The VLDB Journal manuscript No.
(will be inserted by the editor)

Scalable Lighting-fast Temporal Indexing

Panagiotis Simatis · George Christodoulou · Panagiotis Bouros · Nikos Mamoulis

Abstract We study the problem of temporal database indexing, i.e., indexing versions of a database table in an evolving database. Although modern machines include large memory chips, data volumes quickly exceed resources, making it infeasible to keep the entire history in memory. Therefore we require temporal indices that optimize main memory usage while remaining scalable as the history grows. We depart from the classic indexing approach, where all data versions are indexed in a single data structure, and propose LIT, a hybrid index that decouples the management of the current and past states of the indexed column. LIT includes optimized indexing modules for current (i.e., live) and past (i.e., dead) records, supporting efficient queries and updates. Furthermore, our extended approach LIT$^+$handles record versions in memory using LIT bounded by a memory budget, while managing older versions (fossils) that exceed the budget on disk. We show that LIT outperforms state-of-the-art solutions by orders of magnitude while using space linearly proportional to the number of indexed record versions, making it suitable for main-memory temporal data management. In addition, we also show that LIT$^+$ efficiently indexes long database histories on disk while maintaining scalability and query performance.

Keywords Temporal data · Query processing · Indexing

Panagiotis Simatis
Department of Computer Science and Engineering, University of Ioannina, Greece
E-mail: p.simatis@uoi.gr

George Christodoulou
Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Netherlands
E-mail: g.c.christodoulou@tudelft.nl

Panagiotis Bouros
Institute of Computer Science, Johannes Gutenberg University Mainz, Germany
E-mail: bouros@uni-mainz.de

Nikos Mamoulis
Department of Computer Science and Engineering, University of Ioannina, Greece
E-mail: nikos@cse.uoi.gr

1 Introduction

Temporal data management has been studied extensively for at least four decades [10, 11, 24, 30, 51]. 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 regained interest recently [7, 9, 12, 15, 20, 25, 28, 38, 44, 58], due to the increase of cheap storage that makes it often possible to track the versions of a database 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, records are inserted, deleted, or existing attribute values are updated. Figure 1 shows some versions of T, where, at time t_0, T is initialized to include two 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. T evolves as update events arrive; the stream (time-sequence) of update events is also shown in the figure (bottom-left). Insertions (deletions) are modeled by events of type *start* (*end*); each update (i.e., value change) is modeled by a deletion immediately followed by an insertion. Finally, the figure (bottom-right) shows the *validity* intervals of the records and their val-

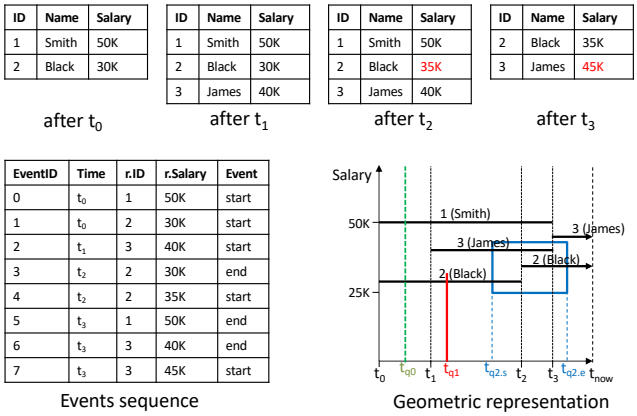


Fig. 1 Example of a time-evolving table

ues in the Salary attribute, as flat line segments. The current time is denoted by t_{now}.

We study the problem of indexing an evolving 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), 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 at time t_{q1} and retrieves the record versions whose timespan (horizontal line segment) is crossed by the vertical line segment 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 horizontal 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). Time-travel queries are included in SQL extensions [27, 34] and implemented in PostgreSQL[1], Oracle Workspace Manager, IBM DB2 [50], Microsoft SQL Server[2], Teradata [1], and MariaDB[3].

Previously proposed temporal indices can be classified to (1) methods for transaction-time and multi-versioned databases (e.g., MVB-tree [4], Timeline index [30]), (2) data structures for (time) intervals [6, 8, 17, 22, 33]. Our work belongs to the first category, where the goal is to support the aforementioned queries, but also real-time version updates, 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.

Contributions. 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 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, developed 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 [4, 30] on mixed workloads of queries and version updates, showing that LIT is orders of magnitude faster.

LIT was originally presented and evaluated in the preliminary version of this paper [19]. Although LIT is appropriate for indexing the brief or recent version history of a data table, eventually the available memory might be exhausted, as the indexing requirements grow with the number of updates. We propose an ex-

[1] https://wiki.postgresql.org/wiki/Temporal-Extensions

[2] https://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables

[3] http://mariadb.com/kb/en/system-versioned-tables/

tension to LIT, denoted by LIT$^+$, which periodically offloads old versions to the disk. LIT$^+$ differentiates between two types of dead records; those that died before a designated timestamp t_f, which are called *fossils*, and those that died after t_f. Fossils are stored in a disk-resident component called FossilIndex, freeing main memory for LiveIndex and DeadIndex to receive new entries. Queries that access timestamps before t_f probe all LiveIndex, DeadIndex, and FossilIndex components, while those referring to recent timestamps are handled in-memory by LiveIndex and DeadIndex. Thus, queries over the recent past are processed efficiently in memory, while queries for older versions are still supported. We show how the three components are updated as data evolves; LiveIndex and DeadIndex by individual updates at current time t_{now}, while FossilIndex, by periodic updates, when the available memory is exhausted. We investigate different options for maintaining LIT$^+$. Our study shows that the most effective strategy combines a partition-aware deletion, tailored to our best implementation of DeadIndex, and aggregating fossils before inserting them into FossilIndex.

Outline. Section 2 reviews related work. In Section 3, we define time-travel queries and the data whereon they apply. Section 4 proposes an extension to the state-of-the-art interval index [17, 18] to manage live and dead record versions in an ever-growing time domain. In Section 5, we present our novel hybrid index LIT for pure time-travel queries. Section 6 extends LIT to index an attribute A of the records besides their temporal validity intervals, for range time-travel queries. Section 7 extends LIT to incorporate a disk-resident component for records that died before a designated timestamp. In Section 8, we discuss topics related to recovery and consistency. Finally, Section 9 presents our experimental analysis, while Section 10 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

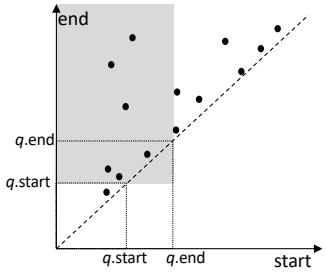
Valid-time temporal databases store record versions which are valid during a well-defined time interval [42]. This interval could refer to the past, the future, or may start at some time in the past and finish in the future (e.g., an activated credit card which expires in the future). The order by which 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 time-travel queries can then be seen as a case of indexing intervals (i.e., one-dimensional ranges), which is a well-studied problem [6, 8, 17, 22, 33]. Classic data structures for intervals include the segment tree [8] and the interval tree [22]. 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 query result size. Their space complexity is $O(n \log n)$ and $O(n)$, respectively. The interval tree also supports *range* queries, i.e., find intervals that overlap with a query interval (value range) in $O(\log n + K)$. Disk-based extensions were presented in [2, 33].

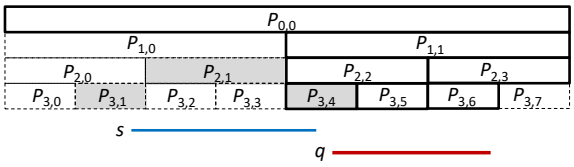
Data structures for multi-dimensional boxes, such as the R-tree [5, 26], can also be used for intervals, which can be considered as 1D boxes. For example, a simple and dynamic data structure for intervals is the 1D-grid, which divides the space into a number of disjoint partitions. Each interval is assigned 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 after dividing the data in each partition to classes based on whether they begin inside or before the partition [17, 39, 54]. Indices which consider both the values and the durations of the intervals are the period index [6] and the RD-Index [16]. These are self-adaptive structures which split the domain into coarse partitions, and then further divide each partition hierarchically to organize the contained intervals based on their positions and durations.

An alternative approach is to map intervals to 2D points [3, 29, 38, 49, 55, 56]. 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 intervals as points in this 2D space. Since $s.start < s.end$ for each interval s, all points lie 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). To index the 2D points, we can directly use a spatial data structure [29, 38, 55]. As a different option, SEB-tree [52] employs a collection of B$^+$-trees.

HINT [17] 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



(a) 2D mapping [3, 29, 38, 49, 55, 56]



(b) HINT [17, 18]

Fig. 2 Interval indices

$m = 3$. 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 [17]. Thus, most query results are reported without performing comparisons, giving HINT a performance advantage over other methods.

Deficiencies of interval indices. While HINT [17] is the best performing index, it shares a weakness of most other interval indexing methods: the domain of the interval endpoints must be known apriori. If the data domain grows (i.e., as in a temporal database), the partitions may have to be updated to cover the new part of the domain, and interval assignments to partitions might change to preserve index properties. On the other hand, the 2D point transformation approach [55] avoids this issue as a 2D spatial index (e.g., R-tree) can adapt to a growing domain. Still, the query regions cover a relatively large part of the mostly sparsely populated 2D space, so this approach is not as efficient as HINT. 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 in the Introduction.

2.2 Indexing Data Versions

Transaction-time databases [41] 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. In contrast, *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.

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 Multiversion B-tree (MVB-tree) [4], which succinctly captures the values of the indexed attribute in all versions of records. For a comprehensive survey of early indexing methods for time-evolving databases see [49]. 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 Section 2.1.

A more recent index for transaction-time DBs is the *Timeline* index [30], which builds upon the Time index [23] and supports very fast updates. In a nutshell, the Timeline index is an Events Sequence Table (see Figure 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. Our approach is a major departure from previous work which indexes dead and live versions in the *same* data structure. Instead, we define separate data structures for live and dead versions; in principle, versions that die are transferred from the first data structure to the second. By decoupling indexing for live and dead versions, we can optimize both data structures. In Section 7, we show how our proposal can be extended to handle dead versions on the disk.

2.3 Other related work

Recent work in temporal databases studies the efficient evaluation of other queries, besides time-travel selections. *Temporal aggregation* [31, 40, 45, 53, 57] 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 [25, 55] are a special case of temporal aggregation. A *temporal join* [13, 14, 28, 44, 47] 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 [15]. Other recent related work includes the definition of new temporal semantics [20], system optimizations in the implementation of temporal and multi-version databases [9, 38, 58], temporal database benchmarking [7], and novel temporal integrity constraints [12].

Inspired by [32], learned multi-dimensional indices [21, 43, 48] are proposed as an alternative to traditional indexing structures. These methods leverage machine learning to capture the data distribution, aiming to build efficient and compact indices. According to [36], some learned indices outperform traditional ones, especially for datasets with uniform distributions, and achieve significant space reductions. However, their construction cost is substantially higher than that of traditional structures, and provide limited support for dynamic operations.

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 [49].

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.*

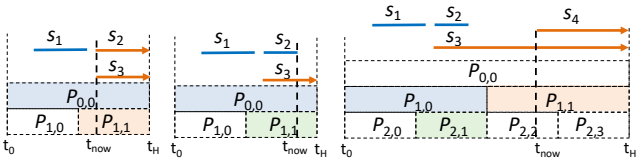
Without loss of generality, we assume query intervals closed at both ends. In addition, for each change in T an update event is generated, which may trigger updates in the indices of T. These update events include: (1) the insertion of a record to T, (2) the deletion of a record from T, and (3) the change 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). Last, we assume a single non-temporal attribute A in T for Query 2; in Section 6.3, we discuss how to handle multiple A attributes.

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 $r.start = t$.

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 exhibit poor search times

compared to interval indexing (Section 2.1). In Section 4, we extent 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).



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

Fig. 3 Example of *te*-HINT

4 Time-evolving HINT

A first attempt to define an efficient in-memory index for time-evolving tables is to convert HINT [17, 18], 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 changes require structural modifications and new update operations compared to HINT [17], 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. On an insertion event, 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 [17], 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. We also insert an entry $\langle r.id, s.start\rangle$ in an auxiliary key-value data structure $\mathcal{H}_{r.id \to start}$ that facilitates finding a live interval in *te*-HINT given the corresponding record id. Figure 3(a) shows a 2-level *te*-HINTexample, 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 $P_{1,1}$ (sub-partition $P^{O_L}_{1,1}$).

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 $\mathcal{H}_{r.id \to start}$ 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 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 $\mathcal{H}_{r.id \to start}$, the partitions ($P^{O_L}_{1,1}$) 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 $P^{O_D}_{1,1}$).

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. 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 reduce the replication of live intervals and facilitate 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 $P_{0,0}$ becomes $P_{1,0}$, etc.). Existing live interval s_3 is added to the new $P^{R_L}_{1,1}$. The new interval s_4 is inserted 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 [17, 18]. 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.

5.1 Overview

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, 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

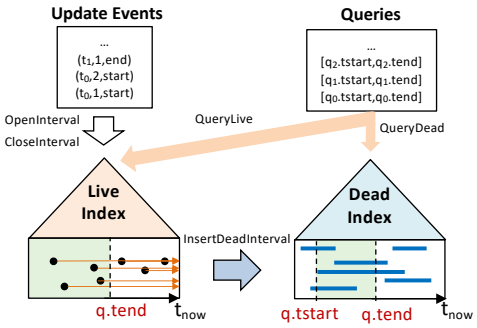


Fig. 4 Overview of LIT

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. Record updates terminate (i.e., "delete") the current (live) version of the record and insert 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 LIT components $\mathcal{I}_L$ and $\mathcal{I}_D$, and describe their key operations.

5.2 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*.

5.2.1 Data structures

We discuss three alternative structures for LiveIndex, aimed at both fast updates and efficient time-travel queries. We experimentally compare them in Section 9.2.1.

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 after accessing the first record that started after *q.tend*. As already mentioned, updates cost $O(1)$ time; however, a query costs $O(n)$, assuming n updates so far, since for each array position p up to *q.tend*, we must check if p is a tombstone.

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. To answer a $[q.tstart, q.tend]$ query, we scan and report entries from the first tree leaf until we find the first entry that has its *start* after *q.tend*. Using this data structure both searches and updates cost $O(\log n)$, where n is the number of updates so far.

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 consider a data structure, which exhibits competitive update time to the array and has lower query time. To this end, we suggest using an *enhanced* hash table, similar to the Gapless hashmap proposed in [46] 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. Hence, updates cost $O(1)$ time as in the sorted array. 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. Hence, queries cost $O(|\mathcal{I}_L|)$, where $|\mathcal{I}_L|$ denotes the number of live entries. This is lower than the $O(n)$ query cost of the sorted array, but still quite significant. In the next section, we suggest partitioning techniques that reduce $\mathcal{I}_L$'s search cost.

5.2.2 Temporal partitioning of LiveIndex

Given a query, a LiveIndex implemented by any of the data structures in Section 5.2.1 would need to conduct comparisons for a large number of live versions (independently of the underlying 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, more importantly, comparisons are conducted only in one 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-width 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) 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

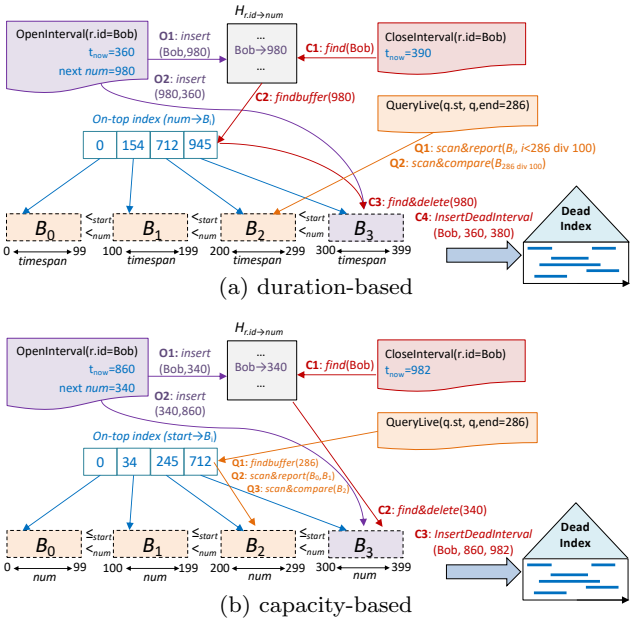


Fig. 5 LiveIndex: partitioning

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) presents 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 imbalanced buffers with respect to the number of contained entries, rendering imbalanced query costs. An alternative approach that results in balanced partitions is to use a capacity constraint C_L, allowing each buffer to hold at most C_L entries. [4] Unlike duration-based partitioning, 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 num/C_L division determines which buffer B contains the recently deceased version. Note that if the last buffer is already full, OpenInterval creates a new buffer B_{new} after the last one and appends the new live version in B_{new}.

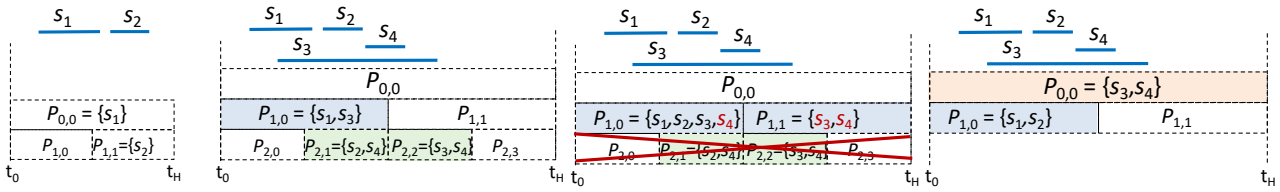
In contrast, it is no longer possible to directly determine buffer B_{end} for a $[q.tstart, q.tend]$ query. In view of this, we define an on-top structure, which stores a $\langle st, ptr \rangle$ entry for each buffer B of the LiveIndex, where st is the lowest $start$ timepoint of a record version 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 query $[q.tstart, q.tend]$, 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 as for 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.

5.2.3 Optimizations

As the timeline evolves and live records die, buffers may become under-utilized or empty. To deal with this, reorganization can be employed for both types of partitioning. For the duration-based LiveIndex, sparsity is expected to occur in the first (early) buffers. So, we merge adjacent sparse buffers and accordingly update the on-top structure.[5] To answer time-travel queries, an 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 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.

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

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



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

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

5.3 The DeadIndex Component

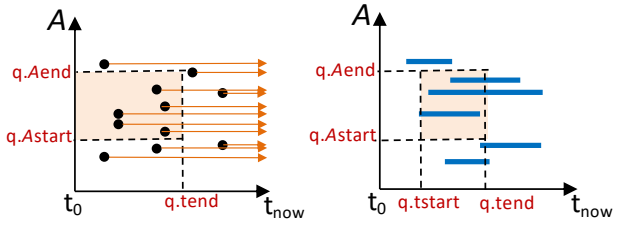
We now turn our focus on indexing dead record versions. Recall that these versions were evicted from the LiveIndex $\mathcal{I}_L$ by the CloseInterval function, after their *end* was read from the input stream. The DeadIndex $\mathcal{I}_D$ offers two key operations. Specifically, (1) $\mathcal{I}_D$ 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 enter $\mathcal{I}_D$, its domain grows. Under this, a straightforward solution for indexing dead record versions is the 2D point transformation approach from [55] as discussed in Section 2.1, where a spatial index (e.g., R-tree), adapts to the growing domain.

An alternative solution is to modify the state-of-the-art interval index HINT [17, 18] 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 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 [17]. 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(b) which is moved to $P_{1,0}$ in Figure 6(c). 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(b) 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(c)). 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(d). 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.

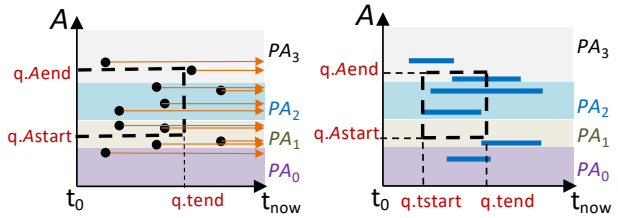
5.4 Complexity

This section briefly analyzes the complexity of LIT. Assuming that we have consumed n update events, LiveIndex occupies space linear to the live records, or $O(n)$ space overall. The space complexity of HINT as a DeadIndex is $O(m \cdot n)$ [17], where m is the number of levels. Regarding time complexity: (1) the cost of consuming a new record event is $O(1)$, if a chain of enhanced hashmaps is used as a LiveIndex; (2) consuming record deletions costs $O(1)$ at LiveIndex and $O(m)$ at DeadIndex [17]. Hence, the cost per update is $O(m)$. This analysis does not consider the cost of restructuring operations (merging live buffers, increasing the number of HINT levels), which are rare and their amortized costs are smaller than $O(m)$. Regarding search, each query costs for the buffer which includes $q.tend$, $O(|I_L|)$ comparisons when duration-based partitioning is used or $O(C_L)$, in case of capacity-based, plus the cost of searching a constant number (4 by expectation [17]) of



(a) Live space (b) Dead space

Fig. 7 Live and Dead space and queries



(a) Live space (b) Dead space

Fig. 8 Live and Dead space A-partitioning

HINT partitions, which are also not expected to hold more than $O(C_L)$ intervals.

6 Indexing Non-Temporal 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 $\{[t_0, q.tend], [q.Astart, q.Aend]\}$ 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.

6.1 The LiveIndex Component

a-LIT's LiveIndex must index current record versions *start* timepoints and their A values simultaneously.

2D space index. A natural approach to do so would be to use a native index for 2D points (e.g., kd-tree, quad-tree, 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 5.2 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 5.2. 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 Figure 8(a)). If a partition is entirely covered by the A-range of the query (e.g., partition PA_2), we evaluate the temporal part of the query, as described in Section 5.2. 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. 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. As before, we can follow either a pure geometric approach or apply an A-partitioning 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 Figure 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 since some record versions in temporal databases are *long-lived*, corresponding to very long

segments that require large node minimum bounding rectangles, 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. Each 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 5.3. 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).

6.3 Handling Multiple Non-temporal Attributes

In case of multiple non-temporal attributes $A_1, \ldots A_m$, both a-LIT index components can be extended to support range time-travel queries with a range predicate to each A_i. Specifically, we can still utilize a single multi-dimensional index for the $(m+1)$-dimensional space defined by time and the A attributes or the approach of multiple pure time indices, using a multi-dimensional grid over the joint attribute domain.

7 Temporal Indexing under Limited Memory

We now shift focus to scenarios where the available memory for temporal indexing is bounded. To handle such cases, we present LIT$^+$, which extends the LIT framework by incorporating a new disk-resident component. For this purpose, we introduce an expiration threshold, termed the *fossilization timestamp* t_f, which defines the temporal boundary between memory-resident and disk-resident records.

7.1 Overview of LIT$^+$

Intuitively, at the core of the LIT$^+$ framework lies the distinction between two types of dead records; those that died before the t_f timestamp and those that died after. The former, called *fossils*, are stored inside the disk-resident component FossilIndex, whereas the latter, inside the DeadIndex, in main memory.

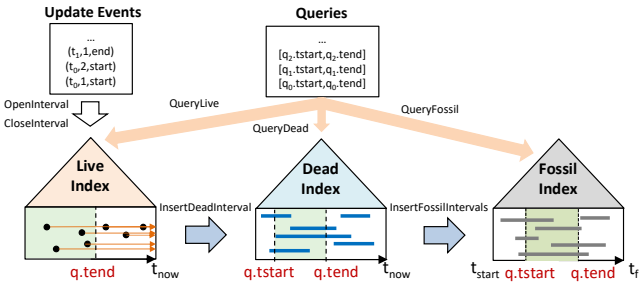


Fig. 9 Overview of LIT$^+$

Figure 9 provides an overview of LIT$^+$, which extends the original LIT framework in Figure 4. The stream of updates to the indexed table T is again consumed by the LiveIndex $\mathcal{I}_L$, which implements the OpenInterval and CloseInterval functions to handle the insert and deletion events, respectively. When a record dies, the corresponding entry is removed from $\mathcal{I}_L$ (Function CloseInterval) and an entry is created in the $\mathcal{I}_D$ DeadIndex (Function InsertDeadInterval). Both components reside in main memory. When the combined memory footprint of $\mathcal{I}_L$ and $\mathcal{I}_D$ exceeds a memory budget M[6], a system event termed *fossilization* is triggered. Fossilization frees space in main memory to accommodate future updates in $\mathcal{I}_L$ and $\mathcal{I}_D$. For this purpose, LIT$^+$ first updates the t_f timestamp by moving it forward in time. Afterwards, all dead records whose *end* timepoint is before the updated t_f are removed from $\mathcal{I}_D$ (Function DeleteFossils) and offloaded to the $\mathcal{I}_F$ FossilIndex on disk (Function InsertFossilIntervals).

To evaluate a pure time-travel query $q = [q.tstart, q.tend]$, we distinguish between two cases. If $q.tstart > t_f$, LIT$^+$ operates exactly as LIT, i.e., we probe only $\mathcal{I}_L$ and $\mathcal{I}_D$ as detailed in Section 5. Otherwise, if $q.tstart \leq t_f$ then we need to probe all three components. Similarly to $\mathcal{I}_D$, $\mathcal{I}_F$ evaluates a typical interval range query to determine all records whose validity interval overlaps q. As discussed in Section 5.1 for LIT, $\mathcal{I}_L$, $\mathcal{I}_D$ and $\mathcal{I}_F$ index disjoint sets of records and therefore, these probing tasks are completely independent to each other and duplicate results are never produced.

In what follows, we elaborate on how the t_f timestamp is updated, on the necessary changes for the DeadIndex and the implementation of FossilIndex; in contrast, LiveIndex remains unchanged.

7.2 Updating the Fossilization Timestamp t_f

A straightforward approach to update t_f is to move it forward to the end of the time domain covered by

[6] The M budget is a system parameter, pre-defined according to the available memory.

DeadIndex. Then, all dead records become fossils and the maximum possible amount of memory is emptied. However, such an extreme approach harms performance since a large number of queries will be evaluated by the FossilIndex, on disk. Therefore, the process of updating t_f encapsulates a typical space-time trade-off. To control this trade-off, we introduce an additional system parameter r, so that $r \cdot M$ specifies the maximum amount of memory we can empty during fossilization; recall that M is the memory budget set by the system.

With parameter r in place, we can set t_f to the time point incurring the highest decrease in the memory footprint of the DeadIndex $\mathcal{I}_D$ that does not exceed the $r \cdot M$ threshold. We can compute this decrease by scanning $\mathcal{I}_D$ and counting the total number of entries (both originals and replicas) for the dead records whose interval ends before the new t_f, i.e., those that will become fossils (Function CountFossilEntries). However, to avoid exhaustively seeking this optimal new t_f inside DeadIndex's entire domain, we choose a value best fit for HINT, which as we show in Section 9.2.1, is the most efficient structure for indexing dead intervals.

Specifically, we first restrict the value of the new t_f among only the *end* of the bottom HINT partitions in $\mathcal{I}_D$. This way we both significantly reduce the search space for t_f and we simplify the scanning process of $\mathcal{I}_D$ performed by the CountFossilEntries function. Second, we examine the candidate t_f values in a binary search fashion. We start off with the center of the time domain covered by $\mathcal{I}_D$ as the new t_f[7], and utilize CountFossilEntries to calculate the amount of main memory M_f to be emptied. If $M_f < r \cdot M$ holds, we need to advance t_f forward and recursively search the second half of the domain; otherwise, we binary search the first half. This process always terminates since there is a fixed number of partitions at the bottom HINT level. Lastly, in the extreme case where the specified new t_f value is equal to the old, we automatically set t_f to the *end* of the directly succeeding partition, emptying this way the minimum possible amount of space in main memory for the system to continue operating.

7.3 The DeadIndex Component

For LIT, DeadIndex operates in an insert-only mode; i.e., dead records are added to the index, but never removed. However, for LIT$^+$, the DeleteFossils function is invoked every time t_f is updated. Essentially, DeleteFossils implements the second step in the fossilization process where all fossil entries in DeadIndex are identified and then the index is accordingly updated.

[7] The center equals the *end* of the $P_{m,2^{m-1}-1}$ partition.

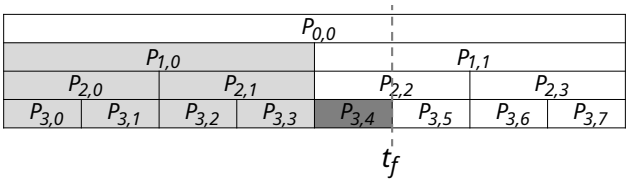


Fig. 10 Classifying HINT DeadIndex partitions to determine fossils

We first elaborate on how to spot the fossil records stored inside a HINT DeadIndex. By definition, fossils end before the t_f fossilization timestamp, which means they are stored as originals inside a partition before t_f. Figure 10 highlights in light and dark gray the partitions which store the fossils' original entries for a HINT DeadIndex with 4 levels; we will explain the difference between the two shades in the next paragraphs. In contrast, the records stored as originals inside a white partition are not fossils as they definitely end after t_f. Note this is true even for the partitions whose timespan contains t_f, i.e., $P_{2,2}$, $P_{1,1}$ and $P_{0,0}$ in Figure 10. By construction, a dead record whose validity interval starts for instance inside $P_{2,2}$, must end after t_f because otherwise the record would have been stored as an original in the level below, i.e., inside $P_{3,4}$.

We next present two approaches for excluding all fossil entries from a HINT DeadIndex $\mathcal{I}_D$[8]. Although we could discuss the maintenance process even for an unoptimized HINT, we assume that the subdivisions optimization from [17] is activated to accelerate the process.[9] In brief, according to this optimization, the originals in every P (see Section 2.1) are further divided into subdivisions $P^{O_{in}}$ and $P^{O_{aft}}$, so that $P^{O_{in}}$ ($P^{O_{aft}}$) holds the record intervals that end inside (resp. after) the partition. Similarly, the replicas inside P are divided into $P^{R_{in}}$ and $P^{R_{aft}}$. Intuitively, the straightforward approach for maintaining $\mathcal{I}_D$ first scans the index to determine all fossils and then drops $\mathcal{I}_D$ to rebuild it using only the remaining dead (non-fossil) records. Under this, it suffices to scan the $P^{O_{in}}$, $P^{R_{in}}$ subdivisions for every partition in $\mathcal{I}_D$. By definition, entries originating from partitions before t_f (light and dark gray, in Figure 10) belong to fossils; these records are collected and sent to FossilIndex (see Section 7.4). In contrast, entries from the remaining partitions (white, in Figure 10) belong to non-fossils which are inserted to the new DeadIndex. Notice how this maintenance process completely ignores $P^{O_{aft}}$, $P^{R_{aft}}$; their contained entries correspond to records that end inside a succeeding P' partition, stored in its $P'^{R_{in}}$ subdivision. Furthermore,

[8] We compare the two approaches in Section 9.3.1

[9] In practice, the subdivisions optimization is always activated as it also enhances query processing [17].

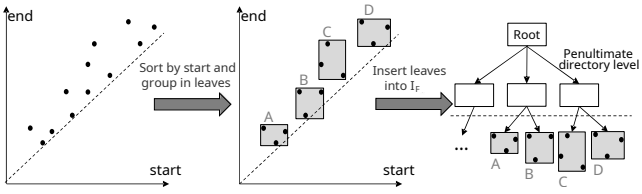


Fig. 11 Batch insertions in LIT$^+$'s R-tree FossilIndex

there is no need for de-duplication because every dead record is stored as a O_{in} or a R_{in}, exactly once.

If the t_f fossilization timestamp is always set at the *end* of a bottom level partition (as proposed in Section 7.2), the above approach requires no comparisons to determine fossils. However, we still need to rebuild the updated $\mathcal{I}_D$ index from scratch. In view of this shortcoming, we next present an alternative approach which updates the subdivisions of the HINT partitions in-situ by removing the fossil entries. Note that we cannot rely on the update process described for HINT in [17] which handles deletions by tombstones. In our setting, we must physically remove the entries to empty space in main memory. Instead, we distinguish between three types of partitions in the current $\mathcal{I}_D$ (colored light gray, dark gray, and white in Figure 10), and handle each type in a different fashion. Let P be a HINT partition, we consider the following cases:

- P **is light gray,** $P.end < t_f$. We directly empty the $P^{O_{in}}$, $P^{R_{in}}$ subdivisions as they store only records that end before t_f, i.e., fossils, by definition. In contrast, for $P^{O_{aft}}$, $P^{R_{aft}}$, fossils are determined by comparing their *end* against t_f.
- P **is dark gray,** $P.end = t_f$. Similar to the light gray case, we directly empty $P^{O_{in}}$, $P^{R_{in}}$, while $P^{O_{aft}}$, $P^{R_{aft}}$ remain intact; by definition, their entries belong to records that end after t_f.
- P **is white,** $P.end > t_f$. All subdivisions remain intact since they exclusively contain entries for records that end after t_f.

To reduce the number of comparisons needed to process $P^{O_{aft}}$, $P^{R_{aft}}$ in the light gray partitions, we rely on the implicit sorting of the subdivisions. By construction, every subdivision is sorted by record *end*, in increasing order. Hence, to clean $P^{O_{aft}}$, $P^{R_{aft}}$, we perform a binary search for the first entry with $end > t_f$ and then remove all entries preceding this pivot inside the subdivision. Lastly, all records inside the $P^{O_{in}}$, $P^{R_{in}}$ subdivisions of the light and dark gray partitions are collected and sent to FossilIndex, similar to the straightforward approach.

7.4 The FossilIndex Component

We next discuss LIT$^+$'s disk-resident component. FossilIndex $\mathcal{I}_F$ operates in an insert-only mode, incorporating the recently removed fossils from the DeadIndex $\mathcal{I}_D$ after the timestamp t_f is updated. Despite indexing dead records similar to $\mathcal{I}_D$, we cannot utilize HINT for $\mathcal{I}_F$ as it was designed for main memory. Instead, we employ the 2D mapping discussed in Section 2.1 under which every record is mapped to a $(start, end)$ point. We index these points on disk using an R-tree [26], which is the dominant structure for spatial data.

We next elaborate on the process of updating $\mathcal{I}_F$. When fossilization is triggered for the first time, we can efficiently index the incoming fossils in a bulk-loading fashion (e.g., with the STR algorithm [35]). For every follow-up event, a straightforward approach would be to directly use the R*-tree insertion algorithm from [5] individually for each recently defined fossil. Despite its simplicity this approach will incur frequent tree adjustments and a large number of I/Os. In view of this, we devise a novel approach which inserts intervals in batches making it suitable for large updates.

Figure 11 exemplifies the key steps in our batch insertion. First, the incoming fossil records are sorted by their *start* to improve spatial locality. Then, the sorted fossils are divided into batches, each sized to fit within an R-tree leaf node. For each batch, a new leaf node is created, which is then directly inserted into an internal node, right above the leaf level of the R-tree FossilIndex. Splits are triggered when necessary to maintain the tree balance. The I/O cost of fossilization is bounded by the number of leaf nodes inserted to the R-tree, since the non-leaves are expected to be too few and may fit in memory (especially, when we use a large node size, e.g. 8KBs). By the last assumption, the I/O cost for searching the FossilIndex is bounded by the number of R-tree leaves which are accessed by the query; this number is expected to be small, estimated as the number of query results divided by the node capacity.

7.5 a-LIT Compatibility

Finally, we discuss the necessary modifications in LIT$^+$ to index record versions on a specific non-temporal attribute A and evaluate range time-travel queries; we denote this extended framework by a-LIT$^+$. In principle, the key challenge we face is that the existence of attribute A increases the memory footprint of a-LIT$^+$ compared to LIT$^+$. Under the memory budget M, this will result in more frequent fossilization events.

Similar to LIT$^+$, a-LIT$^+$ draws a distinction between memory-resident and disk-resident records. For live records and in-memory dead records (i.e., non-fossils), we adopt the multiple indices design for a-LIT in Section 6. In brief, we partition the domain of A and

build a pair of LiveIndex and DeadIndex in each partition; our experiments in Section 9.2.4 show the advantage of this design. In contrast, for fossil records, we rely on a single FossilIndex $\mathcal{I}_F$ and one fossilization timestamp t_f. To incorporate attribute A, we now utilize a 3D R-tree which indexes the $(start, end, A)$ space.

The key difference for the fossilization process of a-LIT$^+$ lies in updating t_f. For this purpose, we prioritize the DeadIndex with the highest memory usage. This approach ensures efficient resource utilization and prevents t_f from advancing excessively. This is beneficial for two reasons; (1) it maintains a balanced distribution of intervals across the dead indices, and (2) it minimizes disk accesses by preventing t_f from becoming too recent, which would degrade query performance.

8 Persistence, Recovery, Concurrency Control

LIT is a main-memory index focused on real-time analytics and handling large volumes of rapidly changing temporal data. However, the volatility of main memory necessitates durability and recovery mechanisms following system failures. Figure 12 illustrates how LIT is integrated into a temporal database system, to support fault tolerance and recovery. Therefore, 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. The backup is merely a dump of 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 system failure at that time), we first load the backups of LiveIndex and DeadIndex in main memory, then replay events starting from 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 "cleanup" the log file by removing all entries up to t_B, for efficiency.

For LIT$^+$, note that the FossilIndex is already on disk and thus, no further action is required in case of a system failure, assuming that disk recovery is handled by another mechanism, such as RAID. However, to ensure atomicity and deal with a failure during fossilization, the process is implemented as a single system transaction guaranteeing that either all dead records to be fossilized are removed from DeadIndex and added to FossilIndex or none. Under this principle, a dead record can be found exclusively either in the DeadIndex or the FossilIndex, but never in both.

As in SAP HANA [30], concurrency control is operated by the transaction manager of the DBMS, which

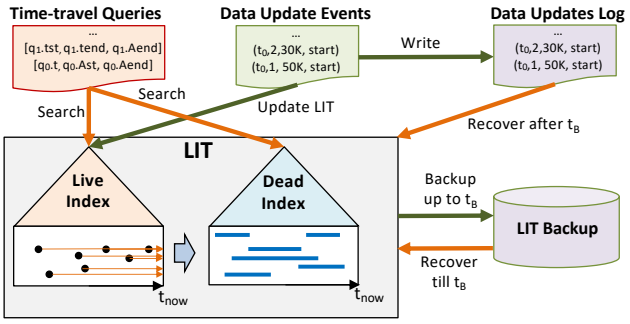


Fig. 12 Persistence and recovery of LIT

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 record cannot be a query result. Time-travel queries do not conflict with deletions/modifications, since only the *start* timepoint of a currently deleted record determines whether it is the result of a concurrent query. Regardless if 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 a record from LiveIndex and before the insertion of that record to DeadIndex, we may get incorrect query results. To ensure correctness, the migration from LiveIndex to DeadIndex is done serially (i.e., 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.

Although possible, parallel fossilization would introduce significant complexity in managing concurrent access to the DeadIndex and FossilIndex. To guarantee data integrity, LIT$^+$ serializes fossilization, granting exclusive control over the DeadIndex and FossilIndex. This design ensures correctness and eliminates race conditions during migration. Furthemore, since fossilization is infrequent, this prioritization has a negligible impact on system performance.

9 Experimental Analysis

This section reports our experiments, which compare (a-)LIT (Section 9.2) and (a-)LIT$^+$ (Section 9.3), against competition. All methods were written in C++ and compiled with gcc using -O3, -mavx, -march=native.

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 information						
Min duration	1 min	1 min	1 min	5 min	1 min	1 hour
Max duration	5 hours	5 hours	7.5 months	12 hours	4 months	1 year
Avg. duration	12 mins	12 mins	16 mins	2.5 hours	28 hours	67 days
Avg. duration [%]	0.0024	0.0024	0.0004	0.0028	0.0135	18.6
search-key A information						
Description	trip fare [USD]	passenger count	rider's birth year	departure delay [secs]	fire extent [acres]	num of fooks lent
Type	real	integer	integer	real	real	integer
Value range	[2.5, 235.5]	[1, 6]	[1940, 2005]	[0, 233400]	[0.0001, 606945]	[1, 38]
Distribution	normal	zipfian	normal	zipfian	zipfian	zipfian

Table 2 Query extents; default values in bold

Stream	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]

9.1 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.[10] 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.[11] 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.[12] WILDFIRES specifies when fire events from 1992 to 2015 in US, were discovered and when declared contained/controlled.[13] 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.[14] 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). 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.

9.2 In-Memory Query Processing

We start off with the in-memory evaluation of queries. Sections 9.2.1 and 9.2.2 study LIT and pure time-travel queries (Query 1). For this, we ignore the search-key A; hence, we use a single TAXIS stream. Sections 9.2.3 and 9.2.4 study a-LIT and range time-travel queries (Query 2) which include selections on the search-key A. For this purpose, we considered an equi-width partitioning of the A domain in 6-7 partitions.[15]

All tests we conducted as single-threaded processes on an AMD Ryzen 9 3950X, clocked at 3.5GHz with 64GBs of DRAM and 1MB L1 Cache, 8MB L2 Cache, 64MB L3 Cache, running Ubuntu Linux.[16]

[10] https://www1.nyc.gov/site/tlc/index.page
[11] https://citibikenyc.com/system-data
[12] https://www.bts.gov
[13] https://www.kaggle.com/datasets/rtatman/188-million-us-wildfires
[14] https://www.odaa.dk

[15] 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.
[16] Code: `https://github.com/GiorgosChristodoulou/LIT`

Table 3 LiveIndex for LIT; total update/query times [secs]

TAXIS

query extent	Append-only array		Search tree		Enhanced hashmap	
[hours]	update	query	update	query	update	query
1	9.92	409	47.9	0.001	12.42	0.011
6	9.92	410	47.9	0.001	12.42	0.011
12	9.92	409	47.9	0.001	12.42	0.011
18	9.92	411	47.9	0.001	12.42	0.011
24	9.92	412	47.9	0.001	12.42	0.011

BOOKS

query extent	Append-only array		Search tree		Enhanced hashmap	
[days]	update	query	update	query	update	query
1	0.125	14.4	1.30	38.0	0.207	6.41
7	0.125	14.8	1.30	37.9	0.207	6.45
14	0.125	14.9	1.30	39.7	0.207	6.46
21	0.125	15.2	1.30	41.9	0.207	6.47
30	0.125	15.6	1.30	42.8	0.207	6.43

9.2.1 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 5.2.1; 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 [46] for the *enhanced hashmap*.[17] 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 analysis from Section 5.2.1. 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 vastly outperformed in updates. 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 streams (TAXIS) will incur a large number of tombstones and significantly slow down the append-only array, while long-lived intervals (BOOKS) increase the size of LiveIndex and slow down the search tree. All data structures are robust to the query extent, which is expected, since $q.tstart$ is ignored in query processing. 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 5.2.2. To determine the best value for the duration constraint D_L and the capacity constraint C_L, we conducted the experiments in Figure 13 where the total time (update plus query time) is reported, while varying 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. Note 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 this experiment, we set $C_L = 10000$ for all streams.

[17] Source code was provided by the authors.

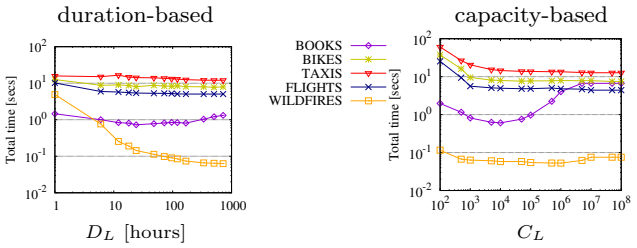


Fig. 13 LiveIndex for LIT partitioning; default query extent

Table 4 LiveIndex for LIT partitioning; update and query times [secs], default query extents

input stream	duration-based			capacity-based		
	insert	delete	query	insert	delete	query
TAXIS	4.65	7.46	0.004	5.25	7.42	0.011
BIKES	5.67	2.36	0.005	3.31	4.14	0.004
FLIGHTS	3.06	2.02	0.004	1.74	2.65	0.011
WILDFIRES	0.027	0.033	0.003	0.023	0.027	0.003
BOOKS	0.083	0.270	0.352	0.083	0.204	0.319

Table 5 DeadIndex for LIT; total update&query times [secs]

TAXIS

query extent	2D R-tree		HINT	
[hours]	insert	query	insert	query
1	69.7	3.21	47.9	0.001
6	69.7	15.5	47.9	0.001
12	69.7	29.8	47.9	0.001
18	69.7	44.3	47.9	0.001
24	69.7	59.2	47.9	0.001

BOOKS

query extent	2D R-tree		HINT	
[days]	insert	query	insert	query
1	0.63	45.9	0.15	0.27
7	0.63	47.8	0.15	1.05
14	0.63	51.2	0.15	1.86
21	0.63	55.2	0.15	1.74
30	0.63	59.1	0.15	2.96

DeadIndex. We compare HINT in the role of DeadIndex as discussed in Section 5.3, against the 2D transformation approach proposed in [55], powered by an in-memory 2D R-tree from the highly optimized Boost.Geometry library. [18] Table 5 reports the insert

[18] Benchmark in [37] showed that Boost.Geometry (https://www.boost.org) R-tree implementations outperform the libspatialindex library (https://libspatialindex.org/).

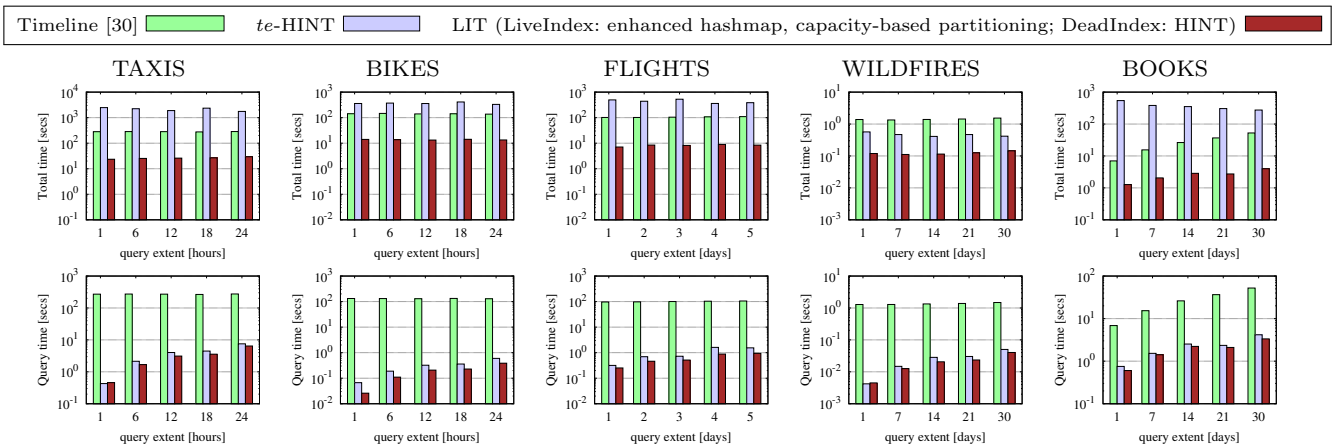


Fig. 14 Pure time-travel queries: LIT against competition

Table 6 Pure time-travel: 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: query time breakdown [secs]; default query 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

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.

9.2.2 Pure time-travel Queries

We now compare the LIT hybrid index against *te*-HINT (Section 4) and the state-of-the-art Timeline index [30] for transactional DBs (re-implemented to fully operate in main memory). Figure 14 (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 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.

9.2.3 Tuning a-LIT

Similar to Section 9.2.1, we first investigate the best setup for a-LIT.

LiveIndex. We implemented the two alternative solutions for the LiveIndex discussed in Section 6.1; an in-memory Boost.Geometry 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 update (insert and delete) and query time for each solution while varying the search-key query extent. Due to lack of space, we only report on 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.[19] Even a single pure time index is still a better option 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.

9.2.4 Range time-travel Queries

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 9.2.2, 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 [4] (re-implemented to fully operate in main memory). The first and the third rows in Figure 15 report the total time of the indices, 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 rea-

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

Table 8 LiveIndex for a-LIT; total update and query times [secs], default temporal query extent

TAXIS

search-key query extent [dollars]	2D R-tree update	2D R-tree query	single pure time index update	single pure time index query	multiple pure time indices update	multiple pure time indices query
1	1163	0.002	16.2	0.02	19.7	0.002
6	1163	0.002	16.2	0.02	19.7	0.002
12	1163	0.002	16.2	0.02	19.7	0.002
18	1163	0.002	16.2	0.02	19.7	0.002
24	1163	0.002	16.2	0.02	19.7	0.002

BOOKS

search-key query extent [books]	2D R-tree update	2D R-tree query	single pure time index update	single pure time index query	multiple pure time indices update	multiple pure time indices query
1	1622	1.6	0.4	0.4	0.5	0.04
7	1622	1.9	0.4	0.4	0.5	0.04
14	1622	2.2	0.4	0.4	0.5	0.04
21	1622	3.2	0.4	0.4	0.5	0.04
30	1622	4.5	0.4	0.4	0.5	0.04

Table 9 DeadIndex for a-LIT; total update and query times [secs], default temporal query extent

TAXIS

search-key query extent [dollars]	3D R-tree insert	3D R-tree query	HINT insert	HINT query	multiple HINTs insert	multiple HINTs query
1	81.9	40.6	9.49	4.11	9.48	0.49
6	81.9	40.5	9.49	4.12	9.48	0.51
12	81.9	40.6	9.49	4.11	9.48	0.40
18	81.9	40.6	9.49	4.12	9.48	0.41
24	81.9	40.5	9.49	4.11	9.48	0.41

BOOKS

search-key query extent [books]	3D R-tree insert	3D R-tree query	HINT insert	HINT query	multiple HINTs insert	multiple HINTs query
1	0.74	4.80	0.15	0.85	0.15	0.26
7	0.74	5.35	0.15	1.75	0.15	0.25
14	0.74	7.86	0.15	2.53	0.15	0.28
21	0.74	9.14	0.15	2.63	0.15	0.27
30	0.74	11.6	0.15	4.14	0.15	0.27

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

input stream	MVB-tree [4]	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

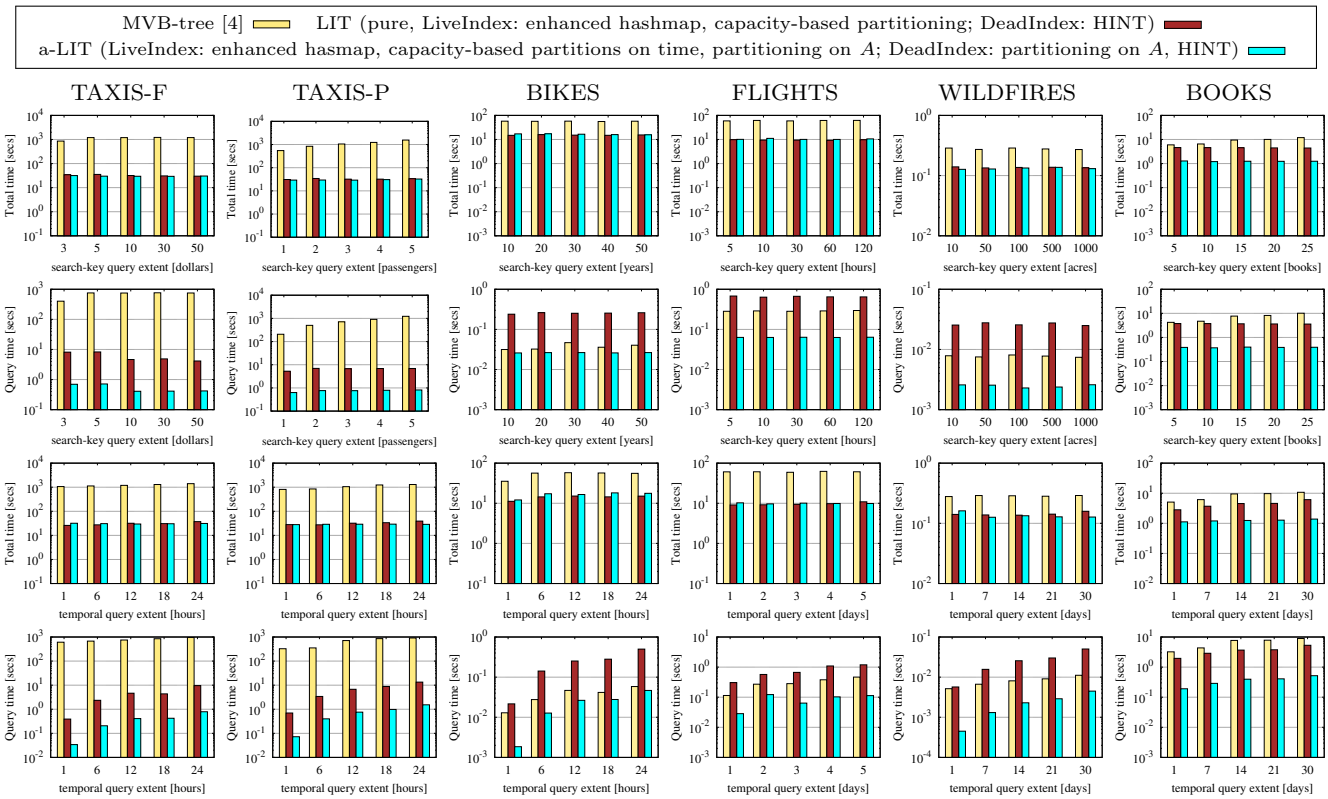


Fig. 15 Range time-travel queries: a-LIT against competition

son 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 15), 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.

9.2.5 Index Size

We conclude our analysis for in-memory query processing 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 all cases as shown in Figure 15. Finally, we study the growth of the LIT's size of time; Figure 16 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.

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

input stream	Timeline [30]	*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: index size [MBs]

input stream	MVB-tree [4]	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

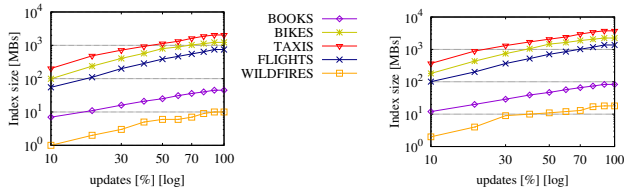


Fig. 16 Size growth over time

9.3 Query Processing under Limited Memory

The second part of our experimental analysis evaluates queries under limited memory for LIT$^+$ and a-LIT$^+$.[20] We ran our tests on the same datasets and input streams used for the first part of the analysis (see Section 9.1). We setup the in-memory LiveIndex and DeadIndex components according to Section 9.2.1 amd 9.2.3; i.e., LIT$^+$ uses an enhanced hashmap with capacity-based partitioning for LiveIndex and a HINT for DeadIndex, while a-LIT$^+$ uses multiple enhanced hashmaps with capacity-based partitions on time and partitioning on search-key A for LiveIndex, and multiple HINTs based on the A-partitioning for DeadIndex. The page size for the on-disk R-trees (both as a FossilIndex and as a competitor) was fixed to 8 KB.

All tests ran (again as single-threaded processes) on an Intel Core i7-14700K clocked at 3.4 GHz with 64 GBs of RAM,.1.7 MB L1 cache, 28 MB L2 cache, and 33 MB L3 cache, and a 4 TB NVMe SSD using PCIe 4.0.

9.3.1 Tuning LIT$^+$ and a-LIT$^+$

Similar to Section 9.2, we first examine the best setting for the DeadIndex (in regards to fossilization) and the FossilIndex of LIT$^+$. Our findings directly apply to a-LIT$^+$; plots omitted due to lack of space.

DeadIndex. We consider the two approaches described in Section 7.3 for implementing the DeleteFossils function. We denote by *Reconstruct* the straightforward approach which drops and rebuilds from scratch the HINT DeadIndex, and by *Update in-situ*, the approach which updates the HINT partitions on site by removing all fossil entries. To evaluate the approaches, we fed LIT$^+$ the input streams and monitored the triggered fossilization events for different values of the memory budget M, as a percentage of the dataset size (see Table 1). Figure 17 reports the total time required to maintain the DeadIndex by removing the fossils. For each M value, we also include the number of fossilization events as annotation over the bars (i.e., how many times the DeleteIntervals function was invoked).[21] Furthermore, Figure 19 shows the average number of fossils removed from DeadIndex per M. As expected, the deletion times for both approaches drop as M increases due to fewer fossilization events taking place, while the average number of fossils increases as larger DeadIndex chunks are removed per fossilization. Most importantly, we observe that the Update in-situ approach always outperforms the Reconstruct one for all datasets; typically there is 6× to 12× improvement. For the rest of our analysis, we always use Update in-situ for maintaining DeadIndex.

FossilIndex. We consider the two approaches detailed in Section 7.4 for updating FossilIndex, i.e., for implementing the InsertFossilIntervals function. To evaluate the approaches, we tested in isolation the last part of the fossilization process. Specifically, we directly submitted chunks of the input streams to the R-tree FossilIndex as fossils. Figure 18 reports the insertion cost to FossilIndex, while varying the number of inserted records as a percentage of the dataset cardinality (see again Table 1). Each experiment is initialized with an empty FossilIndex, followed by inserting a chunk of the dataset; the first chunk is bulk-loaded using the STR algorithm. We observe that the batch insertion approach outperforms R*-tree insertion in all cases. This is expected, as batching minimizes I/O by grouping records into leaf nodes before writing, whereas R*-tree insertion adds records individually, leading to higher cost. The only exception occurs at the 5% insertion of the WILDFIRES dataset, where the number of insertions is small (i.e., 38,921) compared to other streams (e.g., 102,535 for BOOKS). In this case, the sorting overhead of the batching method is more expensive than the restructuring cost of the R*-tree algorithm. Under these findings, we always use batch insertion for updating FossilIndex.

We next study the overall cost of updating LIT$^+$ and a-LIT$^+$ when consuming an input stream. This cost includes the cost of maintaining the LiveIndex, DeadIndex, and FossilIndex components and the cost of updating t_f. Note that a-LIT$^+$ utilizes the same methods for maintaining its DeadIndex and FossilIndex during the fossilization process, as discussed in the previous paragraph. Figure 20 reports our findings for different values of the memory budget M and the reduction factor r (see Section 7.2). Higher values of M significantly decrease the update time because they reduce the frequency of disk migrations and hence the fossilization time. While r has a smaller effect on total update time than M, $r = 10\%$ is the fastest in all settings. Hence, we set $r = 10\%$ for the remaining experiments. Since a-LIT$^+$ has slightly higher memory requirements compared to LIT$^+$ due to attribute A, using the same M value would trigger more frequent fossilization for a-LIT$^+$. So, for

[20] Code in `https://github.com/psimatis/lit_fossils`

[21] Without loss of generality, we set the reduction factor r to 10% in this test; we study its effect later in Figure 20.

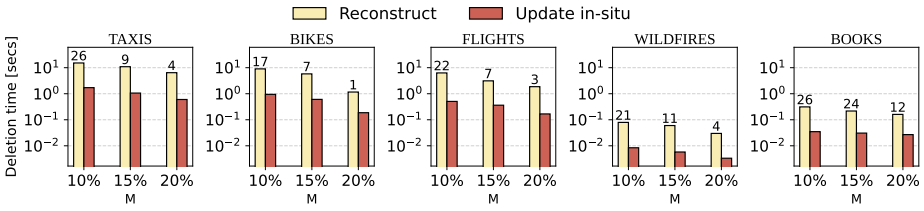


Fig. 17 LIT$^+$ DeadIndex: deletion time varying memory budget M (% of dataset); numbers mark fossilization events, reduction factor $r = 10\%$

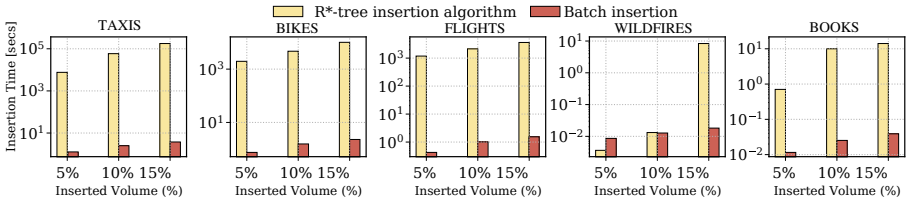


Fig. 18 LIT$^+$ DeadIndex: insertion time varying memory budget M (% of dataset)

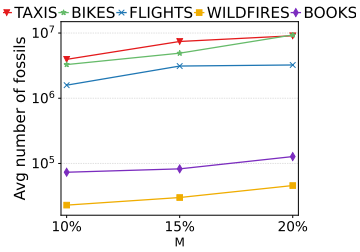


Fig. 19 LIT$^+$: fossilized records

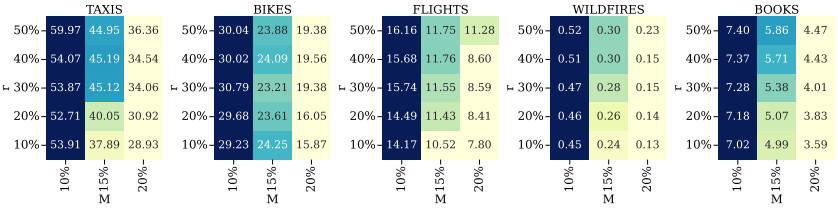


Fig. 20 LIT$^+$: total update time [secs] varying memory budget M and reduction factor r

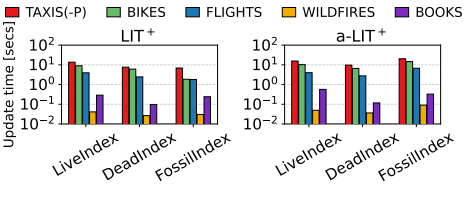


Fig. 21 Update time breakdown: LIT$^+$ ($M = 20\%$) and a-LIT$^+$ ($M = 25\%$), $r = 10\%$

the remaining tests, an empirical +5% adjustment ensures a fair comparison under similar memory pressure.

Figure 21 (left) breaks down LIT$^+$'s update time for budget $M = 20\%$. We observe that dataset cardinality impacts update time; larger steams (e.g., TAXIS) require substantially more time than smaller ones (e.g., WILDFIRES). Overall, LiveIndex dominates, since it processes all insertions and deletions, and DeadIndex is costlier than FossilIndex. A notable exception is BOOKS, where long intervals inflate LiveIndex's memory footprint, thus, leading to frequent fossilizations and a small, cheap to maintain DeadIndex.

Figure 21 (right) shows the same breakdown for a-LIT$^+$. As with LIT$^+$, LiveIndex is the slowest component to maintain. Interestingly, all datasets exhibit the DeadIndex dip seen in BOOKS of Figure 21 (left). This is due to: (1) attribute A increasing memory pressure and triggering frequent fossilizations, and (2) the overhead of the 3D R-tree used for FossilIndex compared to DeadIndex's lighter HINT.

9.3.2 Querying LIT$^+$ and a-LIT$^+$

We study the query performance of the LIT$^+$ and a-LIT$^+$ frameworks. Figure 22 provides a breakdown of the total query time for LIT$^+$, across all three index components while varying the memory budget M. FossilIndex bars are annotated with the percentage of queries that access fossils on disk. As M increases, the fraction of disk-bound queries decreases, leading to lower FossilIndex query times, and an increase in DeadIndex query times. In contrast, LiveIndex remains unaffected by the varying memory budget. This behavior is consistent across all datasets, confirming that reducing I/O overhead improves performance.

We next compare LIT$^+$ against a disk-resident Timeline and a hybrid baseline that shares LIT$^+$'s in-memory LiveIndex with an on-disk 2D R-tree indexes both dead and fossil records. Both these competitors use the available memory as an LRU cache. Figure 24 reports the total and query times; the R-tree based solution was terminated on TAXIS as it was extremely slower than the other methods. LIT$^+$ outperforms Timeline on both metrics, as in the in-memory query processing study (Figure 14). Even though LiveIndex + 2D R-tree avoids transfers from DeadIndex to FossilIndex, it exhibits higher update time than LIT$^+$, because it inserts dead intervals to the on-disk R-tree one-by-one, which is computationally intensive and also incurs many I/Os. Further, it experiences higher query times for queries on recent data than LIT$^+$'s HINT, as each query potentially accesses the disk.

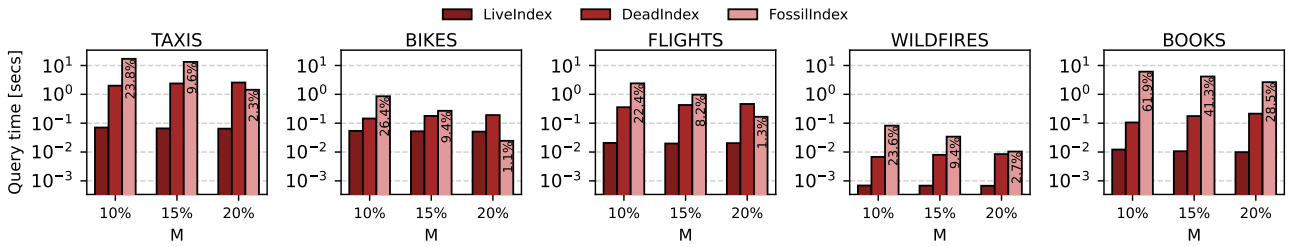


Fig. 22 Pure time-travel queries: LIT$^+$ breakdown; default query extents, $r = 10\%$, annotations show percentage of queries on FossilIndex

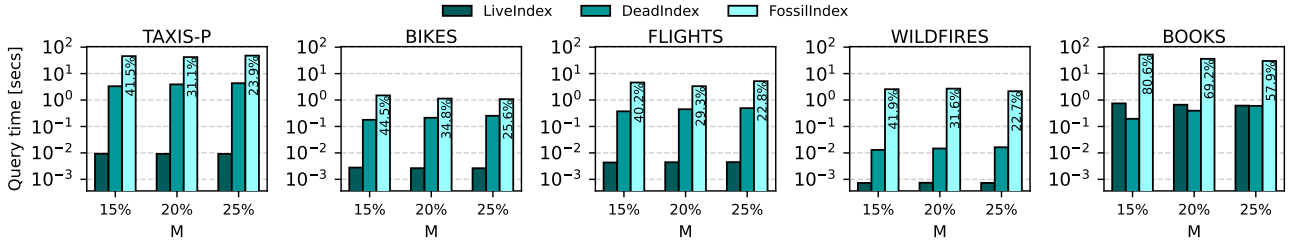


Fig. 23 Range time-travel queries: a-LIT$^+$ breakdown; default query extents, $r = 10\%$, annotations show percentage of queries on FossilIndex

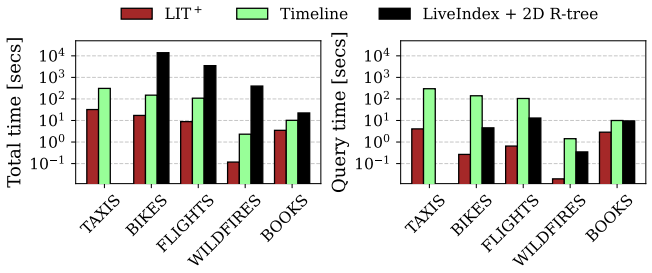


Fig. 24 Pure time-travel queries: LIT$^+$ against competition; default query extents, $M = 20\%$, $r = 10\%$

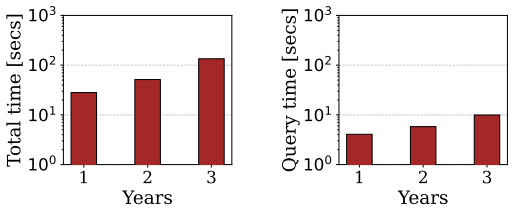


Fig. 25 Pure time-travel queries: LIT$^+$ scalability on TAXIS; default query extents, $M = 20\%$, $r = 10\%$

We also study how LIT$^+$ scales with the size of the input stream. Figure 25 reports total and query times on TAXIS streams spanning 1 to 3 years. Both metrics scale smoothly with stream size indicating stable behavior as the index grows. a-LIT$^+$ exhibits the same scaling behavior; plots omitted due to space constraints.

Figure 23 provides the query time breakdown for a-LIT$^+$, while varying memory budget M. As observed in Figure 22, LiveIndex is the fastest component, followed by DeadIndex, and finally FossilIndex. An exception occurs in BOOKS where LiveIndex is slower than DeadIndex, due to the dataset's long-lived intervals.

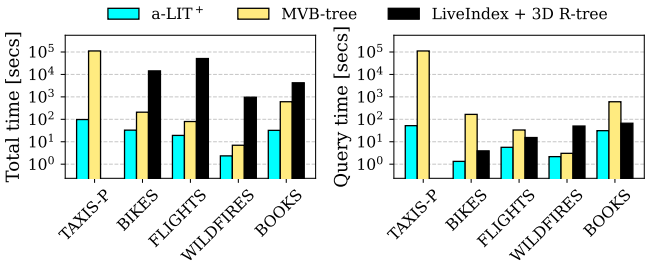


Fig. 26 Range time-travel queries: a-LIT$^+$ against competition; default query extents, $M = 25\%$, $r = 10\%$

Lastly, we compare a-LIT$^+$ against a disk-resident MVB-tree and a hybrid solution, with a single in-memory LiveIndex (similar to a-LIT) and an on-disk 3D R-tree for dead and fossil records. Figure 26 reports total and query times; numbers on TAXIS-P for the R-tree based solution again omitted. a-LIT$^+$ is the best on all datasets in total time; MVB-tree is the runner up. The LiveIndex + 3D R-tree processes queries faster than MVB-tree on every dataset except WILDFIRES.

10 Conclusions and Future Work

We proposed LIT, a hybrid index for time-evolving databases, which decouples the handling of current (live) record versions from past (dead) record versions. We studied different implementation options for the live and dead components to minimize update and query costs. We considered both 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. This work revisits our previously proposed LIT [19] by introducing LIT$^+$. LIT$^+$ extends LIT to manage memory-bounded scenarios, where the indexed version history exceeds the system's memory capacity. LIT$^+$ stores old dead versions on disk, while keeping the recently dead versions in memory. We study mechanisms for batch transfers of versions between LIT$^+$'s components. 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. We also demonstrate the efficiency and scalability of LIT$^+$ in indexing long version histories. Future work includes studying the applicability of LIT on other temporal query types (e.g., aggregation, joins), multi-threaded processing, and LIT's integration into an open-source database system.

References

1. Al-Kateb, M., Ghazal, A., Crolotte, A., Bhashyam, R., Chimanchode, J., Pakala, S.P.: Temporal query processing in teradata. In: EDBT, pp. 573–578 (2013)
2. Arge, L., Vitter, J.S.: Optimal dynamic interval management in external memory (extended abstract). In: FOCS, pp. 560–569 (1996)
3. Arge, L., Vitter, J.S.: Optimal external memory interval management. SIAM J. Comput. **32**(6), 1488–1508 (2003)
4. Becker, B., Gschwind, S., Ohler, T., Seeger, B., Widmayer, P.: An asymptotically optimal multiversion b-tree. VLDB J. **5**(4), 264–275 (1996)
5. Beckmann, N., Kriegel, H., Schneider, R., Seeger, B.: The R*-Tree: An efficient and robust access method for points and rectangles. In: ACM SIGMOD, pp. 322–331 (1990)
6. Behrend, A., Dignös, A., Gamper, J., Schmiegelt, P., Voigt, H., Rottmann, M., Kahl, K.: Period index: A learned 2d hash index for range and duration queries. In: SSTD, pp. 100–109 (2019)
7. Bellomarini, L., Nissl, M., Sallinger, E.: itemporal: An extensible generator of temporal benchmarks. In: IEEE ICDE, pp. 2021–2033 (2022)
8. de Berg, M., Cheong, O., van Kreveld, M.J., Overmars, M.H.: Computational geometry: algorithms and applications, 3rd Edition. Springer (2008)
9. Bernhardt, A., Tamimi, S., Vinçon, T., Knödler, C., Stock, F., Heinz, C., Koch, A., Petrov, I.: neodbms: In-situ snapshots for multi-version DBMS on native computational storage. In: IEEE ICDE, pp. 3170–3173 (2022)
10. Böhlen, M.H., Dignös, A., Gamper, J., Jensen, C.S.: Temporal data management - an overview. In: eBISS, vol. 324, pp. 51–83 (2017)
11. Böhlen, M.H., Snodgrass, R.T., Soo, M.D.: Coalescing in temporal databases. In: VLDB, pp. 180–191 (1996)
12. Bornemann, L., Bleifuß, T., Kalashnikov, D.V., Nargesian, F., Naumann, F., Srivastava, D.: Matching roles from temporal data: Why joe biden is not only president, but also commander-in-chief. Proc. ACM Manag. Data **1**(1), 65:1–65:26 (2023)
13. Bouros, P., Mamoulis, N.: A forward scan based plane sweep algorithm for parallel interval joins. Proc. VLDB Endow. **10**(11), 1346–1357 (2017)
14. Bouros, P., Mamoulis, N., Tsitsigkos, D., Terrovitis, M.: In-memory interval joins. VLDB J. **30**(4), 667–691 (2021)
15. Campbell, F.S., Arab, B.S., Glavic, B.: Efficient answering of historical what-if queries. In: ACM SIGMOD, pp. 1556–1569 (2022)
16. Ceccarello, M., Dignös, A., Gamper, J., Khnaisser, C.: Indexing temporal relations for range-duration queries. In: SSDBM, pp. 3:1–3:12 (2023)
17. Christodoulou, G., Bouros, P., Mamoulis, N.: HINT: A hierarchical index for intervals in main memory. In: ACM SIGMOD, pp. 1257–1270 (2022)
18. Christodoulou, G., Bouros, P., Mamoulis, N.: HINT: a hierarchical interval index for allen relationships. VLDB J. **33**(1), 73–100 (2024)
19. Christodoulou, G., Bouros, P., Mamoulis, N.: LIT: lightning-fast in-memory temporal indexing. Proc. ACM Manag. Data **2**(1), 20:1–20:27 (2024)
20. Dignös, A., Glavic, B., Niu, X., Gamper, J., Böhlen, M.H.: Snapshot semantics for temporal multiset relations. Proc. VLDB Endow. **12**(6), 639–652 (2019)
21. Ding, J., Nathan, V., Alizadeh, M., Kraska, T.: Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. Proc. VLDB Endow. **14**(2), 74–86 (2020)
22. Edelsbrunner, H.: Dynamic rectangle intersection searching. Tech. Rep. 47, Institute for Information Processing, TU Graz, Austria (1980)
23. Elmasri, R., Wuu, G.T.J., Kim, Y.: The time index: An access structure for temporal data. In: VLDB, pp. 1–12 (1990)
24. Gao, D., Jensen, C.S., Snodgrass, R.T., Soo, M.D.: Join operations in temporal databases. VLDB J. **14**(1), 2–29 (2005)
25. Gao, J., Sintos, S., Agarwal, P.K., Yang, J.: Durable top-k instant-stamped temporal records with user-specified scoring functions. In: IEEE ICDE, pp.

720–731 (2021)
26. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: ACM SIGMOD, pp. 47–57 (1984)
27. Hsu, S.H., Jensen, C.S., Snodgrass, R.T.: Valid-time selection and projection. In: R.T. Snodgrass (ed.) The TSQL2 Temporal Query Language, pp. 249–296. Kluwer (1995)
28. Hu, X., Sintos, S., Gao, J., Agarwal, P.K., Yang, J.: Computing complex temporal join queries efficiently. In: ACM SIGMOD, pp. 2076–2090 (2022)
29. Kanellakis, P.C., Ramaswamy, S., Vengroff, D.E., Vitter, J.S.: Indexing for data models with constraints and classes. In: ACM PODS, pp. 233–243 (1993)
30. Kaufmann, M., Manjili, A.A., Vagenas, P., Fischer, P.M., Kossmann, D., Färber, F., May, N.: Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In: ACM SIGMOD, pp. 1173–1184 (2013)
31. Kline, N., Snodgrass, R.T.: Computing temporal aggregates. In: IEEE ICDE, pp. 222–231 (1995)
32. Kraska, T., Beutel, A., Chi, E.H., Dean, J., Polyzotis, N.: The case for learned index structures. In: ACM SIGMOD, pp. 489–504 (2018)
33. Kriegel, H., Pötke, M., Seidl, T.: Managing intervals efficiently in object-relational databases. In: VLDB, pp. 407–418 (2000)
34. Kulkarni, K.G., Michels, J.: Temporal features in SQL: 2011. SIGMOD Rec. **41**(3), 34–43 (2012)
35. Leutenegger, S.T., Edgington, J., López, M.A.: STR: A simple and efficient algorithm for r-tree packing. In: IEEE ICDE, pp. 497–506 (1997)
36. Liu, Q., Li, M., Zeng, Y., Shen, Y., Chen, L.: How good are multi-dimensional learned indexes? an experimental survey. VLDB J. **34**(2), 17 (2025)
37. Loskot, M., Wulkiewicz, A.: (2019). Https://github.com/mloskot/spatial_index_benchmark
38. Lu, W., Zhao, Z., Wang, X., Li, H., Zhang, Z., Shui, Z., Ye, S., Pan, A., Du, X.: A lightweight and efficient temporal database management system in TDSQL. Proc. VLDB Endow. **12**(12), 2035–2046 (2019)
39. Michalopoulos, A., Tsitsigkos, D., Bouros, P., Mamoulis, N., Terrovitis, M.: Efficient nearest neighbor queries on non-point data. In: ACM SIGSPATIAL, pp. 33:1–33:4 (2023)
40. Moon, B., López, I.F.V., Immanuel, V.: Efficient algorithms for large-scale temporal aggregation. IEEE TKDE **15**(3), 744–759 (2003)
41. Moro, M.M., Tsotras, V.J.: Transaction-time indexing. In: Encyclopedia of Database Systems, Second Edition. Springer (2018)
42. Moro, M.M., Tsotras, V.J.: Valid-time indexing. In: Encyclopedia of Database Systems, Second Edition. Springer (2018)
43. Nathan, V., Ding, J., Alizadeh, M., Kraska, T.: Learning multi-dimensional indexes. In: ACM SIGMOD, pp. 985–1000 (2020)
44. Papaioannou, K., Theobald, M., Böhlen, M.H.: Outer and anti joins in temporal-probabilistic databases. In: IEEE ICDE, pp. 1742–1745 (2019)
45. Piatov, D., Helmer, S.: Sweeping-based temporal aggregation. In: SSTD, pp. 125–144 (2017)
46. Piatov, D., Helmer, S., Dignös, A.: An interval join optimized for modern hardware. In: IEEE ICDE, pp. 1098–1109. IEEE Computer Society (2016)
47. Piatov, D., Helmer, S., Dignös, A., Persia, F.: Cache-efficient sweeping-based interval joins for extended allen relation predicates. VLDB J. **30**(3), 379–402 (2021)
48. Qi, J., Liu, G., Jensen, C.S., Kulik, L.: Effectively learning spatial indices. Proc. VLDB Endow. **13**(11), 2341–2354 (2020)
49. Salzberg, B., Tsotras, V.J.: Comparison of access methods for time-evolving data. ACM Comput. Surv. **31**(2), 158–221 (1999)
50. Saracco, C.M., Nicola, M., Gandhi, L.: A matter of time: Temporal data management in db2 10. Tech. rep., IBM (2012)
51. Snodgrass, R.T., Ahn, I.: Temporal databases. Computer **19**(9), 35–42 (1986)
52. Song, Z., Roussopoulos, N.: Seb-tree: An approach to index continuously moving objects. In: MDM, pp. 340–344 (2003)
53. Tao, Y., Papadias, D., Faloutsos, C.: Approximate temporal aggregation. In: IEEE ICDE, pp. 190–201 (2004)
54. Tsitsigkos, D., Lampropoulos, K., Bouros, P., Mamoulis, N., Terrovitis, M.: A two-layer partitioning for non-point spatial data. In: IEEE ICDE, pp. 1787–1798 (2021)
55. U, L.H., Mamoulis, N., Berberich, K., Bedathur, S.J.: Durable top-k search in document archives. In: ACM SIGMOD, pp. 555–566 (2010)
56. Vitter, J.S.: External memory algorithms and data structures. ACM Comput. Surv. **33**(2), 209–271 (2001)
57. Zhang, D., Markowetz, A., Tsotras, V.J., Gunopulos, D., Seeger, B.: Efficient computation of temporal aggregates with range predicates. In: ACM PODS (2001)
58. Zhang, Z., Hu, H., Xue, Z., Chen, C., Yu, Y., Fu, C., Zhou, X., Li, F.: SLIMSTORE: A cloud-based deduplication system for multi-version backups. In: IEEE ICDE, pp. 1841–1846 (2021)