Improving Transaction-Time DBMS Performance and Functionality

David B. Lomet[#], Feifei Li[*]

[#]*Microsoft Research*
Redmond, WA 98052, USA
lomet@microsoft.com

[*]*Department of Computer Science*
Florida State University
Tallahassee, FL 32306, USA
lifeifei@cs.fsu.edu

***Abstract*— Immortal DB is a transaction time database system that is built into a commercial database system rather than being layered on top. This enables it to have performance that is very close to the performance of an unversioned current time database system. Achieving such competitive performance is essential for wide acceptance of this temporal functionality. In this paper we describe further performance improvements in two critical dimensions. First Immortal DB range search performance is improved for current time data via improved current version storage utilization, making this performance essentially the same as unversioned performance. Second, Immortal DB update performance is increased by further reducing the cost for the timestamping of versions. Finally, we show how a simple modification, integrated into the timestamping mechanism, can provide a foundation for auditing database activity. Our algorithms have been incorporated into a commercial database engine and experiments using this database engine demonstrate the effectiveness of our approach.**

I. Introduction

Transaction time database systems have been widely discussed [9, 31]. They provide access to both current and previous database states by the mechanism of creating new versions of data for every transaction, as opposed to doing update in place. Transaction time databases have many important applications and are of increasing interest for auditing, legal compliance, trend analysis, etc.

We have built our Immortal DB transaction time database system [12, 13, 14] into the kernel of a commercial database management system, SQL Server using the TSB-tree [15, 16] to index both current and previous database states. Our objective is to ensure transaction time performance close to that of an unversioned current time database. We also introduced additional functionality to the basic transaction time functionality to provide a further incentive to providing transaction time functionality, e.g., we exploited transaction time versioning to provide recovery from bad user transactions [20]. This work continues with both these threads, i.e. we introduce techniques to further improve Immortal DB performance to a level that is *almost identical* to the unversioned databases, and we add additional functionality to Immortal DB as a further incentive for providing transaction time support.

A. Performance Improvements

Reading a record in the current database will usually not differ much from reading it in a transaction time database. An index, whether it is a B-tree indexing current (single version) data or an MVB-tree [2] or Time-Split B-tree (TSB-tree) [15, 16] indexing transaction time (multi-version) data, will access a data page in logarithmic time, and, should the multi-version tree be a deeper tree because of the multiple versions, the higher level root is most likely in main memory in any event and no extra I/O access will be required. So this performance measure is already satisfactory.

More problematical is the performance of updates and range reads. We discuss these briefly below, and highlight what we have done to address performance in these areas.

1) Update Performance: When a record is updated, one obvious difference is that each update creates a new version. That eventually means that pages fill up faster, and require splitting after fewer updates. We do version compression [14] to reduce this impact, but there is little else that can be done to avoid this as versioning is intrinsic to transaction time support. However, there are multiple updates between page splits in most cases, so this splitting cost is amortized across these multiple updates, which greatly reduces the per update cost.

There is, however, another extra cost in our updating that is more controllable. Each version needs to be timestamped so that we can determine, on subsequent reads, whether the version is relevant to an "as of" read request. Because the version timestamp is not known until a transaction commits so as to keep timestamps consistent with serialization order, this timestamping requires a second "touch" of the record. Doing this second touch at very low cost is essential to making update cost competitive.

We have previously described "lazy" timestamping techniques with modest overhead [13]. Our "lazy" timestamping requires that we retain the mapping from transaction id to timestamp in a persistent table. Here we further reduce the timestamping overhead by leveraging the log to temporarily provide persistence for this mapping.

2) Range Performance: This cost in a B-tree is primarily determined by how many pages need to be read to process the request. Thus, range performance for a versioned "as of" request, to a first order, depends upon the density of the versions on a page that were current at the "as of" time (what we refer to as the single version utilization). Version compression substantially improves upon this single version utilization (*SVU*) for any version, historical or current. However, even with version compression, there is room for further improvement.

While we would like to improve single version utilization for all versions, the utilization for the current version is by far the most important; we anticipate that current version reads will be much more common than historical reads. To that end, a new page splitting strategy is introduced that we call deferred splitting. Deferred splitting improves significantly the utilization for the current version, what we call the single version current utilization (*SVCU*), so that the combination of version compressing and deferred splitting results in single version current utilization that comes very close to the utilization seen in B-trees supporting only current data. And this represents a significant improvement over prior splitting regimes.

B. Functional Enhancement

Transaction time databases will retain all versions of the data. Indeed, this is what it means to support transaction time. We can exploit this basic versioning capability to provide value added services. Previously we had used the versions as a way to recover quickly from user transactions that were erroneous [20]. The basic versioning, together with our timestamping technique, can also be used to provide the underpinnings of database auditing, that is, tracking who is responsible for the updates whose versions appear in the database.

With the emergence of Sarbanes-Oxley requirements, the ability to audit database activity has assumed greatly increased importance [1, 22]. The invaluable thing that a transaction time database supporting an audit capability makes possible is a direct linkage between a user executing a transaction and the version of data records that were updated by the transaction. Such a capability greatly enhances the value of the versions being retained in the transaction time database.

C. Our Contributions

This paper introduces significant improvements to the Immortal DB transaction time database system. Our intent is to increase the desirability of transaction time functionality versus an unversioned database, paving the way for wide acceptance of this transaction time functionality in practice. Our contributions in this work are summarized as follows.

1. We improve update performance via batch updates to the timestamp table used for ensuring that timestamping the record versions can survive a system crash. This consolidates many singleton writes to this table into a small number of batch writes and enables us to actually reduce the total number of entries posted. This is described in section II.
2. We improve range query performance by improving single version current utilization via deferring the key splitting of pages. This is described in section III.
3. We add auditing functionality to our transaction time database system to enable tracing who was responsible for changes to the data. This is described in section IV.

The paper describes related work in section V, and ends with a short discussion in section VI.

II. Timestamping and Updates

A. The Problem

Uniquely identifying versions is easy. One can tag them with a transaction identifier at the time of an update, such that every update of the transaction receives the same tag. The problem arises that the transaction identifier (usually a monotonically increasing transaction sequence number or TSN) by itself only differentiates one transaction's versions from another's. It does not tell us directly which version an "as of" query should see. For that, one needs to be able to relate the "tag" on the version to the serialization order of the transaction. When a query is asked "as of" some point in the serialization order, the correct version of a record needs to be determined. This correct version is the last version in the serialization order that is at or earlier than the "as of" request time. One usually wants, in addition, to be able to relate points in the serialization order to "wall clock time" so that a query can be asked "as of" some user understood time. When viewed this way, version tagging becomes version "timestamping". Identifying the correct version then involves finding the version with the latest timestamp earlier or equal to the as of time of the query.

B. Previous Solutions

There have been a number of ways proposed for doing timestamping in support of transaction time databases. They fall into three broad categories.

1) Timestamp Order Concurrency Control: Choose the timestamp for a transaction at the time that a transaction starts execution, or when it makes its first update. Then, one can tag the versions created by the transaction's updates with the already chosen timestamp. This is very simple. Unfortunately, when choosing a timestamp this early, the transaction serialization order may not agree with timestamp order. When an active transaction's timestamp does not agree with its serialization order, the transaction is aborted. This is called timestamp order concurrency control [4]. We know of no system that uses this strategy because of concerns about the frequency of aborts.

2) Late Choice Timestamps: The other methods of timestamping all involve choosing the timestamp when the transaction commits. When using the common concurrency control protocols like strict two phase locking, the commit order is consistent with the serialization order. So choosing the time then will result in a timestamp that agrees with serialization order. Note, however, that this is at the end of the transaction, after an update of the transaction has created a new version of a record. Thus, we must re-visit the version to provide it with a timestamp. The other approaches all involve this second visit.

Eager Timestamping: Update all versions of records created by the transaction with their timestamp using a normal transactional update (one that does not create yet another version), and log this as an update of this transaction. Do this while the transaction is active, and then commit as usual. This exploits existing database mechanisms. Timestamping becomes simply a subsequent update within the transaction. It is just like any other update that changes the same version of a record a number of times. The logging and recovery are entirely conventional. This was the first Immortal DB timestamping technique [13].

While simple, and minimizing the need for new mechanisms, this approach has undesirable execution, bookkeeping, and concurrency control impacts. Treating timestamping as updates can double the number of updates in a transaction, doubling also the number of log records for the transaction. It also requires that we maintain a list of all records updated by the transaction so that we can later find and timestamp them. Finally, all these actions are within the transaction, meaning that when using strict two phase locking, the locks are held for an extended time, impeding concurrency. Our very first implementation within Immortal DB used eager timestamping because of its minimal new mechanism, but we found its cost to be unacceptable.

Lazy Timestamping: Instead of doing the timestamping within the transaction, we can do it later. Obviously, we need to do the timestamping prior to the record versions being needed for a query, or indeed, given the way we maintain the historical data, prior to the time that versions are moved to historical pages of the database. But we can wait until that subsequent visit to timestamp the version. We have explored a number of lazy techniques in our effort to find the lowest cost one [13].

The lazy techniques all require that versions created by a transaction's updates be initially tagged with a transaction identifier (we use a transaction sequence number or TSN). The TSN is then associated with a timestamp TS, which is a time or a reliable proxy for time, that is chosen when the transaction commits. This association must be made persistent as part of the transaction so that the timestamping activity can be completed even if the system should crash. Then, as versions are accessed, should the system find a version with a TSN tag, this tag is translated to a TS that is then used to identify whether the version is relevant to a query. Thus, one advantage of lazy techniques is that the timestamping is piggybacked on a subsequent access to the version.

3) Timestamping Issues: There are two primary questions that need to be answered for any lazy timestamping technique. There are too many variations to fully discuss all combinations. Rather than discussing combinations, we discuss possible answers to these questions independently, and then present our latest proposal, briefly contrasting it with our previous approach.

1. Mapping Stability: How is the mapping from TSN to TS stored stably at least until the timestamping for a transaction is complete?
2. Garbage Collection: Do we make efforts to remove mapping information that we no longer need so as to minimize the storage, and perhaps the maintenance costs, of this information?

Mapping Stability: Two techniques have been previously suggested.

1. Define an ordinary (and hence stable) table that contains the mapping information. Such a table can be made accessible by TSN key, hence speeding the translation process. Usually, recent mappings are retained in a main memory cache. This is the approach that we used earlier in Immortal DB [13] and in Postgres [30]. It is useful to keep the table small so that main memory caching of its entries is effective, so garbage collecting entries is useful.
2. Maintain the mapping table in main memory, with the information made stable by including it also in log records on the recovery log [18]. So long as the table does not become too large, this approach can be quite efficient, as writing a log record has less overhead than updating the table in approach 1. However, to keep costs under control, it is imperative that the table be kept modest in size as this approach requires that the table be copied forward in the checkpointing process to ensure that mapping entries can survive the checkpointing induced truncation of the recovery log. Hence this approach must garbage collect entries that are no longer needed.

Garbage Collection: It is useful to keep the mapping information modest in size in both the above approaches by discarding mapping entries that are no longer needed for timestamping, i.e. the timestamping for the transaction is complete. Three alternatives have been suggested.

1. Do not perform garbage collection. The mapping table grows as transactions are committed. A small active part of the table is kept in main memory to speed the translation of TSN to TS. This method can work when the mapping table is an ordinary table. But it is inappropriate for the log based table approach, as the table quickly becomes too large to maintain in main memory, and it becomes too expensive to copy it forward in checkpoint information when the log is truncated.
2. Perform stable reference counting to garbage collect the mapping entries for which the timestamping activity is complete. This has been suggested with the log based approach, where it is essential to keep the mapping

information small because of the need to "forward" the mapping table across checkpoints. Salzberg proposed "stable" reference counting [24]. It requires writing log records to document the timestamping so as to be able to stably decrement reference counts. This, of course, adds to the cost.

3. Perform volatile reference counting to garbage collect most mapping entries for which the timestamping activity is complete. Transactions whose timestamping is incomplete when the system crashes will lose their reference counts. We will not be able to garbage collect their entries. However this failure occurs only when the system crashes, which is a rare event. So the mapping table grows very slowly. We used this approach previously in Immortal DB [13], together with an ordinary table to provide mapping stability.

C. *Prior Immortal DB Approach*

We now provide a more complete description of what we did previously with Immortal DB. We stored the mapping information in an ordinary table that we call the persistent timestamp table (PTT). The PTT is updated as part of every transaction, so it is guaranteed to be stable. We performed volatile reference counting, the least expensive way we know, to provide garbage collection for this information. Unlike Salzberg's approach [24], no logging of the reference counting is done. Because of this, a system crash will lose track of the reference counts and some entries in the PTT will not be garbage collected even when their timestamping is complete. But this is a good trade-off. One gets very low cost reference counting and a PTT that grows a bit in size when the system crashes.

Because the reference counting and timestamping is volatile (not logged), we need to know that the pages containing the timestamps for a transaction are all stable before we delete the transaction's mapping entry from the mapping table. We did that using the checkpointing process. This exploits log sequence numbers (LSNs) which order the log records. A checkpoint identifies an LSN (a point on the log) as the redo scan start point LSN. Once this is greater than the end-of-log LSN (EOL LSN) at the time that the timestamping was completed and the page was marked as dirty, the page containing the timestamps has been written to disk. This must be true for the checkpoint to operate correctly and only truncate log entries that are no longer required. At that point, we can delete the PTT entry for the transaction.

We believe the Immortal DB prior approach, exploiting volatile timestamping, is more efficient than any competing approach. However, it does raise costs, particularly for short transactions that update one or a small number of records. *Every* update transaction requires an insert of the TSN:TS mapping entry to the PTT, and eventually its deletion, both logged. We wanted to avoid this overhead.

D. *Our New Approach*

To reduce overhead, we focus on removing the need to update the PTT in every transaction. This update always requires writing a log record for the update, and eventually requires the writing of the page of the PTT containing the record. In our experience, this adds about 60% overhead to the cost of executing a transaction that updates exactly one record. (Updating the PTT means that a "one user update" transaction is doing two updates.)

To avoid the PTT update, we combine elements from the two schemes described above for making the mapping information stable. We temporarily make a transaction's TSN:TS entry stable by including it in the commit record for the transaction, as in [18]. At checkpoint intervals, we execute a system transaction that updates the PTT with a batch of inserts of mapping items. Updating the PTT in a batch is much more efficient than updating it during each transaction. In addition, we exclude from the batch all transactions for which the timestamping is complete and the record versions with their timestamps are stable. Thus, for these transactions we save both the insert and the delete of the PTT.

In all cases, the reference counting is kept volatilely (volatile RCNT) in a volatile timestamp table (VTT) until it is no longer needed. The VTT acts as a cache for the PTT, and so it contains the TSN:TS mapping as well as the count of the number of records not yet timestamped (in a way that we will explain next), and the end of log LSN (EOL LSN). Figure 1 shows the format of the original VTT and our new VTT. Orchestrating the transition from having the TSN:TS mapping temporarily stored in the commit record and the VTT to being persistently stored in the PTT involves subtle considerations with respect to the reference counting and when the timestamping for versions is known to be stable.

Original VTT

TSN TS (volatile RCNT) (EOL LSN)

New VTT

TSN TS (stable RCNT)

Figure 1: Formats for the Volatile Timestamp Table.

In earlier Immortal DB implementations, we decremented the reference count for a transaction's VTT entry as soon as TS replaced TSN in the record version in the database cache. Thus, we tracked the remaining records not yet timestamped in volatile memory. We then used checkpointing information to determine when the pages in the database buffer are known to be stable. Unfortunately, this gives us the stability information later than we need it for gaining the maximum benefit.

The problem with this former technique is that we know the timestamping to be stable only AFTER the checkpoint, which is too late for a strategy that can entirely avoid posting the TSN:TS mapping to the PTT if it is known that the timestamping is stable **prior to the checkpoint completion**. We need to have a "seamless" story in which the TSN:TS mapping is always stable before all timestamping is stable for the versions in database pages. That is, the TSN:TS mapping for a transaction must be either (1) in one or both of the stable (and accessible) part of the log or the PTT, or (2) stable in all pages updated by the transaction. Thus, we need to know the updated pages are stable *as soon as they are stable*, i.e. before the checkpoint process truncates the log. Otherwise, we have to insert the TSN:TS mapping entry into the PTT.

To know that timestamping information is stable as early as possible, we maintain our reference count in the VTT based on timestamping that we know is already stable (stable RCNT). To do this, we keep track of timestamping information on a per page basis. With each page in the database buffer, we maintain the timestamping activity in the page since the last time the page was written to disk. When we write the page back to disk, we complete timestamping for all committed transactions with records on the page. Once the disk write has been confirmed as having completed, we know that this timestamping is now stable. At that point, we update the VTT reference count field (stable RCNT), now tracking the *stably* timestamped records, in particular, the number of records for which the timestamping is not yet stable. This count reflects the number of versions either untimestamped or timestamped but not yet written to disk.

Thus, as soon as a page is written to disk, we know the impact on the reference counting for the transaction. Pages written as part of an effort to enable a checkpoint to be taken are now known to be stable before the checkpoint. We expect that the vast majority of transactions have their timestamping completed prior to their log records being truncated.

Now, before we complete a checkpoint, which involves truncating the log, we know precisely the transactions whose timestamping is both complete and stable. Those transactions entries can be dropped from our volatile timestamp table. We now form the group of TSN to TS mapping items that are then batch inserted into the persistent timestamp table. And most transactions are not included in this batch because their timestamping is known to be complete and stable.

E. Experimental Results

We ran experiments to determine the effect of our new strategy on the performance of single update transactions, which is a worst case in terms of impact because the incremental cost of updating the TSN to TS mapping table is highest as a percent of execution time. The results of our experiments are shown in Figure 2. The top line (highest cost) is our previous timestamping technique. The next line (second highest cost) is when all transaction mappings (equivalent to no transactions with completed timestamping) are added in batch to the mapping table. This shows that simply batching the updates substantially improves performance. The next line (third highest cost) is when 50% of the transactions have not completed timestamping prior to a checkpoint and still need to be added to the table, which means that 50% have timestamping completed. Then we show the cost when 80% of the transactions do not need to be included in the batch because their timestamping is complete, leaving only 20% to be added to the PTT. And finally, we show the performance of an unversioned database executing the same updates.

Figure 2 demonstrates that the cost of timestamping has been reduced dramatically, and even when 20% of the transactions still need to be timestamped at a checkpoint (which we believe to be much higher than what will be encountered under real load), the cost of updating a versioned database is only 10% higher than the update cost in an unversioned database.

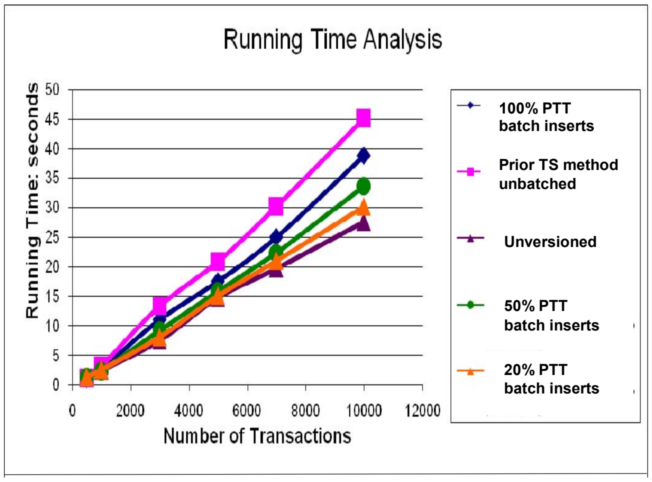

Figure 2: Performance of single update transactions under varying timestamping methods, compared with no versioning.

III. Deferred Splitting and "Current" Range Reads

As with update efficiency, we have made strenuous efforts to make range search performance for transaction time data as close as possible to the corresponding performance in an unversioned (current time) database. Certainly, version compression substantially improved range search performance, as we showed in [14]. However, this performance continues to be up to 20% worse than unversioned performance on a large part of the experimental space. This is truly unfortunate for range queries on current versions of data in a versioned database, as it imposes on users interested only in current versions the negative performance impact of versioning.

A. The TSB-tree

The TSB-tree [15, 16], which we use as an integrated index for both current and historical versions, splits pages both by key and by time. Each page of the tree indexes a rectangular key-time region of the search space. To ensure that all versions in the key-time rectangle are present on the page, we replicate versions whose lifetimes cross any time boundary. Thus time splits add copies of versions to each page when the version lifetimes cross the time boundary of the split. This increases the total space required for the data being indexed.

We control whether a page is split by key or by time via a utilization threshold *Th*. Only pages containing current data are ever split as only current data can be updated, and hence only current pages can become over full. We time split a page when it fills whenever the current versions in the page occupy less than *Th* of the page. Thus, *Th* acts as a lower bound on the utilization seen by current data. We cannot be sure that the maximum utilization for current data will exceed *Th*. Average utilization is *max(utilization)*ln(2)*. Thus, we guarantee that utilization for current versions will be at least *Th*ln(2)*. Only by increasing *Th*, bringing it closer to 100%, can we guarantee to improve current utilization. But increasing *Th* increases the number of time splits that we do prior to performing a key split. Hence it increases the number of pages needed to store the multi-versioned data.

As pointed out in [2], the only way to guarantee the storage utilization seen by every version in a multi-version tree like the TSB-tree is to perform a time split whenever there is a key split. Doing this ensures that the maximum current version utilization is actually captured in the historical page, before it is halved by the key splitting of the page. This also makes the key split very similar to the B-tree key split since only current versions are in the page when the key split is done.

This form of key splitting (never do a "naked" key split) has been an aspect of many multi-version indexing techniques beginning with the write-once B-tree (WOB-tree) [5]. The WOB-tree was designed to work on write-once media. Hence, a time split was "built into" the very nature of the splitting process. The original page became the historical page, and only the current data from the original page was then re-written to one or more new pages. For a pure time split, the current data was written to a single new page. For a key and time split, it was written to two new pages. Thus, a key split could not be performed in any other way, since it was impossible to remove data previously written in a page.

B. Exploiting Re-writable Media

While a write-once medium permits only the WOB-tree splitting strategy, other splitting strategies have been explored when a re-writeable medium such as a hard disk is used [16]. These strategies improve certain aspects of the TSB-tree index, e.g. reducing total space consumed. But, other splitting techniques do not guarantee that each version has a lower bound on storage utilization [2]. Hence, it is not possible to guarantee range performance, either for current or for historical versions.

C. A New Splitting Strategy

We want to "have our cake and eat it too." That is, we wanted improved storage utilization without having to increase the value of *Th*. And, when using re-writable media like magnetic disks, this turns out to be possible. The new splitting strategy works as follows. When the utilization of the current version in a full current page (called single version current utilization or *SVCU*) is less than *Th*, the page is time split as before. When *SVCU* >= *Th*, then instead of doing a time split followed by a key split, we only do a time split. But, we remember that we have exceeded *Th* by marking the page. When the page fills again, we then do a key split without doing an immediately preceding time split. Rather, the earlier time split substitutes for this. Thus, we have not changed the time of the time split, but we have deferred its associated key split until the current page fills again.

What has this new strategy accomplished? Historical pages are unchanged by this. The time splits and when the time splits occur happen exactly as they did when using the WOB-tree splitting strategy. What have changed are the current pages and the utilization seen on these pages. During the time between the preceding time split and the eventual key split, the current pages in this situation have twice the storage utilization of the pages had the key split been done immediately. Thus, we get one more opportunity to fill up the current page before we finally do this key split. And this is done without (1) increasing the number of time splits and hence the number of versions that are replicated and (2) without changing the single version utilization (*SVU*) seen for any historical version. But because of the one extra time the current page is filled before it is key split, we increase *SVCU*.

D. Deferred Splitting and Current Versions

Here we derive the deferred splitting impact analytically and confirm this via experiments. In [14], we derived *SVCU*. We repeat part of that analysis for completeness and because it extends naturally to deal with deferred splitting. This is an asymptotic analysis, not a probabilistic one. Our results are presented in terms of the mix of inserts (resulting in new records) and updates (resulting in new versions of existing records), where, for example a "percent of update" of 0.20 indicates 20% of database modifications are updates while 80% of them are inserts. We use "percent of update" in the presentation to be consistent with the results presented in [16], which served as a sanity check for our results. We compute *SVCU* as a function of the fraction of updates that are inserts, called the insert ratio *In*. Thus "percent of updates" is *1.0 – In*. We use *In* because it makes the equations simpler.

We also include the impact of compressing all versions of a record except the latest version on the page. Compression ratio is denoted as *CR* in the analysis, where *CR* is compressed version size divided by uncompressed version size. The format for a database page is shown in Figure 3. It is important to note that the current versions (more generally, the most recent versions) on the page are not compressed. All older versions of existing records are delta compressed; i.e. only the difference between the version and its predecessor is retained, along with the timestamping information associated with the version and the version chain pointers. *CR* denotes the full size in bytes of a delta version divided by the full size in bytes of the uncompressed version.

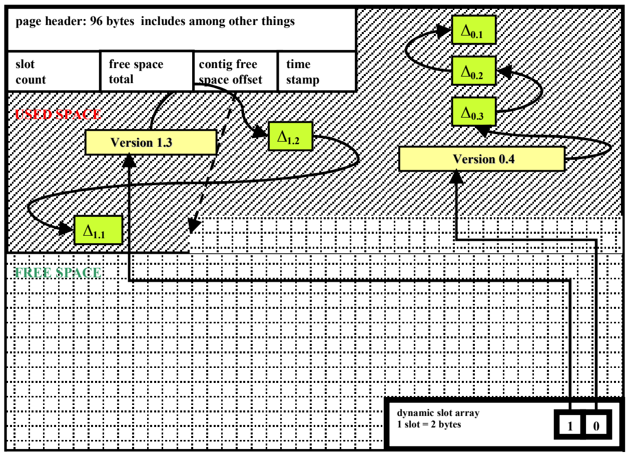

Figure 3: Data page layout showing how uncompressed latest versions are chained to earlier delta compressed versions.

When a data page is split by time, it is always split as of the current time. Its entire contents are moved to the resulting historical page. The original current page is then updated by removing all historical records (the deltas) from the page. The following analysis is based on this form of time splitting.

We compute the maximum value for *SVCU* iteratively until it reaches a fixed point. Consider a page split at iteration *i*'s maximum value $SVCU_i$. We iterate through multiple splitting steps until this maximum converges. Once we have determined the maximum value for *SVCU*, we then compute $SVCU_{avg}$ as $SVCU*ln(2)$.

A newly key split page at the $(i+1)^{th}$ step has utilization $SVCU_{(i+1)min} = 0.5*SVCU_i$. That is, we have removed all historical delta records during the preceding time split, and we have divided the remaining current records in half, allotting each half to one of the resulting pages. We then fill the page with entries divided between updates and inserts as given by the update ratio. The current entries when the page next fills are represented by these initial entries plus the inserts (but not the updates). We need to capture the impact of compression and hence we want to know how the space in the page is divided. This results in the following iteration formula. We start calculating this using *Th* as $SVCU_0$. The value converges rapidly (five iterations). At iteration $i+1$, we fill the unused space $(1 -0.5*SVCU_i)$ with insertions in their ratio of insertion space over the total space for new versions, taking into account that updates lead to compression of the supplanted version. Once the value of $SVCU_i$ are "clipped" by threshold *Th*, guaranteeing that *Th* is the minimum value for the maximum utilization that is permitted. Thus:

$$SVCU_0 = Th$$

$$SVCU_{i+1}=$$

$$Max(Th,0.5*SVCU_i+(1- 0.5*SVCU_i)*(In/(In+C*Up)))$$

These values are $SVCU_{max}$, the maximum value reached by *SVCU* before the page is key split. For average, we then get:

$$SVCU_{avg} = SVCU_{max} * ln(2).$$

We plot $SVCU_{avg}$ against update ratio in Figure 4 based on our prior experiments (the "no defer" results). Our analysis suggests that *Th* limits $SVCU_{max}$ at lower update ratios more than found in the experiments, but has less of an impact at mid-range update ratios before *Th* limits are strong.

To derive the improvement in *SVCU* resulting from our delayed splitting strategy, we continue our analysis by determining the impact on *SVCU* of one extra filling of the page prior to performance of a key split. Let us call the new single version current utilization $SVCU^d$ indicating it is for the deferred split case. Then, because of the deferred split, we have one more opportunity to fill the page, starting at the maximum fill (utilization) reached for the original case. Thus

$$SVCU^d_{max}= SVCU_{max} +(1-SVCU_{max})* [In/(In +C*Up)]$$

To get the new average $SVCU^d_{avg}$, we multiply the maximum utilization by $ln(2)$, which is now quite standard. Thus

$$SVCU^d_{avg}= SVCU^d_{max}*ln(2)$$

We plot the result of our analysis for the deferred case in Figure 4 comparing our analytic results with both deferred and non-deferred (WOB-tree) experimental results, both with and without version compression. Figure 4's experimental results are produced from experiments that we ran using Immortal DB as the test vehicle. The difference between analysis and experiment are minor, never differing by more than a few percent, and usually less.

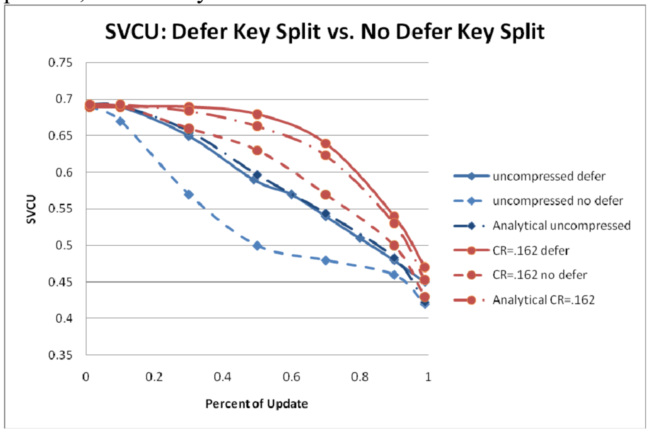

Figure 4: Single version utilization: deferred and non-deferred splitting results.

What we see in looking at Figure 4 is that our analysis matches our experimental results for the deferred case very closely. Deferred splitting improves *SVCU* substantially in the middle of the update ratio range, but the effect is reduced at both end points. When we have almost all inserts (percent of updates below say 0.2), then regardless of the threshold *Th*, the page is almost full without deferred splitting since there are very few historical versions to prevent this. Hence there is very little room for improvement. The resulting *SVCU* is already comparable to B-tree utilization. At the high end of the percent of updates, almost all updates modify records, producing more historical versions. Even though there is space in the page to fill after the final time split (which removes all historical versions), filling it with almost all historical versions does not improve *SVCU* much. It is in the middle of the update range where deferred splitting has most impact. And it is this part of the update ratio range that we expect to encounter in real system deployments. What we see, especially when we have decent compression (a compression ratio of $C = .162$ that might be produced when one or two fields of a multi-field record are updated) is that deferred splitting keeps *SVCU* within 10% of B-tree utilization out to around an update ratio of 0.8. Experiments confirm the analysis.

E. *Deferred Splitting and Historical Versions*

Delayed splitting also has an impact on what we have called multi-version utilization (*MVU*). Here we want to determine the effective storage utilization, where each version is counted only once, regardless of how often it might be duplicated during a time split of an overfull page. We also want to calculate it assuming that each version is uncompressed, so that we can clearly see the benefit of compression as well as deferred splitting.

As with *SVCU*, we adapt our results for *MVU* to determine the impact of deferred splitting. Our analysis for MVU did not determine its value at all points. We will not repeat the analysis from [14] here. Deferred splitting changes only the amount of space taken by data pages containing current data. Even for update percent of zero, current pages make up only *1/2* of the pages. So the number of current pages is always smaller than the number of historical pages. Over most of the range of update percent, the number of current pages is small. What we see then is that deferred splitting has only a very modest impact on multi-version utilization. This is because it only impacts single version current utilization, and the pages containing the current versions can be a very modest fraction of the total pages. Most of the impact of deferred splitting is in the significant increase in current version utilization, and hence current version range read performance.

At the end points of the update percent range, the deferred splitting has no impact. That is because at the end points, it has no impact on *SVCU*. The biggest impact, evident from the graph in Figure 5, occurs between *0.5* and *0.9*. This is the range that we expect most applications to operate in. However, even here, the impact is not large. Hence, the primary justification for deferred splitting lies in its improvement in current version range search. On the other hand, we see that delta compression makes a very large difference in *MVU* over most of the update percent range. It is especially helpful when there are a large percentage of updates, and hence a large number of historical versions. The space savings are very substantial at percent of updates.

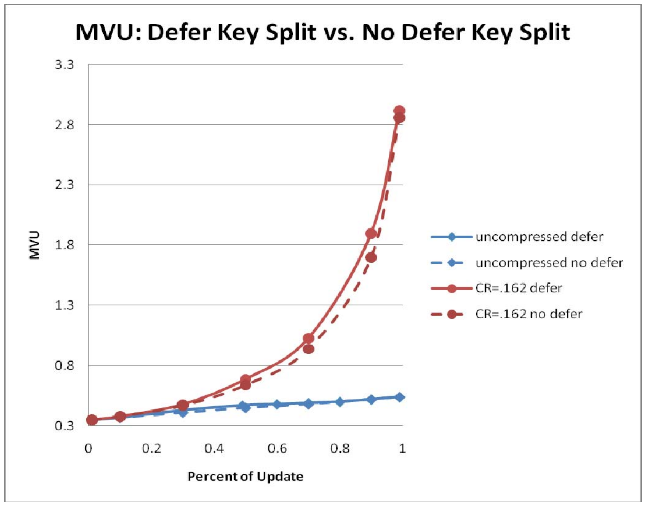

Figure 5: Multi-version utilization: deferred and non-deferred splitting results.

IV. Auditing Who Did What

Auditing, e.g. to track company finances, is a complex subject with many esoteric tests that are performed on a company's data. A database platform cannot hope to provide built-in support for these tests. Rather, the role of a platform is to provide the underlying information that enables the tests to be made. Note that a transaction time database, because it retains the entire history of the database, already helps substantially with this.

Part of auditing is tracing responsibility. Transaction time databases do not, with their basic functionality, address this. However, we have found it straightforward to add the ability to track which user (actually user id) is responsible for each change made to the database. And the impact on system performance is modest.

To support timestamping of versions, Immortal DB maintains the PTT table. This permits the system to replace TSNs by timestamps lazily after commit. We have been garbage collecting the PTT's entries once the timestamping activity is completed for the corresponding transactions. However, garbage collection is discretionary, and is purely a space optimization. If we retain the time table entries, the system continues to operate correctly.

Based on the preceding observation, our support for the audit function adds a user id field (UID) to the PTT. The UID permits us to remember who executed the transaction. To be consistent with our timestamping approach, we also add the UID to the information stored in each transaction's commit record and to the VTT, the volatile timestamp table that we cache in main memory to speed up the timestamping process. Coupled with a no delete policy for the PTT, we can now remember not only the "what" of a transaction but also the "who". That is, every record contains a timestamp. This timestamp can be used to search the PTT to find the UID of the user on whose behalf the transaction executed.

We have already measured the performance impact of providing this audit support. We measured it in our experiment to determine timestamping overhead (see Figure 2). Recall that in the timestamp information is added in "batches" to the PTT. Because it is added in a batch, the overhead of doing this is modest. The correct result to use in determining the auditing cost is the 50% batch inserts line. This line denotes the cost of adding only 50% of the transaction entries to the PTT. However, it also includes the cost of deleting those entries as well, hence being approximately equivalent to a pure insert of all transactions to the PTT. Recall that what Figure 2 gives is the overhead when executing single record update transactions, which is a worst case. Auditing thus adds (from Figure 2) about 25% overhead in this case. Remember, however, that what is being measured here is response time, not throughput. The throughput impact will be less as there are fewer page writes per transaction when the system is more heavily loaded, and multiple users are updating a transaction time database.

V. Related Work

Many database applications require that multiple versions of records be stored and retrieved. The effort to satisfy the diverse needs of these applications has led to a number of versioning solutions. A more complete history of related work is given in [14]. There have been a number of papers discussing aspects of timestamping [10, 19, 32]. Further, there are a large number of proposed indexing techniques used for temporal data, e.g. [2, 7, 25, 26]. Temporal database bibliographies are in [11, 26, 33]. Here we focus on multi-version support in general purpose database systems.

A. Postgres

There was some conceptual temporal database work in the early 1980's ([28] is an example and contains citations to even earlier work). However, the first database system offering temporal functionality was Postgres [30], which provided reasonably complete transaction time functionality. R-trees [6] are used in Postgres to index historical data, with recent data residing in a B+tree. This separation is important as R-trees, a general multi-attribute index, have difficulty supporting, in a straightforward way, data that is current and hence does not yet have an end time.

The movement of data from the B+tree to the R-tree occurs at a later time, after transaction commit. Versions that had not yet been timestamped, can be timestamped during this process, called "vacuuming", i.e. committed versions whose "end times" are sufficiently old are moved from the current part of the database to the historical part. This means that the Postgres version of our PTT can be garbage collected after each vacuuming scan completes, as all transactions committed prior to the time specified for the vacuuming can be guaranteed to be timestamped.

The Postgres approach does mean, however, that queries accessing historical "as of" record versions need to access both B+tree and R-tree. A record version valid at a given time may either (1) be in the B+tree if it has not been subsequently updated or (2) be in the R-tree if it has been subsequently updated. Range search performance in either of these trees is limited by the lack of time splitting support. Thus, the B+tree only splits by key. The R-tree splits in both dimensions, but is forced to index intervals, resulting in a reduction in the single version utilization and hence as of query performance.

In [23], a time-travel service is implemented for a replication DBMS. The time-travel semantics is defined using snapshot isolation in PostgreSQL and allows retrieval of older snapshots in replication systems.

B. DEC Rdb

DEC Rdb [8] (now owned by Oracle and called Oracle Rdb) provides support for read-only transactions without impeding update transactions via a transient versioning technique in which the transient versions are accessed by being linked to the current data. Transient versioning methods are also described in [29] for the same purpose. Rdb uses a technique called "commit lists" to identify the versions that should be seen in an "as of" query. General "as of" queries are not supported. Rather, a user can issue a query within a read only transaction. As in snapshot isolation, the version that is current as of the time that a read only transaction begins is selected as the version to be read. The system keeps track of transaction identities via TSNs. When a read only transaction starts, the system points the transaction to the set of transactions that have already been committed, together with their commit sequence numbers (CSN). The CSN is assigned at transaction commit in the order that transactions have committed, and hence in their serialization order.

This "commit list" permits the read only transaction to identify which version of a record it should read, i.e. the version with the highest CSN that is earlier than the last CSN that it is permitted to see. TSNs are stored with the updated record versions. Finding the correct version entails translating the TSN to its associated CSN.

The "commit list" approach avoids storing all transaction TSNs over all of history by truncating the list at an active transaction "low water mark", a TSN that is smaller than any active transaction. All earlier TSNs encountered when reading versions are assumed to be readable by the read only transaction. Further, all versions with TSNs greater than the TSN associated with the last CSN the read only transaction is permitted to see are bypassed in a backward scan to find the correct version. Thus a "commit list" can be easily represented by a list of limited size.

Rdb chains back to a separate version store for its recent versions. This works fine when only recent versions are read. Microsoft SQL Server borrows from DEC Rdb both the commit list approach for handling "timestamping" and the backward chaining to recent versions for its support of snapshot isolation concurrency control [29] which is a form of multi-version concurrency control that improves concurrency and reduces locking.

C. Oracle

Oracle has long supported a form of versioned data. Its undo recovery method keeps prior committed versions available in database pages, where a transaction abort removes the uncommitted version, restoring the prior version to its status as current version. It exploits this versioning in concurrency control, being a very early supporter of multi-version concurrency control. It calls the most stringent of its isolation levels "serializable", but this has been more precisely identified as "snapshot isolation" [3], with its own set of slightly weaker guarantees. Oracle has, over time, enhanced its versioning capability to support transaction time functionality.

With Oracle 9i, Oracle announced support for transaction time functionality [21], which it called "FlashBack". FlashBack queries allow the application to access prior transaction time states of their database. Oracle 10g extended FlashBack queries to retrieve all the versions of a row between two transaction times (a key-transaction time-range query) and allowed tables and databases to be rolled back to a previous transaction time, discarding all changes after that time. This is equivalent to "point in time" recovery and is used to deal with removing the effects of bad user transactions. The Oracle 10g Workspace Manager includes the time period data type, valid-time support, transaction time support, support for bitemporal tables, and support for sequenced primary keys, sequenced uniqueness, sequenced referential integrity, and sequenced selection and projection. They do not index historical versions, however, so historical version queries must go through current time versions and then search backward "linearly" in time.

More recently, Oracle has announced the "Total Recall" feature for Oracle 11g [22]. Building on FlashBack, Total Recall supports the long time archiving of transaction time versions, supporting the migration of the versions to archival media. "As of" queries, supported with FlashBack, execute "seamlessly" on the archive maintained by Total Recall. Built in security enforces that the Total Recall archive is strictly read-only. A form of compression is supported to reduce the storage cost of retaining the more extensive database history. Centralized management supports a deletion policy that can "age out" old versions, based on business policy. Total Recall is promoted as supporting Sarbanes-Oxley compliance.

D. Comparing with Our Work

In none of this related work on implemented systems were versions indexed using a multi-version temporal access method. Since such an access method was not used, it is not surprising that deferred splitting was not an aspect of the work.

Aside from Postgres, the other system implementations do not replace transaction sequence numbers with timestamps. We have worked hard to make this process as efficient as possible. This has a large payoff as soon as significant numbers of queries to past database states are executed, making our system a much better fit for the actual exercise of transaction-time functionality than prior work.

Our auditing approach is a very simple, effective, and high performance way of supporting audit functionality. Oracle "Total Recall" also provides an auditing capability, but we have not found an explanation that is sufficiently detailed to provide a meaningful discussion.

VI. DISCUSSION

A. Performance

We have stressed throughout this paper the importance of temporal support having performance that is close to the performance of a non-versioned (non-temporal) database. This is important even when only supporting versioning for snapshot isolation. If versioning performance is not comparable to unversioned performance, few users will run their database code using the versioning technology. Without competitive performance, users will not be inclined to exploit any more general database temporal functionality. This is at the heart of our rationale for building multi-version support into the database kernel.

In this paper we have shown how to make performance noticeably better than we had achieved in the past.

1. For transaction timestamping, we exploited (1) the better performance of batch updating, enabled by storing mapping information in a transaction's commit record; and (2) a more timely reference counting method to reduce the number of entries needing insertion into our PTT table.
2. We improved the range search performance for current data by exploiting re-writable media. Delaying key splitting in the way that we have may seem an obvious thing to do. But there is a long history of multi-versioned indexing in which this opportunity was not noticed or exploited. And it has a noticeable impact on range query performance for current data.

Both of these performance improvements together narrow the performance gap with unversioned data. It can truly be claimed that this gap is now no longer a serious impediment to supporting applications that require transaction time temporal support.

B. Functionality

Reducing the performance impact is one way of making temporal functionality more attractive. Increasing the functionality supported is another way, where both together improve the cost/benefit ratio seen by database users. Already suggested has been high performance media recovery [17] as well as the Immortal DB fast recovery from bad user transactions [20].

In this paper, we have added a foundation for audit support. Auditing database systems (and their data) has always been important for businesses. And with the passing of the Sarbanes-Oxley legislation, auditing has assumed even greater importance.

It is important to understand that we have added a foundational element of audit functionality. We have added only the tracking of who was responsible for the execution of a transaction. This, together with the versioning already supported by Immortal DB, permits special purpose application programs to be written that can trace every change made in the database, and assign the change to the responsible user id. Further, the performance impact of this is quite modest.

Adding audit support is very much in the same spirit as our previous functional additions to versioning databases. Like the audit support, they exploit the existence of versioned data to accomplish in a simple and high performance way, a highly useful capability. For bad user transaction recovery, earlier versions are used to replace subsequently corrupted later versions. This is done with much higher performance and much greater selectivity than is done by the classical "point in time" recovery, which has to install a database backup and then roll forward changes from the media recovery log to a point just earlier than the bad transaction. And then, finally, "point in time" recovery removes the effects of all later transactions, hence "de-committing" them. Using multiple versions, we avoid installing a backup, need no roll forward step, and selectively de-commit only directly impacted transactions.

C. Conclusion

Immortal DB in its current state has demonstrated that supporting multiple versions does not incur any serious performance degradation. And it provides the foundation for useful additional functionality exploiting multiple versions. This provides a strong incentive for the adoption of database systems that support transaction time functionality.

REFERENCES

[1] R. Agrawal and R. J. Bayardo Jr. and C. Faloutsos and J. Kiernan and R. Rantzau and R. Srikant: Auditing Compliance with a Hippocratic Database. *VLDB 2004.*

[2] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An Asymptotically Optimal Multiversion B+tree. *VLDB J.* 5, 4, 264--275, 1996.

[3] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil: A Critique of ANSI SQL Isolation Levels. *SIGMOD*, 1--10. 1995.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman: *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[5] M. Easton: Key-Sequence Data Sets on Inedible Storage. *IBM J. R & D* 30, 3, 230--241, 1986.

[6] A. Guttman, "R-trees: a dynamic index structure for spatial searching", *SIGMOD*, pp. 47--57, 1984

[7] M. Hadjieleftheriou, G. Kollios, V.J. Tsotras, and D. Gunopulos: Efficient Indexing of Spatiotemporal Objects. *EDBT*, 251 -- 268, 2002.

[8] L. Hobbs, K. England. *Rdb: A Comprehensive Guide.* Digital Press, 1995.

[9] C. S. Jensen and R. T. Snodgrass: Temporal Data Management. *IEEE TKDE*, 11, 1, 36--44, 1999.

[10] C. S. Jensen and D. B. Lomet: Transaction Timestamping in (Temporal) Databases. *VLDB*, 441--450, 2001.

[11] N. Kline: An Update of the Tcmporal Database Bibliography, *SIGMOD Record*, 22, 4, 66--80, 1993.

[12] D. B. Lomet, R. Barga, M. Mokbel, G. Shegalov, R. Wang, and Y. Zhu: Immortal DB: Transaction Time Support for Sql Server. *SIGMOD*, 939--941, 2005.

[13] D. B. Lomet, R. Barga, M. Mokbel, G. Shegalov, R. Wang, and Y. Zhu: Transaction Time Support Inside a Database Engine. *ICDE*, 35, 2006.

[14] D. B. Lomet, M. Hong, R. Nehme, R. Zhang: Transaction Time Indexing with Version Compression. *VLDB*, 2008 (to appear).

[15] D. B. Lomet and B. Salzberg: Access Methods for Multiversion Data. *SIGMOD*, 315--324, 1989.

[16] D. B. Lomet and B. Salzberg: The Performance of a Multiversion Access Method. *SIGMOD*, 353--363, 1990.

[17] D. B. Lomet and B. Salzberg: Exploiting A History Database for Backup. *VLDB*, 380--390, 1993.

[18] Lomet, D. and Salzberg, B. Transaction-Time Databases. Chapter in *Temporal Databases: Theory, Design, and Implementation.* A Tansel et al eds., Benjamin/Cummings(1993)

[19] D. B. Lomet, R. T. Snodgrass, and C. S. Jensen: Using the Lock Manager to Choose Timestamps. *IDEAS*, 357--368, 2005.

[20] D.B. Lomet, Z. Vagena, and R. Barga: Recovery from "Bad" User Transactions. *SIGMOD*, 337--346, 2006.

[21] Oracle:_Oracle_Flashback_Technology. http//www.oracle.com/technology/deploy/availability/htdocs/Flasflashback_Overview.htm, 2005

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

[23] C. Plattner, A. Wapf, and G. Alonso: Searching in Time. *SIGMOD*, 754--756, 2006.

[24] Betty Salzberg: Timestamping After Commit. PDIS 1994: 160-167

[25] B. Salzberg and V.J. Tsotras: Comparison of access methods for time-evolving data. *ACM Comput. Surv.* 31, 2, 158--221, 1999.

[26] M.D. Sao: Bibliography on Temporal Databases. *SIGMOD Record*, 20, 1, 14--23, 1991.

[27] H. Shen, B.C. Ooi, and H. Lu: The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases. *ICDE*, 274--281, 1994

[28] Richard T. Snodgrass, "The Temporal Query Language TQuel," In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS'84)*, Waterloo, Ontario, Canada, April 1984, pp. 204–213.

[29] SQL Server: *Inside Microsoft SQL Server 2005: The Storage Engine,* MS Press, 2005.

[30] M. Stonebraker. The Design of the POSTGRES Storage System. *VLDB*, 289--300, 1987.

[31] U. Tansel, J. Clifford, S. K. Gadia, A. Segev, and R. T. Snodgrass: *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings, 1993.

[32] K. Torp, R. T. Snodgrass, C. S. Jensen. Effective Timestamping in Databases. *VLDB J.*, 8, 4, 267--288, 2000.

[33] V.J. Tsotras and A. Kumar: Temporal Database Bibliography Update. *SIGMOD Record*, 25, 1, 41--51, 1996.