation, the existing partitions are renamed to reflect their new level, but their contents remain intact.

Increasing the number of levels in a HINT that implements $\mathcal{I}_D$ to a very large number may negatively affect its search performance and size, as there could be far too many partitions for the number of indexed intervals [12]. A naïve approach to reduce the number of HINT levels by one is to construct a new HINT with one less level and insert all intervals in it. We propose a more efficient algorithm for deleting the lowest level of HINT, which progressively moves intervals from the deleted level to an appropriate partition above, while maintaining the HINT property (i.e., each interval s should be assigned at the smallest set of partitions from all level that define s). Each interval at level m (to be deleted) is stored in at most two level-m partitions. Intervals that begin and end in exactly one partition $P_{m,i}$ are directly moved to $P_{m-1,i\div 2}$ and no further action is needed. This is the case of s_2 in Figure 6(a) which is moved to $P_{1,0}$ in Figure 6(b). Intervals that begin in a $P_{m,i}$, for an odd i, are *temporarily moved* to $P_{m-1,i\div 2}$; the same holds for intervals that end in a $P_{m,i}$, for an even i. For instance, s_3 in Figure 6(a) is temporarily moved to partition $P_{1,1}$ because it ends in $P_{2,2}$, while s_4 is temporarily moved to both $P_{1,0}$ and $P_{1,1}$ (see Figure 6(b)). Temporary partitions $P_{m-1,j}$ at each level $\ell < m$ for an even j are set-intersected with the next partition at the same level holding replicas, at the potential of moving intervals to the previous level $\ell - 1$ as finalized or temporary. Symmetrically, temporary partitions $P_{\ell,j}$ at level ℓ for an odd j are set-intersected with the previous partition $P_{\ell,j-1}$. While there are temporary partitions at each level, intervals may propagate upwards until their correct partition is found. For instance, intervals s_3 and s_4, which, after the deletion of level 2, were stored in (temporary) partitions $P_{1,0}$ and $P_{1,1}$ at level 1 are eventually propagated at $P_{0,0}$ of level 0, as shown in the final HINT at Figure 6(c). A pseudocode of the drop level algorithm is skipped due to space constraints. Note that the same method can be used to delete the last level of *te*-HINT.

6 INDEXING RECORD ATTRIBUTES

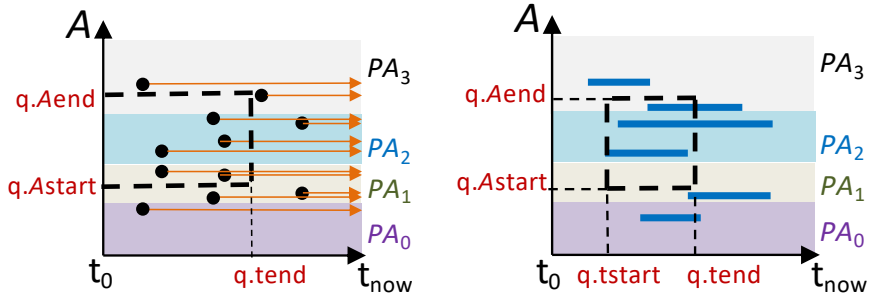
We now discuss how to modify LIT and index record versions on a specific attribute A for range time-travel queries, where not only a timepoint/range is specified but also a selection predicate on A. We denote a LIT that indexes an attribute A (besides time) by a-LIT.

Before describing a-LIT we discuss the requirements of a LiveIndex and a DeadIndex in the presence of the attribute A. Figure 7 illustrates the information that should be stored about live and dead record versions. As shown in Figure 7(a), to be able to answer range time-travel queries against LiveIndex, we need for each live version its *start* point and its A-value. So, the live version is a 2D point in the time-A space. A range time-travel query can then be modeled as a rectangular range $\{[t_0, q.end], [q.Astart, q.Aend]\}$ in the time-A space. Regarding the DeadIndex, we need for each dead version its *start*, *end* and its A-value. Figure 7(b) illustrates some dead versions in the time-A space and a range time-travel query, which is modeled as a 2D rectangle, defined by the query bounds.



(a) Live space (b) Dead space

Fig. 7. Live and Dead space and queries



(a) Live space A-partitioning (b) Dead space A-partitioning

Fig. 8. Live and Dead space A-partitioning

6.1 The LiveIndex Component

The LiveIndex of a-LIT should index the *start* timepoints of the current record versions and their values on A simultaneously.

2D space index. A natural approach to do so would be to use a native index for 2D points (e.g., kd-tree, quadtree, R-tree). Besides the 2D-space index, we also need an auxiliary structure $\mathcal{H}_{r.id \rightarrow (start,A)}$ that maps record *ids* to the *start* points of their live versions and their A values. Otherwise, it would not be possible to find and remove an indexed point from the 2D index, when the corresponding version dies (i.e., CloseInterval). Hence, the OpenInterval operation inserts the $(start = t_{now}, A)$ entry of a new live version to both the 2D index and $\mathcal{H}_{r.id \rightarrow (start,A)}$. Operation CloseInterval uses $\mathcal{H}_{r.id \rightarrow (start,A)}$ to find the coordinates of the ending version in the 2D index, searches and removes it, and relays the dead record version to DeadIndex. Finally, QueryLive issues a 2D query to the 2D index to retrieve the qualifying live versions.

Use multiple pure time indices. Another indexing approach is to divide the domain of A into partitions (e.g., equi-width) and develop a LiveIndex as described in Section 5.1 for each partition. The data structures and temporal partitioning methods are defined separately for each partition. The only difference is that the mapping mechanism $\mathcal{H}_{r.id \rightarrow num}$ of record *ids* to *num* values should also capture the A-partition identifier wherein a live version is located. By this, CloseInterval can identify and delete a live version from the correct A-partition of the LiveIndex. Figure 8(a) illustrates an A-partitioning of the live data space into four divisions (PA_0 to PA_3). For each of them, we can define a pure temporal LiveIndex, as described in Section 5.1. Given a range time-travel query, we use the selection predicate on A to identify the partitions that overlap with the query range in the A-domain (i.e., PA_1, PA_2, and PA_3 in Fig. 8(a)). If a partition is entirely covered by the A-range of

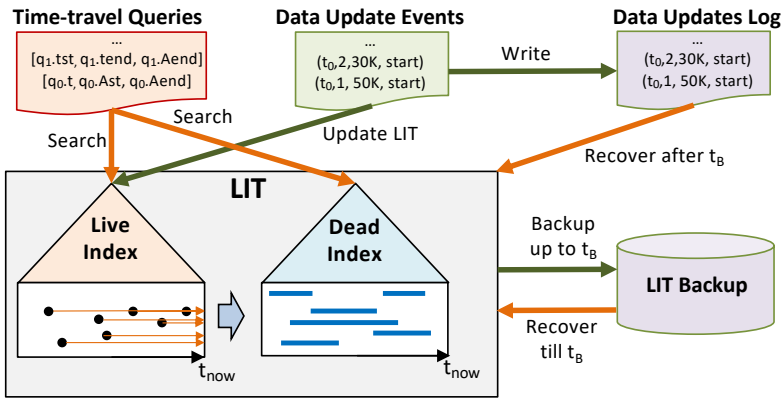


Fig. 9. Persistence and recovery of LIT

the query (e.g., partition PA_2), we evaluate the temporal part of the query, as described in Sec. 5.1. Otherwise (e.g., in PA_1 and PA_3), for each result obtained by the LiveIndex of the partition, we verify the A-predicate of the query. This verification is applied for at most two A-partitions containing the query boundaries. Updates on this A-partitioning approach are expected to be faster than updates on a 2D index, due to the fast hashing mechanisms it incorporates.

6.2 The DeadIndex Component

Now we turn to DeadIndex options for a-LIT. Like before, we can follow either a pure geometric approach or apply an A-partitioning technique to take advantage of the efficiency of pure time indices.

3D index. A straightforward approach is to index the line segments of the dead space (see Fig. 7(b)) directly by a native 2D index for geometric objects (e.g., an R-tree). However, such a method is not expected to perform well because some record versions in temporal databases are *long-lived* and correspond to very long segments that require large node MBRs, rendering the index inefficient. A more effective approach is to model each dead version as a 3D point $(s.start, s.end, r.A)$ in the (time, time, A) space, and index these points using a 3D index (e.g., a 3D R-tree). Figure 2(a) shows how this can be done for pure time intervals; the idea is to add one more dimension for A. Every query in this 3D space is then modeled as a $([0, q.tend], [q.tstart, t_{now}], [q.Astart, q.Aend])$ 3D box.

Use multiple pure time indices. Similar to the case of LiveIndex, we may also partition the domain of A to define a number of partitions, as shown in Figure 8(b). For each partition (e.g., PA_0 to PA_3), we use an optimized interval index, such as the modified HINT to support domain extension, discussed in Section 5.2. Given a range time-travel query, we first identify the A-partitions that overlap with the query A-range (e.g., PA_1, PA_2, PA_3) and then evaluate a pure time-travel query in each such partition, verifying the A-predicate against its results if necessary (e.g., in PA_1 and PA_3).

7 PERSISTENCE, RECOVERY, AND CONCURRENCY CONTROL

LIT is a main-memory index that facilitates real-time analytics, high-performance querying, and handling large volumes of rapidly changing temporal data. However, since main memory is volatile, we should ensure durability and recoverability, after power or system failures. Figure 9 illustrates how LIT is integrated into a temporal database system, to support fault tolerance and recovery. For this, each update event is written to a log file. In addition, a backup of LIT is taken periodically and written to the hard disk for persistence and faster recovery. The backup is merely a dump of

Table 1. Characteristics of tested datasets

		TAXIS-F	TAXIS-P	BIKES	FLIGHTS	WILDFIRES	BOOKS
	Cardinality	169290307	169290307	101472950	61328124	778410	2050707
	Domain extent	1 year	1 year	8 years	10 years	24 years	1 year
	Size [MBs]	5498	5498	3247	1963	26	66
temporal	**Min duration**	1 min	1 min	1 min	5 min	1 min	1 hour
temporal	**Max duration**	5 hours	5 hours	7.5 months	12 hours	4 months	1 year
temporal	**Avg. duration**	12 mins	12 mins	16 mins	2.5 hours	28 hours	67 days
search-key	**Description**	trip fare [USD]	passengers count	rider's birth year	departure delay [secs]	fire extent [acres]	num of books lent
search-key	**Type**	real	integer	integer	real	real	integer
search-key	**Value range**	[2.5, 235.5]	[1, 6]	[1940, 2005]	[0, 233400]	[0.0001, 606945]	[1, 38]
search-key	**Distribution**	normal	zipfian	normal	zipfian	zipfian	zipfian

Table 2. Query extents; default values in bold

input stream	**query extent**	
	temporal	**search-key**
TAXIS-F	1, 6, **12**, 18, 24 [hours]	3, 5, **10**, 30, 50 [dollars]
TAXIS-P	1, 6, **12**, 18, 24 [hours]	1, 2, **3**, 4, 5 [passengers]
BIKES	1, 6, **12**, 18, 24 [hours]	10, 20, **30**, 40, 50 [years]
FLIGHTS	1, 2, **3**, 4, 5 [days]	5, 10, **30**, 60, 120 [mins]
WILDFIRES	1, 7, **14**, 21, 30 [days]	10, 50, **100**, 500, 1000 [acres]
BOOKS	1, 7, **14**, 21, 30 [days]	5, 10, **15**, 20, 25 [books]

the main memory data structures for LiveIndex and DeadIndex. Assuming that the last checkpoint where the last backup has been taken is t_B, to recover LIT at a time $t_{now} > t_B$ (e.g., due to a power failure at that time), we first load the backups of LiveIndex and DeadIndex in main memory, then find the first event after t_B in the log file, and finally ingest all events after t_B to evolve LiveIndex and DeadIndex to their current state at t_{now}. Since all states up to t_B are captured by the LIT backup, we can even "cleanup" the log file by removing all entries up to t_B, to avoid searching it.

As in SAP HANA [24], concurrency control is operated by the transaction manager of the DBMS, which manages the *current* database state, and is independent to our proposed index. LIT ingests committed updates by the transaction manager, as shown in the Events sequence table of Figure 1. As the time-travel queries refer to the past, they do not conflict with insertions, which always happen at t_{now}. So, a newly inserted item cannot be a query result. Time-travel queries do not conflict with deletions/modifications, since only the *start* timepoint of a currently deleted item determines whether it is the result of a concurrent query. Regardless whether the item is in the LiveIndex (before the deletion) or in the LiveIndex (after the deletion) it will be reported as a query result if its *start* is before the *end* timepoint of the query. However, when a query is evaluated after the deletion of an item from LiveIndex and before the insertion of that item to DeadIndex, we may get incorrect query results. To ensure correctness, the migration of an item from LiveIndex to DeadIndex is done serially (i.e., it is not interleaved with concurrent queries). The total cost of a migration is extremely low (around 150 nanoseconds, as shown in our experiments), so the serial migration requirement does not affect the performance.

8 EXPERIMENTAL ANALYSIS

We last present our experiments. All indices were implemented in C++[3], compiled with gcc (v9.4.0) and -O3, -mavx, -march=native.[4] The tests ran on an AMD Ryzen 9 3950X, at 3.5GHz with 1MB L1 Cache, 8MB L2 Cache, 64MB L3 Cache, running Ubuntu Linux.

[3]Code available in https://github.com/GiorgosChristodoulou/LIT.
[4]Timeline was re-implemented according to the details provided in [24].

8.1 Setup

Datasets. We experimented with six real temporal datasets with an additional search-key A; Table 1 summarizes their characteristics. TAXIS-F(-P) contain the pick-up and drop-off timepoints of taxi trips (same intervals in both datasets) in NYC from 2009.[5] In TAXIS-F, A is the paid fare, and in TAXIS-P A is the number of passengers. BIKES contains the pick-up and drop-off timepoints of bike rides in NYC from 2014 to 2021; the search-key A is the birth year of the rider.[6] FLIGHTS contains the take-off and landing timepoints of flights recorded by the US Transportation Department from 2013 to 2022, and the occurred departure delay.[7] WILDFIRES specifies when fire events from 1992 to 2015 in US, were discovered and when declared contained/controlled.[8] As search-key A, we use an estimate of the area burnt. BOOKS contains the periods of time when books were lent out by Aarhus libraries in 2013, and the number of books during each period.[9] BOOKS, WILDFIRES include objects with long validity intervals, while in TAXIS, BIKES intervals are extremely short; FLIGHTS lies in the middle of the spectrum. As search-key, we consider both real and integer values; A's domain varies from extremely small (TAXIS-P) to extremely large (WILDFIRES). Last, the values of A follow either a normal or a Zipfian distribution.

Input streams. We created an event stream (workload) for every dataset, by splitting each interval to an insert and a deletion event, and interleaving 10K queries. Queries are positioned uniformly inside the active timeline, i.e., the period between the *start* of the very first interval until current t_{now}. The nature of the created streams varies from extremely update-heavy for TAXIS, BIKES and FLIGHTS with a 34000/1, 20000/1 and 13000/1 ratio of updates over queries, respectively, to moderate for BOOKS and WILDFIRES, with a 410/1 and 156/1 ratio, respectively. We considered two types of query extents; for pure time-travel queries, the extent of the $[q.tstart, q.tend]$ interval while for range time-travel queries, additionally the extent of the $[q.Astart, q.Aend]$ range. Table 2 lists the values for the query extents; the defaults are in bold. In each test, we measure the update time (for some indices, broken down to insert and delete time) and the query time.

8.2 Pure time-travel Queries

We start our evaluation with pure time-travel queries (Query 1). As we ignore the search-key A, we consider a single TAXIS stream.

8.2.1 Tuning LIT. We first investigate the best setting for the LiveIndex and the DeadIndex of LIT.

LiveIndex: data structure. We implemented the alternative structures from Section 5.1.1; STL C++ vector class was used for the *append-only array*, STL C++ ordered_map class (Red-Black tree) for the *search tree* and the Gapless hashmap from [35] for the *enhanced hashmap*.[10] Table 3 summarizes the results of our tests; for the interest of space, we report only TAXIS and BOOKS, which contain long and short intervals, respectively. The tests back up our intuition from Section 5.1.1. The append-only array exhibits the best (lowest) update times due to its simplicity. The enhanced hashmap however is always competitive, even for the update-heavy stream of TAXIS. The search tree on the other hand is outperformed by an order of magnitude for both inserts and deletions. Regarding queries, the enhanced hashmap is the most robust structure; the efficiency of the other two is affected by the nature of the input stream and/or the length of the intervals. Update-heavy

[5] https://www1.nyc.gov/site/tlc/index.page
[6] https://citibikenyc.com/system-data
[7] https://www.bts.gov
[8] https://www.kaggle.com/datasets/rtatman/188-million-us-wildfires
[9] https://www.odaa.dk
[10] Source code was provided by the authors.

Table 3. LiveIndex for LIT; time in secs, default query extents

TAXIS

q extent	Append-only array				Search tree				Enhanced hashmap			
[hours]	insert	delete	query	*total*	insert	delete	query	*total*	insert	delete	query	*total*
1	4.61	5.31	409	418.9	18.6	29.3	0.001	47.90	5.10	7.32	0.011	12.43
6	4.61	5.31	410	419.9	18.6	29.3	0.001	47.90	5.10	7.32	0.011	12.43
12	4.61	5.31	409	419.1	18.6	29.3	0.001	47.90	5.10	7.32	0.011	12.43
18	4.61	5.31	411	420.9	18.6	29.3	0.001	47.90	5.10	7.32	0.011	12.43
24	4.61	5.31	412	421.9	18.6	29.3	0.001	47.90	5.10	7.32	0.011	12.43

BOOKS

q extent	Append-only array				Search tree				Enhanced hashmap			
[hours]	insert	delete	query	*total*	insert	delete	query	*total*	insert	delete	query	*total*
1	0.057	0.068	14.4	14.52	0.336	0.967	38.0	39.30	0.065	0.142	6.41	6.61
7	0.057	0.068	14.8	14.92	0.336	0.967	37.9	39.20	0.065	0.142	6.45	6.65
14	0.057	0.068	14.9	14.93	0.336	0.967	39.7	41.0	0.065	0.142	6.46	6.66
21	0.057	0.068	15.2	15.32	0.336	0.967	41.9	43.20	0.065	0.142	6.47	6.66
30	0.057	0.068	15.6	15.72	0.336	0.967	42.8	44.10	0.065	0.142	6.43	6.63

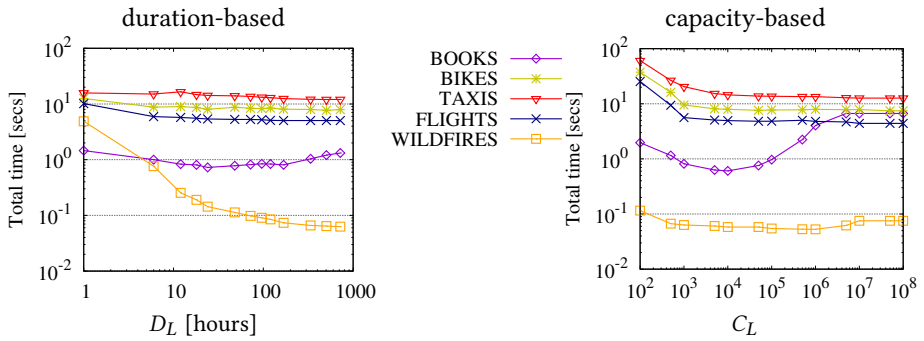


Fig. 10. LiveIndex for LIT tuning; default query extent

Table 4. LiveIndex for LIT; in msecs; default extents

input	duration-based				capacity-based			
stream	insert	delete	query	*total*	insert	delete	query	*total*
TAXIS	4653	7462	4	12121	5252	7418	11.4	12681
BIKES	5667	2358	5	8030	3305	4140	4.3	7449
FLIGHTS	3059	2018	4	5083	1742	2653	11.1	4405
WILDFIRES	27	33	3	63	23	27	2.8	52.8
BOOKS	83	270	352	706	82.6	204	319	606

streams will incur a large number of tombstones and significantly slow down the append-only array, while long-lived intervals increase the size of LiveIndex and slow down the search tree. Overall, the enhanced hashmap offers the best trade-off between updates and queries, exhibiting always the lowest total time. For the rest of our experiments, we use the enhanced hashmap to store the LiveIndex.

LiveIndex: partitioning. We implemented both partitioning approaches from Section 5.1.2. To determine the best value for the duration constraint D_L and the capacity constraint C_L, we conducted the experiments in Figure 10 where the total time (update plus query time) is reported, while varying

Table 5. DeadIndex for LIT; times in secs

TAXIS

q. extent [hours]	2D R-tree [41]			HINT		
	insert	query	*total*	insert	query	*total*
1	69.7	3.21	72.9	8.43	0.28	8.71
6	69.7	15.5	85.2	8.43	1.54	9.97
12	69.7	29.8	99.5	8.43	2.96	11.4
18	69.7	44.3	114	8.43	3.39	11.8
24	69.7	59.2	128	8.43	6.20	14.6

BOOKS

q. extent [days]	2D R-tree [41]			HINT		
	insert	query	*total*	insert	query	*total*
1	0.63	45.9	46.5	0.15	0.27	0.42
7	0.63	47.8	48.6	0.15	1.05	1.20
14	0.63	51.2	51.8	0.15	1.86	2.01
21	0.63	55.2	55.7	0.15	1.74	1.89
30	0.63	59.1	59.7	0.15	2.96	3.11

Table 6. Pure time-travel queries: total update time [secs]

input stream	Timeline	*te*-HINT	LIT		
			LiveIndex	DeadIndex	total
TAXIS	12.3	1886	14.5	8.43	22.89
BIKES	10.4	357	7.93	5.13	13.06
FLIGHTS	4.08	526	4.68	3.01	7.69
WILDFIRES	0.05	0.38	0.07	0.04	0.11
BOOKS	0.19	349	0.49	0.14	0.63

Table 7. Pure time-travel queries: in secs; default extents

Component	**input stream**				
	TAXIS	BIKES	FLIGHTS	WILDFIRES	BOOKS
LIT: LiveIndex	0.157	0.005	0.011	0.001	0.371
LIT: DeadIndex	2.96	0.203	0.504	0.019	1.85

D_L and C_L. Note that as the value of both constraints increases, the number of LiveIndex buffers always drops. With the best observed values for each input stream in place, we compare the two approaches in Table 4 for the default query extents, which also includes a runtime breakdown for each approach. We observe that the capacity-based partitioning always outperforms the duration-based by 10%, on average. For the rest of our analysis, the LiveIndex of LIT will use the capacity-based partitioning; also, based on Figure 10's experiment, we set $C_L = 10000$ for all streams.

DeadIndex. We compare HINT in the role of DeadIndex as discussed in Section 5.2, against the 2D transformation approach proposed in [41], powered by a 2D R-tree from the highly optimized Boost.Geometry library.[11] Table 5 reports the insert time and the query time for each DeadIndex approach, while varying the query extent. Due to lack of space, we show again only the numbers for TAXIS and BOOKS. HINT outperforms the 2D R-tree on computing pure time-travel queries by at least one order of magnitude (usually two orders), while for ingesting dead records, the 2D R-tree is competitive only in case of BOOKS, which contains significantly fewer updates than TAXIS. In contrast, for the update-heavy TAXIS, the 2D R-tree is an order of magnitude slower than HINT for indexing new dead records. In view of the above, LIT will use HINT as its DeadIndex component for the rest of our analysis.

8.2.2 LIT against the competition. We now compare the LIT hybrid index against *te*-HINT (Section 4) and the state-of-the-art Timeline index [24] for transactional DBs. Figure 11 (first row) reports the total time (updates and queries) for each index to ingest the input streams, while varying the query extent. Our tests clearly show that LIT is the most efficient index for all input streams, followed in almost all cases by the Timeline index, while *te*-HINT ranks last, with the exception of WILDFIRES. To better understand these results, the second row of the figure reports

[11] Benchmark in [27] showed that Boost.Geometry (https://www.boost.org) R-tree implementations outperform the libspatialindex library (https://libspatialindex.org/).

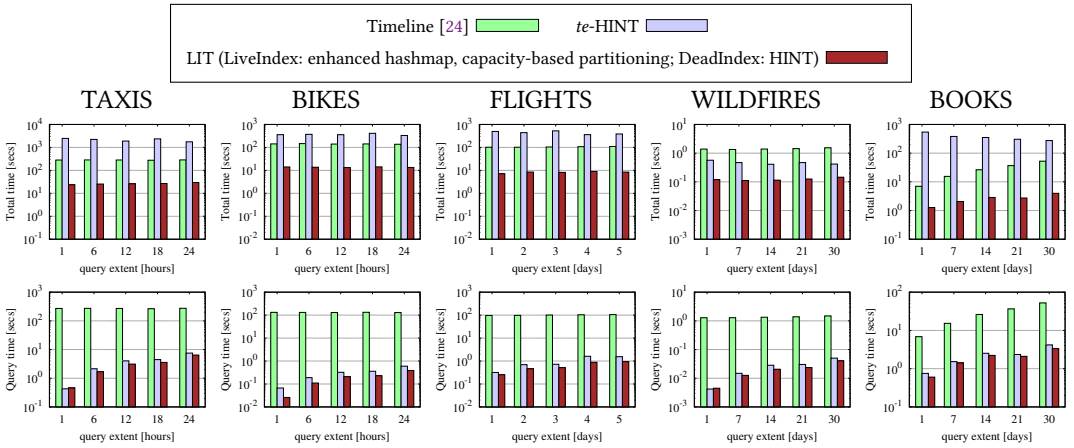


Fig. 11. Pure time-travel queries

the accumulated time over the 10K queries of the stream and Table 6 reports the accumulated update time. The query costs of LIT and *te*-HINT are always lower compared to those of Timeline; *te*-HINT is competitive to LIT but in all cases slower. For updates, Table 6 shows the advantage of Timeline; recall from Section 2 that Timeline is designed for the support of fast updates in transaction-time DBs. Nevertheless, LIT is competitive to Timeline. Also, observe that the total updating cost is almost equally divided in between the LiveIndex and the DeadIndex. In contrast, *te*-HINT is orders of magnitude slower than LIT and Timeline in updates, mainly due to the high cost of moving intervals between partitions at different levels, as the timeline evolves and deletion events arrive. Overall, LIT offers the best tradeoff between updates and queries, resulting in the lowest total time, even for update-heavy streams such as TAXIS and BIKES. Lastly, we provide a breakdown to the query time of LIT in Table 7.

8.3 Range time-travel Queries

We next switch gears and evaluate range time-travel queries (Query 2), which include selections on the search-key A. For a-LIT, we considered an equi-width partitioning of the A domain in 6-7 partitions.[12]

8.3.1 Tuning a-LIT. We first investigate the best setup for a-LIT.

LiveIndex. We implemented the two alternative solutions for the LiveIndex discussed in Section 6.1; a Boost 2D R-tree which directly indexes the $start$-A 2D space and a series of pure time indices (using enhanced hashmap and capacity-based partitioning), one for each partition of the A domain. For completeness, we also include the approach of a single pure time index (again with enhanced hashmap and capacity-based partitioning); this captures the case of an extremely skewed distribution of A-values, where the vast majority of the objects fall inside one A-partition. Table 8 reports the total insert, delete times, and the total query time for each solution while varying the search-key query extent. Due to lack of space, we only report for the TAXIS-F and BOOKS streams. The performance of the 2D R-tree LiveIndex is severely affected by the cost of updates, especially by deletions, rendering this solution impractical.[13] Even a single pure time index is still a better option

[12]Our tests (not included due to lack of space) showed that this number of partitions is sufficient to provide good total times in all tested streams.

[13]This is expected, as R-trees typically suffer from high maintenance costs.

Table 8. LiveIndex for a-LIT; time in secs, default query extents

TAXIS-F

search-key query extent [dollars]	2D R-tree				single pure time index				multiple pure time indices			
	insert	delete	query	*total*	insert	delete	query	*total*	insert	delete	query	*total*
3	57.8	1105.5	0.002	1163.3	6.7	9.5	0.02	16.2	8.4	11.3	0.002	19.7
5	57.8	1105.5	0.002	1163.3	6.7	9.5	0.02	16.2	8.4	11.3	0.002	19.7
10	57.8	1105.5	0.002	1163.3	6.7	9.5	0.02	16.2	8.4	11.3	0.002	19.7
30	57.8	1105.5	0.002	1163.3	6.7	9.5	0.02	16.2	8.4	11.3	0.002	19.7
50	57.8	1105.5	0.002	1163.3	6.7	9.5	0.02	16.2	8.4	11.3	0.002	19.7

BOOKS

search-key query extent [books]	2D R-tree				single pure time index				multiple pure time indices			
	insert	delete	query	*total*	insert	delete	query	*total*	insert	delete	query	*total*
5	0.9	1621.4	1.6	1623.9	0.1	0.3	0.4	0.8	0.1	0.4	0.04	0.54
10	0.9	1621.4	1.9	1624.2	0.1	0.3	0.4	0.8	0.1	0.4	0.04	0.54
15	0.9	1621.4	2.2	1624.5	0.1	0.3	0.4	0.8	0.1	0.4	0.04	0.54
20	0.9	1621.4	3.2	1625.5	0.1	0.3	0.4	0.8	0.1	0.4	0.04	0.54
25	0.9	1621.4	4.5	1626.8	0.1	0.3	0.4	0.8	0.1	0.4	0.04	0.54

for the LiveIndex than a 2D R-tree which indexes both time and A dimensions. Finally, regarding the comparison between the single and the multiple time indices solutions, we observe an expected tradeoff. The single time index solution is faster for updates, especially in update-heavy streams like TAXIS-F, while using multiple indices has an order of magnitude lower time on queries. As the decrease in the total time from using a multiple time indices LiveIndex for query-intensive streams (BOOKS) is larger than the increase of the total time on update-heavy streams (TAXIS-F), in the rest of our analysis, a-LIT will use the multiple time indices solution, i.e., maintaining a LiveIndex for each partition of the search-key A domain.

DeadIndex. We implemented the two options discussed in Section 6.2; a 3D R-tree which directly indexes both the validity interval of a dead version and its search-key A, and a series of pure time indices powered by HINT, one for each partition of the A-domain. For completeness, we also include the case when a single HINT is used as the DeadIndex, which again captures the case of an extremely skewed data distribution, where the vast majority of the objects are indexed by a single HINT. Table 9 reports the total update (insert) and query time for each approach, while varying the search-key query extent; again, due to lack of space, we only report on the TAXIS-F and BOOKS streams. The table clearly shows the advantage of the multiple pure time indices option in the role of the DeadIndex for a-LIT. The 3D R-tree DeadIndex is always slower both for updating (insertions of dead record versions) and querying, while using a single pure time index is only competitive for updating. In the rest of our analysis, a-LIT will maintain a HINT powered DeadIndex for each partition of the search-key A domain.

8.3.2 a-LIT against competition. We compare a-LIT against two competitors. The first is a *time-first* baseline, which directly employs the pure LIT and does not index the search-key attribute A. Pure LIT employs the same setup considered for pure time-travel queries comparison in Section 8.2.2, i.e., an enhanced hashmap with capacity-based partitioning as the LiveIndex and HINT as the DeadIndex. To answer a range time-travel query q, this (pure) LIT first executes a pure time-travel query with $[q.tstart, q.tend]$ and then, checks the attribute A of every intermediate result against the $[q.Astart, q.Aend]$ range. The second competitor is the state-of-the-art index for multi-versioned DBs, MVB-tree [1]. The first and the third rows in Figure 12 report the total time of the indices,

Table 9. DeadIndex for a-LIT; time in secs, default query extents

TAXIS-F

search-key	**3D R-tree [41]**			**HINT**			**multiple HINTs**		
query extent [dollars]	insert	query	*total*	inset	query	*total*	insert	query	*total*
3	81.9	40.6	123	9.49	4.11	13.6	9.48	0.49	9.97
5	81.9	40.5	122	9.49	4.12	13.61	9.48	0.51	9.99
10	81.9	40.6	123	9.49	4.11	13.6	9.48	0.40	9.88
30	81.9	40.6	123	9.49	4.12	13.61	9.48	0.41	9.89
50	81.9	40.5	122	9.49	4.11	13.6	9.48	0.41	0.89

BOOKS

search-key	**3D R-tree [41]**			**HINT**			**multiple HINTs**		
query extent [books]	insert	query	*total*	insert	query	*total*	insert	query	*total*
5	0.74	4.80	5.52	0.15	0.85	1.0	0.15	0.26	0.41
10	0.74	5.35	6.07	0.1 5	1.75	1.9	0.15	0.25	0.40
15	0.74	7.86	8.60	0.15	2.53	2.68	0.15	0.28	0.43
20	0.74	9.14	9.88	0.15	2.63	2.78	0.15	0.27	0.42
25	0.74	11.6	12.3	0.15	4.14	4.14	0.15	0.27	0.42

Table 10. Range time-travel queries: total update time [secs]

input stream	MVB-tree [1]	LIT (pure)	a-LIT
TAXIS-F(-P)	341	27.9	29.3
BIKES	57.8	15.7	16.5
FLIGHTS	61.6	8.76	9.89
WILDFIRES	0.28	0.12	0.14
BOOKS	1.86	0.85	0.87

while varying the *A*-range of the query and the temporal query extent, respectively. Observe that both LIT-based indices outperform the MVB-tree, in all tests. The reason is the high cost of update handling by the MVB-tree; the performance gap is larger for the TAXIS and BIKES (update-heavy streams). As Table 10 shows, LIT (pure) and a-LIT capitalize on the LiveIndex to cope with updates. In fact, the MVB-tree is competitive only in BOOKS, which has the smallest number of updates and so, queries significantly contribute to the total time. a-LIT always outperforms LIT (pure) as expected for range time-travel queries (second and fourth row in Figure 12), since LIT (pure) cannot prune the search space using the search-key attribute. Overall, a-LIT exhibits a good tradeoff between updating and querying, being able to efficiently handle both update-heavy and moderate streams. Based on our tests, we expect an even bigger advantage over LIT (pure) for query-heavy streams.

8.4 Index Size

We conclude our analysis with a study on the index size. First, we compare LIT and a-LIT against the competition; Tables 11 and 12 report the maximum size for each index for pure time-travel and range time-travel queries, respectively. For all indices, this maximum value is observed after the entire input stream was ingested. In Table 11, observe that for all streams LIT occupies less space than the Timeline index. On the other hand, *te*-HINT has an identical maximum footprint to LIT because both approaches eventually build identical HINT indices. As Table 12 shows, a-LIT always occupies less space than the MVB-tree. Compared to a-LIT, (pure) LIT has a slightly smaller footprint due to building a single HINT, but at the expense of an inferior performance, in almost

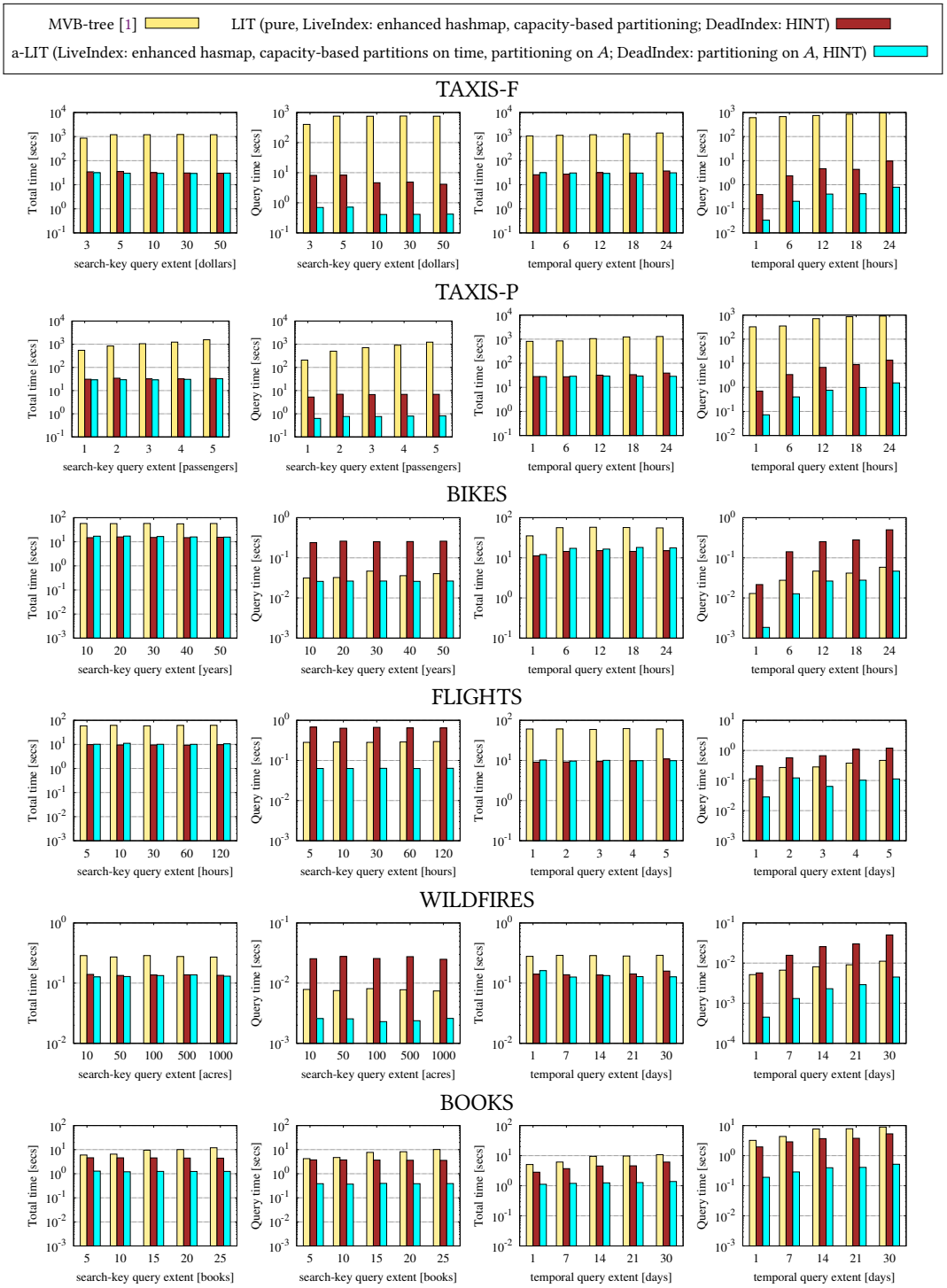


Fig. 12. Range time-travel queries

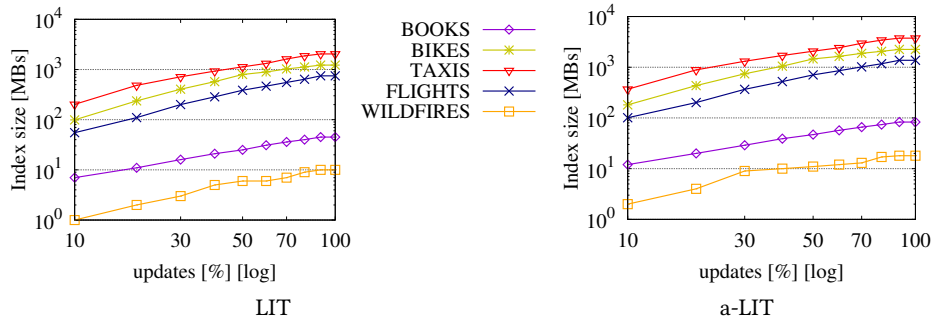


Fig. 13. Size growth over time

Table 11. Pure time-travel queries: index size [MBs]

input stream	Timeline [24]	*te*-HINT	LIT
TAXIS	3086	2042	2042
BIKES	1851	1226	1226
FLIGHTS	1129	747	747
WILDFIRES	15	10	10
BOOKS	69	45	45

Table 12. Range time-travel queries: index size [MBs]

input stream	MVB-tree [1]	LIT (pure)	a-LIT
TAXIS-F(-P)	8522	3404	3744
BIKES	5433	2043	2247
FLIGHTS	4739	1246	1370
WILDFIRES	35	16	18
BOOKS	282	75	83

all cases as shown in Figure 12. Finally, we study the growth of the LIT's size of time; Figure 13 plots its size as a function of the percentage of the updates in each stream. Observe that LIT's space increases linearly with the number of updates, which makes it appropriate for in-memory management of time-evolving data.

9 CONCLUSIONS AND FUTURE WORK

We proposed LIT, a hybrid index for time-evolving databases, which decouples the handling of current (live) record versions from the management of past (dead) record versions. We studied options for implementing the live and dead index components, focusing on minimizing the cost of index updates and queries. We considered pure time-travel queries that retrieve active record versions at some time point or period in the past, and range time-travel queries, which additionally apply a selection predicate on a search-key attribute. Our tests unveil the best approaches for handling live and dead record versions in LIT and shows that LIT is orders of magnitude faster than temporal indices that index live and dead versions in the same structure. LIT uses linear space to the number of record versions, which renders it suitable for in-memory indexing of temporal data. In the future, we will study the applicability and effectiveness of LIT for other query types (e.g., temporal aggregation, temporal joins) and its integration into an open-source database system. We will also investigate the (partial) storage of dead records on disk-based data structures, to support the indexing of big temporal data.

ACKNOWLEDGMENTS

Partially funded by the Hellenic Foundation for Research and Innovation (HFRI) under the "2nd Call for HFRI Research Projects to support Faculty Members & Researchers" (Project No. 2757)

REFERENCES

[1] 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

[2] 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

[3] 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

[4] Luigi Bellomarini, Markus Nissl, and Emanuel Sallinger. 2022. iTemporal: An Extensible Generator of Temporal Benchmarks. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2021–2033. https://doi.org/10.1109/ICDE53745.2022.00197

[5] Arthur Bernhardt, Sajjad Tamimi, Tobias Vinçon, Christian Knödler, Florian Stock, Carsten Heinz, Andreas Koch, and Ilia Petrov. 2022. neoDBMS: In-situ Snapshots for Multi-Version DBMS on Native Computational Storage. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 3170–3173. https://doi.org/10.1109/ICDE53745.2022.00290

[6] 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

[7] 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

[8] Leon Bornemann, Tobias Bleifuß, Dmitri V. Kalashnikov, Fatemeh Nargesian, Felix Naumann, and Divesh Srivastava. 2023. Matching Roles from Temporal Data: Why Joe Biden is not only President, but also Commander-in-Chief. *Proc. ACM Manag. Data* 1, 1 (2023), 65:1–65:26. https://doi.org/10.1145/3588919

[9] Panagiotis Bouros and Nikos Mamoulis. 2017. A Forward Scan based Plane Sweep Algorithm for Parallel Interval Joins. *Proc. VLDB Endow.* 10, 11 (2017), 1346–1357. https://doi.org/10.14778/3137628.3137644

[10] Panagiotis Bouros, Nikos Mamoulis, Dimitrios Tsitsigkos, and Manolis Terrovitis. 2021. In-Memory Interval Joins. *VLDB J.* 30, 4 (2021), 667–691. https://doi.org/10.1007/S00778-020-00639-0

[11] Felix S. Campbell, Bahareh Sadat Arab, and Boris Glavic. 2022. Efficient Answering of Historical What-if Queries. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 1556–1569. https://doi.org/10.1145/3514221.3526138

[12] 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

[13] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2023. HINT: A Hierarchical Interval Index for Allen Relationships. *VLDB J.* (2023). https://doi.org/10.1007/s00778-023-00798-w

[14] Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. 2008. *Computational geometry: algorithms and applications, 3rd Edition*. Springer.

[15] Anton Dignös, Boris Glavic, Xing Niu, Johann Gamper, and Michael H. Böhlen. 2019. Snapshot Semantics for Temporal Multiset Relations. *Proc. VLDB Endow.* 12, 6 (2019), 639–652. https://doi.org/10.14778/3311880.3311882

[16] 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

[17] Herbert Edelsbrunner. 1980. *Dynamic Rectangle Intersection Searching*. Technical Report 47. Institute for Information Processing, TU Graz, Austria.

[18] 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

[19] Volker Gaede and Oliver Günther. 1998. Multidimensional Access Methods. *ACM Comput. Surv.* 30, 2 (1998), 170–231. https://doi.org/10.1145/280277.280279

[20] 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

[21] Junyang Gao, Stavros Sintos, Pankaj K. Agarwal, and Jun Yang. 2021. Durable Top-K Instant-Stamped Temporal Records with User-Specified Scoring Functions. In *37th IEEE International Conference on Data Engineering, ICDE 2021,*

Chania, Greece, April 19-22, 2021. IEEE, 720–731. https://doi.org/10.1109/ICDE51399.2021.00068

[22] 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

[23] 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

[24] 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

[25] Nick Kline and Richard T. Snodgrass. 1995. Computing Temporal Aggregates. In *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*. IEEE Computer Society, 222–231. https://doi.org/10.1109/ICDE.1995.380389

[26] 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

[27] Mateusz Loskot and Adam Wulkiewicz. 2019. https://github.com/mloskot/spatial_index_benchmark.

[28] Wei Lu, Zhanhao Zhao, Xiaoyu Wang, Haixiang Li, Zhenmiao Zhang, Zhiyu Shui, Sheng Ye, Anqun Pan, and Xiaoyong Du. 2019. A Lightweight and Efficient Temporal Database Management System in TDSQL. *Proc. VLDB Endow.* 12, 12 (2019), 2035–2046. https://doi.org/10.14778/3352063.3352122

[29] Achilleas Michalopoulos, Dimitrios Tsitsigkos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2023. Efficient Nearest Neighbor Queries on Non-point Data. In *Proceedings of the 31st ACM International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2023, Hamburg, Germany, November 13-16, 2023*. ACM, 33:1–33:4. https://doi.org/10.1145/3589132.3625609

[30] Bongki Moon, Inés Fernando Vega López, and Vijaykumar Immanuel. 2003. Efficient Algorithms for Large-Scale Temporal Aggregation. *IEEE Trans. Knowl. Data Eng.* 15, 3 (2003), 744–759. https://doi.org/10.1109/TKDE.2003.1198403

[31] Mirella M. Moro and Vassilis J. Tsotras. 2018. Transaction-Time Indexing. In *Encyclopedia of Database Systems, Second Edition*. Springer. https://doi.org/10.1007/978-1-4614-8265-9_399

[32] Mirella M. Moro and Vassilis J. Tsotras. 2018. Valid-Time Indexing. In *Encyclopedia of Database Systems, Second Edition*. Springer. https://doi.org/10.1007/978-1-4614-8265-9_1513

[33] Katerina Papaioannou, Martin Theobald, and Michael H. Böhlen. 2019. Outer and Anti Joins in Temporal-Probabilistic Databases. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 1742–1745. https://doi.org/10.1109/ICDE.2019.00187

[34] Danila Piatov and Sven Helmer. 2017. Sweeping-Based Temporal Aggregation. In *Advances in Spatial and Temporal Databases - 15th International Symposium, SSTD 2017, Arlington, VA, USA, August 21-23, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10411)*. Springer, 125–144. https://doi.org/10.1007/978-3-319-64367-0_7

[35] Danila Piatov, Sven Helmer, and Anton Dignös. 2016. An interval join optimized for modern hardware. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. IEEE Computer Society, 1098–1109. https://doi.org/10.1109/ICDE.2016.7498316

[36] Danila Piatov, Sven Helmer, Anton Dignös, and Fabio Persia. 2021. Cache-efficient sweeping-based interval joins for extended Allen relation predicates. *VLDB J.* 30, 3 (2021), 379–402. https://doi.org/10.1007/S00778-020-00650-5

[37] 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

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

[39] Yufei Tao, Dimitris Papadias, and Christos Faloutsos. 2004. Approximate Temporal Aggregation. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*. IEEE Computer Society, 190–201. https://doi.org/10.1109/ICDE.2004.1319996

[40] Dimitrios Tsitsigkos, Konstantinos Lampropoulos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2021. A Two-layer Partitioning for Non-point Spatial Data. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 1787–1798. https://doi.org/10.1109/ICDE51399.2021.00157

[41] Leong Hou U, Nikos Mamoulis, Klaus Berberich, and Srikanta J. Bedathur. 2010. Durable top-k search in document archives. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM, 555–566. https://doi.org/10.1145/1807167.1807228

[42] Donghui Zhang, Alexander Markowetz, Vassilis J. Tsotras, Dimitrios Gunopulos, and Bernhard Seeger. 2001. Efficient Computation of Temporal Aggregates with Range Predicates. In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*. ACM. https:

//doi.org/10.1145/375551.375600

[43] Zihao Zhang, Huiqi Hu, Zhihui Xue, Changcheng Chen, Yang Yu, Cuiyun Fu, Xuan Zhou, and Feifei Li. 2021. SLIM-STORE: A Cloud-based Deduplication System for Multi-version Backups. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 1841–1846. https://doi.org/10.1109/ICDE51399.2021.00164

Received July 2023; revised October 2023; accepted November 2023