Transaction Time Support Inside a Database Engine

David Lomet, Roger Barga
Microsoft Research
Redmond, WA
{lomet,barga}@microsoft.com

Mohamed F. Mokbel *
University of Minnesota
Minneapolis, MN
mokbel@cs.umn.edu

German Shegalov *
Max Planck Institute
Saarbrucken, Germany
shegalov@mpi-sb.mpg.de

Rui Wang *
Northeastern University
Boston, MA
bradrui@ccs.neu.edu

Yunyue Zhu *
New York University
New York, NY
yunyue@cs.nyu.edu

Abstract

Transaction time databases retain and provide access to prior states of a database. An update "inserts" a new record while preserving the old version. **Immortal DB** *builds transaction time database support into a database engine, not in middleware. It supports* **as of** *queries returning records current at the specified time. It also supports snapshot isolation concurrency control. Versions are stamped with the "clock times" of their updating transactions. The timestamp order agrees with transaction serialization order.* **Lazy timestamping** *propagates timestamps to transaction updates after commit. Versions are kept in an integrated storage structure, with historical versions initially stored with current data.* **Time-splits** *of pages permit large histories to be maintained, and enable time based indexing, which is essential for high performance historical queries. Experiments show that Immortal DB introduces little overhead for accessing recent database states while providing access to past states.*

1 Introduction

Research in temporal databases has been pursued for almost twenty years (e.g., see [2, 9, 12, 14, 18, 20, 29, 31, 34]). However, the migration of this research into commercial databases is limited. Historical data in commercial databases is usually supported only for the very recent past, providing what is called *snapshot isolation* [1, 3, 23]. With snapshot isolation, reads are not blocked by concurrent updates. A reader reads a *recent* version instead of waiting for access to the current version. However, there are no commercial database systems that support temporal functionality with historical versions organized for rapid retrieval, nor with timestamping that is highly precise. Further, layering temporal support on top of a database system is cumbersome and typically is not practical [16, 35].

In this paper, we present **Immortal DB**, our research prototype that provides transaction time database support *built into* the SQL Server database engine, not layered on top. With *Immortal DB*, regular insert/update/delete actions never remove information from the database. Rather, these actions add new data versions, thus maintaining such a complete, query-able history of states of the database.

1.1 Temporal Databases

Two forms of temporal database functionality have been identified:

Valid time is the real world time at which information recorded in the database becomes true (or no longer true) [10, 11]. For example, J. Smith might have started work at X Corp. on May 1, 1994 and left it on Oct. 7, 2001.

Transaction time is the time at which information is posted to the database [14, 15, 25]. In our example, J. Smith's original record might not have been posted until the end of the quarter, e.g. June 30, 1994.

Often valid time and transaction time are very close, perhaps identical. For example, the transaction time for an online purchase of a book at Amazon is also the "valid time" for the sale of the book. A database that supports both functionalities is termed bi-temporal (e.g., see [5, 17]).

In the *Immortal DB* project, we focus on transaction-time functionality. There are several scenarios in which transaction-time support is particularly useful.

* Work done while on Microsoft internship.

Snapshot isolation [3]. Snapshot isolation needs to access recent data versions in a transaction consistent way.

Data auditing [6]. For auditing purposes, a bank finds it useful to keep previous states of the database to check that account balances are correct and to provide customers with a detailed history of their account.

Data analysis [24]. A retailer keeps versions of the sales transaction data to answer queries about changes in inventory, and may want to mine that data for sales trends.

Backup [22]. The data versions preserved in a transaction time database can be used to provide backup for the current database state. Such a backup is done incrementally, is query-able, and can always be online.

Moving objects [7]. Keeping historical data supports tracing the trajectory of moving objects in location-aware service applications.

1.2 Immortal DB Project

An *Immortal DB* database maintains multiple versions of data. When a transaction inserts/updates new data records into the database, it stores with the new record version a timestamp T_i that indicates the beginning of the lifetime of the newly inserted/updated data. With a subsequent *update*, a new version of the data is *inserted* into the database, marked with its timestamp T_j ($T_j > T_i$), indicating its start time. The prior T_i version of data is marked implicitly as having an end time of T_j. A delete is treated as an update that produces a special new version, called a "delete stub", that indicates when the record was deleted. Old versions of records are *immortal* (i.e., are never updated in place). Each new version of a record is linked to its immediate predecessor version via a *version chain*.

The *Immortal DB* prototype, demo'd earlier [19], extends DBMS functionality without disturbing much of the original SQL Server code. Changes include:

SQL DDL syntax. A table is defined as a transaction time table via an **"Immortal"** attribute in the table create statement.

Query syntax. A new transaction statement **"AS OF"** clause specifies queries of historical data, including both persistent versions and snapshot isolation versions.

Commit processing. Timestamping of versions after transaction commit guarantees each version a timestamp that is consistent with serialization order.

Page manager. Record versions are chained together within a page.

Recovery manager. New log operations are defined to enable recovery redo and undo the "versioned" updates required for transaction time support.

Storage manager. New pages are acquired via time based page splitting to permit the space for versions to grow while facilitating high performance time based accesses.

Immortal DB is distinguished from conventional database systems and from some prior transaction time research systems in both its approaches to timestamping and to managing versions.

1. A record version *timestamp* is chosen to agree with transaction serialization order. A unique *lazy timestamping* mechanism, which **does not require logging**, propagates the timestamp to all updates of the transaction.
2. Versions of a record are stored in an integrated storage structure where historical versions initially share the same disk page as the current record version and are linked via a *version chain*. Pages of this structure are split based on time to support arbitrary numbers of historical versions.

Immortal DB is the first transaction time database implementation that organizes versions for high performance access. It provides precise time based query support via high resolution clock-based timestamping. It demonstrates that it is possible to provide this functionality without compromising performance for conventional database tables or for conventional current state database functionality. It also supports snapshot isolation with excellent performance, as confirmed by our experimental study. Performance is our rationale for building transaction time support into the database engine. Layered approaches find high performance difficult to provide.

1.3 Paper Overview

The rest of this paper is organized as follows: Managing timestamps (i.e., timestamp representation and assignment) in our *Immortal DB* prototype is outlined in Section 2. Section 3 outlines the version management (i.e., versioning chain of both records and splitting pages) in the *Immortal DB* prototype. The basic functionalities of *Immortal DB* are discussed in Section 4. Experiments described in Section 5 investigate the overhead introduced to provide transaction-time support. In Section 6, we discuss prior work in terms of its timestamp and version management. Finally, Section 7 summarizes the paper and points out future research directions for the *Immortal DB* prototype.

2 Immortal DB Timestamping

Timestamp management needs to deal with two main issues:

1. How is a transaction's timestamp chosen? It must be close to the clock time at which a transaction is executing and be consistent with the ordering of transactions.

2. How is this time propagated to the versions of records updated by a transaction, which all need the same timestamp?

2.1 Choosing a Transaction's Time

What Time, When Chosen

SQL Server supports a number of isolation modes, including serializable, via fine grained locking and ARIES style recovery [26]. Recently, it has added support for snapshot isolation which permits reads to proceed without locking and hence without conflicts with on-going updates. The timestamps of transactions must order the transactions correctly, including serialization ordering when that option is chosen.

Choosing a timestamp at transaction start, and using the start time as the transaction's time makes the timestamp available whenever a record is updated, and so it can be posted to the record version during the update. Further, timestamp order (TO) concurrency [4] correctly serializes transactions using this time. Unfortunately, TO can result in a large number of aborts as transactions that serialize in an order different from their already chosen timestamps must be aborted.

Instead of early choice, *Immortal DB* chooses a timestamp as late as possible. In the absence of requests for CURRENT TIME (a SQL feature in which a transaction can request its time during transaction execution [14]), this chooses the timestamp at transaction commit, a choice also suggested by others [30, 32]. Late choice means that the timestamp can be chosen when transaction serialization order is known. Thus, in *Immortal DB*, we choose a transaction's timestamp to be its commit time, guaranteeing that it is consistent with transaction serialization order. However, late choice means records updated during the transaction will need to be re-visited to timestamp them.

Representing and Storing Timestamps

It is useful for time used for timestamping record versions to be represented in the same way as time in the database itself or easily convertible to such a representation. This enables us to treat our timestamp as a user visible attribute that can be accessed in a WHERE clause. It also easily supports "as of" queries using a user sensible time representation.

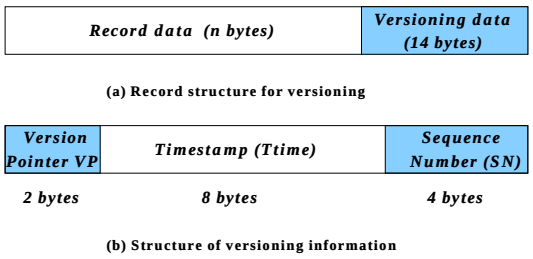


Figure 1. Record structure in *Immortal DB*.

In SQL Server, the SQL date/time function returns an eight byte time with a resolution of 20ms, which is not sufficient precision to give each transaction a unique time. Without a unique time, it is impossible to identify every database state among the versions stored in *Immortal DB*. Thus, we extend this SQL value with an additional four byte sequence number. This extension permits us to distinguish 2**32 distinct transactions within a 20 ms span, more than enough for any conceivable transaction processing system.

SQL Server snapshot isolation versioning adds 14 bytes to the tail of each record. To minimize our changes to SQL Server, *Immortal DB* utilizes these same bytes for its versioning information. Figure 1a gives the record layout used in SQL Server. Figure 1b gives the *Immortal DB* layout of these 14 bytes, specified as follows:

Version pointer VP(2 bytes). VP contains a pointer to the previous version of the record that existed prior to this update. This intra-page pointer need not be larger than 2 bytes. This is discussed more completely in Section 3.

Timestamp time Ttime (8 bytes). We initially store the TID of the updating transaction in Ttime. After a transaction commits, the TID is replaced with the time chosen for the transaction.

Sequence number SN (4 bytes). We store the sequence number used to extend the precision of our timestamp in SN to provide every transaction with a unique and correctly ordered timestamp. SN is assigned when the record is timestamped.

2.2 The Timestamping Process

Because it uses late timestamping, *Immortal DB* must revisit updated records in order to timestamp them. Two revisiting strategies have been proposed, *eager timestamping* and *lazy timestamping*. We discuss both below and explain why we chose lazy timestamping.

Eager Timestamping

Eager timestamping adds the timestamp to updated record versions before the updating transaction commits. To accomplish this, a list is maintained of the record versions

produced by this transaction. At transaction commit, we know the transaction time, and we add timestamps to all record versions on the list. Eager timestamping has two main advantages: (i) The idea is intuitive and simple to implement, involving treating timestamping as a normal update. Hence it exploits existing mechanisms. These are "in place" updates and do not themselves produce new versions. (ii) Once a transaction commits, all the records inserted/updated by this transaction are timestamped, simplifying subsequent accesses.

However, eager timestamping has drawbacks: (i) When we revisit records for timestamping, some of them may not be in main memory. This can result in extra I/O's. (ii) Transaction commit is delayed until timestamping is done, extending transaction duration, and reducing system throughput because locks are held for a longer period. (iii) Timestamping needs to be logged as well, because recovery needs to redo the timestamping should the system crash. Extra log operations reduce system throughput. The delay of transaction commit and the extra logging are serious performance liabilities that cause us to pursue the alternative below.

Lazy Timestamping

With *lazy* timestamping, timestamping is done when records are visited on their next normal access or when the page is visited for some reason, such as during a page split. We maintain a *persistent timestamp table* that contains the mapping from transaction id to timestamp, as was done in Postgres [32, 33]. While timestamp table update serializes transactions, lock duration is very short as transaction commit follows immediately. Hence, transactions are shorter and concurrency and system throughput are increased. Uniquely, *Immortal DB* garbage collects the timestamp table incrementally, which we show below can be done without logging the timestamping.

Data Structures:
We maintain two data structures to perform *lazy* timestamping:

Persistent timestamp table (PTT). PTT is a disk table that has the format $(TID, Ttime, SN)$, where TID is the transaction identifier, $Ttime$ is the commit (clock) time of the transaction, and SN is the sequence number of the transaction. (Recall that our timestamp is a concatenation of $Ttime$ and SN.) This table is a B-tree based table ordered by TID, which permits fast access based on TID to find the related timestamp information. Since TIDs are assigned in ascending order, this also means that all recent table entries are at the tail of the table.

Volatile timestamp table (VTT). The VTT is a main memory hash table that has the format $(TID, Ttime, SN, RefCount)$. We use $RefCount$ to count versions not yet timestamped. At commit, $RefCount$ is the number of records inserted/updated/deleted by this transaction, i.e. the number of record versions that need timestamping. The VTT caches the recent and hence likely to be used entries of the PTT. It speeds the translation from TID to timestamp, and counts the number of versions for each transaction that remain to be timestamped. Note that the $RefCount$ entry is only kept in the VTT, and hence, maintaining it has a very low cost.

The Timestamping Process:
The *lazy* timestamping process has four stages:
I: Transaction Begin. A VTT entry is created when a transaction starts. At that time, we assign the transaction a TID and enter it into the $Ttime$ field, initialize its $RefCount$ field to zero, and set its SN to "invalid", meaning that the transaction is currently active and does not yet have a transaction time.
II: Inserting/updating/deleting records. New versions are initially marked with the TID of the updating transaction in the 8-byte $Ttime$ field at the tail of the record. A record marked by a TID is called a *non-timestamped* record. We also increment the $RefCount$ field for the transaction in the VTT.
III: Transaction commit. At transaction commit, we determine a timestamp for the transaction, consisting of the transaction commit time stored in $Ttime$ in the volatile table, and sequence number, stored in SN of the volatile table. The $RefCount$ in the VTT has already been set during update actions. We do not revisit updated data records as in *eager* timestamping. Rather *lazy* timestamping performs a single update to the PTT that includes the TID, $Ttime$ and SN from the VTT. These entries will be used in Stage IV to map from TID to timestamp.
IV: Accessing a *non-timestamped* record. When a *non-timestamped* record is accessed after its transaction has committed, we replace its TID with the transaction's timestamp. We consult the VTT for an entry with this TID and we replace the TID with the VTT's timestamp ($Ttime$ and SN). We decrement the VTT entry's $RefCount$ to track the number of *non-timestamped* records of this transaction.

If we don't find an entry for the TID in the VTT, we retrieve the corresponding entry from PTT, replace the TID with the retrieved transaction timestamp, and cache this entry in the VTT. We set the $RefCount$ for the entry to "undefined" so that we don't garbage collect its PTT entry.

Activities that trigger *lazy* timestamping include:

- When we update a *non-timestamped* version of a record with a later version, all existing versions must be committed, and we timestamp them all.
- Just before a cached page is flushed to disk, we check whether the page contains any *non-timestamped*

records from committed transactions. If so, we timestamp them.

- If a transaction reads a *non-timestamped* version, we timestamp it so that we can determine whether the version is in the database state being queried. We use the SN field in the VTT to indicate when the version is part of an uncommitted transaction.
- When we time split a page of the current database to provide room for additional versions, we timestamp all versions from committed transactions. Only if we know the timestamps for versions of records can we determine whether they belong on the history page. We describe this more completely below.

Lazy timestamping of *non-timestamped* data records requires that an exclusive latch be obtained on the page to enable the change to be made. We mark such pages as "dirty" so that they will be flushed to the disk appropriately. Once a record is timestamped, a subsequent read of the record will need only a read latch, and the "dirty" flag will not be changed.

Garbage collection:
If we do not remove unneeded entries from it, the PTT eventually becomes very large. Not only does this needlessly consume disk storage, but it can increase the cost for a TID lookup to find its timestamp . To keep the PTT relatively small and high performance, we garbage collect PTT entries for which timestamping is completed. However, to determine when timestamping is complete for a particular transaction is subtle.

Immortal DB does volatile reference counting using the $RefCount$ field in the VTT, of the number of *non-timestamped* records for each transaction. When the count goes to zero *AND* all pages containing the now timestamped records have been written to disk, we delete the entry for the transaction from the PTT (and VTT). We can know when the pages have been written to disk by tracking database *checkpoints*. This is essential since we are not logging timestamping and hence cannot redo it after a system crash.

Immortal DB records the LSN of the end of the log in the VTT at the time that timestamping for a transaction is complete (i.e. when its VTT $RefCount$ goes to zero). When the redo scan start point LSN becomes greater than its VTT LSN, we know that all timestamping for the transaction is stable on the disk. The redo scan start point moves as a result of checkpointing, which shortens the section of the log that needs to be scanned during redo recovery [26].

Volatile reference counting avoids the extra logging and disk write costs of stable reference counting. However, there is a small risk of a system crash before all timestamping is done and timestamped records are on the disk. Volatile reference counts would then be lost. This is not a tragedy since crashes are rare. We simply end up with certain PTT entries that cannot be deleted. However, once timestamping is done for these entries, even though we would not know this, these entries would no longer be accessed. So, at the cost of storing "in flight" transaction timestamps persistently should the system crash (a modest cost), we make reference counting very low in cost.

Because the PTT is organized as a B-tree ordered by TID, its active part, consisting of entries for the most recent transactions, is clustered at the tail of the table. So access to the PTT remains fast, even though the PTT grows whenever the system crashes. It is possible to delete the otherwise ungarbage collectible entries by forcing all pages to eventually time-split and become historical pages, as was described in [22], which was similar to the "vacuuming" process in Postgres [32].

Finally, *Immortal DB* also supports snapshot versions (for snapshot isolation concurrency control). Snapshot versions do not need to persist across system crashes. Thus, we do not store $< TID, timestamp >$ entries for snapshot transactions in the PTT. Entries are stored only in the VTT. Further, we can drop the VTT entry for a snapshot transaction immediately upon its reference count going to zero.

3 Immortal DB Versioning

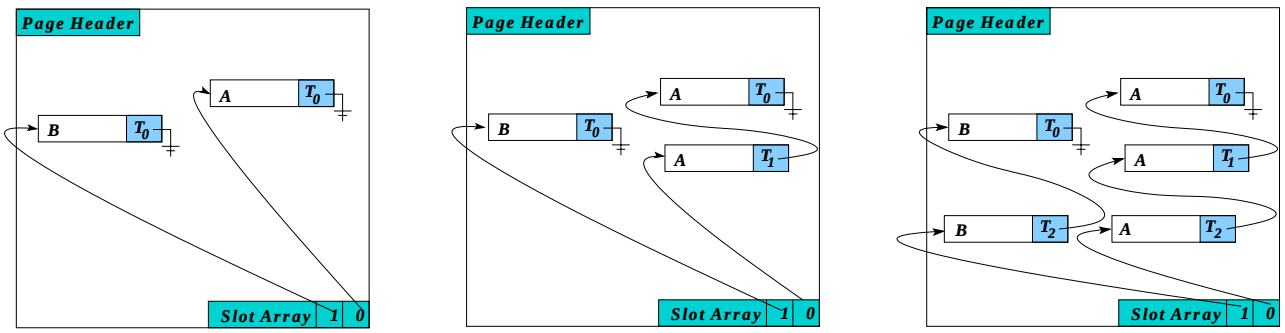
We change the database storage engine to maintain a record version for each transaction updating a record. Changes are made in record structure, page structure, page splitting, and indexing. The result is a single integrated data access structure that houses all versions, current and historical. Versioning requirements differ for transaction time vs snapshot isolation:

Snapshots: Only recent versions need to be maintained. *Immortal DB* keeps track of the time of the oldest active snapshot transaction O. Versions earlier than the version seen by O are garbage collected. Because snapshot transactions are aborted at a system crash, snapshot versions need not persist across crashes.

Transaction Time: Versions persist for an extended time. Because of the potential sizable number of historical pages, it is desirable to index directly to the right page to avoid the overhead of searching back along the version list. In addition, versions need to persist across system crashes, and so are not garbage collected.

3.1 Record Structure

Storing versioning and timestamping information requires having additional bytes with each data record. *Immortal DB* utilizes 14 bytes at the tail of the record structure currently used by SQL Server for snapshot isolation versioning, but with a different format. Figure 1 gives the



(a) Transaction I: Insert A, Insert B **(b) Transaction II: Update A.** **(c) Transaction III: Update A, Update B.**

Figure 2. Page structure in *Immortal DB*.

structure of the extra bytes to store both timestamp and version information. Timestamp information utilizes 12 bytes and is described in Section 2.1. The remaining two bytes are used to point to the previous version of the record.

Immortal DB initially stores older versions of a record in the same page as the current record version. A 2-byte pointer is sufficient to point to the previous version in the same page. All versions are accessed through a versioning chain that originates from the most recent record, until the disk page is full. When the page becomes full, the older version are moved to another page and the version pointer (VP) field is used to store the slot number of the earlier version in the "historical" page. We maintain a pointer in the current page that references the historical page containing these slots.

3.2 Page Structure

The page structure in conventional database systems includes a header at the start of the page and a slot array at the end of the page. All records in the page are accessible through a link from the slot array. The *Immortal DB* prototype has two main changes to this page structure:

Versioning chain. Historical versions for the current records are accessed through the version chain and are only indirectly accessible from the slot array. A current transaction sees the same records via the slot array as it would with the conventional page structure.

Page header. Two fields are added to the disk page header;

- *history pointer*. This points to the page that contains versions for records that had once been in the current page at an earlier time. These older versions can be traced through different pages via a version chain of pages.
- *split time*. This field contains the time used to time split the page, i.e. the start time for versions in the page. By examining this field during an "as of" query, we may skip searching for records in this page.

History pointer and *split time* are assigned in the page time split procedure (Section 3.3).

Figure 2 illustrates the layout of a page and describes how it changes during three transactions. Transaction I (Figure 2a) inserts two records A and B. This insertion is done as in conventional databases where two entries in the slot array are allocated to point to the newly inserted records. Transaction II updates record A, allocating a new version of A in the disk page. This version points to the old version of A. The slot array entry now points to the new version of A. Figure 2b gives the layout of the disk page after executing transaction II. Transaction III (Figure 2c) updates both records A and B. New versions are allocated for A and B that point back to their previous versions. The slot array entries now point to these latest versions of each record.

3.3 Page Time Split

Time splitting is a unique feature of *Immortal DB* and greatly increases the efficiency of its queries. Historical versions are initially stored in the current page. When the current page is full, we either do a B-tree style key split or a time split. A key split splits the records by key range and distributes records over two pages, both holding current data. A time split, which we do using the current time, moves historical (non-current) record versions out of a current page into a historical page separate from the current data.

A time split is unlike a key split which partitions the record versions of an existing page. For a page time split, we distinguish four cases when assigning record versions between current page and historical. These are illustrated in Figure 3 and described below:

1. Record versions with end times before the split time are moved to the historical page.

2. Record versions whose lifetimes span the split time are copied to the historical page. Also, they (redundantly) stay in the current page.
3. Record versions whose lifetimes start after the split time remain in the current page.
4. Uncommitted records remain in the current page.

Some record versions are replicated to both pages. In Figure 3, this is true for the only version of Record A, the earlier version of record B, and the center version of record C. The most recent version of B is only in the current page, the earliest version of C is only in the history page, a delete stub for C is only in the current page. Delete stubs earlier than the split time are removed from the current page since their purpose is to provide an end time for an immediately prior version in the same page.

The essential point is that each page contains all the versions that are alive in the key and time region of the page. With a time split, we assign a historical page the time range that starts at the split (start) time for the current page before the time split (this becomes the historical page's start time), and ends at the new value for split time of the current page after the time split. Records that cross the new split time appear in both pages. Because historical versions are read-only and never updated, no update difficulties are introduced due to this redundancy.

We key split a page in addition to performing a time split if storage utilization after a time split is above some threshold T, say 70%. This ensures that, in the absence of deletes, storage utilization for any time slice will, under usual assumptions, be $T * ln(2)$. The impact of this choice under varying assumptions is explored in [21]. Current time range search performance is lower than in B-trees, where $T = 100\%$ due to the reduced utilization of current records per page.

When time splitting a current page, we include in it a pointer to the newly created historical page. We include with that pointer the split time we used to divide the versions between current and history pages. This chain of pages, each with an associated earliest time enables faster querying of versioned data.

3.4 Indexing Versions

When accessing recent versions for a snapshot isolation transaction, the split time in the current page tells us whether we will find the version of interest in the current page or need to access an historical page. We expect to usually find the desired recent version, even if it is not the current version, in the current page. Occasionally, we will need to access the first historical page, more rarely will we need to traverse deeper into the page chain.

Thus, searching for recent versions requires a visit to the current page. If not there, we search the time split chain of

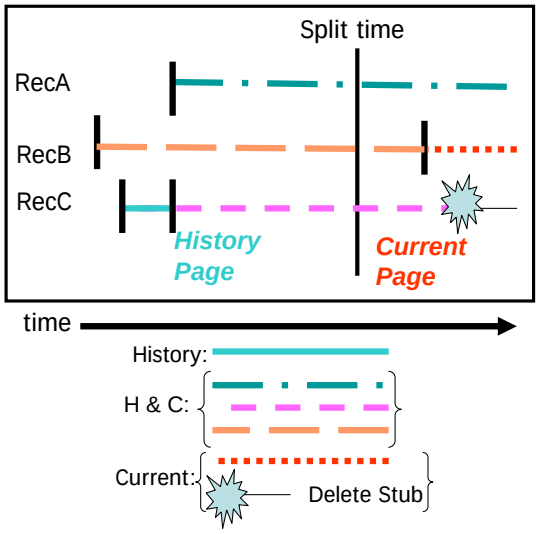


Figure 3. Time Splits in *Immortal DB*.

pages back until we encountered a page containing versions in the time range of interest. This behavior is more than acceptable for snapshot isolation, but is unacceptable for persistent versions, where the time split page chain can be quite long.

The time-split B-tree (TSB-tree) [20, 21] was designed for indexing historical versions. It indexes the collection of time split and key split data pages that we have been describing. Once the TSB-tree is supported in *Immortal DB*, we will directly access the data pages that are needed to satisfy any historical query. This indexing is enabled by time splitting, which ensures that all versions that lived in each page's time range will be stored there. Storing a version of a record that lives across the time split boundary in both of the pages resulting from the time split accomplishes this.

4 Immortal DB Functionality

In this section, we describe how a user, writing in the SQL language and using our extensions, can specify the temporal functionality supported by *Immortal DB*.

4.1 Defining an IMMORTAL table

We define an *immortal* property on database tables, i.e., we explicitly specify if a certain table is *immortal*. The keyword `Immortal` is added to the `Create Table` statement to indicate that the table should have persistent versions. Non-immortal tables, i.e., conventional tables, can still make use of our prototype for supporting snapshot versions, i.e., recent versions used for concurrency control, by enabling snapshot isolation using an `Alter Table` statement.

The following example shows the creation of an *immortal* table named `MovingObjects`. Figure 4 shows a screen shot of our application using this table. This table is used for our experiments in Section 5.

```
Create IMMORTAL Table MovingObjects
(Oid smallint PRIMARY KEY,
LocationX int,
LocationY int) ON [PRIMARY]
```

By recognizing the new keyword `IMMORTAL`, we set a flag in the table catalog that indicates the *immortal* property of that table. This flag is visible to the storage engine. The "immortal" flag in the table catalog determines three things:

1. There is no garbage collection of historical versions in an *immortal* table.
2. When an update transaction commits, we insert an entry for it into the PTT for lazy timestamping of its updates.
3. *Immortal* tables enable "AS OF" historical queries of versioned data.

The rest of *Immortal DB* functionality (e.g., volatile timestamping, versioning chain, etc.) can also be exercised via conventional tables enabled for snapshot isolation, since this requires keeping track of recent versions with their timestamps.

4.2 Querying in Immortal DB

Immortal DB can support many forms of temporal applications (e.g., the list of applications mentioned in Section 1). Some of these need new query processing/optimization techniques. Our *Immortal DB* implementation focused on changes to the SQL Server storage engine. However, as a proof of concept, it supports "as of" queries. We did this by modifying the snapshot isolation capability already present. We extended the SQL Server syntax for the "Begin Transaction" statement by adding the keyword clause `AS OF`. The query below asks for the first ten moving objects "as of" some earlier time.

```
Begin Tran AS OF "8/12/2004 14:15:20"
SELECT * FROM MovingObjects
WHERE Oid < 10
Commit Tran
```

To process this query, we traverse a B-tree index on a primary key to find current records for the ten objects. Then we traverse record version chains starting at these current versions. We describe here only our impending optimized page chain traversal. For this, we check the current page's split time. If "as of" time is later than split time, the version we want is in the current page. Otherwise we follow the page chain to the page holding versions for the requested time. When we replace the B-tree by a TSB-tree, we will index directly to the appropriate page, avoiding the cost of searching down the page time split chain.

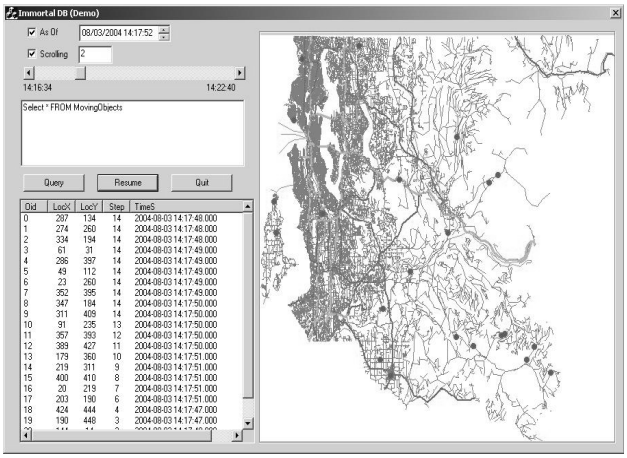


Figure 4. Moving objects in the Seattle Area.

We then follow the record versioning chains in the identified page to find the versions with the largest times earlier than "as of" time. Thus we only follow record version chains in the one page that must contain the version of interest.

The layout and structure of both records and pages allow for supporting several kinds of queries that exploit and/or mine the history. In particular, our versioning is capable of supporting "time travel" where we ask to see the history of a particular object(s) over time. This functionality requires changing the query processor, which is beyond the scope of our project.

5 Experimental Results

We performed several experiments with *Immortal DB* to evaluate its performance. The experiments focused on two aspects:

- The overhead of regular transactions. As *Immortal DB* provides new functionality, one expects to see additional overhead. However, our experiments show this overhead is quite low.
- The performance of "AS OF" queries. Here one expects to encounter some overhead as we need to look through the versions maintained to find the appropriate version.

We used the *Network-based Generator of Moving Objects* [8] to generate a set of moving objects (e.g., vehicles, trucks, cyclists, etc) on a road network. Figure 4 gives a screen shot of the generator where objects (the circles) are plotted on the road network of the Seattle area. Once an

object appears on the map, it sends an `Insert` transaction to the *Immortal DB* server that includes the object ID and location. Our *Immortal DB* server appends a timestamp to the inserted record using the lazy timestamping method. When an object moves, it sends an `update` transaction to the *Immortal DB* server to update its location. Moving objects have variable speeds, i.e., they submit `update` transactions at different rates. Also, a moving object has a predetermined source and destination. Once an object reaches its destination, it stops sending update transactions to the server. Thus, not all moving objects have the same number of updates.

The experiments in this section were run on an Intel Pentium IV 2.8GHz processor with 256MB of RAM, and running Windows XP. The base DBMS is SQL Server, and *Immortal DB* is implemented using C++ extensions to SQL Server. DB Page size is 8KB.

5.1 Transaction Overhead

We investigated the overhead of insert/update transactions. Figure 5 compares the execution of up to 32,000 transactions for the *Immortal DB* and a traditional database. Among the 32K transactions, only 500 are `Insert` transactions, the rest are updates. The performance of the *Immortal DB* is greatly affected by the number of inserted/updated records per transaction. Regardless of transaction size, each separate transaction requires an update to the persistent timestamp table. The lowest overhead case is when all the 32,000 records are inserted/updated within only one transaction. In this case, *Immortal DB* updates the timestamp table only once. The highest overhead case is when each transaction updates or inserts only one single record. In this case, we update the timestamp table 32K times, which results in more I/O and CPU processing. We do not show the lowest overhead case as it is indistinguishable from non-timestamped updates. The highest overhead case with 32K transactions resulted in *Immortal DB* having an 11% overhead over conventional tables.

As expected, *Immortal DB* has an overhead over the traditional database where it provides an additional functionality. The overhead results from two factors:

- With a transaction-time table, 32K transactions result in 32K records being stored on the disk. In a traditional database, only 500 records are stored on the disk. Thus, *Immortal DB* needs to allocate and access more disk storage when providing transaction time functionality.
- In every `update` transaction (we have 31,500 update transactions), *Immortal DB* consults the timestamp table to timestamp the previous record versions. We don't consult the timestamp table for `inserts` (500 transactions) where a newly inserted record is initialized by the transaction ID as its temporary timestamp. However, each `update` timestamps, not the newly inserted record version, but the prior version by replacing its transaction ID with its assigned timestamp from the timestamp table.

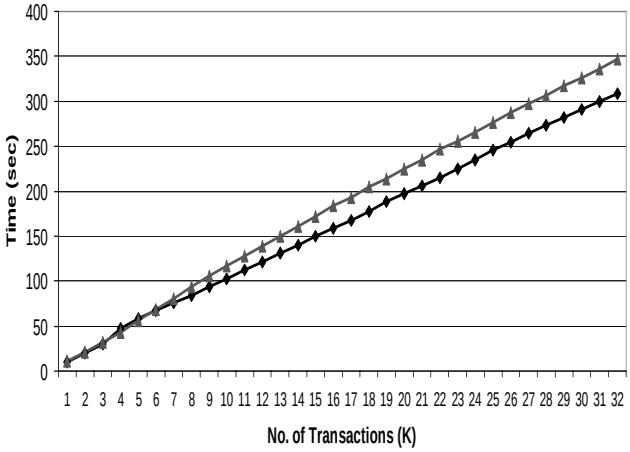


Figure 5. Transaction overhead in *Immortal DB*.

For the case of 32K transactions, transactions for a conventional table average in 9.6 msec. To support a transaction time table, we add an additional 1.1 msec to each transaction, or about 11%. This is really a worst case measure as each transaction updates a single record. If we include many updates within one transaction, we would have about the same 1.1 msec overhead, but the overhead percentage would be much lower. The reason for this small overhead is that there is a large common part in the execution of both transaction time (*immortal*) and conventional tables.

5.2 AS OF Queries

Next we investigated full table scan "AS OF" queries. Figure 6 graphs the results of experiments with 36,000 transactions having different ratios between the number of updates and the number of insertions. The experiment was repeated four times with insertions of 500, 1000, 2000, and 4000. We restrict the number of transactions to be equal for each experiment. Thus, for these experiments, each record is updated 72 times for the 500 insert case, 36, 18, and 9 times for the others respectively. An "as of" query that asks about the recent history will have better performance with lower number of inserts, basically because the number of retrieved records is smaller. However, as we go back in history, the performance advantage reverses because those records are updated more frequently. The more updates,

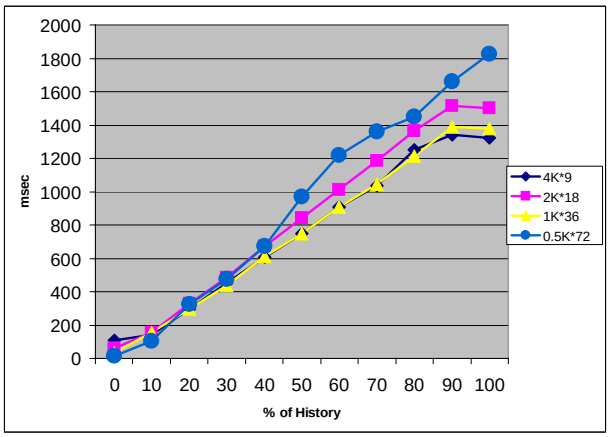


Figure 6. The effect of insertions/updates on AS OF queries.

the lower the performance, because the version chains are longer.

It is not surprising that a query over older data has a much larger response time. We currently sequentially scan the chain of pages starting at the current page, looking for the page responsible for records with the "as of" time. Then, within the required page, we search for the qualified records. We expect that the performance of "as of" queries, independent of the time requested, to equal current time queries once we implement the TSB-tree to index the versions.

6 Related Work

Here we describe how timestamp management and version management, are performed in three database systems that support versioning and historical queries of some form. Not described are middleware systems supporting some flavor of snapshot versioning, frequently used for higher availability via replication. They are rarely intended to provide "as of" query capability to multiple versions. Transaction time functionality is either not present, is greatly circumscribed or has limited performance.

6.1 Rdb

Oracle Rdb [13] supports snapshot isolation, and serializable transactions with fine grained locking. It requires version ordering based on transaction serialization order, which is determined at transaction end, a problem it shares with Immortal DB. Because it only supports snapshots, however, it can take a different approach, using commit-lists [13, 27], a clever technique that avoids having to revisit versions to do timestamping.

An update transaction stamps its versions with its TSN, like Immortal DB. At the start of a snapshot read transaction (Rdb only supports reads for snapshots), the transaction is given the list of TSNs of committed transactions. When accessing a record, the snapshot transaction searches back in time from the current version for the first version whose TSN appears on its commit list. Only transactions close in time to the snapshot need to be explicitly on the list. The others are either known (i) to be committed because they have TSNs before a lower bound or (ii) to be uncommitted because their TSNs are after the TSN of the snapshot transaction. Unfortunately, the commit list approach does not generalize to support queries that ask for results as of an arbitrary past time. Only snapshot transactions are given a commit list, and only for the time when they begin executing. Generating commit lists for earlier times is not possible.

Rdb stores historical versions in a separate storage area. Both current and historical storage areas need to be accessed for snapshot queries, unless the current version has a TSN on the commit list. Versions do not survive a crash and are not indexed by time.

6.2 Oracle and Oracle Flashback

Oracle supports snapshot isolation, but it's support of serializable transactions is only with coarse grained locking. Most users do not use atomic (serializable) transactions. The Oracle snapshot isolation technique reduces locking conflicts but makes it very difficult to include high concurrency serializable transactions. Exactly how Oracle handles timestamping of its versions is not easily accessible.

Oracle Flashback [28] supports persistent versioning in which versions are re-created from retained undo information. It searches for the right version in the undo information consisting of prior versions of records. This capability is described mostly as dealing with database corruption introduced by erroneous transactions. It solves this problem via "point in time" recovery back to a time prior to the execution of the erroneous transaction, without the need for installing a database backup and rolling forward using a redo log. So Flashback is capable of shortening the outage produced by erroneous transactions, which is a real problem.

While Flashback supports flashback ("as of") queries, that does not seem to be its design center, nor does it appear to be tuned for that purpose. If a query uses clock time for its "as of" time, the result is only approximate, since versions are identified by something analogous to a transaction identifier, not a time. Search starts with the current state, and scans back through the undo versions for the version of interest. One would expect performance to degrade the farther back in time one goes.

6.3 Postgres

Postgres [32] is closest to Immortal DB in functionality, but uses different techniques both for timestamp management and for version management.

Postgres stamps records with a "real" time that needs to be consistent with transaction serialization order. For that reason, Postgres, like Immortal DB, revisits records after commit, either on next access, or via a background "vacuuming" demon. "Vacuuming" moves old versions to an R-tree organized archival storage but also degrades current database performance [33]. Postgres uses a form of persistent timestamp table (PTT), but without garbage collection. It relies on "vacuuming" to limit the growth of the PTT.

Because Postgres moves historical verisons to a separate access structure, most "as of" queries need to access both current and historical storage structures. Otherwise, it is impossible, in general, to determine whether the query has seen the record version with the largest timestamp less than the "as of" time.

The R-tree limits Postgres in two ways. First, Postgres cannot simply maintain all the versions, including the current versions, in the R-tree because the R-tree cannot handle records with unknown end times. Second, R-tree performance on "as of" queries is problematic. Storage utilization for some timeslices in an R-tree page can be very low. R-trees do not have the kind of storage utilization characteristics produced by our time-splitting process.

7 Discussion

7.1 Research Status

We have described our *Immortal DB* effort. Its main goal is to provide transaction time support built into a commercial database system, not layered on top. This functionality also supports snapshot isolation concurrency control. *Immortal DB* does not impose any storage overhead for record structures when conventional tables are defined. Like snapshot isolation versioning, it uses extra bytes, indeed the same bytes as in SQL Server, but in a different format.

Each database change *inserts* a new version for the changed record in the database that points back to the prior version of the same record. We dealt with two main challenges: (1) Timestamping. *Immortal DB* does lazy timestamping after commit. (2) Storing versions. *Immortal DB* maintains all versions of records in a single unified storage scheme. This avoids the large cost of separately accessing current and historical data.

We extended SQL syntax to support creating *immortal* tables. We then implemented "AS OF" queries asking for the state of the database "as of" some time in the past. Our experiments show that the additional *Immortal DB* functionality results in only modest additional overhead for current time function.

7.2 Next Steps

There are a number of important features not yet included in *Immortal DB*.

Temporal Indexing: It is essential that we provide efficient access to historical versions of data. While our current performance is quite good for accessing recent versions of data, the longer a database is updated, the worse the performance is for accessing the same version (i.e. the version with the same "as of" time). This is not acceptable behavior. Fortunately, we know how to do the time based indexing [20] and providing it is next on our "to do" list.

CURRENT TIME: A SQL query can ask for CURRENT TIME within a transaction. This request needs to return a time consistent with the transaction's timestamp. This forces a transaction's timestamp to be chosen earlier than its commit time. Our main idea is to extend the conventional lock manager to keep track of earlier conflicting transactions and their timestamps. This improves on the more limited functionality in [14].

Queryable Backup: Prior work [22] described how to use a TSB-tree to implement a versioned database, where the historical pages could be used as a backup of the current database. Such a backup has three advantages. (i) It is always installed and ready to be "rolled forward", hence saving *restore* time. (ii) It is done incrementally. (iii) It can be queried. This increases the value of keeping the historical versions.

As temporal support is deployed in commercial databases, it will trigger much wider use of this functionality. And that will make it clear that we have barely started in developing database technology for this important area.

References

[1] A. Adya, B. Liskov, and P. E. O'Neil. Generalized Isolation Level Definitions. *ICDE*, 67–78, 2000.

[2] I. Ahn and R. T. Snodgrass. Performance Evaluation of a Temporal Database Management System. *SIGMOD*, 96–107, 1986.

[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] R. Bliujute, C. S. Jensen, S. Saltenis, and G. Slivinskas. R-Tree Based Indexing of Now-Relative Bitemporal Data. *VLDB*, 345–356, 1998.

[6] B. Bloor. Audit the Data - or Else. Un-audited Data Access Puts Business at High Risk. (2004) http://www.baroudi.com/pdfs/LumigentFinal.pdf.

[7] M. Breunig, C. Turker, M. H. Bohlen, S. Dieker, R. H. Guting, C. S. Jensen, L. Relly, P. Rigaux, H.-J. Schek, and M. Scholl. Architectures and Implementations of Spatio-temporal Database Management Systems. *LNCS 2520*, 263–318. Springer, 2003.

[8] T. Brinkhoff. A Framework for Generating Network- Based Moving Objects. *GeoInformatica*, 6(2), 2002.

[9] J. Clifford and A. Tuzhilin. *Recent Advances in Temporal Databases, International Workshop on Temporal Databases*. Springer, 1995.

[10] C. E. Dyreson and R. T. Snodgrass. Valid-time Indeterminancy. *ICDE*, 335–343, 1993.

[11] C. E. Dyreson and R. T. Snodgrass. Supporting Valid-Time Indeterminacy. *ACM TODS*, 23(1):1–57, 1998.

[12] S. K. Gadia and C.-S. Yeung. A Generalized Model for a Relational Temporal Database. *SIGMOD*, 251–259, 1988.

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

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

[15] C. S. Jensen, L. Mark, and N. Roussopoulos. Incremental Implementation Model for Relational Databases with Transaction Time. *IEEE TKDE*, 3(4):461–473, 1991.

[16] C. S. Jensen and R. T. Snodgrass. Temporal Data Management. *IEEE TKDE*, 11(1):36–44, 1999.

[17] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing Access Methods for Bitemporal Databases. *IEEE TKDE*, 10(1):1–20, 1998.

[18] T. Y. C. Leung and R. R. Muntz. Query Processing for Temporal Databases. *ICDE*, 200–208, 1990.

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

[20] D. B. Lomet and B. Salzberg. Access Methods for Multiversion Data. *SIGMOD*, 315–324, 1989.

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

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

[23] S. Lu, A. J. Bernstein, and P. M. Lewis. Correct Execution of Transactions at Different Isolation Levels. *IEEE TKDE*, 16(9):1070–1081, 2004.

[24] M. S. Mazer. Data Access Accountability - Who Did What to Your Data When? A Lumigent Data Access Accountability Series White Paper: (2004) http://www.lumigent.com/files/Whitepaper_DAA.pdf

[25] L. E. McKenzie and R. T. Snodgrass. Extending the Relational Algebra to Support Transaction Time. *SIGMOD*, 467–478, 1987.

[26] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17,1 (Mar. 1992) 94-162.

[27] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions. *SIGMOD*, 124–133, 1992.

[28] Oracle. Oracle Flashback Technology. (2005) `http://www.oracle.com/technology/deploy/availability/htdocs/Flashback_Overview.htm`

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

[30] B. Salzberg. Timestamping After Commit. *PDIS*, 160–167, 1994.

[31] R. B. Stam and R. T. Snodgrass. A bibliography on temporal databases. *IEEE Data Engineering Bulletin*, 11(4):53–61, 1988.

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

[33] M. Stonebraker, L. A. Rowe, and M. Hirohama. The Implementation of Postgres. *IEEE TKDE* 2(1):125–142, 1990.

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

[35] K. Torp, C. S. Jensen, and R. T. Snodgrass. Stratum Approaches to Temporal DBMS Implementation. *IDEAS*, 4–13, 1998.