Immortal DB: Transaction Time Support for SQL Server

David Lomet, Roger Barga
Microsoft Research
Redmond, WA

Mohamed F. Mokbel *
Purdue University
Lafayette, IN

German Shegalov *
Max Planck Institute
Saarbruecken, Germany

Rui Wang *
Northeastern University
Boston, MA

Yunyue Zhu *
New York University
New York City, NY

ABSTRACT

Immortal DB builds transaction time database support into the SQL Server engine, not in middleware. Transaction time databases retain and provide access to prior states of a database. An update "inserts" a new record while preserving the old version. The system supports *as of* queries returning records current at the specified time. It also supports snapshot isolation concurrency control. Versions are stamped with the times of their updating transactions. The timestamp order agrees with transaction serialization order. *Lazy timestamping* propagates timestamps to all updates of a transaction after commit. All 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. We demonstrate Immortal DB with a moving objects application that tracks cars in the Seattle area.

1. INTRODUCTION

1.1 Overview

Researchers have identified two forms of temporal database functionality: (1) *Valid time* [3] is the real world time at which information recorded in the database becomes true (or the time when they are no longer true). (2) *Transaction time* [5] is the time at which information is posted to the database. However, transfer of this research into commercial databases is limited. Historical data in commercial databases is usually only for the very recent past for *snapshot isolation* [1], a multi-version concurrency control method where a reader reads, without locking, a *recent* version instead of accessing the current version.

Immortal DB, our research prototype, builds transaction time support into a commercial database system, Sql Server, not on top. It shows that one can provide this functionality without impairing performance for conventional functionality applied to current data. It also supports snapshot isolation with excellent performance.

*Work done while on Microsoft internship.

Permission to make digital or hard copies of all or part 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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA
Copyright 2005 ACM 1-59593-060-4/05/06 $5.00.

1.2 Immortal DB Project

Immortal DB extends Sql Server without disturbing most of the original code. Regular insert/update/delete actions never remove information from the database. Rather, these actions add new versions to maintain a complete and queryable history of states of the database. *Immortal DB* makes changes in three areas that distinguish it from conventional database systems and some previous efforts to provide transaction time functionality.

Timestamping. Each record version is stamped with a clock time T_i that serializes it with the commit order of other transactions. T_i indicates the beginning of the lifetime of the newly inserted/updated data. When that data changes (e.g., with a subsequent *update*), a new version is *inserted*, marked with the timestamp T_j ($T_j > T_i$) of its transaction, indicating its start time. The old version implicitly has an end time of T_j. A record delete is treated as an update that produces a special version, called a "delete stub" that indicates when the record was deleted. Versions are thus *immortal* (i.e., never updated in place). A unique *lazy timestamping* mechanism propagates the timestamp to the transaction updates after commit.

Version management. Historical versions of a record are stored in a version chain within a page. A time-based page split grows space associated with versions beyond a single page and facilitates time based access to all versions via a unified temporal index [4].

SQL syntax. We extend the data definition language (DDL) to include an **Immortal** attribute that defines a table to be a transaction-time table. We enable historical queries by means of an **"AS OF"** clause on the "Begin Transaction" statement, requesting the status of the database *as of* some time in the past.

2. IMMORTAL DB TIMESTAMPING

2.1 When to Timestamp

Timestamps must reflect a correct serialization order for their transactions. One might choose a timestamp at transaction start, using the start time as the timestamp. The timestamp is then available whenever a record is updated, so it can be posted to the record version during the update. Timestamp order (TO) concurrency [1] correctly serializes transactions using this time. Unfortunately, transactions that serialize in an order different from the order of their

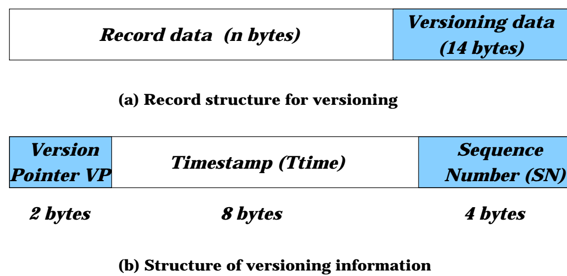

Figure 1: Record structure in *Immortal DB*.

already chosen timestamps must be aborted.

Immortal DB hence chooses a transaction's timestamp as late as possible. At commit, the transaction serialization order is already known. Thus, *Immortal DB* currently chooses a transaction's timestamp to be its commit time, hence guaranteeing that it is consistent with transaction serialization order.

2.2 Timestamp Representation

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

Transaction time Ttime (8 bytes). We initially store the transaction identifier (TID) of the updating transaction in Ttime. When the transaction commits, the TID is replaced with the transaction's commit time.

Sequence number SN (4 bytes). In SqlServer, the SQL date/time function returns an eight byte time with a resolution of 20ms, which is insufficient to give each transaction a unique time. Thus, we extend it with a four byte sequence number, SN, to give every transaction a unique, correctly ordered timestamp. SN is chosen when the transaction commits.

Version Pointer VP(2 bytes). VP contains a pointer to the immediate previous version of the record. Two bytes is sufficient for this intra-page pointer.

2.3 The Timestamping Process

With transaction time not known until the end of the transaction, *Immortal DB* revisits updated records in order to timestamp them. A *lazy* timestamping strategy is used. Records are only re-visited on the next access or when the page is re-visited for some reason (e.g., a page split). This eliminates any delay at the transaction commit, resulting in shorter duration transactions, and increasing concurrency and system throughput.

Lazy timestamping is not logged. Instead, we maintain a disk-based table, the *persistent timestamp table* (PTT) containing the mapping from TID to Ttime. In addition, we maintain a memory-based hash table, the *volatile timestamp table* (VTT). The VTT serves two purposes: (1) It caches some of the entries from PTT, thus enabling in-memory timestamping. Only entries that are not found in the VTT require disk accesses to search the PTT. (2) It maintains a reference count of the number of records not yet timestamped for each transaction. This permits garbage collection of PTT entries. Once a transaction's reference count is zero *AND* all pages containing its timestamped records have been written to disk, the transaction's entry is deleted from PTT (and VTT). We know when the pages have been written to disk by tracking database `"checkpoints"`.

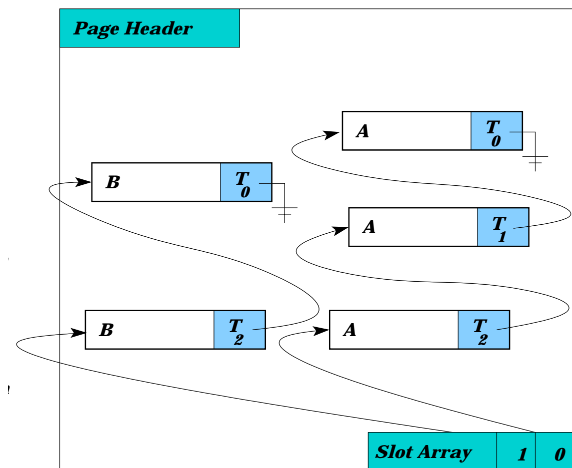

Figure 2: Page structure in *Immortal DB*.

Lazy timestamping has four stages:

Transaction Begin. A VTT entry with the reference counter set to zero is created when a transaction starts.

Inserting/updating/deleting records. New versions are initially marked with the *TID* of the updating transaction in the version's *Ttime* field. We also increment the reference counter for the transaction in VTT.

Transaction commit. At transaction commit, we set the transaction timestamp in both VTT and PTT.

Accessing a record. When a *non-timestamped* record is accessed, we replace its *TID* with the timestamp for its transaction by consulting either VTT or PTT. If an entry is found in VTT, we decrement its reference count, otherwise, we consult PTT.

3. IMMORTAL DB VERSIONING

Immortal DB supports both persistent versions for transaction-time functionality and snapshot versions for snapshot isolation concurrency control. It stores prior versions of a record in the same page as the current version. All versions are accessed through a versioning chain that originates from the most recent record, until the disk page is full. Figure 2 gives an example of the layout of the page structure after inserting three versions of record A and two versions of record B.

When the current page fills up, we either do a key split, as in a conventional B-tree, or a time split. With a key split, we go from one to two pages holding current versions of records, partitioning them by key range. A time split, unlike a key split, does not partitions the versions. With a time split: (1) a version that ends its lifetime before the split time is moved to the historical page; (2) a version whose lifetime spans the split time is copied to the historical page (It (redundantly) stays in the current page. Thus, a time split does not partition record versions.); (3) a version that starts after the split time remains in the current page; (4) an uncommitted version remains in the current page.

Immortal DB adds two fields to the page header: (1) *History pointer*. This field points to the page that contains ver-

sions that had once been in the current page but at an earlier time. In this case, older versions can be traced through different pages via a chain of pages. (2) *Split time*. This field contains the time used in splitting the page. By examining this field during an "as of" query, we may skip searching for records in this page. Both the *history pointer* and *split time* fields are assigned during the time split procedure.

The time-split B-tree (TSB-tree) [4] indexes the collection of time split and key split pages described above. Using the TSB-tree in *Immortal DB*[1] permits us to directly access the pages needed to satisfy any historical query. This indexing is enabled by time splitting, which ensures that each page contains all versions with lifetimes within its time range. This improves access performance while increasing somewhat the number of version copies.

4. IMMORTAL DB FUNCTIONALITY

In this section, we describe how a user specifies temporal functionality in *Immortal DB's* extended SQL language.

4.1 Defining an IMMORTAL table

The keyword `Immortal` is added to the `Create Table` statement to indicate that the table should have persistent versions. Conventional tables, can be used in our prototype to support snapshot versions, i.e., recent versions used for concurrency control, by enabling snapshot isolation with an `Alter Table` statement. The following example shows the creation of an *immortal* table named `MovingObjects`.

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

The keyword `IMMORTAL` causes us to set a flag in the catalog entry for the table, indicating the *immortal* property. This has three effects: (1) It disables garbage collection of historical versions; (2) It enables persistent timestamping; and (3) It enables "as of" queries. The rest of *Immortal DB* functionality (e.g., volatile timestamping, versioning chain, etc.) can be exercised via conventional tables enabled for snapshot isolation.

4.2 Querying in Immortal DB

To demonstrate *Immortal DB* transaction time functionality, we support queries "as of" some time in the past. We modify the snapshot isolation capability already present. We extend the SQL syntax for the "Begin Transaction" statement by adding the keyword clause `As Of`. The following query asks for information about 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 < 11
Commit Tran
```

5. IMMORTAL DB IN ACTION

We demonstrate *Immortal DB* functionality with a moving objects application. We first define an `Immortal` table.

[1]We have not yet implemented the TSB-tree.

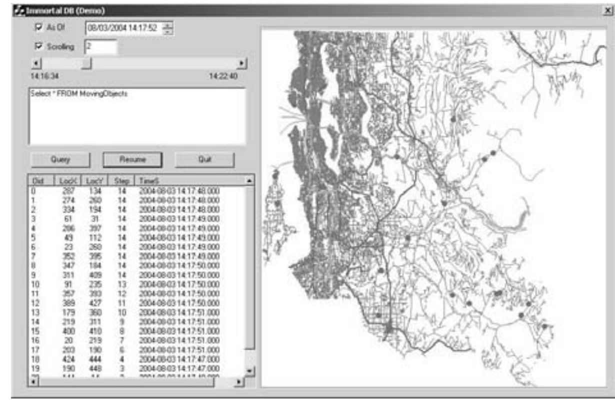

Figure 3: Screen Shot: Scrolling through history.

Then, we use the *Network-based Generator of Moving Objects* [2] to generate a set of moving objects (e.g., vehicles, trucks, cyclists, etc), in our case on the road network of the Seattle, Washington area. Each moving object is shown on a map of the area.

When an object appears on the map, an `Insert` transaction adds a record for it to the MovingObjects table that includes the object ID and location. *Immortal DB* timestamps the inserted record using the lazy timestamping method. When an object moves, an `update` transaction for it posts the new location to the MovingObjects table. *Immortal DB* maintains the complete history of locations for each moving object. Objects have pre-determined sources and destinations and move at variable speeds, submitting `update` transactions at different rates. Once an object reaches its destination, it stops sending updates. Thus, moving objects have different numbers of `update` transactions.

An interesting feature of our demo is that we can scroll over history. When the SQL query (e.g., *select * from MovingObjects where LocX >50 AND LocX<350*). is submitted to *Immortal DB*, we start a transaction that asks for database state "as of" a specified time. Figure 3 gives a screen shot of the demo in the scrolling mode. First, we specify the scrolling resolution (i.e., the amount of time separating "as of" queries as we travel through time via scrolling). Then, we specify the SQL query. Finally, with each movement of the scroll bar, we display the result "as of" the scroll bar time in tabular form as well as on the map. By scrolling over the history, we can traverse the trajectory of a set of moving objects in the road network.

6. REFERENCES

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

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

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

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

[5] D. B. Lomet and B. Salzberg. *Temporal Databases: Theory, Design, and Implementation*, chapter Transaction-Time Databases, 388–417. Benjamin/Cummings, 1993.