Transaction Time Indexing with Version Compression

David Lomet
Microsoft Research
Redmond, WA
lomet@microsoft.com

Mingsheng Hong*
Cornell University
Ithaca, NY
mshong@cs.cornell.edu

Rimma Nehme*
Purdue University
West Lafayette, IN
rnehme@purdue.edu

Rui Zhang*
University of Melbourne
Melbourne, Australia
rui@csse.unimelb.edu.au

ABSTRACT

Immortal DB is a transaction time database system designed to enable high performance for temporal applications. It is built into a commercial database engine, Microsoft SQL Server. This paper describes how we integrated a temporal indexing technique, the TSB-tree, into Immortal DB to serve as the core access method. The TSB-tree provides high performance access and update for both current and historical data. A main challenge was integrating TSB-tree functionality while preserving original B+tree functionality, including concurrency control and recovery. We discuss the overall architecture, including our unique treatment of index terms, and practical issues such as uncommitted data and log management. Performance is a primary concern. To increase performance, versions are locally delta compressed, exploiting the commonality between adjacent versions of the same record. This technique is also applied to index terms in index pages. There is a tradeoff between query performance and storage space. We discuss optimizing performance regarding this tradeoff throughout the paper. The result of our efforts is a high-performance transaction time database system built *into* an RDBMS engine, which has not been achieved before. We include a thorough experimental study and analysis that confirms the very good performance that it achieves.

1 INTRODUCTION

1.1 Overview

Transaction time database systems [21, 42] provide access to both current and historical information, and have many important applications. Temporal functionality is of increasing interest to database customers for auditing, legal compliance, trend analysis, etc. We have built a transaction time database system to provide access to both current and historical data. Our system, *Immortal DB* [26, 27], uses versions to support both "as of" queries to access data at an arbitrary time in the past and snapshot isolation [6], which requires access to recent versions. We believe that poor access performance for historical data has impeded the adoption of temporal functionality. Layering temporal support *on top of* a database system is cumbersome and typically is not practical [43]. For that reason, we have implemented Immortal DB *inside* an RDBMS engine, by modifying the SQL Server storage engine [40]. Insert/update/delete actions never remove information. Rather, these actions add new data versions, thus maintaining a complete, query-able history of database states.

We have extended our Immortal DB prototype [27] to use the TSB-tree [28] as an integrated index for accessing both current and historical versions of data. This enables it to provide logarithmic (in the number of versions) access to all versions of record. Further, range query performance, after the initial logarithmic probe, is linear in the size of the range. Our TSB-tree is derived from SQL Server's B+tree implementation and uses its concurrency control and recovery framework. Although based on SQL Server, our approach is more widely relevant as SQL Server's B-tree has a fairly standard architecture.

Other B-tree variants index multiversion data, e.g. the MVBT [5] and WOB-tree [13]. We use a TSB-tree variant primarily because it migrates historical data from a current data store to a historical store during node splitting. This is important. It preserves the performance of current queries as current data can remain clustered on high performance media. We can then move historical data to independent and less expensive slower disks and even WORM storage. Both MVBT and WOB-tree leave historical data in place and move current data during node splits.

Our TSB-tree performs both key splits, like a B+tree, and time splits. Time splits are required to index both by key and by time. To ensure that the density of records within a range has a good guaranteed minimum for all "as of" queries, we always time split [13] before we key split. This will be described in more detail, along with how we use a utilization threshold to control the choice between making a pure time split and doing a time split followed by a key split. This minimum version density guarantees that the cost for any "as of" range queries is linear in the number of records within the range [5].

Performance is further enhanced by compressing historical versions, both of data records and of index terms. We use a form of delta compression that is derived from the undo log record structure used in SQL Server. All data is compressed only locally within a page so that it may be uncompressed by accessing information only within the same page upon which it resides.

1.2 Immortal DB System

Immortal DB supports databases with multiple versions of data. When a transaction committing at time T_i inserts a new data record into the database, Immortal DB creates a version of the record with a timestamp T_i that indicates the beginning of the version lifetime. Each subsequent *update* creates another version of the data that is *inserted* into the database, marked with its timestamp, say T_j ($T_j > T_i$), indicating its start time. The prior T_i version of data implicitly then has an end time of T_j. A *delete* produces a special new version, called a "delete stub", that indicates when the record was deleted, and hence it provides an end time for the last version of the record. Record versions are

* Work done while interning at Microsoft Research

Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright for components of this work owned by others than VLDB Endowment must be honored.
Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept., ACM, Inc. Fax +1 (212) 869-0481 or permissions@acm.org.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

immutable (are never updated in place). A record version is linked to its immediate predecessor version via a *version chain*.
The Immortal DB prototype [26, 27] provides transaction time functionality via a collection of modest changes to SQL Server.
SQL DDL syntax. A transaction time table is specified via an "IMMORTAL" attribute in the table create statement. SQL Server already supports an alter table statement to turn on snapshot versioning.
Query syntax. An "AS OF" clause added to the transaction statement specifies queries on historical data. A "SNAPSHOT" clause already exists in SQL Server to indicate snapshot isolation queries [6, 40].
Commit processing. Version timestamps are chosen at transaction commit and are lazily posted to versions after commit. Timestamps are consistent with serialization order [7, 22, 31].
Page manager. A page is organized as a slotted array of records, with the most recent record version pointed to directly by an array entry. Older versions of the record are chained together within a page in the order of their create time.
Record Format. Each relational tuple has 14 bytes appended at the end that contains versioning information. This includes a timestamp, a sequence number, and a version chain pointer.
Recovery manager. New log operations are defined to enable recovery redo and undo of the "versioned" updates required for transaction time support.
Storage manager. Growing the number of unique records is accommodated via key splitting, which is done via modification of the existing B+tree key splitting already in SQL Server. New pages are acquired via time splitting to permit the space for versions to grow.

Earlier, only indexing by key to a current version was supported. Historical versions were found by searching back a linked list of versions, across multiple pages if needed [27]. Further, each version consisted of the entire uncompressed record. This made a record version very simple, at the expense of storing data from one version to the next that was unchanged by an update.

1.3 Time Split B+tree

The first focus of this paper is the integrated indexing of historical and current records, by both key and by time, into Immortal DB using the *Time Split B+tree*, or *TSB-tree* [28]. Given the performance results reported in [29], we expected performance to be quite good. We chose to use the WOB-tree splitting strategy [13] because using it guarantees that the storage utilization for any version on a page is always greater than a minimum value equal to half the storage threshold used to determine when to key split a page [5]. This strategy involves always doing a time split immediately prior to doing a key split, without intervening updates.

A time split is a special form of page split. Consider a collection of versions of records. Record versions span time intervals. Almost always a split time choice will cross the interval representing the lifetime of some version. In particular, when we use the WOB-tree splitting strategy, we always split data pages at current time. All records alive at current time have their lifetimes "split" by this strategy. The TSB-tree index partitions key-time space into rectangles where all versions with records in the key range that have lived within the time range defined by the page must be on the page. This can only be accomplished by having the versions with lifetimes that cross the split time appear in both the resulting pages.

A TSB-tree time split posts an index term describing the split to the parent index page. This requires changing the format of B+tree index pages, as we previously changed the format of B+tree data pages. It further requires that we time split index pages as well as key splitting them. Index page splits, though identical in concept, are subtly different from data page splits.

1.4 Compression

In the past several decades, disk costs have been dropping rapidly. Nonetheless, disks still constitute a significant portion of database management cost, not only for the hardware, but also for the human labor cost of managing it. Further, range query performance depends upon the density of records (records per page accessed) for the version of interest. This record density is reduced compared to non-versioned databases due to the existence of multiple versions of records in the same page. This is especially important when the historical versions share a page with the current versions, as accessing current time data is more frequent than accessing any given historical time data. Compressing versions is thus important for both reducing storage costs and for improving query performance.

1.5 Our Contributions

The contributions of this paper can be summarized as follows:

1. We integrate a temporal index (TSB-tree) into a commercial database system by adding TSB-tree functionality to the SQL Server B+tree, while preserving B+tree functionality for backward compatibility. Importantly, accessing current data is largely done via existing B+tree code. In this effort, we deal with technical issues such as concurrency control, recovery, handling uncommitted data, and log management.
2. We detail our unique designs of version chaining and treating index terms as versioned records to achieve the TSB-tree implementation with backward compatibility with B+tree.
3. We describe the tradeoff between query performance and storage space. We explain how to control this tradeoff through a parameter and discuss our designs to optimize the performance regarding the tradeoff throughout the paper.
4. We implement a data compression scheme in the TSB-tree. Our compression reduces substantially the storage needed for preserving historical data. For efficiency, all compression and decompression is local to a page.
5. We present experiments confirming gains that are achieved, both for storage utilization and range search performance. Compression improves range search by reducing the number of data pages that need to be accessed. Our analysis gives a simple, intuitive picture that explains our results and can predict performance under different conditions.

1.6 Paper Overview

The rest of this paper is organized as follows. How data pages in the TSB-tree are organized and split is discussed in Section 2. Section 3 extends our design to index pages of the TSB-tree and describes the index page splitting strategy. Section 4 covers the compression scheme used on data pages and index pages. Once one can index by both time and key, it is storage utilization that largely determines any extra cost compared to a non-versioned database. This is true for both storage cost and for range query performance, which depends upon how many pages are accessed. Section 5 describes our analysis and experiments which confirm that compressing records has an enormous positive impact on both these costs. We briefly survey related work in Section 6, and conclude the paper with a short discussion in Section 7.

2 DATA PAGES

2.1 Data Page Version Support

Immortal DB bases its data page layout on the now standard slotted page organization also used by SQL Server. An Immortal DB page is illustrated in Figure 1. Each entry in the slot array points to a data record that is the latest version of the record on the page. Every record version is linked to its immediate predecessor version to form a *version chain*. When the page contains current data, the latest record is either the last committed version of the record or an uncommitted version from a still executing transaction. Data manipulation operations do the following.

- **Insert:** An insert produces a record that is directly referenced by a slot in the slot array, records with higher keys being moved up one slot to make room for the insert. The newly inserted record's back pointer is set to null.
- **Update:** An update produces a new version for an existing record. The slot for the version is updated to point to the new version. The new version's back pointer references the version that had been current prior to the update.
- **Delete:** A delete is handled like an update in that a new version is created and linked into the chain of versions at its start. But this new version is special and is called a *delete stub*. The delete stub is marked as a "ghost", a SQL Server feature that makes the version invisible to ordinary queries.

Each of the operations here is logged with special log records signifying "versioned" operations. If a system failure occurs, the redo operations will ensure that any missing versions are restored, while the undo operations will remove uncommitted versions, and update the slots to reference the earlier version.

Version chaining is important for two reasons. (1) It provides *backward compatibility with B+tree*. For current time queries, the existing SQL Server code sees what looks like an unversioned page, since it only accesses record versions pointed at directly by the slot array. Thus, the unchanged B+tree access method read continues to work when versioning is provided. (2) It makes it easier to find ancestor versions when performing historical searches, key splits and compression, which are explained later.

We use the deleted record, with a timestamp that is the delete time, as a delete stub so that the key, which may be multiple fields, can be easily found when looking for historical records on the page, without the SQL Server kernel needing to decode fields of the record. The delete stub is marked as a "ghost" record and ghosts are ignored during queries. The immediately prior record version is the full deleted version with version start time as its timestamp, hence correctly representing the version and its lifetime. When a data page is time split, which uses current time, we remove delete stubs from the current page since the records whose ends they are marking are no longer in the current page.

2.2 Data Page Splitting

The TSB-tree does both key splits, which are similar to key splits in the B+tree, and time splits. For time splits, versions whose end times are before the split time are moved to the historical page. Versions with start times later than the split time are put in the current page. Because versions have interval time extents, versions that span the time chosen for the split must be present in both of the resulting pages. Readers are referred to [28] for details on these operations. We focus more on performance tradeoffs and handling uncommitted data in this subsection.

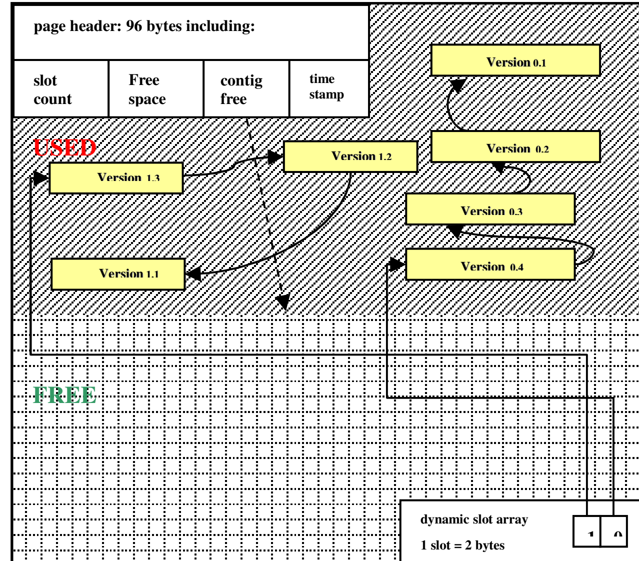

Figure 1: Data page layout with uncompressed versioned data.

One point to stress is that when a page is time split, we create a new page for the historical data. The original page remains the current page. This assignment of pages makes it possible to progressively move historical data to a different medium, e.g. a different disk [26,32]. This is different from structures such as the MVBT [5], which makes the new page for the current data.

2.2.1 Splitting Policy Threshold

We control the version density via a key splitting threshold. That is, we do not split a data page by key until the utilization achieved by the current data reaches a threshold. There are some subtleties involved and we need to answer the question of why this threshold should not be 100% (or indeed too close to 100%).

Whenever a data page fills up, it is split. We need to decide whether the split is a time split, a key split, or both. Immortal DB never does an isolated key split. In Immortal DB, if a key split is needed, we always perform a time split before it, which we call *the WOB-tree split policy* [13]. This policy ensures that any version (as seen by an as-of query) has at least a minimum storage utilization [5]. So our choice is between a time split by itself and a time split followed immediately by a key split. This choice is controlled by the *current version utilization* in the page being split.

We define *single version current utilization* for a page ($SVCU_{page}$) as the size of the page's current data divided by the page size (both in bytes). We specify a threshold value *Thresh* for this utilization to control page splits. If, when a page fills completely, $SVCU_{page} > Thresh$ then we do a key split after a time split. Otherwise, we perform only a time split. We control the tradeoff between the space required to store versions and the storage utilization seen by as-of queries via *Thresh.* The higher the value of *Thresh* the more pure time splits are done, the lower the value, the more often key splits are done as well as time splits.

2.2.2 Version Redundancy

We always split at the current time T_{curr}. This requires that all committed versions, current and historical, of the original page be present in the newly created historical page since they all have start times earlier than the current time. This new historical page is then written to a separate storage partition that holds the

historical data. The original page, continuing as the current page, is then updated by removing historical versions because their end times are earlier than the split time. This leaves only the last committed version of each record along with any uncommitted versions in the current page.

Thus, current committed versions appear in both the historical page and the current page, since their lifetimes cross the time boundary chosen for splitting the page, i.e. the current time. The more frequently a page is time split, the more redundant versions are introduced. Thus, if key splitting is delayed excessively, many record versions might appear redundantly in one, two, or more data pages as the result of time splitting.

By setting our key splitting threshold below 100%, we permit data pages to be split at an earlier time. The lower the threshold is, the fewer the redundant record versions are. However, as we reduce this splitting threshold, we also reduce the storage utilization of the current data (*SVCU*). Thus, there is a trade-off between how much redundancy is introduced and how large *SVCU* is.

The original TSB-tree study [29] suggested a threshold of 0.67, which means that the utilization of any version (versions being defined by selection of an as-of time) will be at least 0.67 of the utilization were we not supporting multiple versions. We make this more precise in Section 5 on storage utilization.

2.2.3 Uncommitted Data

There may be several records on a data page that are being actively updated and hence have uncommitted versions as their latest versions at the time when a page becomes full. These uncommitted records do not have lifetimes that cross the time boundary used to split a full page. Nonetheless, they will appear in both resulting pages. Why?

The uncommitted versions must appear in the current page resulting from the time split since this is the page in which the version lives if their transaction commits. They appear in the historical page for a number of practical reasons.

- A simple strategy of (byte) copying the current page to a newly allocated historical page copies these versions as well.
- Our compression method leaves the most recent version in a page uncompressed (see Section 4). Uncommitted versions are the most recent versions. Having them in the history pages avoids needing to uncompress other versions.
- A historical page is never subsequently updated, so any free space on the page cannot be put to use in accommodating new record versions.

Even though uncommitted data is on an historical page, it will be invisible and never returned as a result of a query.

The issue to be addressed is what role uncommitted data plays in dealing with our key splitting threshold. Recall that we set the threshold to less than 100% in order to reduce the frequency of time splitting and hence of version redundancy. Since uncommitted versions remain in the current page, they consume space like committed data. Therefore, we choose to treat uncommitted data just like the committed data when determining whether we have reached the key splitting threshold.

3 INDEX PAGES

3.1 TSB-tree Index Requirements

A TSB-tree partitions data by key and time into key-time rectangles. Each TSB-tree index term that references a data page

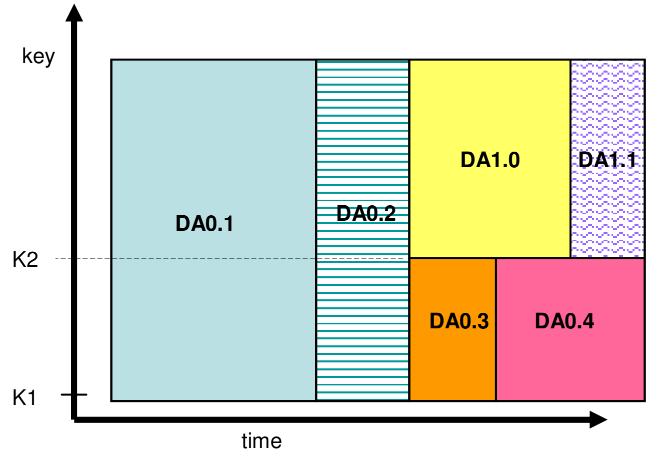

Figure 2: Index terms represent key-time areas, not merely key points with time intervals.

includes a description of this key-time rectangle and a pointer to the page containing data in this rectangle. This is the first difference between an index term (and its versions) and a data record version which has a (point) key and a time interval in which it lives. Figure 2 illustrates the division of key-time space as might arise in a TSB-tree. Note that key adjacent index terms for later times can share an historical index term for a page from an earlier time because they reference pages resulting from a key split of the earlier page.

A second difference is that index terms are not directly associated with any transaction. Thus, a new index term, together with an appropriate key-time rectangle description typically is generated during the execution of a transaction but will not share the timestamp of any transaction. Rather, the timestamps are used to partition the time dimension of the TSB-tree index, and hence we have some flexibility in how we choose the time.

Finally, index terms referring to current data, i.e. the space that they describe includes the current time, are not immutable. When we split a current page, the key-time rectangle is divided into two spaces. The index term referencing now historical data must point to the new historical page since we move the historical versions, not the current versions, to a new page.

The differences between index term and data record mean that we must manage and split TSB-tree index pages with their "versioned" index terms somewhat differently from data pages.

3.2 Index Page Version Support

In SQL Server, B+tree index pages are treated much like data pages, with index terms treated like records. Common formatting means that code to manage index pages is similar to and in some cases identical to code used for data pages. Thus, in Immortal DB, we designed our index page organization to be similar to data pages described in the previous section. We use the lower left hand corner of a key-time rectangle to describe the region, i.e., <low key, low time>. Full boundaries can be derived from the descriptions of the adjacent regions.

Index term "historical" versions are maintained in the same way that record versions are maintained, i.e. in a linked list starting at the current (or most recent) version. The difference here is that index terms on the list are not historical versions of the later index term. Rather they are index terms that reference earlier versions

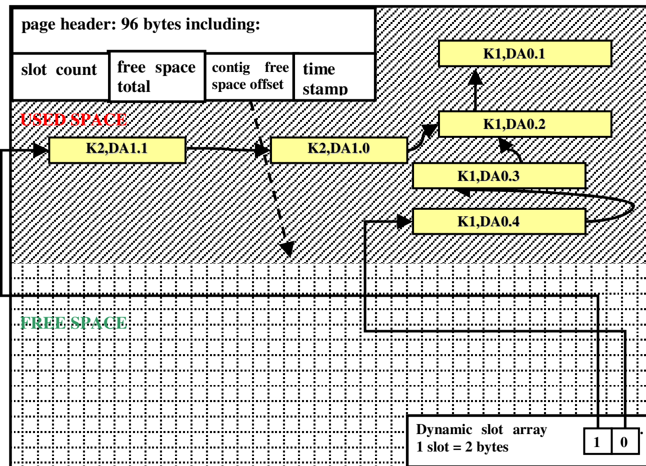

Figure 3: An Immortal DB index page for the space shown in Figure 2, with index terms treated like versioned records.

of the data indexed by these later index terms. Figure 3 shows how we represent index terms within an index page for the space partitioning shown in Figure 2. Note how two adjacent index terms share an earlier index term in this chain. This representation shares the data page characteristic that unversioned read access to a TSB-tree does not require that the reader know that the data is versioned. It differs from the scheme presented in [28], where pointers to all index terms are present in the slot array.

3.3 Index Page Splitting

Complications can arise when splitting index pages because index terms denote regions in key-time space. An index term whose region crosses a key split boundary must, like the record versions of data pages that cross a time boundary, be present in both pages resulting from the split.

3.3.1 Key Split

Two current index terms may share the same historical index term *I*. When a data page is first time split and then key split, the key of one of the current index terms divides the key space of *I*. Shared historical index terms are illustrated in Figure 3. How such shared index terms may be partitioned in a key split is illustrated in Figure 2, where *K2* is used as the splitting key, and *K2* divides the spaces of the index terms denoted by DA0.1 and DA0.2. When *K2* splits an index page, key-time regions of both DA0.1 and DA0.2 cross the *K2* key boundary. Thus, they will need to appear in both index pages resulting from the key split.

Thus, historical index terms may need to be stored in both resulting pages of the key split. Because they point to pages that are part of the historical tree, these pages will never be updated or split. Thus their index terms will never need to be updated. We can have such duplicated historical index terms without any concern about future updating difficulties.

3.3.2 Time Split

Because index terms denote time ranges, like data page records, an index term will frequently need to be in both pages resulting from a time split. We use the same technique with them that we use with historical data versions (and with index terms whose key range crosses a key split boundary). That is, we simply put the index term in both pages resulting from the time split.

We are faced, however, with a complication when an index term that references a current page crosses the time split boundary. Because current data is updatable, its page can subsequently be split (by key or time, or both). Thus, the index term for a current page is itself updatable. When such an index term is in a "current" index page, this poses no problem. However, if a time split of an index page cause a current index term to appear in both historical and current resulting pages, this produces two problems. (1) We now have a "historical" index page that can be updated. (2) The page referenced by the current index term now has two parents that need to be updated should it split. There are three main ways of dealing with this.

1. Find a time (boundary) for splitting the full index page that does not cross the time interval of any index term accessing current data. This is our preferred tactic. We choose it whenever it permits the page to be effectively time split. Inevitably, there will be index terms that cross the boundary. Historical index terms can cross the boundary because they can be in both resulting pages without complication as they will never be updated. However, when we cannot avoid a current index term crossing the boundary, we use the 2nd way.
2. Perform a key split instead of a time split for the index page. Key splits can always be done so that no index term accessing current data need be copied. This is a second choice because it reduces the fan-out of the resulting index pages for any as-of time slice query, including accessing current data. However, so long as fan-out is not permitted to get too low, this is an acceptable strategy as a fallback to 1.
3. As we do for index terms referencing historical data, we can also duplicate the index term for current data when the key-time region it references crosses the time split boundary. We will then need to update this index term when the current page it references is subsequently time split or key split.

We did not pursue option 3, primarily because of the extra code complexity, but also because it compromised the invariant that *historical pages are immutable*. This is no trivial matter, as mutable historical pages require latches to prevent concurrent access during updates. By avoiding option 3, we preserve immutability of historical pages, and hence avoid the need to latch them during range searches. However, the risk (which we have not encountered in our experiments and tests) exists that index page fan-out will be reduced in some cases.

3.4 Rest of Structure Modification Protocol

The rest of our protocol is derived directly from the SQL Server protocol. For both reads and updates, we latch couple down the tree then up the key range to assure deadlock avoidance via resource ordering. The structure modification process involves a second traversal of the tree should an update find a page to be full, with splitting preemptively down the tree. A page, once split and committed via a system transaction, is not undone even when the triggering update transaction aborts. This is multi-level transactions within an ARIES style of recovery.

4 COMPRESSION

4.1 Record Version Compression

The TSB-tree clusters records by key and time, storing in a page all versions of records within a key range that exist within the page's time range. All versions of a given record share at least the primary key field(s) in common. These versions may share many fields, with an update frequently changing only a small number of

fields of a record. Immortal DB compresses record versions using a backward delta compression scheme that exploits the frequently large commonality between a record version and its immediate predecessor version.

For version compression, we modify the SQL Server update log record [40]. This log record contains information that identifies the record. The log record contains, for each change, two byte fields for a *change offset*, a length of the data before updating (*delete length*) and a length of the data after the update (*insert length*) plus the before and after data. This permits recovery to remove the old value from the record and replace it with the new value, without knowing about attributes. Rather, it can perform the update entirely by byte replacement based on this information.

A record identifier is unnecessary in our delta records. Our link from an earlier uncompressed record version on the page provides this. We also reduce the size of the delete and insert lengths to one byte, optimizing for small updates, at the cost of having to break large updates (greater than 255 bytes) into multiple changes within our delta record. We only need undo information and so do not store redo information. Finally, we also squeeze other parts of the uncompressed record in producing our delta record, e.g. the timestamp field. Figure 4 illustrates our delta record format in contrast to the uncompressed original record.

Figure 5 illustrates how deltas are tied into a record's version chain on the page in the example of Figure 1. The latest version of a record on a page is uncompressed. This means that current versions are uncompressed and that current time database functionality is unaltered by compression. All predecessors are delta compressed. We expect most updates will be to a single attribute of a record. With 10 to 20 attributes for a record, a compressed record might be expected to be around a tenth the size of an uncompressed one.

4.2 Delete Stub Compression

When we delete a record, we use a delete stub to provide the end time for the last version of the record. In our initial work, this delete stub consisted of a *complete copy* of the prior version, with a new timestamp and the ghost flag set. The reason for this is that we need to remember the key value for the record so that we can place records correctly on the page, i.e. in key order. The SQL Server storage engine wants to find keys in all records in exactly the same way, so we leave the entire record, since the key can be anywhere in the record.

This technique of using the prior record as a delete stub is logically effective but obviously is expensive, since its sole purpose is to provide an end time for the preceding record version. With compression, we have the chance to reclaim the extra space. The delete stub is still the entire preceding record (the deleted record), and is thus unchanged from before, continuing to also provide the key for the record. However, the preceding record can now itself be replaced with what we call the *empty delta compressed record*, since this preceding record is identical to the record version in the delete stub, except for the timestamps and the ghost flag. This is illustrated in Figure 6. Note that this technique continues the "rule" that the latest version of any record in a page is uncompressed, with compression applied to earlier versions.

Note that the empty delta record of Figure 6 contains no change descriptors, just control and timestamp information. So, while we cannot actually compress the delete stub because we continue to need its key information, the record for which it is a delete stub can be reduced to an empty delta record.

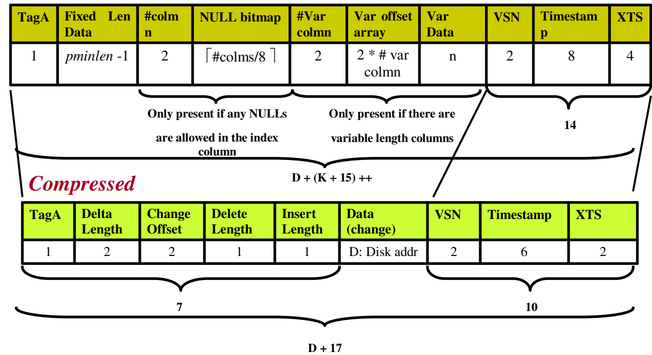

Figure 4: Compressed and uncompressed record formats.

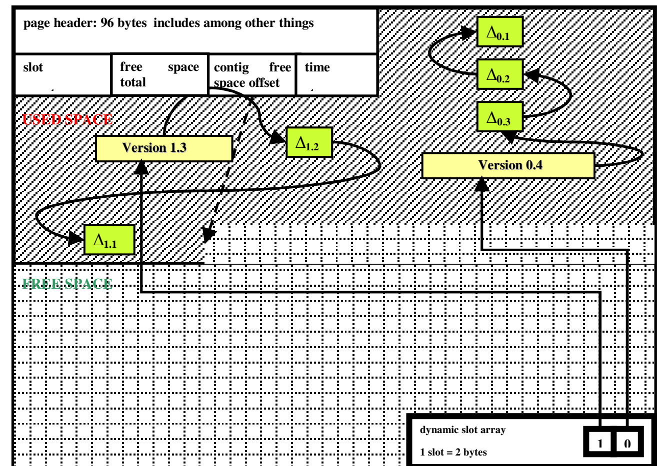

Figure 5: Data page layout showing compressed versions for the page in Figure 1.

Empty Delta

TagA	Delta Length	VSN	Timestamp	XTS
1	2	2	6	2

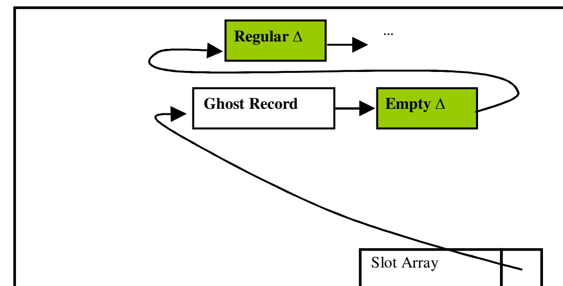

Figure 6: The format of an empty delta, and its position in the version chain when a record is deleted.

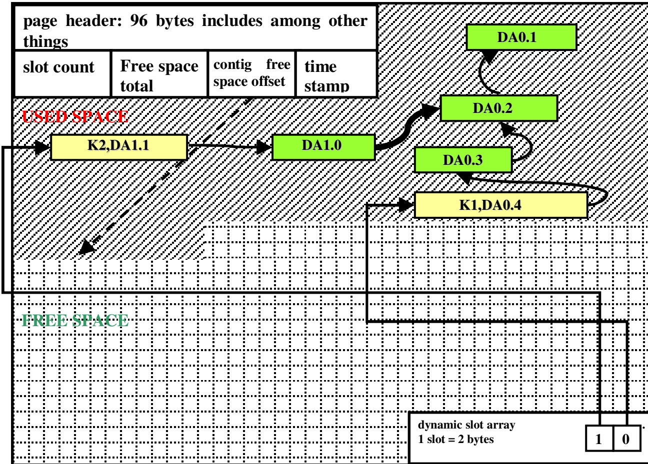

Figure 7: Immortal DB index page of Figure 3 with compressed index terms.

4.3 Index Term Compression

We compress index terms on an index page similarly as we compress record versions on a data page. Index terms can be compressed very effectively. Recall that an index term space description is the <low key, low time> corner of the region referenced by the index term. In addition to the space description, an index term includes a pointer to the child page that the term references. The key of an index term is unchanged in a time split, which is how "historical" index terms are created. Hence, the index term pointing to the same key range but in an earlier time period only differs in child page pointer and timestamp.

Two later index terms with different low key values can both point to the same compressed historical index term in which the key value is omitted; i.e., an historical index term can be shared with multiple index terms that precede it on the chain (an index term preceding another on the chain indexes a later part of the key-time space). Its omitted key is the lower key value for only one of the terms resulting from a key split. But the historical index term will nonetheless point to the correct child page containing the data for the regions identified by different key values. Figure 7 illustrates this compression for the index page shown in Figure 3. In all index terms referring to only historical data, the key value is omitted. During lookups, the key value from the index term referring to current data that precedes the index term on the version list guides the search. In Figure 7, the historical index term pointing to disk address DA0.2 contains historical information for both K1 and K2 current regions.

5 STORAGE UTILIZATION

There are two reasons why storage utilization is exceptionally important in a transaction time database.

1. Disk storage cost can be a significant factor in the hardware cost of supporting a transaction time database. Disks are becoming cheaper and indeed that is a reason why transaction time databases are increasingly important. However, even for a constant size current database, its transaction time cousin can continue to grow, consuming ever more storage. So providing good overall storage utilization is very important. The quantity we focus on is multiversion total utilization (*MVTU*), the size of all versions (uncompressed) divided by the storage size needed to contain them.
2. The density of record versions relevant to any single as-of query determines how many pages need to be accessed to satisfy an as-of range query. This is single version utilization (*SVU*). Because all versions share the same approximate average utilization, we focus on the single version utilization provided for the current version (*SVCU*).

Unfortunately, we cannot simultaneously optimize both *SVCU* and *MVTU*. Both are impacted by the key split threshold (*Thresh*), the utilization required to be attained by the current version within a page before we perform a key split in addition to the time split that is always done when a page is full. The higher we set *Thresh*, the higher will be *SVCU*, as it is always at least *Thresh*ln(2)*. However, the higher *Thresh* is set, the more time splits are performed. This leads to more redundant versions, since any version that lives across a time split must be duplicated to be present in each of the resulting pages. This reduces *MVTU* as more duplicate versions require more storage.

In this section, we explore this tradeoff between *SVCU* and *MVTU* and the impact of compression. We chose our experimental parameters based on [29], which serve to confirm the results that we report when working with uncompressed data. We provide also a "back-of-the-envelop" analysis that further confirms our experiments for a subset of the cases the experiments cover. This gives an intuition as to how and why the performance is achieved, and can be used to predict performance under other conditions. The notation we use for this analysis is given in Table 1.

5.1 Experimental Setup

We used our implementation of the TSB-tree in Immortal DB as the vehicle for doing experiments. For our experiments, we set our key splitting threshold at $Thresh = 0.67$, inserting and updating a total of *50,000* versions, using uniformly distributed random keys. We varied the update/insert ratio from 1% updates to 99% updates (the values used are given in the reported results), reproducing the experimental parameters reported in [29].

Table 1:

Notation used in our analysis and experiments.

Term	Denoting	Computation
P_{size}	page size	
R_{size}	record size	
R_{cur}	# of current records	
C	# current pages	
R_{hist}	# of history records	Without duplicates
H	# of history pages	
R_{comp}	Compressed record size	
CR	Compression ratio	R_{comp}/R_{size}
$SVCU$	Single version current utilization	$R_{cur}*R_{size}/P_{cur} * P_{size}$
$MVTU$	Multiversion total utilization	$(R_{cur}+R_{hist})*R_{size}/(C+H)*P_{size}$
$Thresh$	Utilization threshold	
In	Insertion ratio	$(1 – Up)$
Up	Update ratio	$(1 – In)$
D	Uncompressed record storage	$(R_{cur} + R_{hist})*R_{size}$

We extended the experiments for version compression, repeating the experiments for different compression ratios *CR*, which were controlled by updating a character string field with varying size strings. We ran four sets of experiments, uncompressed (*CR = 1.0*), 2:1 compression (*CR = 0.515*, where the data portion is compressed at 2:1, but *CR* includes the storage overhead of timestamps, etc.), 4:1 (*CR = 0.295*), and 10:1 (*CR = 0.162*).

5.2 Single Version Current Utilization

Supplementing the experiments, we did an approximate analysis of the expected results for values of *SVCU* at all experimental points.

The analysis used to produce the average value for *SVCU* is given below. This is an "asymptotic" analysis, not a probabilistic one. $SVCU_{avg}$ is the average utilization seen in current database pages for the current versions. It is, in fact, also the average utilization of any "as-of" version.

As a starting point, imagine that a data page has been split at the prior iteration *i*'s maximum value $SVCU_i$. We want to iterate on this until this maximum converges. We can then compute $SVCU_{avg}$ in the usual way as $SVCU_i*ln(2)$.

After a key split, the new page has utilization $SVCU_{(i+1)min} = 0.5*SVCU_i$. 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. We need to capture the impact of compression and hence we want to know how the space is divided. This results in the following iteration formula. We start calculating this using *Thresh* 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. All maximum values of $SVCU_i$ are "clipped" by threshold *Thresh*. Thus:

$$SVCU_0 = Thresh$$

$$SVCU_{i+1}=$$

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

These values are $SVCU_{max}$, the maximum value reached by *SVCU* before the page is key split. For average, we multiply by ln(2).

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

These results closely match our experiments, as indicated in Figure 8.

Generally, the analysis suggests that *Thresh* limits $SVCU_{max}$ at lower update ratios than found in the experiments, but has less of an impact at mid-range update ratios before *Thresh* limits are strong. The difference between analysis and experiment are mostly minor, never differing by more than four or five percent, and usually less.

5.3 Multiversion Total Utilization

We also determined multiversion total utilization *MVTU*. Since we compress old versions, one should not be surprised that *MVTU* improves as more old versions are created via a higher update ratio. Indeed, because of compression, the effective *MVTU*, which is calculated based on the size of uncompressed data, can be larger than one, in some cases substantially larger.

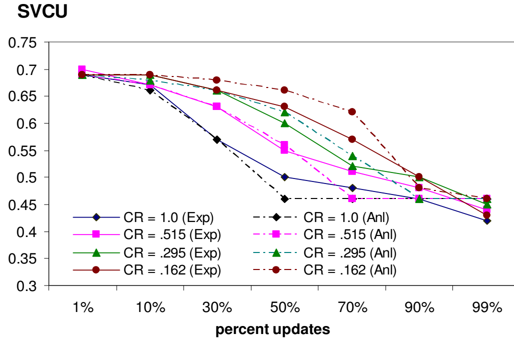

Figure 8: Single Version Current Utilization (*SVCU*) with the key split threshold *T = 0.67*

Our simple analysis for *SVCU* provided results for all update ratios. Our analysis for *MVTU* is more limited, applying only to the end points of the update ratio range. Thus we can confirm the experiments only for some of the cases we considered in the experiments.

5.3.1 Update Rate near Zero

When the update rate *Up=0*, we have only inserts. Hence, all versions are current versions. For this case, *Thresh* and compression ratio *CR* have no impact. We always fill up the page before splitting the page. And all versions are current, so none are compressed. Each page is both time split and key split at this point. This results in two current pages and one historical page. This binary process, over time, then produces a "binary tree" of data pages, formed by this "two current pages for each history page" splitting regime. Given our uniformly distributed insertions, this results in a balanced tree of pages. The number of leaf pages (current pages) in a balanced tree is equal to the total umber of non-leaf pages (historical pages). Hence, because all versions are current, and they are spread over twice the number of current pages,

$$MVTU = 0.5 * SVCU_{avg} = 0.5 * ln(2) = 0.346$$

This is close to our experimental results reported in Figure 9.

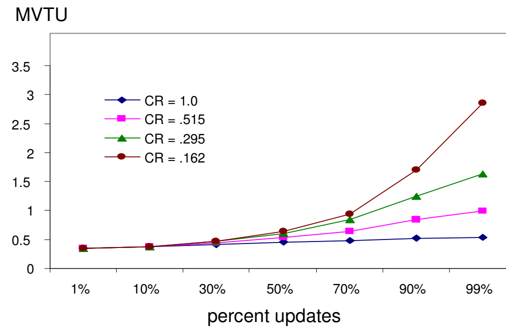

Figure 9: Multiversion total Utilization (*MVTU*) with a threshold of *T = .67*.

5.3.2 Update Rate near One

We can also confirm the experiments at *Up* approaching *1.0 (0.99),* strongly for the uncompressed case, and suggestively for

compressed cases. With Up near 1.0, the number of current pages is very small compared with the number of history pages. Further, the last (and always uncompressed) versions on any page are fully redundant with compressed versions of the more recent historical page. Hence, all non-redundant versions of such a page exist as "historical versions" on some page. (Only the uncompressed versions are redundant.) We know that the average utilization for the uncompressed versions is $SVCU_{avg}$. So we can subtract that from 1.0 to determine $MVTU$ for the uncompressed case. This gives, at Up near one

$$MVTU = 1 - SVCU_{avg} = 1 - 0.462 = 0.538$$

Again, this fairly closely matches our experimental results for uncompressed data closely at $Up = 0.99$, as shown in Figure 9.

We need a more accurate analysis for compression factors smaller than one and $Up = 0.99$. The reason for this is that the relative number of current pages increases compared to the uncompressed case. So we take a weighted average of the storage utilization in the history pages and the current pages. Thus we need to determine how many pages are history pages H, how many are current pages C, and then divide the total data space (assuming all versions are uncompressed) by the total space in all pages.

Current Pages

We can derive the number of current pages C from the $SVCU_{avg}$ (which involves only current data in current pages) and the amount of current data. Since we are treating the update percent of 0.99 case, the amount of current data is $0.01*D$, where D represents the size of all the versions. Thus

$$SVCU_{avg} = 0.01*D/(C*P_{size})$$

Further, $SVCU_{avg} = Thresh*ln(2) = 0.462$ at Up near 1.0, so

$$0.462 = 0.01D/(C * P_{size}) \text{ and}$$

$$C = 0.0216(D/P_{size})$$

Historical Pages

Our first approximation for $MVTU$ was an approximate calculation for historical pages only. We will refine that calculation, and then join it with the current page computation to produce the final result.

For our experiments, we chose a page size of exactly 35 uncompressed records. When records are uncompressed, each historical page is completely full. But when records are compressed, historical pages do not quite fill up. On average, half an uncompressed record of capacity remains. So, each historical page can exploit[1]

$$[34.5/35 - SVCU_{avg}]*P_{size} = [0.986 - Thresh*ln(2)]*P_{size} = 0.524*P_{size}$$

Page size P_{size} is in terms of the number of uncompressed records. To determine the number of compressed records, we need to divide that by the compression ratio CR. The amount of historical data is $0.99*D$. Thus

$$0.524*P_{size}*(1/CR))*H = 0.99D$$

Solving for H gives us

$$H = 0.99D/((0.524/CR)*P_{size}) = 1.89*CR * (D/P_{size})$$

[1] For uncompressed records, we use $(1-SVCU_{avg}) = 0.538$.

All Pages

Finally, by definition, $MVTU = D/(C + H)$, so

$$MVTU= D/[\{0.0216 *(D/P_{size}) + (1.89*CR *(D/P_{size}))\}*P_{size}]$$

$$\text{or } MVTU = 1/(0.0216 + 1.89*CR)$$

The analytic results are compared with our experiments in Table 2. The analysis, approximate though it is, produces results that are quite close to the experimental results. For uncompressed data, where we did not adjust the page size computation because exactly 35 records did fill the page, experiment and analysis agree "exactly".

Table 2: Comparison of experimental and analytic results.

Compression Ratio *CR*	***MVTU* Analysis**	***MVTU* Experiment**
1.000	*0.54*	*0.54*
0.515	*1.01*	*0.99*
0.295	*1.73*	*1.63*
0.162	*3.05*	*2.86*

5.4 Compression to Improve Performance

As we indicated in the introduction, one can use compression not only to save space but also to improve query performance. Compressing versions can be used to impact both the total number of pages required to store versions as well as the utilization that will be seen by an "as-of" query. This is determined by how we choose $Thresh$. If we leave $Thresh$ unchanged when we introduce compression, we reduce the number of pages required to store our versions, hence improving $MVTU$. Alternatively, we can try to keep the number of pages unchanged by increasing $Thresh$, which improves $SVCU_{curr}$, and the effective utilization seen by all "as-of" queries. In this subsection, we show the impact of compression on the multiversion between $SVCU_{curr}$ and $MVTU$.

We ran a set of experiments on compressed data in which we varied the key splitting threshold $Thresh$ for the compressed cases until the compressed cases produced the value for $MVTU$ achieved for the uncompressed case. We found that we were able to raise $Thresh$ substantially. This translates the compression benefit into a performance improvement for range queries.

Figure 10 displays the results of our experiments. At low update ratios, there is a very broad range of thresholds that produce similar results. This is because $Thresh$ plays a smaller role at low update ratios Up since pages frequently exceed the threshold at the point when splitting occurs. At high Up, small changes in $Thresh$ can produce large changes in the number of pages and hence in $MVTU$. This is because many updates can occur at utilizations just under $Thresh$, and these might each lead to more page time splits.

As with our prior results, we perform an approximate analysis that at least partially explains the nature of the results. This permits us to compute an approximate value for $Thresh$ analytically.

For update rates near 1.0, we have for the uncompressed case, and leaving $Thresh$ as an unknown:

$$MVTU = (0.986 - Thresh*ln(2))/CR$$

Setting *MVTU* for the compressed case equal to the uncompressed value (for *Up* near *1.0*) yields

$$(0.986 - Thresh*ln(2))/CR = 0.54$$

Finally, solving for *Thresh* yields

$$Thresh = 1.41-0.78*CR$$

For *CR = 0.515*, we get a value for *Thresh* of *1.01*. This implies that one can let pages fill completely for most compression ratios. This neglects that for probabilistic distributions (as opposed to this deterministic analysis), extra time splitting makes this an overly aggressive strategy. But note that *Thresh* does get close to *1.0* at high update rates in our experiments. At smaller update rates, our experiments suggest one should be less aggressive, but setting *Thresh = 0.9* (even for our modest "2:1" compression case, i.e. *CR = 0.515*), is a sound strategy.

Our experiments and this approximate analysis both indicate that one can turn compression into a range search performance improvement, with that performance being within 10% of the performance of a conventional unversioned database.

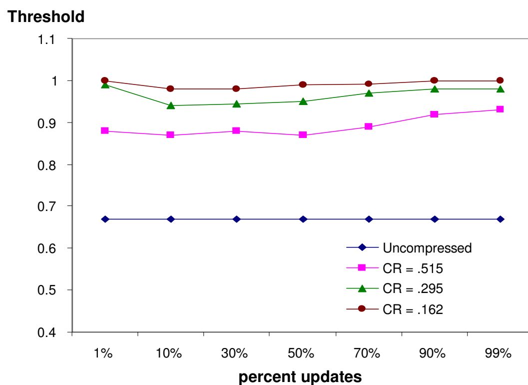

Figure 10: Key split threshold adjustment

6 RELATED WORK

There has been extensive research on temporal and versioned databases and their applications [11, 35, 41, 42]. Much work, especially earlier papers, focused on theoretical foundations, not on practical considerations such as storage efficiency and indexing versioned data, this paper's focus. We briefly review some of the work done in the area. For a good survey we refer the reader to [35]. Extensive bibliographies have also been compiled [24, 38, 44].

6.1 Temporal Indexing and Compression

Many indexing structures [2, 5, 9, 14, 16, 23, 25, 39] have been proposed for versioned and temporal data. A good survey of temporal indexing has appeared in [37]. Most of these alternative indexing techniques have drawbacks.

The drawback of the Time Index [14] is the size of the index, which is quadratic in the number of indexed time ranges. The Time Index+ [25] improves upon the Time Index, substantially reducing the storage needed for the index while improving query performance. However, worst case storage remains quadratic.

The TP-index [39] maps a (one-dimensional) time range to a point in two-dimensional space (<low time, high time>), and the querying is reduced to a spatial search problem. It is more space efficient than the Time Index, but is biased toward some types of queries. Moreover, it is highly specialized to the mapping, and the integration into existing RDBMSs is challenging.

The Interval B-tree (IB-tree) [2] has also been developed to overcome the weaknesses of the Time Index. The original main interval tree memory model is transformed to an efficient secondary storage structure while preserving optimal space and time complexity. The disadvantage of the IB-tree is that the complex three-fold structure of the interval tree is retained, and a dedicated structure of its own is used for each level. This complexity makes the implementation inside a commercial RDBMS challenging.

The Interval B+-tree (IB+-tree) [9] addresses the problem of indexing the temporal dimension in valid time databases where the temporal information of data objects are represented as valid time intervals. Here, the concept of time splits is introduced as a successful heuristic to avoid large fruitless scans. However, a limitation of the proposed structure is that time-splits are applied only to the leaf level. Moreover, the IB+-tree also requires a complex nested data structure, which makes it difficult to integrate into existing DBMSs.

The monotonic B+tree [14], the Append-Only Tree [16], and the Snapshot Index [45] also aim at indexing time-based data. None of these indexes, however, employ multiversion compression which both saves space and improves query performance.

A recent paper [23] studies the problem of efficiently indexing data with "branched evolution". The main contributions here are the extension of temporal index structures to data with branched evolution and a steady state analysis that estimates the performance of the different index structures and provides guidelines for the selection of the most appropriate one.

The multiversion B+tree (MVBT) [5] has fine performance. However, as discussed early in this paper, the MVBT moves current data instead of historical data during a time split, and hence does not progressively move historical data to another storage medium as the TSB-tree does. Moving historical data to a new page is essential if one wants historical data on an archival medium while continuing to access current data on its original medium. In addition, the MVTB's root* is not as good a fit with the SQL Server B+tree implementation as is the TSB-tree. Finally, the MTBT performs page merges, which we decided to avoid because it causes complications when we represent index terms like chains of data record versions. Permitting page merges would require that an index version chain fork at the merged page, and hence further complicate index page splitting. This complication would be on top of the one introduced by the TSB-tree moving of historical pages in a split.

Related to our version compression technique is the idea of temporal coalescing [12]. Temporal coalescing merges the temporal extents of value-equivalent tuples. Our compression technique, however, stores only the incremental differences between the values and the timestamps of the versions.

6.2 Version Support

Many database applications require the storage and manipulation of different versions of data objects. To satisfy the diverse needs

of these applications, a number of versioning solutions for database systems have been proposed.

The first system offering transaction time functionality was Postgres [41]. Postgres had reasonably complete transaction time functionality, but it depended, in part, on a recovery technique that exploited stable random access memory for the cache, making it less than ideal as an evolutionary starting point. Postgres used R-trees [17] to index historical data, but not current data, for which a B+tree was used. This was important as R-trees have difficulty supporting, in a straightforward way, data that is current and hence does not yet have an end time.

Recently, support for multiple versions of complex data, e.g., XML [10], object oriented [8], and spatio-temporal data [18] have been proposed. In [3], the authors describe a versioning model that uses signature patterns, a hash table and B+ trees to support multiple versions. In [1], VQL, a language designed for querying data stored in multiversion databases is introduced. VQL is based on a first order calculus and provides users with the ability to navigate through object versions modeled by the database.

DEC (now Oracle) Rdb [19] 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 [15] for the same reason.

In [36], 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.

Multiversion support in data warehouses has been addressed in [46]. Here the authors maintain a data warehouse under changes of schemas and contents based on explicit versioning of the whole data warehouse (i.e. schema and data). The model of a multiversion data warehouse can maintain real and alternative versions of the whole data warehouse and allows running queries that span multiple versions and compare various factors computed in those versions, as well as to create and manage alternative virtual business scenarios required for the what-if analysis. The focus of [46] is on physical sharing of data between several data warehouse versions which is similar in spirit to our proposed version compression scheme.

6.3 Industrial Interest

Transaction time functionality has also received some industrial interest, particularly from Oracle. Oracle 9i included support for transaction time [34]. Its 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. In comparison, our work is the first industrial effort to provide logarithmic time access to historical versions of data.

Other database-related products also begin to provide temporal support. LogExplorer from Lumigent [33] provides an analysis tool for Microsoft SQL Server logs, to allow viewing how rows change over time (a nonsequenced transaction time query) and then to selectively back out and replay changes, on both relational data and the schema (it effectively treats the schema as a transaction-versioned schema). aTempo's Time Navigator [4] is a data replication tool for DB2, Oracle, Microsoft SQL Server and Sybase that extracts information from a database to build a slice repository, thereby enabling image-based restoration of a past slice; these are transaction time slice queries. IBM's DataPropagator [20] can use replication of a DB2 log to create both before and after images of each row modification to create a transaction time database that can be later queried. These products, however, are built outside the database engine, and do not employ any transaction time indexing for storage. Further, when processing queries, they may incur significant storage and processing overhead.

7 CONCLUSIONS AND FUTURE WORK

7.1 Summary

Temporal support is becoming increasingly important in the commercial market as indicated by the FlashBack temporal functionality provided by Oracle [34]. Oracle has been actively advocating that the SQL standard be extended in this direction.

It has been an essential goal of Immortal DB to be able to index historical versions effectively. Thus, we have implemented the TSB-tree by modifying the SQL Server B+tree implementation. This was both an added complication, requiring dealing with a very large code base, but also a great help as the B+tree gave us an existing framework upon which to build.

Our TSB-tree deals with the full set of implementation issues: representing and managing index terms, page splitting and splitting policies, range searches, etc. Our overall goal has been to provide performance for the TSB-tree that is very close to that provided by the SQL Server B+tree. Indeed, Immortal DB executes SQL Server B+tree code for current queries.

Version compression further improves storage efficiency and range search performance. Our backward delta technique works very well within the TSB-tree context, where the last version of any record or index term on a page is uncompressed. Thus, compression is completely handled within a single page. The result of compression is to improve, at high compression ratios dramatically, both storage efficiency and performance. This was confirmed both by experiments and analysis.

7.2 Future Work

We continue to strive to narrow even further the performance differences that exist between transaction time database functionality and current time functionality, both for update and for range query. So we continue our search for additional optimization opportunities. We also want to further enhance the utility of Immortal DB. We have already implemented recovery from bad user transactions [32]. Using transaction time historical versions to provide a backup for current data, as previously suggested [30], remains on our agenda.

8 REFERENCES

[1] T. Abdessalem and G. Jomier: VQL: A Query Language for Multiversion Databases. *International Workshop on Database Programming Languages*, 160--179, 1998.

[2] C.-H. Ang and K.-P. Tan: The Interval B+tree. *Information Processing Letters*, 53, 2, 85--89, 1995.

[3] G. Arumugam and M. Thangaraj: An efficient multiversion access control in a Temporal Object Oriented Database. *Journal of Object Technology*. 2006.

[4] aTempo: *aTempo*. http://www.atempo.com/

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

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

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

[8] A. Björnerstedt and C. Hultén: Version control in an Object-oriented Architecture. *Object-Oriented Concepts, Databases, and Applications*, 451--485, 1989.

[9] T. Bozkaya and M. Ozsoyoglu: Indexing Valid Time Intervals. *DEXA*,. 541--550, 1998.

[10] S. Chien, V. Tsotras, C. Zaniolo, and D. Zhang: Efficient Complex Query Support for Multiversion XML Documents. *EDBT*, 161--178, 2002.

[11] J. Clifford, C. Dyreson, T. Isakowitz, C. Jensen, R. Snodgrass: On the Semantics of "Now" in Databases, *ACM TODS* 22, 2, 171--214, 1997.

[12] C. Dyreson: Temporal Coalescing with Now Granularity, and Incomplete Information. *SIGMOD* 169-180, 2003.

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

[14] R. Elmasri, G. Wun, and V. Kouramajian: The Time Index and the Monotonic B+ tree. In [42] Chapter 18, 433--456, 1993.

[15] S. Gukal, E. Omiecinski, and U. Ramachandran: An Efficient Transient Versioning Method. *British National Conference on Databases*. 155--171, 1995.

[16] H. Gunadhi and A. Segev: Efficient Indexing Methods for Temporal Relations, *IEEE TKDE* 5,3, 496--509, 1993.

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

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

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

[20] IBM: IBM Data Propagator. http://www306.ibm.com/software/data/integration/replication

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

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

[23] K. Jouini, and G. Jomier: Indexing multiversion databases. *CIKM,* 915 -- 918, 2007.

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

[25] V. Kouramajian et al: The Time Index+: An Incremental Access Structure for Temporal Databases. *CKIM*, 296--303, 1994

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

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

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

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

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

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

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

[33] Lumigent: Lumigent Log Explorer. http://www.ssw.com.au/ssw/LogExplorer/

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

[35] G. Ozsoyoglu and R. Snodgrass: Temporal and Real-Time Databases: A Survey. *IEEE TKDE*, 7, 4, 513--532, 1995.

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

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

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

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

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

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

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

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

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

[45] V. Tsotras and N. Kangelaris. The Snapshot Index, An I/0 Optimal Access Method for Timeslice Queries. *Information Systems*, 3, 20, pp. 237--260, 1995.

[46] R. Wrembel and T. Morzy: Managing and Querying Versions of Multiversion Data Warehouse. *EDBT*, 1121--1124, 2006.