High Performance Temporal Indexing on Modern Hardware

David B. Lomet [1], Faisal Nawab [2]
[1] *Microsoft Research, Redmond, WA 98052 USA*
lomet@microsoft.com
[2] *Department of Computer Science, University of California, Santa Barbara, Santa Barbara, CA 93106 USA*
nawab@cs.ucsb.edu

***Abstract*—Transaction time databases can be put to a number of valuable uses, auditing, regulatory compliance, readable backups, and enabling multi-version concurrency control. While additional storage for retaining multiple versions is unavoidable, compression and the declining cost of disk storage largely removes that impediment to supporting multi-version data. Not clear has been whether effective indexing of historical versions, can be achieved at high performance. The current paper shows how temporal indexing can exploit the latch-free infrastructure provided for the Bw-tree by the LLAMA cache/storage subsystem to support high performance. Further, it demonstrates how the LLAMA mapping table can be exploited to simultaneously enable migration of historical data, *e.g.* to cloud storage, while overcoming the index node time splitting difficulty that has arisen in the past when historical nodes are migrated.**

Keywords—Modern hardware, latch-free, log structured, temporal database, multi-version indexing

I. Introduction

A. Temporal Data

Temporal functionality has been discussed within the database community for many years. This functionality includes both transaction time [28], [29] and valid time variants, and bi-temporal that includes both variants.

1) Transaction Time Support: Our emphasis here is on transaction time functionality. Longer term storage of versions enables both (1) *time travel* that explores how a data item has changed over time and (2) *as of queries* which provide access to the state of the database some time in the past. Transaction time databases (TTDBs) provide access to both current and previous database states by creating new versions of data for every transactional update, as opposed to doing update in place. TTDB's have a "long history" in the database research community [10], [29]. In addition, the SQL standard specifies temporal functionality and this is supported commercially by Oracle [1], [2] and IBM.

In this paper we show how transaction time indexing can be provided with very high performance. We do this by exploiting many of the same techniques pioneered in the Bw-tree, a non-temporal B-tree index [18]. In particular, we make use of an augmented form of the LLAMA [16] cache/storage subsystem.

2) Uses for Transaction Time Data: TTDB's have been used for auditing, legal compliance, trend analysis, etc. [3], [23]. Such uses derive from their ability to execute queries *as of* a past time and to trace data items forward and backward in time, called *time travel.*

In addition to external user functionality, TTDB's serve useful **systems** purposes such as backup and restore. The historical data can serve as a query-able online backup that can be used for media recovery and point-in-time recovery. Transaction time versions have been exploited commercially in a number of these **system aspects**. For example, Rdb [9] uses these versions for a transaction consistent checkpoint. Oracle uses its Flashback versions for point-in-time recovery [1] to remove erroneous user transactions. The Immortal DB research prototype [20] used versions in a similar but more precise way, by being able to automatically remove only versions that depended upon the erroneous data.

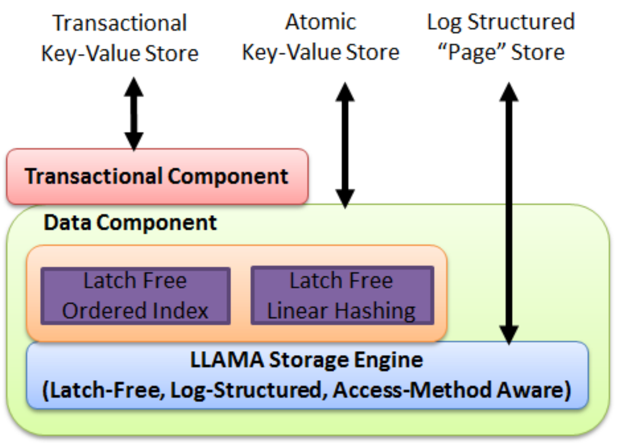
B. Temporal Indexing

Temporal data is two dimensional (2D). A key or record id uniquely identifies a point in key space. In the time dimension, versions have lifetimes, naturally represented as a line segment. This characteristic of the time dimension led several temporal indexes to exploit the notion of a *time split*, introduced in the Write-Once B-tree (WOB-tree) [6].

A time split separates versions of a page by time into two pages. Versions only in the time region corresponding to one page only reside in that page. But any version that crosses the time boundary between the pages appears in both pages. This permits a search that will always find a given version of a record within key and time boundaries of a single leaf page responsible for a disjoint part of the search space.

Splits only occur on pages containing current data. Pages with only historical data are not updated and there is no need to partition them further. How one handles historical and current pages that result from a time split varies among the temporal methods. The WOB-tree moved the current data to a new page since it was using a write-once medium that prevented it from updating the existing page, which was hence assigned to the purely historical data.

The time-split B-tree (TSB-tree) [22] moved historical data nodes elsewhere, *e.g.* to perhaps less expensive media, so as to keep the current data as highly accessible as possible. But this limited index node time splitting, even perhaps leading to the need to time split lower level nodes so that an index node could be time split effectively (see section IV).



Transactional Key-Value Store
Atomic Key-Value Store
Log Structured "Page" Store
Transactional Component
Data Component
Latch Free Ordered Index
Latch Free Linear Hashing
LLAMA Storage Engine
(Latch-Free, Log-Structured, Access-Method Aware)

Fig. 1: Deuteronomy transactional key value store architecture. Note that the intent of the architecture is to permit multiple access methods to be "slipped into" the system.

1) Modern Hardware: The promise of temporal functionality has always been weighed against the cost of providing it. Much of the focus of the Immortal DB project [21] was on showing that the performance for the current data would not be adversely impacted by the costs of storing and managing historical versions.

The context for temporal functionality has now changed from prior efforts. Improved exploitation of modern hardware has produced dramatically higher performance for current time database systems [5], [7], [12], [16]. The question is whether these performance techniques can be applied to multi-version data, with comparable positive performance impact.

2) "Modern" Temporal Indexing: This paper introduces the TSBw-tree, a temporal access method that performs time splitting and, like the TSB-tree, migrates historical data. The TSBw-tree exploits the LLAMA cache/storage manager used by the Bw-tree as its latch-free, log structured infrastructure. The architecture of this system [17] is shown in Figure 1.

There are subtleties in the TSBw-tree, both in its time splitting and in the migration of historical pages to a different location. This paper describes the nature of the problems encountered and how the TSBw-tree solves them using a modified LLAMA subsystem.

C. Contributions

The TSBw-tree is a new temporal indexing method like the TSB-tree that layers on LLAMA, with its latch-free and log structured implementation that so effectively exploits modern hardware. There are three major contributions in doing this.

1) High Performance: LLAMA latch freedom means that readers never collide with writers and that threads are never blocked or forced to do a context switch. This is a key part of the LLAMA "secret sauce", and accounts for the unparalleled high Bw-tree main memory performance. The log structuring means that the number of writes needed to make multi-versioned data stable is greatly reduced. This combination of thread and I/O efficiency carries over from the Bw-tree [18] to the TSBw-tree setting, resulting in the highest performing temporal access method yet reported. This also illustrates LLAMA generality in supporting multiple access methods.

2) Index Node Splitting: The TSB-tree, during a time split, moves the historical page produced by the split to a new location. Unfortunately, moving historical nodes introduces a serious complication to the splitting of index nodes. In particular, the TSB-tree had trouble splitting index pages exactly because a historical page can move and require the updating of its index term. And this index term might be on a historical index page, or might even be on both current and historical pages. The TSBw-tree avoids this problem, despite the migration of historical pages. This is a feature of our mapping table architecture and is described more fully in Section IV.

3) History Migration: We extend LLAMA to support migration functionality. This enables crash recovery for the log structured store without the need to scan historical data. Migration of historical data solves a LLAMA difficulty. Log structured file system (LFS) cleaning (garbage collection) moves unchanged data from the early part of the log to the end of the log to make contiguous space available for reuse. Historical data is never changed but can persist for an extended period. Without migration, a historical page is rewritten whenever cleaning encounters it. Migration moves this historical page to a different location where it avoids further cleaning induced moves. The migration protocol is described in Section V.

D. Outline

The paper is organized as follows. We give background on LLAMA in Section II and describe our high performance temporal indexing in Section III. The major technical innovations are described in detail in Sections IV(node time splitting) and V (migration). Section VI presents our performance results, while Section VII discusses related work. We present some conclusions in Section VIII.

II. LATCH-FREE, LOG STRUCTURED

As with the Bw-tree, we exploit LLAMA to produce a high performance index, now for temporal data. Exploiting the LLAMA mapping table avoids the index maintenance difficulties experienced by earlier temporal indexing methods that relocated historical data. The mapping table also facilitates migration of historical data to a less expensive storage medium where we can minimize its impact on current time database performance. Migration also reduces the write amplification that is a characteristic of log structured approaches.

A. Latch-free Updating

The Bw-tree [18] demonstrated that eliminating latches and reducing update-in-place produces high performance for a B-tree style index. All Bw-tree updates are installed using a compare and swap (CAS) instruction. The effect is that readers never wait for writers, or the reverse. Writers sometimes run into each other (low probability), producing an update CAS failure that requires repeating the update.

In addition to being latch-free, many updates are *delta updates* that avoid update-in-place. The deltas are accumulated

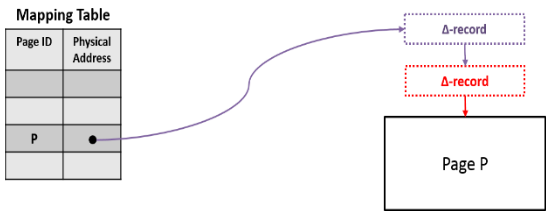


Fig. 2: The Mapping Table showing delta updating. The mapping table isolates a page physical location from the logical references to it. A logical reference may remain unchanged while the physical location of the page changes.

into a batch that is eventually merged into a read-optimized state. This has two benefits. (1) The number of state reorganizations is reduced by the size of the batch, reducing the instruction path per update. (2) While accumulating the batch of deltas, the cached state of the page is preserved, greatly improving the hit ratios of the processors caches.

LLAMA supports both latch-free state changes and delta updating. It enables an access method to focus solely on the indexing aspects of its task. For the Bw-tree, that was indexing the current key identified data. Here, we use LLAMA for a revised implementation of a time-split B-tree (TSB-tree) [22].

B. Mapping Table

The mapping table provides a central array with pointers to each page, the pages identified by the array index used as a logical page id (PID). All references to the page outside of a page use the PID. Pointers acquired while an operation is accessing the page are acquired via the mapping table translation of the PID. These pointers have a lifetime coincident with operation execution. Between operations, some or all of the pointers derived from the mapping table can change.

The level of indirection provided by the mapping table permits us to use the mapping table PID entry for the page to make atomic state changes to it. A state change can involve more than simply data state change, as page flushing and page structure modifications also change page state (see [18] and section IV-A2 of this paper).

C. Operations on Pages

LLAMA is latch-free and supports the following user operations to read and write pages of data:

- *Delta Update (D-Update)*: The D-update starts by observing the current state of the page. The current state is represented by the current address associated with the PID in the mapping table. Then, a delta record that describes the necessary changes is created. This record has a pointer to the current state (address) of the page. A CAS on the current state as represented in the mapping table entry is then executed. If successful, the D-update is installed and the changes are observed. Figure 2 illustrates this. A D-Update is invoked as Update-D(PID, in-ptr, out-ptr, data). PID is the page id. The in-ptr is a pointer to the prior state of the page and out-ptr points to the new state of the page. The data parameter can take different forms depending on the delta type. For example, a delta to update a record consists of a key and a value.
- *Replacement Update (R-Update)*: An R-Update installs its state in a similar fashion. In this case however, a totally new state is constructed and a pointer to the new state is installed using a CAS. The storage for the old state becomes garbage at this point. However, because other threads may still be accessing it, an epoch mechanism is used to protect the storage for the old state until no thread can have seen it and hence still depend on it. An R-Update is invoked as Update-R(PID, in-ptr, out-ptr, data). The parameters have the same function as the ones for the D-Update. However, in a R-Update the data parameter describes the entire state of the new page, not the changes to the old state.
- *Read*: A page read returns the page's address in main memory. If it is only on secondary storage, the read first fetches the page to main memory and then returns the address. The interface for reads is Read(PID, out-ptr), which reads a page with id PID and returns a pointer to the page in main memory via out-ptr.

Cache management produces state changes to pages of the cache in the same way that data operations produce state changes. For example, when a page is flushed, the flush is indicated by prepending a flush delta to the prior state of the page and installing it with a CAS on the mapping table entry. There is the added requirement to coordinate this with the posting of the page contents to the log structured storage (LSS) log buffer. This coordination is quite subtle and is explained fully in [16]. Our temporal index will exploit the same operational paradigm, as described in Section V.

D. Log Structured Storage

LLAMA stores its data on secondary storage in a log-structured manner [16]. As implemented previously, it managed its data as a single circular buffer, cleaning (garbage collecting) at the logical buffer start, while posting data to the logical buffer end. The cleaned storage at buffer start is recycled to become the free storage at the logical buffer end where newly updated pages are written. As we discuss below, this storage management strategy is not very effective when historical data is mixed with current updatable data.

Managing the persistence of data is provided by the following operations:

- *Flush*: Flush copies page state from cache to LSS buffer. Flush uses a latch-free approach that never blocks. Flush is invoked via Flush(PID, in-ptr, out-ptr, annotation). The in-ptr is the old state of the page and out-ptr is the new (flushed) state of the page. Each flush has an annotation that is set by the caller and is opaque to LLAMA. It can be used for a multitude of purposes as shown later. LLAMA returns the memory address of a “flush delta” that includes the location of the page in the log structured store (LSS) and the annotation. This flush delta is part of the page state copied to the LSS buffer.
- *Mk-Stable*: A Mk-Stable ensures that flushed pages are made stable. The interface is Mk-Stable(LSS

address). The LSS address is an address in the LSS. The interface returns only after all data up to the provided LSS address is stable in secondary storage.

- *Hi-Stable*: Hi-Stable returns the highest LSS offset of persistent LLAMA data. The interface is Hi-Stable(out-LSS address).
- *Allocate*: An allocate creates a new page with a unique Page ID (PID) using the interface Allocate(out-PID).
- *Free*: A free deallocates a page and make the PID available for reuse using the interface Free(PID).

E. System Transactions

LLAMA supports a limited transaction facility called a system transaction. System transactions are used for structure modifications needed by an access method, *e.g.* node splits. Inside a transaction, only the following operations are permitted: (1) Allocate, (2) Free, and (3) Update-D. Also, Allocate and Free must be called inside a transaction since the allocation state of a page needs to be persistent. A transaction works by reserving space in the LSS buffer for the operations that are inside the transaction and treating them as a single unit. This enables the stability of operations within a system transaction to be all or nothing, thus avoiding the need for system transaction undo to cope with partially executed transactions.

Durability and atomicity are provided via system transaction operations:

- TBegin(out-*TID*): begin a transaction and return a transaction ID (*TID*).
- TCommit(*TID*): commit a transaction identified by *TID*.
- TAbort(*TID*): abort transaction *TID*.

Operations that use the result of a committed transaction are guaranteed to come after the transaction's commit in the LSS. This enables the LSS to act as a durable recovery log. The use of delta updates makes logging the state in the LSS efficient compared to logging the state of the whole page.

The Bw-tree performs node splits in a latch-free manner [18] using a system transaction. The split is triggered when the page size exceeds a system threshold. A leaf page maintains a side link to the most recent split page. The side link provides a valid search path after installing the split page. This allows separating the split into two atomic actions. The B-link atomic split installation technique is employed to perform the split [15] in two phases.

1) The split at the leaf level is done in a system transaction that allocates a new page and installs a split delta.
2) The parent node is then atomically updated to contain the new index term pointing to the new page.

The process continues recursively up the tree if the size of the parent node exceeds the threshold.

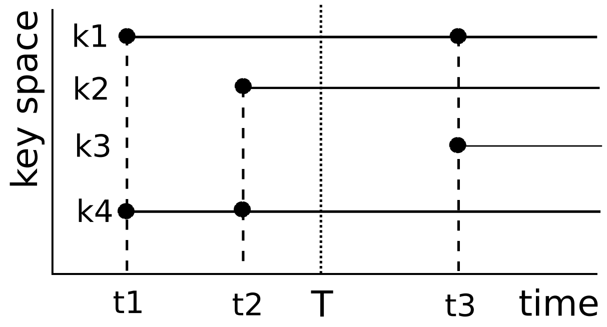


Fig. 3: A key-time representation of a page. A solid line represents the existence of the key at the corresponding time and a circle denotes an update/insert of the key.

III. THE TSBw-TREE

The TSBw-tree leverages LLAMA's latch-free and log-structured infrastructure to provide temporal indexing support similar to that provided by the TSB-tree, but exploiting modern hardware in the way that was done by the Bw-tree. In this section we begin by presenting the temporal indexing methods of the TSB-tree. Then, we show how the TSB-tree is adapted to LLAMA to form the TSBw-tree.

A. Time-Split B-tree

1) Indexing Versioned Data: Temporal data records contain key, value, and version timestamp. Data records reside in leaves of the TSB-tree. A leaf node can have many versions of a record with the same key. A record update is treated as an insertion of a new version. The TSB-tree indexes records by both key and time. There are lower and upper bounds on key values and timestamps in a leaf node. Figure 3 shows a key-time diagram of a leaf node with four keys. Each record is represented with a horizontal line. The beginning of the line is when the first record with that key was inserted. A circle denotes an update to the key. For example, key k_1 was inserted at time t_1 and updated at t_3. This node contains six records, two of which are historical.

Index nodes contain entries pointing to leaf pages or other lower level index nodes. An entry contains key, page pointer, and timestamp. Multiple entries can have the same key, at most one being current and the rest historical. The timestamps of entries pointing to historical pages denote the split times at which they were separated from more recent versions. Historical nodes contain the values within the key and timestamp ranges for the page. Once a time split is performed, the historical page is never updated. Only current pages are updated. The current page's timestamp is a special "infinity" time that is logically higher than all other timestamps.

To find a current record, search starts at the TSB-tree root. At each level, search looks for entries with the largest key that does not exceed the search key. Search then follows the entry pointing to a current node. Repeat this process until search arrives at a leaf node. All updating of the TSB-tree starts with locating the leaf node with the current version of the record with a given key. The new record version is added to that node.

2) Key Splitting: In many cases, it does not make sense to split by time. When time split, a node that contains mostly current records will result in two pages with a lot of redundant records. In such cases a key split is favored with a split key chosen to split the node. The index node pointing to the original node is then updated to contain an entry pointing to the new page with its new key range boundaries.

3) Time Splitting: With temporal data, an update is treated as an insert with a higher timestamp. If updates continue for records in a page, the page will eventually store mostly historical records, diluting the density of current records in the key range stored on each page. So several temporal indexes separate historical versions from pages storing current versions via a time split.

Temporal access methods, including the TSB-tree, can exploit the fact that historical versions are not updated. Hence, these methods can use "time splitting", introduced in the WOB-tree [6]. Unlike keys of current time databases, time associated with a version is an interval. Splitting at a time then requires duplicating versions whose validity interval crosses the split time boundary, enabling a page to contain all versions present in its time interval.

Consider the leaf node at Figure 3 for example. A record is denoted by $< key, timestamp >$. If the node was split with split time T it will result in a historical node with four records: $< k_1, t_1 >$, $< k_2, t_2 >$, $< k_4, t_1 >$, and $< k_4, t_2 >$. The new current node will contain the records: $< k_1, t_1 >$, $< k_1, t_3 >$, $< k_2, t_2 >$, $< k_3, t_3 >$, and $< k_4, t_2 >$. The records $< k_1, t_1 >$, $< k_2, t_2 >$, and $< k_4, t_2 >$ exist in both pages because their validity crosses the split time T. Duplication of data across a time split consumes additional space, but need not introduce extra version updating complexity as only the current version can be updated. "Historical" versions are read-only. Indeed, even current versions are read-only at times earlier than the current time.

4) Indexing Historical Data: The TSB-tree, using time-splits, indexes historical nodes as well as current nodes. Finding any leaf node then involves using an index that is larger than one accessing only current data. Thus, indexing historical nodes increases the access time to reach current nodes. However, TSB-tree access time to all data is logarithmic in the size of both current and historical data. Hence the height of the TSB-tree is rarely more than one level greater than the height of a simple B-tree over current data.

B. Temporal Indexing With LLAMA

The TSBw-tree leverages LLAMA to efficiently provide TSB-tree temporal indexing. Much of the TSB-tree function is mapped to LLAMA's interface in a way analogous to the Bw-tree. Updates are done using D-Update and R-Update operations, such that all versions are timestamped and no versions are discarded.

Structure modification operations (SMOs) [24] require multi-page changes. Splitting a node by key or time is divided into two atomic actions, as done in the Bw-tree. Each atomic action is a system transaction. The algorithm for node splits is shown in Algorithm 1. A first transaction installs a split delta at the full page that points to a new page and, if successful, a second transaction installs an index update delta to the parent index page.

Algorithm 1: Performing a key or time split with system transactions

```
1: // begin the split transaction with (TID$_{split}$)
2: TID$_{split}$ = TBegin ()
3: Read ($P_{pid}$, $P$) // read page $P$
4: Allocate($Q_{pid}$) // allocate a PID for a new page $Q$
5: $K_P$ := set of keys that will be in $P$ after the split
6: $K_Q$ := set of keys that will be in $Q$ after the split
7: prepare ($Q$, $K_Q$) // populate a page $Q$ and prepare it
8: Update-R ($Q_{pid}$, $Q$) // install the new page $Q$
9: // install the split delta
10: Update-D ($P_{pid}$, split-delta ($K_P$, $K_Q$))
11: // if the split delta failed, abort
12: If $fail$: Free($Q_{pid}$), TAbort(TID$_{split}$) and exit
13: TCommit (TID$_{split}$)
14: // begin the index delta system transaction with (TID$_{index}$)
15: TID$_{index}$ = TBegin ()
16: // install the index delta
17: Update-D ($P_{parent-pid}$, Index-entry-delta ($K_P$, $K_Q$))
18: // if the index delta failed, abort
19: If $fail$: TAbort(TID$_{index}$) and exit
20: TCommit (TID$_{index}$)
```

The first transaction with TID_{split} (line 2) is initiated. Then page P, is read (line 3), and a new page Q is allocated as the new right page of the split (line 4). For key splits, two sets of keys are created: (1) K_P: for keys that will be in P, and (2) K_Q: for keys that will be in Q (lines 5-6). Page Q is prepared for installation (lines 7-8). It is first populated with K_Q then its annotation is set by the higher layers input (*e.g.*, log position, timestamps,etc.). Also, a side link to what was P's right neighbor is set. The critical part is installing the split delta on P (line 10). A split delta contains information about the key sets to allow it to redirect operations to Q if the corresponding key was moved there. When installment of the split delta fails, Q is freed and the transaction is aborted (line 12). Otherwise, the transaction commits (line 13).

The second system transaction installs a new index entry (line 15) at the parent node. This uses a delta update to install a new index term (line 17). If the delta update fails, the system transaction is aborted (line 19) and is retried. If the index delta is installed, the system transaction commits (line 20).

Although the logic is similar for both key and time splits, time splits have challenges that have design implications for splitting both data and index nodes. This is discussed in Section IV.

IV. NODE TIME SPLITS

We want historical nodes to be read only to enable historical queries to execute without locking or latching. Thus, when we do a time split, we want to ensure that no future updates will be made to the historical node generated. This requires that we take care in choosing a split time for both data and index nodes.

A. Avoiding Updates to Historical Index Terms

1) Time Splitting Conundrum: Many temporal indexing techniques derive from the write-once B-tree (WOB-tree) [6], which was designed to provide B-tree style indexing for write-once disk storage. To cope with write-once storage, nothing was updated in place. Rather, new versions of data were added, which, in a transactional setting can support transaction time functionality. This "no update in place" was a requirement of the media, and meant that once a page was written, further updating required that a new page be the home for the updates. This was accomplished by a "time split".

Since each record version has a duration at which it represents the state of the record, and because record updating is normally not coordinated, a split time will usually intersect the record version lifetimes of most records. With the WOB-tree, this led to time splits as of the current time, copying the current versions to a new current node. A further result was that historical nodes were never updated, including historical index nodes, when split in the same way.

The TSB-tree [22] moved the historical node when a current node was time split. The intent was to keep the current data more accessible on, perhaps, a higher performance medium, while storing historical data on a lower cost medium. However, when historical nodes move, ensuring that only the current parent of the node is updated constrains the time at which we can split the parent to times earlier than the begin time of its oldest current child node. Otherwise, updating the child's new historical index term page pointer requires an update to the historical index node. This is very restrictive with respect to split times, and can result in the need for preemptive time splitting of child nodes to enable their parent index node to split at more convenient times.

2) LLAMA Mapping Table: Like the TSB-tree, and unlike the WOB-tree, the TSBw-tree migrates historical data, while leaving the current data on the same high performance medium on which it started. But it does this in a new way. In earlier indexing, the "node name" and its physical location were the same. With the mapping table (see Figure 2), these are separated, with the result that a *historical node does not change its logical PID address, while still being subject to migration of its physical location*. This permits the TSBw-tree to avoid the index node splitting TSB-tree problem where moving a historical node required updating the page pointer of its (historical) index term in a historical index page.

B. Avoiding Insertion of Current Items to Historical Nodes

We need to ensure that split times are chosen such that a *current item (index term or record) never needs to be included in a historical node*. This requires care in choosing split times for both data and index nodes. We discuss this below.

1) Data Nodes: LLAMA is designed to work in a concurrent environment where multiple threads, of unknown performance, update pages. While threads usually have similar execution rates, robust concurrent systems avoid assumptions about execution rate. User updates, with associated timestamps, will be generated and submitted in approximate timestamp order. But the timestamp order of updates at a page may not be monotonically increasing because of concurrent activity. Hence, when splitting data nodes, we need to choose a split time that guarantees there are no unapplied updates with earlier times, which would require updating the historical page.

In the Deuteronomy architecture [17], a transactional component (TC) sends an "end of stable log" (EOSL) control operation to a Data Component (DC), that indicates a low water mark LSN such that all updates with lower LSNs than the EOSL LSN are guaranteed to be on the stable log, and more importantly, that there are *no outstanding updates with lower log sequence numbers*. We can augment EOSL information to include a *low water mark timestamp* EOSL-TS such that there are no updates with lower timestamps than EOSL-TS that remain active and may need to update a data node. Hence, we can use EOSL-TS as the time at which we time split data nodes in the TSBw-tree and be sure that no earlier timestamped updates will present themselves that would need to update a historical node.

Although it is *safe* to choose any split time that is less than EOSL-TS, it is preferred to make the split time equal to EOSL-TS. Splitting with a time earlier than EOSL-TS will not move out as many versions to the historical page. This leads to having more records in the current page after a time split, which speeds the process of filling the page and triggering more splits. Choosing EOSL-TS as the split time will reduce the number of data node splits compared to using an earlier time.

2) Index Nodes: Like record updating, a data node split can be concurrent with other data node splits and with parent index node splits. Thus we cannot be sure that data node splitting, and associated index term posting, complete in monotonically ascending timestamp order. We need to ensure that an index node split time is earlier than the timestamp of any subsequent current index term posted as a result of a lower level time split. This ensures that the index term is added to the current parent index node, not to a historical node. For this we need a low water mark timestamp at each index level L similar to EOSL-TS that guarantees this. This will ensure that active time splits at level L produce index terms that need only be posted to their "current" parent at level $L+1$.

Thus, for each level L of the tree, we track the active time splits. At each level L we maintain a split timestamp ordered list of active time splits for L with list elements <*next*, *key*, *timestamp*, *PID*>. The *next* attribute points to the earlier time splits on the list. The remaining fields are the fields of the index term that the time split will post. An index term must be posted to this list prior to the beginning of its associated time split. We maintain each list using latch-free techniques. New entries, with later timestamps, are prepended to the start of the list with a CAS. When a time split is completed at level L, including the posting of its index term, we remove its entry from the level L list. We do this by marking the entry as deleted, and periodically "consolidate" the list to remove marked elements. Should the list at level L become empty, we remove all its elements but retain in the list header a lower time bound for all subsequent time splits at level L of the tree.

When a time split begins at $L+1$, we choose its split time to be earlier than the time of any active time splits at L. This ensures that index terms for these active level L time splits are all posted to current pages produced by time splits at level $L+1$. Thus, when we time split at level $L+1$, we use, as our split time, $min\{timestamp \| < key, timestamp, PID > in\ L\}$. There can be no future index terms from level L that can have earlier times than this, which is exactly what we need.

V. MANAGING HISTORICAL DATA

A. *Why Archive Historical Data*

A valuable feature of the TSBw-tree is its ability to isolate historical records from current ones. Current records are the ones undergoing more access and manipulation. It is possible to exploit this isolation both to improve access to current data and to reduce the cost of maintaining historical records.

The storage medium for current records might be capacity limited flash storage. And even if it is disk storage, one might want to isolate historical data from current data to reduce access interference in order to improve access to current data. Thus, there will frequently be a performance advantage to using a separate medium for the storage of a growing historical archive.

It is also more cost effective to leverage an alternative storage medium for historical data, a medium that has a larger capacity but possibly lower performance. Using flash for current data and migrating historical data to disk or cloud storage are examples of this paradigm. The choice of the archival medium affects access to migrated historical data.

LLAMA currently manages its log structured store (LSS) with a single multi-buffer circular queue. New writes for nodes go to the "logically" ever increasing end of this queue. At the head of the queue, underlying space is being reclaimed (LFS cleaning) by a relocation-based garbage collector. Overwritten nodes are the garbage that is being reclaimed. However, nodes that are still active need to be relocated (re-written) to the end of the queue.

This scheme, if unchanged (without migration) results in historical nodes, which are accessible but never updated, being continuously re-written, resulting in an unbounded write amplification for historical data. Migration solves this problem directly by moving historical data out of the LSS used for the current data. Thus the TSBw-tree exploits a two level log structured storage. The way that the TSBw-tree manages this two level store results in confining all log structured recovery from system crashes to the LSS containing the current data (LSS-C). The LSS containing historical data (LSS-H) never needs to be accessed during recovery. And LSS-H requires garbage collection only if historical data is being dropped, which might happen but on a very different time scale and where a simpler and cheaper cleaning paradigm can be used (see section V-C).

B. *Moving History Nodes*

1) Safety Problem: Adding archiving functionalities to LLAMA results in two stores that need to be managed. Entries in the mapping table point to pages both in the current storage and historical archive. The content and stability of LSS-C and LSS-H determine the state of the cache/storage subsystem. LLAMA must guarantee that a failure at an inopportune time does not cause a corrupt state after recovery. Migrating a page to the archive will *invalidate* it in the current storage. Garbage collection will reclaim these invalidated pages but will not rewrite their contents in LSS-C. The problem that arises is that an invalidated page might be garbage collected prior to being stable in LSS-H. If that happens, a crash would lose the page while in the middle of its migration.

Our modified LLAMA does not allow garbage collection of a migrated page from the LSS-C unless it is stable in LSS-H and visible stably from LSS-C. A system crash during the midst of migration, and before the archived page is visible stably will result in a state after recovery as if the page were not migrated and is still part of LSS-C. This can lead to wasted space in the archive. This is a cost incurred to ensure no data loss caused by dangling pointers in the mapping table for migrating pages. And the wasted space is minor as crashes are rare.

2) Interfaces: A deployment of another storage medium to preserve data requires changes to the cache/storage layers and their interfaces. The main idea behind the following modifications is to add a migration component that will be responsible for managing historical information. This migration component is exposed in the cache layer interface and will act as a second storage layer "adjacent" to the storage layer for current records.

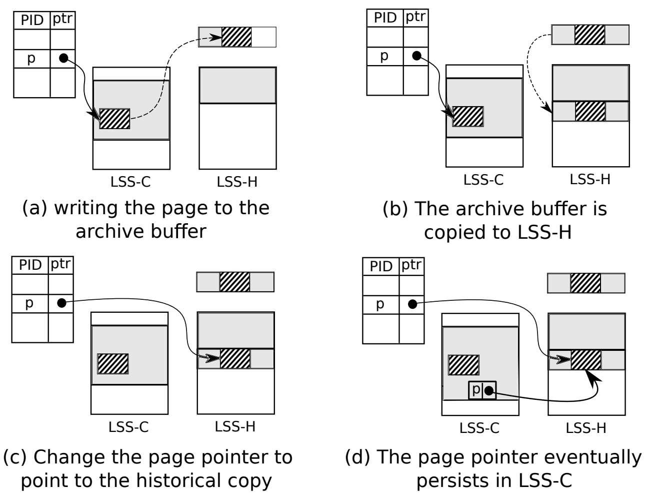
Since LLAMA is not aware of the contents of pages, it is not possible for LLAMA to be responsible for deciding whether pages are to be migrated or not. The migration component itself needs to provide a specific interface at the cache layer to permit the access method to invoke migration as it sees fit. This interface is in the form of Migrate(in-id). This operation is similar to a flush operation that writes the page to the LSS-C but performs the similar function of flushing the page to LSS-H. The in-id identifies the page to be migrated. Flushing a migrated page's state to the LSS-C will store a "flush delta" that indicates that the page has been migrated and include its offset in the LSS-H.

In addition, the functionality of Read(id, out-ptr) needs to be extended to enable reading an archived page. Equivalent operations to Mk-stable and Hi-Stable, but now applicable for LSS-H are also provided.

The mapping table needs to maintain information about the location of the page. An entry in the mapping table can point to a page in main memory, flash storage, or archive. The mapping is to an address if the page is in main memory, to an offset if it is in the current or historical storage. The type of each mapping table entry is identified by a different tag in the "ptr" field.

3) Migration: Migrating a page starts with a call from the upper layers. Then a sequence of events follow for a successful migration. These events are shown in Figure 4. The figure shows the mapping table and the LSS-C and LSS-H. Also, it shows the buffer for the archive. An entry for page P in the mapping table points to the location of the page in either the storage or the archive. The gray area in the LSS-C, LSS-H, and archive buffer are filled with data while the white space is empty or reclaimed in the case of the LSS-C. The sequence of events to migrate the page P is:

1) The page P is read from LSS-C and a consolidated representation of it is written to the LSS-H archive buffer (Figure 4(a)). Requests will be serviced by the page as it exists in LSS-C, as the archiving effort is not yet visible. The LSS-C page P will not be garbage collected because it can still be reached and seen as active via the mapping table entry. At this point, a crash will have no effect on the state as the content



(a) writing the page to the archive buffer

(b) The archive buffer is copied to LSS-H

(c) Change the page pointer to point to the historical copy

(d) The page pointer eventually persists in LSS-C

Fig. 4: A migration example. The solid line is a mapping table pointer and dashed lines represent copying to another location.

of the archive buffer will be lost. However, the state will continue with *P* being in LSS-C.

2) After the archive buffer is filled, its content is copied to the LSS-H (Figure 4(b)). A crash after copying *P* to the archive will cause the state after recovery to continue with *P* in LSS-C, as the archiving process remains invisible. However, the space occupied by *P* in the archive will be wasted space and another attempt to migrate *P* will occupy additional space.
3) *P* is now persistent in the archive. The process that copied the buffer to the archive will update the mapping table to point to the new location of *P* (Figure 4(c)) in LSS-H with a new LSS-H flush delta. This update is guaranteed to succeed since *P* is a historical page and no further data updates are performed on it.
4) The archival flush delta to *P*, which is now stable in the archive, is eventually persisted by being flushed to LSS-C (Figure 4(d)). After that, should an earlier state of *P* in the LSS-C be examined for possible garbage collection, like any other such page, the garbage collector will check whether there is a stable subsequent state for *P*. Once the archival flush delta for *P* is stable, all prior states of *P* in the LSS-C can be garbage collected. At this point, the migration of *P* is complete and will survive system crashes.

C. Archive Garbage Collection

The archive contains only historical pages that are never updated. This obliterates the need for LSS-H garbage collection since no space in the LSS-H can be reclaimed because of subsequent updates. However, it can be desirable to allow discarding data that exceeds an age threshold. In this case, the space occupied by historical nodes can be reclaimed in the LSS-H in a fairly simple circular fashion.

Each historical node is a result of a time split. The split time represents an upper bound on the validity ranges of all record versions in the page. A page in the archive can be reclaimed if its split time is lower than the time threshold. To mark the page for garbage collection we might set the pointer to that page to null in the mapping table and use the null pointer to "deter" the access. This "page update" in the mapping table, like the pointer update for the page in the mapping table during archiving, would enable garbage collection once the update is stable in the LSS-C.

However, an access to an invalid page does not necessarily need to access the mapping table to check whether a page id has a null pointer. An access to such a page from the TSBw-tree index can be aware of the age threshold. Then, accesses to versions with lifetimes that end before the time threshold are treated as invalid accesses. So all we would need to do is set the threshold and examine it before making an access. This has three advantages. (1) We do not need the mapping table entry for the archive page to persist and serve as a "tombstone" for an invalid page. Hence we can reclaim the mapping table entry. (2) While we need to persist the threshold, that is global information that can "invalidate" multiple pages. (3) Garbage collection can simply advance by discarding all pages with split times earlier than the threshold. And, because pages are written to the archive in approximate timestamp order, it can simply pause GC until every page that it encounters is garbage.

VI. Performance

In this section, we evaluate the TSBw-tree by comparing its performance to the Bw-tree. Our purpose is to illustrate the performance difference incurred to maintain historical versions. The effect of migration on the overall performance is also examined by experiments enabling and disabling writes to LSS-C and/or LSS-H.

A. Implementation and Setup

Implementation. LLAMA is written in C++ code and consists of approximately 12,000 lines of code. The Bw-tree is implemented on top of LLAMA with approximately 4,000 lines of code. The TSBw-tree is implemented by applying the necessary changes and additions to an implementation of the Bw-tree to support multi-versioning and migration. These changes added approximately 4,000 lines of code. Performing the CAS operations is done through the Windows InterlockedCompareExchange64.

Experiment machine. Our experiment machine is an Intel Xeon W3550 (at 3.07 GHz) with 24 GB of RAM with a 160GB Fusion IO flash SSD drive. The machine has four cores that we hyperthread to eight in all of our experiments.

Workload. The experiments in this section are performed with a synthetic workload. The size of keys and values is 8 bytes each. Prior to the experiment the index begins with an initial population of 1M entries generated using a uniform random distribution. The workload issues insert, update, and read operations. The proportions of these operations are fixed for each experiment. The workload for each experiment issues a total of 10.24M operations.

Defaults. The workload is served by 8 worker threads which is equivalent to the number of hyperthreads. The default page size is 4KB. A time split is triggered if at least a third of the page is comprised of historical records when the page is full. Unless otherwise mentioned, data is written to flash and migration is enabled. No garbage collection is performed on

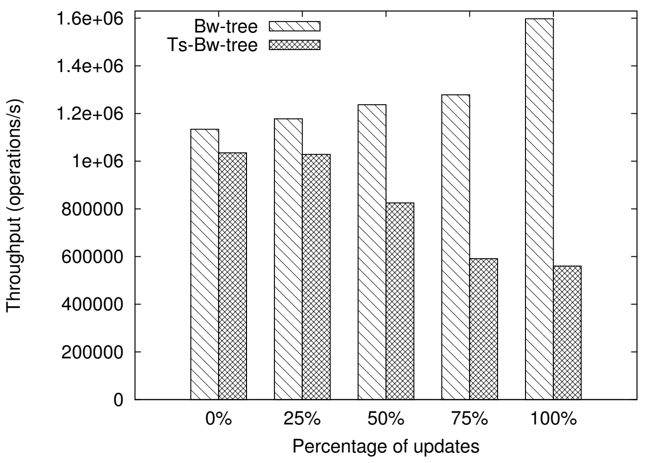


Fig. 5: Performance overhead while varying the percentage of updates in a workload with updates and inserts only

the archive. Page consolidations for both the TSBw-tree and Bw-tree are done after ten or more deltas are prepended to a base page.

B. TSBw-tree Performance

Updates to the TSBw-tree result in adding new versions of existing records. This results in increasing the index size since it indexes both current and historical records. A larger index may lead to a performance penalty because of increased pressure on the cache and, if the number of tree levels increases, the need to traverse an extra node. However, the interior nodes of the TSBw-tree are a very small fraction (approximately 1%) of the overall file size, so the effect is minor.

Additionally, a record is eventually written to the archive at least once, as part of an archived page. However, it is possible that it is written multiple times because a record might exist in both the historical and current nodes after a split, thus leading to more than one copy of the same version. There are other sources of overhead in the TSBw-tree that affect performance to a lesser degree. These include the overhead of manipulating a compound key consisting of both a key and a timestamp. The experiments described below show the cumulative performance cost of supporting multi-versioned data.

1) Synthetic workload with no reads: The first set of experiments run a workload that consists of inserts and updates only, *i.e.*, no read operations. The percentage of updates to inserts is varied to shed light on the difference in cost between updates in the TSBw-tree and the Bw-tree. Figure 5 plots the results of these experiments for both the TSBw-tree and the Bw-tree. The throughput in operations per seconds (*ops/s*) is shown for different updates to inserts ratios.

Consider first the case of all inserts. The Bw-tree achieves 1.133M *ops/s* and the TSBw-tree achieves 91% of that value. Here, the TSBw-tree is isolated from the write overhead caused by historical records since no updates are performed. The reported numbers differ because there is extra logic needed for the TSBw-tree to check versions, and the versions themselves have compound keys.

As the fraction of updates in the workload is increased, the performance of the Bw-tree improves. A workload with half updates and half inserts results in a throughput 9% higher than the workload with all inserts. A scenario with only updates achieves a throughput that is 40.9% higher than the scenario with all inserts. Unlike inserts, updates in the Bw-tree do not contribute to the size of the data and hence page splitting does not occur. Inserts lead to an increase in the size of the data, the more inserts the greater the index size. This explains the performance decrease observed for the Bw-tree.

Increasing the fraction of updates to inserts in the TSBw-tree leads to poorer performance as additional time splitting introduces duplicate versions and these versions are archived. The performance penalty for increasing updates from a workload with no updates to a workload with half updates and half inserts is a decrease of 20.3%. This penalty is 45.9% when going from a workload with all inserts to a workload with all updates.

Workloads with a higher update ratio produce a greater performance difference between the TSBw-tree and the Bw-tree. The TSBw-tree achieves 87.3% of the throughput of the Bw-tree for a workload with 25% updates. For the extreme workload with all updates, the TSBw-tree's throughput is 35.0% of that achieved by the Bw-tree.

2) Results Discussion: Here, we will focus our discussion on two workloads, pure inserts and pure updates. The nature of these pure workloads allows us to deduce interesting information about the relative characteristics of the TSBw-tree in comparison to the Bw-tree. These deductions are first presented and then verified with the experimental results.

Pure inserts. A key characteristic of the workload with pure inserts is that there are no historical records in the TSBw-tree. All splits are key splits. The insert operations in this case contribute to the index in an identical way for both the Bw-tree and the TSBw-tree. This leads to the observation that the number of key splits for both systems is similar. The Bw-tree experienced 120353 key splits and the TSBw-tree experienced 120003; a difference that is less than 1%. No historical records also means that nothing is written to the archive and that each inserted record occurs only once in the tree, i.e., it is not written again later to the archive. This is reflected by the similar final LSS-C sizes of 598MB for the Bw-tree and 587MB for the TSBw-tree; a 2% difference[1].

Pure updates. A pure update workload does not change the Bw-tree size. On the other hand, the TSBw-tree creates new versions of existing records and thus increases in size. A sufficiently long experiment will observe almost all splits to be time splits for the pure updates workload. The time split results in two pages: a full historical page and a new current page two thirds the size of a full page (the threshold for a time split is filling at least one third of the page with historical records). Thus, the time split will cause the historical full page to be written to the archive. Also, it will cause the historical full page *and* the new current page to be written to LSS-C. This makes time splits write 1.66 pages in LSS-C for each page written in LSS-H. To verify this we took the ratio of the size

[1]We controlled for key size by making the Bw-tree key size be equal to compound key size used in the TSBw-tree. If we had not done that, the TSBw-tree file would have been even larger than the Bw-tree file.

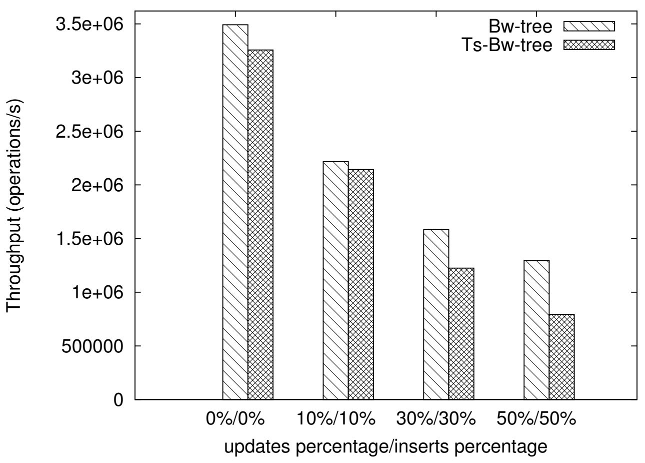


Fig. 6: Performance overhead while varying the percentage of updates and inserts

of LSS-C to the size of LSS-H. We subtracted the original sizes of LSS-C and LSS-H that were present after we initially populated the trees and before we ran the experiments for the ratio calculation. The resulting size ratio of LSS-C to LSS-H is 1.61.

3) Synthetic workload with reads: The workload used in this second set of experiments consists of update, insert, and read operations. For each experiment in the set the ratio of updates is equal to the ratio of inserts. The rest of the operations are reads. For example, a 20% updates/20% inserts workload will consist of 60% read operations. We plot the throughput in operations per second in Figure 6.

Consider first the read-only case. The throughput of the TSBw-tree is 93.3% of that achieved by the Bw-tree. The TSBw-tree reduced throughput is caused by the additional logic and manipulation of a compound key. Introducing inserts and updates to the workload with 10% each results in decreasing the throughput by 36.5% for the Bw-tree and 34.3% for the TSBw-tree. This performance hit is similar for both systems. However, as the ratio of updates and inserts increase to 30% and 50% each, the throughput of the TSBw-tree decreases more rapidly than the Bw-tree. The throughput of the TSBw-tree is 77.3% of the throughput of the Bw-tree for 30% updates and inserts, decreasing even more for the half updates and half inserts workload to 61.9%. The 50% experiment no longer contains reads and produces, in Figure 6, the same result as the 50% case in Figure 5. Essentially, we are simply seeing the reads diluting the performance differences already revealed in the initial experiment with no reads, with a slight variation at the 10% case that might be due to caching effects between reads and modification operations.

It should be noted that in the above results, we did not exploit compression of the sort used in the Immortal DB effort [19]. Immortal DB found that "delta" compression of historical versions on a page basis, where the most recent version on a page is uncompressed was highly effective for two reasons: (1) it reduced the size of the data being stored, hence reducing the index size as well; (2) it reduced the number of page splits. This would go a long way toward shrinking the throughput decline that we see with increased updates.

Scenario	Throughput (Mops/s)	Normalized
In-memory	2.47	1.00
Write to flash	2.17	0.88
Flash and archive	2.14	0.86

TABLE I: Overhead of persistence and archiving

4) Migration and Storage Costs: The experiments so far were persisting the data in flash and migrating historical pages to the archive. Now we examine the overhead of persistence and migration to the performance of an in-memory TSBw-tree. The TSBw-tree was run three times: (1) in-memory with no persistence to flash and no migration; (2) with persistence to flash but no migration; and (3) with both flash persistence and migration. The workload used for this scenario consisted of 10% inserts, 10% updates, and 80% reads.

Table I shows the throughput in the three scenarios. The rightmost column normalizes the throughput to the throughput of the in-memory case. The in-memory TSBw-tree executes at 2.47 million operations per second. Persisting data to flash causes the throughput to decrease to 2.17 Mops/sec, 12% lower than the in-memory case. Note that although migration is not enabled, time splits still occur and the resulting historical pages are written to LSS-C in addition to the current pages. When migration is enabled the historical pages will be written to LSS-C *and* to LSS-H (the archive). This leads to a throughput of 2.14 Mops/sec. This is 86% of the throughput of the in-memory case. Turning on archiving decreases the performance by only 1.8% when compared to the case when data is persisted to the flash.

This archiving overhead result is very re-assuring. It means that removing historical pages from the current database essentially preserves the indexing performance of the TSBw-tree. And it makes it possible to move historical data to a lower cost hard disk, and exploit flash storage for the current data. This permits a much larger current database size to experience the performance benefits of flash storage, and without giving it back with overhead increases due to the archiving itself. Further, an important rationale for archiving is to avoid having to repeatedly need to re-write historical versions in LSS-C during garbage collection. Migration succeeds in avoiding this extra garbage collection while incurring only modest archiving overhead.

VII. Related work

A. Temporal and Multi-version Databases

Support for temporal data became a mainstream database research topic in 1987 with its inclusion as a fundamental feature of Postgres [29], [30]. In Postgres, records are timestamped with clock time and reflect the serialization order of transactions. A B+-tree maintains current records, with historical records moved from the current B+-tree to a separate access structure via a background "vacuuming" process. An R-tree [8] is responsible for indexing historical records. Managing historical records in an R-tree was a bit awkward, and querying frequently required access to both current and historical indexes.

In the early 80's, coincident with database research community temporal interest, multi-version support began to appear in commercial databases. Oracle was an early supporter of multi-version concurrency control, supporting an isolation level now referred to as snapshot isolation. This introduced transient versions that permitted much greater concurrency by removing read-write conflicts, at the sacrifice of strictly serializable execution. DEC (now owned by Oracle) Rdb [9] introduced transient versioning to support read-only transactions, which could execute without interfering with current serializable updates.

While commercial systems started with transient versions restricted to enhancing concurrency, this evolved into more persistent versioning with more explicit temporal support. FlashBack [1] is the name of the transaction time functionality that was announced with Oracle 9i and extended in Oracle 10g. It allows accessing previous states in the database. Point-in-time recovery is supported by rolling back to a previous state prior to bad transactions. Historical records are not part of the index structure. This makes any access to historical data pass through the current versions first and then travel backward in time.

B. Temporal and Multi-version Indexing

Indexing of multi-version data began in the mid 1980's, as a matter of necessity. The write-once B-tree (WOB-tree) [6] was designed to store data on Write-Once Read-Many (WORM) optical disks. The WOB-tree indexes both the keys and the versions. It does both key splits and time splits. Because of the write-once media, it is the current data that must be relocated during a time split. Time splits permit the writing of new data to current nodes. A by-product of this is to keep the current data in a small number of nodes, allowing more efficient access to them. The multiversion B+tree [4], which is similar, also moves current data during its time split, which is done in such a way as to guarantee a lower bound on page utilization for any time-slice it stores.

The TSB-tree [22] is designed to exploit high performance storage for current data and cheaper, perhaps less performant storage for historical data. Thus, the TSB-tree migrates historical records to the less expensive storage medium. This is done by its time split, which moves historical data to a new page. The major difficulty with the TSB-tree is that the historical part of a current page is relocated, and the historical index term produced during the splitting of a current page requires updating. Thus any historical index page (produced as a result of time splitting an index page) must not include current index terms if it is to keep historical pages read-only. The TSBw-tree avoids this complication because it relocates historical pages without changing their index terms.

C. Exploiting Modern Hardware

Starting with AlphaSort [25] the database community has increasingly paid attention to the characteristics of modern hardware, initially the memory hierarchy, then in the past several years, to multi-core processors and flash storage. A new collection of techniques were developed to improve the performance of database systems. An interesting characteristic of focus on performance is that to achieve really big gains requires looking at the entire software stack, from user input through processors and memory hierarchies to secondary storage.

Much of the performance work was in the context of main memory database systems [11], [13], [26], and it has very quickly been adopted in commercial database products [5], [7], [31]. A focus of this work was on removing latch overhead, done mostly by partitioning the data such that only a single thread would access data. Hekaton was unique in adopting multi-threading latch-free technology, and the Bw-tree [18] was the key range index used in Hekaton.

Another area pursued in the quest for higher performance was exploiting flash storage as the stable storage medium. Because flash requires erasing before write and is wear limited, flash vendors supply a flash translation layer (FTL) to relocate and spread writes over the raw storage. To cope with the characteristics of flash, techniques like "in page logging" [14], parallelism within the flash technique have been introduced. Finally, even though the FTL usually uses log structured techniques, LLAMA [16] have found that "application level" log structuring has a big performance payoff.

It is LLAMA, with its latch-free approach, coupled with its log structuring of storage that are the enablers for the TSBw-tree's great performance.

VIII. Conclusion

We have designed and implemented the TSBw-tree, a temporal index with higher performance than has been reported elsewhere. The techniques that we used demonstrate that temporal indexing can be done effectively at very high performance. The TSBw-tree owes this performance to its exploitation of LLAMA, the latch-free, log structured cache/storage subsystem [16] that has been used previously in support of the Bw-tree.

Our implementation of the TSBw-tree required us to build technology to deal with a number of issues.

1) To migrate historical data, as done with the TSB-tree, we enhanced LLAMA functionality to include a migration capability. This permitted us to exploit inexpensive media (e.g. disks) for cold historical data while exploiting high performance media (e.g. flash) for hot current data.
2) Migration further solved a problem observed originally in LFS [27], i.e. that cleaning efficiency can be a substantial cost if data is not partitioned. Thus, cleaning efficiency is greatly enhanced when cold data (in an update sense) is separated from hot data.
3) We further sketched a way in which it is possible to "clean" the archive very efficiently when the oldest parts of it are no longer needed. This cleaning required merely dropping, in a careful manner, historical pages containing data that is no longer needed.
4) Because LLAMA has been structured to maximize concurrent execution, it is essential to handle time splits concurrently with on-going updating. This led us to exercise great care in choosing timestamps for time splits, so as to ensure that the historical page never needed further updating.

In building the TSBw-tree on top of LLAMA, we achieved great performance, while simultaneously solving a TSB-tree problem caused by the moving of historical nodes. Using

LLAMA meant that we could move historical pages without changing their index terms, due to the relocation possible through the indirection enabled by LLAMA's mapping table.

Finally, our TSBw-tree implementation is like a first version of a product. There are ways to further enhance both its performance and its storage efficiency. The "low hanging fruit" is to exploit the historical version compression technique used in Immortal DB [19]. This reduces the storage footprint of historical data, and improves performance by reducing, especially in a high update setting, the number of time splits needed to accommodate historical versions.

References

[1] Oracle Flashback Technology. http://www.oracle.com/technology/deploy/availability/htdocs/Flashback_Overview.htm, 2005.

[2] Oracle Total Recall. http://www.oracle.com/technology/products/database/oracle11g/pdf/flashback-data-archive-whitepaper.pdf, 2008.

[3] R. Agrawal, R. J. B. Jr., C. Faloutsos, J. Kiernan, R. Rantzau, and R. Srikant. Auditing compliance with a hippocratic database. In *VLDB*, pages 516–527, 2004.

[4] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. In *VLDB*, volume 5, pages 264–275, 1996.

[5] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *SIGMOD*, pages 1243–1254. ACM, 2013.

[6] M. C. Easton. Key-sequence data sets on indelible storage. *IBM Journal of Research and Development*, 30(3):230–241, 1986.

[7] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. Sap hana database: data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2011.

[8] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57. ACM, 1984.

[9] L. Hobbs, I. Smith, and K. England. *Rdb: a comprehensive guide*. Digital Press, 1999.

[10] C. S. Jensen and R. T. Snodgrass. Temporal data management. *IEEE Trans. Knowl. Data Eng.*, 11(1):36–44, 1999.

[11] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[12] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206. IEEE, 2011.

[13] A. Kemper and T. Neumann. Main-memory database systems. In *ICDE*, pages 1310–1310. IEEE, 2014.

[14] S.-W. Lee and B. Moon. Transactional in-page logging for multiversion read consistency and recovery. In *ICDE*, pages 876–887. IEEE, 2011.

[15] P. L. Lehman et al. Efficient locking for concurrent operations on b-trees. *ACM TODS*, 6(4):650–670, 1981.

[16] J. Levandoski, D. Lomet, and S. Sengupta. Llama: A cache/storage subsystem for modern hardware. *VLDB*, 6(10):877–888, 2013.

[17] J. J. Levandoski, D. B. Lomet, M. F. Mokbel, and K. Zhao. Deuteronomy: Transaction support for cloud data. In *CIDR*, volume 11, pages 123–133, 2011.

[18] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In *ICDE*, pages 302–313. IEEE, 2013.

[19] D. Lomet, R. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Immortal db: transaction time support for sql server. In *ACM SIGMOD*, pages 939–941, 2005.

[20] D. Lomet, Z. Vagena, and R. Barga. Recovery from bad user transactions. In *ACM SIGMOD*, pages 337–346, 2006.

[21] D. B. Lomet, R. S. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Transaction time support inside a database engine. In *ICDE*, page 35, 2006.

[22] D. B. Lomet and B. Salzberg. Access methods for multiversion data. In *ACM SIGMOD*, pages 315–324, 1989.

[23] S. Mitra, M. Winslett, R. T. Snodgrass, S. Yaduvanshi, and S. Ambokar. An architecture for regulatory compliant database management. In *ICDE*, pages 162–173, 2009.

[24] C. Mohan and F. E. Levine. Aries/im: An efficient and high concurrency index management method using write-ahead logging. In *ACM SIGMOD*, pages 371–380, 1992.

[25] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. Alphasort: A risc machine sort. In *SIGMOD*, pages 233–242, 1994.

[26] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. Plp: page latch-free shared-everything oltp. *VLDB*, 4(10):610–621, 2011.

[27] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM TOCS*, 10(1):26–52, 1992.

[28] R. T. Snodgrass. The temporal query language tquel. In *PODS*, pages 204–213, 1984.

[29] M. Stonebraker. The design of the postgres storage system. In *VLDB*, pages 289–300, 1987.

[30] M. Stonebraker, L. A. Rowe, and M. Hirohama. The implementation of postgres. *IEEE Trans. Knowl. Data Eng.*, 2(1):125–142, 1990.

[31] M. Stonebraker and A. Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.