Multi-Version Concurrency via Timestamp Range Conflict Management

David Lomet[#], Alan Fekete[*], Rui Wang[#], Peter Ward[*]

[#]*Microsoft Research*

Redmond, WA, USA 98052

{lomet, ruiwang}@microsoft.com

[*]*University of Sydney*

Sydney, Australia

{alan.fekete, pwar3236}@sydney.edu.au

***Abstract*— A database supporting multiple versions of records may use the versions to support queries of the past or to increase concurrency by enabling reads and writes to be concurrent. We introduce a new concurrency control approach that enables all SQL isolation levels including serializability to utilize multiple versions to increase concurrency while also supporting transaction time database functionality. The key insight is to manage a range of possible timestamps for each transaction that captures the impact of conflicts that have occurred. Using these ranges as constraints often permits concurrent access where lock based concurrency control would block. This can also allow blocking instead of some aborts that are common in earlier multi-version concurrency techniques. Also, timestamp ranges can be used to conservatively find deadlocks without graph based cycle detection. Thus, our multi-version support can enhance performance of current time data access via improved concurrency, while supporting transaction time functionality.**

I. INTRODUCTION

A. Our Goals

Some database systems support multiple versions, which can be used to allow more sophisticated concurrency control than traditional strict two-phase locking (S2PL), and/or to support transaction time databases [12] with time travel queries of past states. Commercially, Oracle was the earliest with snapshot isolation[1], and transaction time support in its Flashback and Total Recall features[21][22]. Among research prototypes, we mention the original Postgres [26] and Immortal DB [17][18][19] which use S2PL concurrency control and support transaction time. Others have offered serializable transactions where writers do not conflict with reads that are in pre-declared read-only transactions [4][5][12]. Two recent prototypes have concurrency control that resembles snapshot isolation (allowing reads to run concurrently with writes by accessing older versions) and yet they ensure serializable execution [7][24].

The existing approaches each have disadvantages. Snapshot isolation, while preventing several concurrency anomalies, does not prevent them all[1]; so executions do not satisfy the textbook definition of serializability, and undeclared integrity constraints can be violated (this has been observed in production code[13]). For systems with S2PL for current-time transactions, versioning reduces the performance of current time transactions below that of unversioned S2PL [19], and this is often significantly less than the performance of snapshot isolation, because in S2PL, a read operation is blocked by any concurrent transaction that writes the same data. The proposals of Cahill et al [7] (we call this "CRF") and Revilak et al ("ROO")[24] do not support time-travel queries, and adding these would be problematic, because neither CRF nor ROO ensures that timestamps assigned to versions agree with a serialization order of transactions. Thus a query that looks at versions based on a given timestamp would not necessarily report a state that agreed with any serial history that leads to the observed final state.

We want all the features together, so we can have true serializablilty (as well as weaker isolation levels), concurrent readers and writers leading to better performance than S2PL, and time travel queries. No previous proposal does all this.

B. General Approach

With multi-versioning, each version is tagged with a timestamp associated with the transaction that created the version. Usually this timestamp is assigned to the transaction when it commits. *Our key idea, which is unique to our approach, is to dynamically track the bounds on what a transaction's timestamp can be as it executes.* Whenever a transaction accesses a resource and conflicts with another transaction, we adjust the range within which its timestamp must lie. Concurrent read/write access occurs by reading of a version earlier than the one that an updating transaction is concurrently creating, and adjusting timestamp ranges so that reader's timestamp range is earlier than the writer's.

Creating versions, and finding a version to access based on a timestamp, are done as in other multi-version systems. We employ a more or less conventional lock manager[10][28] to *detect* conflicting accesses, but once it is detected, we handle it differently. We use the usual lock modes to determine conflicts. The nature of the current access, read or write, determines how transaction timestamp ranges are impacted and whether concurrent access is possible. When no range extent can be found for a transaction, it is aborted. Using bounds for timestamps is similar in spirit to the timestamping lock manager of [20] for dealing with a SQL CURRENT TIME request that provides a latest possible transaction time,

but [20] did not permit concurrent read/write access, nor were the ranges modified to enable greater concurrency. Bernstein et al [3] track timestamp bounds, but they are not based on conflicts, nor is serializability guaranteed.

C. Contributions

This paper proposes a new form of conflict manager that we call a timestamping conflict manager or TCM. It tags committed transactions with transaction timestamps that can be made consistent with transaction isolation order, including serializable, while enabling concurrent read/write access. This is unique to our work here. Our primary contributions are that our approach can:

1. Provide concurrent read/write access via multiple versions, whose timestamps are consistent with the transaction isolation level requested, including serializable. To facilitate natural temporal queries, a transaction's timestamp is between its start and its commit clock times.
2. Reduce the incidence of aborts, a downside of many prior multi-version approaches. The usual snapshot isolation techniques abort in write/write conflict cases; CRF and ROO also abort for some read/write conflicts. Our method has the flexibility of being able to block in many cases on WW and RW conflicts that would abort using other multi-version techniques.
3. Detect deadlocks via the timestamp range mechanism so that wait-for graph cycle detection is not needed. Timestamp ranges do "everything". A transaction's timestamp range reflects all conflicts encountered thus far.

II. TIMESTAMPS AND LOCKING

A. Timestamps for Concurrency Control

We assume that each version is stamped by the transaction that created the version. This "stamp" can be simply the transaction identifier for the updater, or it can be a timestamp assigned at the time an update transaction "commits". If an identifier, the "stamp" needs to be convertible into a "timestamp" ordered consistent with the isolation order of the transactions. Version timestamps permit us to choose the appropriate version to be read by a transaction.

Timestamps have been used before in concurrency control, notably with timestamp order based methods [2], where the timestamp for a transaction is the transaction start time, and conflicts not ordered in agreement with those timestamps are aborted. However, these methods have not been widely adopted because of fears of an excessive number of aborts. Many techniques that assign timestamps to temporal versions [17][25][26] choose timestamps for transactions at commit time, using a conventional lock manager to handle conflicts and guarantee isolation via two phase locking, and hence do not provide concurrent reads and writes. Version IDs are used also for snapshot isolation with an optimistic approach [22]. This is also used in [26], where timestamps are negotiated during certification. However, optimistic methods preclude blocking instead of abort, as read versions have already been acted on by the time of certification.

B. Conflict Detection and Timestamps

CRF showed how the advantages of having multiple versions can, when coupled with a more or less conventional lock manager (insofar as conflict detection is concerned), provide serializable transactions, while frequently enabling read-write conflicts to proceed concurrently by having the reader read an earlier version of the data.

We want the same high concurrency using multiple versions, but with an additional requirement. We want transaction timestamps to use with versions for transaction time functionality. This requires that timestamps be ordered consistently with the isolation order of transactions, including serializable. The CRF approach does not enable this choice of timestamps. Timestamping and concurrency control in Immortal DB [17], which provided consistent timestamping, did not support read/write concurrency. This prompted us to take another look at the notion of a timestamping conflict manager (TCM) of a more pervasive form.

The conventional lock manager blocks a new access to a resource when the new access conflicts with an existing access from another transaction. This conventional lock manager typically is used to support access to current data, even when transaction time versions are supported. But our intent is to make it version aware to improve concurrency.

Our TCM is similar in many respects to a conventional lock manager, and can exploit similar data structures because conflict detection is the same. The non-conflict case, which is the most common, is unchanged from a conventional lock manager. This is the fast path in and out of lock managers. Only the conflict cases are handled differently, where we need to inspect timestamp ranges. *The reward for this "inspection" is that many read/write conflicts can proceed concurrently.*

The basic idea is for the TCM to maintain a range of timestamps for a transaction. This range is adjusted at each conflicting access of the transaction with some other transaction so that the timestamp range is consistent with the conflicts that the transaction has encountered. Because there is a range of timestamps, we can more flexibly re-order transactions compared with timestamp order methods [2]. This re-ordering reduces the abort rate while preserving our ability to choose an appropriate timestamp.

A lock manager maintains a conceptual lock matrix for transactions and resources. Whenever a transaction locks a resource, an entry is made for the appropriate lock matrix entry, linking the resource and the transaction via the lock. The matrix is usually accessed via hashing resource IDs, dividing them into "hash buckets". At each resource, active accesses to a resource are enqueued. Each transaction is likewise accessible via a hash table that refers to a set of transactions, each with a transaction control block (TCB). A TCB points to the set of resources accessed by the transaction so that when the transaction is no longer "alive", it is straightforward to remove its TCM entries.

Like CRF, the TCM maintains information not just on active transactions but also on recently committed transactions (that are still "alive" in that they can affect the timestamps of active transactions). An active transaction might commit with

a timestamp earlier than an already committed transaction. We need to detect such cases, and adjust timestamp ranges appropriately as these extended conflicts are detected.

C. Blocking Instead of Abort

When two snapshot isolation transactions conflict on writes, one is aborted. When one has already committed, the currently active transaction is aborted. When both transactions are still uncommitted, we can choose either as a victim. Many implementations abort as soon as a WW conflict is detected (eager abort). Our TCM can sometimes have the new writer wait for a resource as is done in conventional locking. Indeed, it is also possible to deal with some of the "abort" cases for RW conflicts by having the requestor wait. Always blocking is possible, which leads to conventional locking. With TCM, we block only when the alternative is abort. We opt for increased concurrency when timestamp ranges permit it.

Blocking can lead to circular waiting or deadlock. Deadlocks are low frequency but they need to be dealt with lest some resources be tied up indefinitely and some transactions be prevented from completing. We identify deadlocks using the same timestamp range validation as used elsewhere, not by tracing conflict edges in a graph looking for circular waiting.

The key to using timestamps in dealing with conflicts is to ensure that the timestamp range associated with each transaction fully reflects all the conflicts it has seen so far. Then when circular waiting arises, this will be identified by the new conflict not being resolvable via adjustments to the timestamp range. We can be conservative here, aborting in some cases where it might have been avoided. The low frequency of deadlock and even of multiple enqueued blocked transactions means deadlock should not be a costly issue.

D. Transaction Attributes

We associate each transaction with a transaction control block (TCB) that is linked in the TCM to resources being accessed by the transaction. Each transaction has at least the following attributes:

1. **TID**: Transaction identifier X
2. **X.early**: earliest time at which X can commit. This is initially set to the time the transaction begins.
3. **X.end**: (true) X.late is not null; (false) X.late is null. X.end is initially false.
4. **X.late**: latest time before which X must commit.
5. **X.committed:** (true) X.timestamp is not null; (false) X.timestamp is null.
6. **X.timestamp**: time at which X is committed (Note: One can use X.early or X.late to store X.timestamp instead of a separate field. It is kept separate here to make the exposition simpler.)
7. **Isolation level**: Conflicts and response to them may depend on transaction isolation level.

In a transaction time database, X.timestamp becomes the timestamp of all versions updated by X when X commits. Another transaction Y, reading at Y.early can see a transaction consistent view of the database by reading a version with the largest timestamp less than Y.early.

It is not necessary to physically store timestamps in versions. The timestamp in the TCB is sufficient so long as there is a mapping from a TID in the version to the TCB. However, for transaction time functionality, driving the timestamps into the versions as done in Immortal DB is desirable to avoid runtime translation overhead.

E. Conflict Principles

How timestamp ranges are managed is dictated by the following principles. In what follows, we use X and Y to denote transactions making requests, and R to denote a resource being accessed. The principles are organized by the role they play.

Constraining transaction timestamp in a timestamp range:

1. *When a transaction X commits, it chooses a value in [X.early, X.late) as X.timestamp, and sets (X.early, X.late) ←X.timestamp.* Thus, for a committed transaction, the timestamp range collapses to X.timestamp.

Accessing the correct version:

2. *When access by X is granted for a request for R, the version of R accessed is the one with the largest timestamp less than X.early (after access is granted).* This ensures that X's timestamp is appropriate for the database state that it reads, at the time of access.
3. *A request by Y to write R is never moved ahead of a prior request for R from any X that has accessed earlier versions of R.* This ensures that the version of R accessed by X at the time of its request is never changed by a later writer, even should X's start time be pushed back by a write that it may do subsequently.
4. *A transaction X's timestamp range must never include any versions of resource it is accessing other than those it created.* This guarantees that the version it accesses from other transactions will not change regardless of where its timestamp is chosen in its range.

Ordering conflicting accesses and timestamps:

5. *New conflicts never grow the range [X.early, X.late) of a transaction.* Thus, prior conflict imposed constraints will continue to be honored, in particular the order of transactions and their timestamps, while a new conflict may impose a more stringent constraint.
6. *When a request by X is granted to read a version of R earlier than the version of a writer Y for R, X.late will not be later than Y.early.* This ensures that the reading transaction X comes before the writing transaction Y, which is required for the version that it reads to be the correct version.
7. *When Y blocks behind X waiting for R, X's timestamp range must be earlier than and disjoint from Y's timestamp range.* This ensures that deadlocks are avoided by the timestamp range technique of section II.C.

And when all else fails:

8. *Where the above principles cannot be applied, abort of one of the conflicting transactions.*

F. Correctness

We argue for the correctness of our TCM approach in the common "page" model [28], where a transaction consists of read and write operations on named data items (that is, we simplify the model to avoid predicate evaluation).

An essential lock management invariant is that, whenever T has accessed item X, then T holds an appropriate lock on X. For a system like TCM where version order is the same as the creation order of the versions, the multi-version serialization graph (MVSG) is generated by edges from T_1 to T_2 that arise in one of four cases: (i) T_1 writes a version X_1 and then T_2 writes another version X_2 of the same item X, (ii) T_1 writes X_1 and then T_2 reads that version X_1, (iii) T_1 reads version X_1 and then T_2 writes X_2, (iv) T_2 writes X_2, and then T_1 reads a version X_1 that precedes X_2 in the version order. In each of these cases, at the point in the execution where the second access occurs, the timestamp ranges for the transactions will be set so that the timestamp range for T_1 is entirely before the range for T_2. All subsequent steps of TCM preserve this relationship, until eventually T_1 and T_2 are allocated point timestamps such that

$$timestamp(T_1) < timestamp(T_2).$$

That is, every edge in the serialization graph is compatible with timestamp order, and so there are no cycles in the MVSG.

III. LOGIC FOR CONFLICTS

A. Basic Capabilities

We determine, for conflicting accesses, if one of the transactions can precede the other, and then provide appropriate access, adjusting the timestamp ranges accordingly. For example, a writer cannot be moved ahead of a reader (principle 3). If the reader in a read/write (R/W) conflict reads a version earlier than the writer's version, the reader must have an earlier timestamp than the writer. Further, once we have ordered the transactions in a conflict, we need to make sure that the timestamp ranges of the conflicting transactions are consistent with this ordering. That is, the transaction that we have placed first has a timestamp range that strictly precedes the timestamp range of the transaction that is ordered later.

To test if we can order the transactions as required, we invoke the **can_be_before(A,B)** function on transactions A and B that returns true if the timestamp ranges for A and B permit A to precede B. If we can order A and B according to **can_be_before**, then we make the timestamp ranges of A and B consistent with this ordering by invoking **put_before(A,B)**. This procedure ensures that the timestamp range for A is disjoint from the timestamp of B and precedes it. These functions are described in Figure 1.

The procedure **put_before(A,B)** reflects our tactic, in R/W conflicts, of giving readers as much of the timestamp range as is possible without aborting the writer. And for W/W conflicts, we give as much of the timestamp range as is possible to the earlier writer currently accessing the resource. Other tactics are possible for adjusting timestamp ranges during conflicts.

```
can_be_before(A, B) returns(boolean)
if B.end & (B.late ≤ A.early+1) then
   return(false);
else /* when B doesn't have late bound, A can always
        precede B */
   return(true);
end

put_before(A, B)
/* assumes can_be_before(A, B) has previously returned true, i.e.
¬(B.late ≤ A.early+1) or B.late is not yet defined */
/* Since A must precede B, A.end */
if ~A.end then
   {A.end = true; /* in all cases, A.end becomes true */
    A.late = current_time;
   }
if B is reader then /* A must be a writer */
   { /* we give most of range to B */
    B.early = max{B.early, A.early+1};
    A.late = min{A.late, B.early};
   }
else /* Either A is a reader and gets most of the time range or
     A is a writer that currently holds a resource */
   { if B.end then
        {A.late = min{A.late, B.late-1};
     B.early = max{B.early, A.late};
     }
end
```

Fig. 1. Testing and ordering timestamp ranges.

B. Handling Read and Write Requests

We now show how read and write accesses are handled. This is shown in Figure 2. For both forms of access, conflicts are, in fact, uncommon. Well-tuned database systems have relatively low conflict rates. And the no-conflict path through a lock manager is very efficient. Our conflict manager shares that efficient no-conflict path, as shown in Figure 2. It is when conflicts occur that the TCM has a very large advantage over conventional lock managers.

While the logic within the TCM seems more complicated than the corresponding lock manager (LM) logic, whenever the TCM avoids blocking, it avoids the substantial overhead (thousands of instruction cycles) of a thread switch. This thread switching overhead does not appear in the logic of the LM, but it is operating system code path to implement the LM "block". Further, because the TCM blocks much less frequently, there will typically be fewer active transactions in the system at any given time, with smaller average latency. Hence the conflict frequency of dealing with conflicts will also decrease, which is part of the "virtuous cycle" caused by the reduced blocking (and hence increased concurrency) of the TCM.

C. Aborts and Blocking

Figure 2 superimposes two concurrency control schemes. The usual multi-version concurrency control (MV-CC) approach is to abort in two conflict cases:

1. R/W: when it is impossible to have the reader precede the writer and gain concurrent access;
2. W/W: always

There is a fundamental reason for this. Almost all MV-CC approaches are examples of optimistic concurrency control (CRF is an exception). Most optimistic approaches check for conflicts at the end of the transaction. This is after potential conflicts where the transaction proceeded despite these conflicts. Hence the transaction has already chosen and used a version. If that version is incorrect, abort is about the only possible outcome.

In Figure 2, the abort approach is revealed by removing the "WITH BLOCKING PERMITTED" boxes from the code. Without that code, abort occurs exactly as indicated in the list above. However, using our TCM, we identify conflicts early (pessimistically), and can react differently. Since aborts clearly waste the prior work of transactions, it is almost always better if a transaction is blocked instead.

Blocking is enabled by including the "WITH BLOCKING PERMITTED" boxes. The TCM has not chosen the version of the resource that the requestor will read. Hence, if a requestor's timestamp range does not permit it to access a version earlier than an uncommitted one, the requestor (reader or writer) blocks waiting for the earlier writer to commit. It then accesses the newly committed version.

With blocking, abort frequency is substantially reduced. A read request virtually never leads to an abort. Only when reader and writer have identical degenerate timestamp ranges (exactly one time point in the range) is it impossible for the reader to be ahead or behind the writer.

Abort is a bit more common with write requests, though still much reduced when blocking is permitted. A writer can never be moved ahead of an earlier reader because that violates our principle 4 which protects the earlier holder from later write requests changing the version it is reading. Permitting blocking does not avoid an abort when the reader (holder) cannot be earlier than the writer.

When the conflict is W/W, most MVCC techniques require one of the writers to be aborted. However, as with conventional locking, when blocking is possible (i.e., timestamp ranges permit the holder to precede the requestor), the write request can block waiting for the earlier writer transaction to commit and release the lock on the resource.

IV. LOCKING ISSUES

In the preceding, we have discussed conflicts in a general setting, referring to readers and writers and without relating them to specific lock modes. Here we want to discuss lock modes more generally and discuss handling deadlocks.

A. Lock Modes

How our TCM identifies conflicts between locks is the same as in a conventional lock manager [10] [28]. The lock mode conflict matrix is unchanged. Further, as noted previously, the non-conflict case is unchanged in going from LM to TCM.

1) Readers and Writers: To deal with conflicts, we need to associate lock modes with read and/or write access so that we can use the pseudo-code in Figures 1 and 2 to determine whether concurrent access, blocking, or abort results.

```
Read Request
If no-conflict then return /* 95% case; immediate access */
else /* holder must be a writer */
 {  /* put reader ahead of writer if possible */
  if can_be_before(requestor, holder) then
    { put_before(requestor, holder);
     return; /* concurrent access */ }

   /* WITH BLOCKING PERMITTED */
   if can_be_before(holder, requestor) then
      put_before(holder, requestor);
      BLOCK; /* block until lock released */
      return; }

abort holder}  /* we do not abort readers */
 /* Abort is rare: both ranges must be same degenerate range. */
-----------------------------------------------------------------------------
Write Request
If no-conflict then return /* 95% case, immediate access */
else  /* test whether holder is reader or writer */
  {if can_be_before(holder, requestor) then
        {if holder is reader then /* put reader ahead of writer */
        {put_before(holder, requestor);
         return; /* concurrent access */}

      /* WITH BLOCKING PERMITTED */
       else  /* holder is a writer */
          {put_before(holder, requestor);
          BLOCK; /* block until lock released */
          return; }
abort requestor; }
/*Abort occurs when new writer cannot be after holder*/
```

Fig. 2. Handling Read and Write Requests

The rule we adopt is simple. If a lock mode conflicts with S, it is treated as a write during conflicting accesses. A request lock mode not conflicting with S is treated as a read when it conflicts with other operations (from writers). Further, while a write lock mode permits change to a data object, it does not reveal whether the object will be read prior to being changed. Thus, we conservatively assume that all writes are also reads. This is the high frequency case, and is the reason we can never move a later write request ahead of any earlier request, read or write.

2) Multi-granularity Locking: Using the above technique with the traditional multi-granularity hierarchy, S and IS are readers, and IX, SIX, and X are writers. How to treat these lock modes in our analysis of conflicts flows directly from this classification. Note that this is conservative. Someone holding an IX lock on a table will conflict with a transaction with an S lock on the table. But the IX locker might never actually write. This is similar to the conventional LM case where one of the transactions involved will block (perhaps unnecessarily).

Two IX locks do not conflict, even though we treat them both as writers. This is not a problem. The impact of locking on timestamp ranges only occurs in conflict cases. There is no conflict at this point. If a conflict arises, it will be when both transactions attempt to X lock the same lower level resource, e.g. a record.

3) Update Locks: Update (U) mode locks are frequently taken on a resource when it is necessary to read the resource to decide whether it should be modified or not. Two U mode locks conflict to prevent a potential deadlock that would arise if two transactions had S mode locks on a resource, and both wanted to write (modify) it. Instead, a transaction that might write must take a U lock on the resource before it can upgrade the lock to an X. S lockers cannot upgrade to a U. They must take the U mode lock on a previously unlocked resource. U mode locks do not conflict with S mode locks, a big plus for reducing conflicts.

Our policy above classifies a U locker as a reader as U does not conflict with S. The upgrade to X required for writing produces the write conflict when we need it. However, this does not tell us how to handle UU conflicts.

When readers conflict, as in the UU case (but not SS or US), we choose to block the requestor, forcing it to have a timestamp range later than the current U lock holder. If the holder upgrades to X, the read/write conflict cannot result in concurrent access as the U requestor already has a timestamp range strictly after the range of the holder. The UU conflict simply is transformed into a UX conflict and the U requestor continues to wait. However, if the holder downgrades to S, the requestor can then read and decide whether it needs to upgrade. If so, its write is concurrent, producing a version after the earlier reader. If not, it downgrades to S and continues executing.

When a U request produces a UX conflict, the requestor may read an earlier version than the X holder. Most of the time, this will be fine as most U mode locks are not converted, and concurrency is increased. However, if the U requestor upgrades to X, it will have to abort as we cannot move a writer ahead of a prior access.

4) Key Range Locks: If a transaction T_1 reads a range of keys that include k_1 and k_4 of a table in which these keys are adjacent, when using key range locking it will obtain a key range lock on k_4 which covers the range between k_1 and k_4.

If transaction T_2 inserts a key k_3, T_2 conflicts with T_1 and the insert is permitted as a concurrent RW conflict, with the insert occurring temporally after the range read. However T_1's range lock on k_4 now appears to cover only (k_3, k_4]. So if another transaction T_3 inserts k_2 in the gap between k1 and k3, it now does not appear to conflict with T_1 as the next key seen is k_3, which is not locked with T_1's key range lock. So if the insertion were allowed to happen, and if T_1 then rereads the table, it might now see k_2, which is a phantom.

We found a way, called lock propagation, of dealing with this potential difficulty. Lock propagation acts on locks at the time when k_3 is inserted and splits the (k_1, k_4] range. When a transaction inserts a record, it copies all key range locks from the following key. In our example, the key range lock on k_4 is propagated to the newly inserted k_3. This propagation will also include the timestamp range information of the respective transactions. Then, when T_3 inserts k_2, T_1 will have a key range lock on k_3, so T_3 will be seen to conflict with T_1.

B. Deadlock Detection

In Figure 2, the abort cases shown "WITH BLOCKING PERMITTED" strictly include the cases where deadlocks arise. This timestamp based deadlock handling technique is enabled by our adjusting timestamp ranges to agree with conflict ordering at the time that a transaction blocks. Every blocked transaction has a timestamp range that is disjoint from and later than transactions earlier in the queue.

Whenever there is circular waiting, a transaction A that completes that cycle (resulting in a deadlock) has blocked other transactions. All these transactions will have timestamp ranges disjoint with and later than A's timestamp range. A is now being blocked by one of these transactions or by another transaction blocked by these. The requirement, when A is blocked, is that it be later than the transaction blocking it. But this is impossible. Hence the deadlock is "detected" and one of the blocking transactions must be aborted. The inability to reconcile timestamp ranges is a necessary but not a sufficient condition for deadlock and hence is conservative as some aborts occur without a deadlock.

V. SYSTEM OPERATION

We want to show how our conflict analysis fits into a wider system. We discuss three aspects.

1 What does the TCM do when conflicts are detected? Detecting conflicts and the no-conflict case handling are unchanged.
2 How are significant milestones in a transaction's execution impacted? We present our versions of modified operations along the lines of CRF.
3 When do we garbage collect the "locks" of committed transactions in the TCM? Like CRF, we cannot use transaction commit for this.

A. Conflict Handling

A requesting transaction may conflict with more than one transaction that has "locked" the resource. For example, a writer conflicts with every reader of a resource. Further, when transactions block waiting for a resource, the requesting transaction can conflict with multiple blocked transactions enqueued on the resource. We describe these cases next.

1) Resource with at least one reader and no writers: A writer, if it is to proceed concurrently with existing readers of a resource, needs to come after all readers. Thus the writer start time must be later than the end times of all readers. Hence, A.early for writing transaction A is the latest time required among the readers.

While a writer might be able to execute concurrently with some readers, for the writer to proceed, it must be able to execute concurrently with all the readers. If not, this is because the writer timestamp range forces it to precede at least one of the readers. In this case, the writer is aborted as the "problem" reader has already read a version that is different from the one that would be created by the writer.

2) Resource with at least one writer: When a transaction has written a resource, we permit readers to execute

concurrently by reading an earlier version, when that is possible. If a new reader does not have a timestamp range that permits it to read the earlier version, that reader is blocked. Both in the concurrent case and the blocked case, the analysis we have done earlier in sections II and III can be used to determine each reader's timestamp range. A read requestor almost never needs to abort (see III.C); it proceeds either concurrently with the writer by reading an earlier version, or by waiting for the writer to commit and then reading this new version.

A reader will be moved forward in the queue of writers waiting on the "locked" resource as far as possible, until it reaches a writer that must precede it. If there is no such writer, the read can execute concurrently with the active writer. A new reader's timestamp range is impacted solely by the writers of the resource, not by the concurrent (or blocked and waiting) readers and a read requestor is never delayed by blocked readers, except for update lock mode conflicts (see section IV.A.)

If a writer accesses a resource with an existing writer, it must either abort or block waiting for the resource. To block, the new writer must have a timestamp range that permits it to follow the writer and all the concurrent readers and/or blocked readers and writers as well. Otherwise it is aborted.

B. Transaction Stages

There are a number of milestones in the life cycle of a transaction that we describe in this section.

BeginTransaction(A): Execute existing begin code, then:

- *A.early ← current time; A.end ← false; A.commit ←false*; both *A.late* and *A.timestamp* are indicated as being undefined because *A.end* and *A.commit* are "false".

EndTransaction(A, commit/abort): For commit, execute existing commit code, then:

- *A.commit ← true; A.timestamp ← A.early; A.late ← A.early*. We choose the earliest time for commit. There are good reasons for this choice (see V.C). Other choices are possible. We do not remove A's locks from the TCM.

For abort, execute existing abort code, then:

- Remove transaction *A* locks from the TCM.

At transaction end, we unblock transactions waiting on *A*'s locks, permitting them to resume execution.

C. TCM Garbage Collection

We need to wait until *A*'s commit timestamp can no longer impact the timestamp ranges of active write transactions before removing *A* from the TCM. By choosing the earliest timestamp in the acceptable range, we hasten the time we can remove *A*. (Read only transactions are discussed in section VIII below.) *A*'s having read a version of a datum *D* forces writers of *D* to be later than *A*. Thus, once no *B* exists with $B.early \leqslant A.timestamp$, A can be removed.

We can be lazier than necessary about removing *A*'s "locks", but we must not be more eager. Thus we need to track *B.early* for active writers *B*, but can do it in a conservative way. One approach (among many) is to count the number of active transactions *B* with *B.early* in a small interval Δt. When the count in an interval Δt is *0*, we find the earliest non-zero interval ΔT and delete committed transactions in the TCM with timestamps earlier than ΔT.

VI. IMPLEMENTING A TCM

We have successfully produced the core of a timestamp range conflict manager using InnoDB's multi-version record support. This section documents changes we made to InnoDB.

A. Transaction Timestamp Ranges

Our TCM keeps track of the earliest and latest possible times a transaction can commit. This is the timestamp range of the transaction. To keep track of this timestamp range in InnoDB, we added extra attributes to the transaction structure.

When a transaction starts, it is assigned the current time as the earliest time it can commit and the end time is unbounded. As this transaction is involved in conflicts with other transactions, this range shrinks. When the transaction commits, we need to shrink the range down to a single point in time.

In our implementation, we use a 64-bit unsigned integer counter to represent time. To get current time, the counter value is retrieved and incremented. As mentioned in section VII.B this method has limitations. A technique that provided a sparser set of timestamps would provide more flexible timestamp range adjustment possibilities when handling conflicts, potentially reducing the number of aborts.

In interval notation, a transaction *trx* can commit at any time in the interval *[trx.early, trx.late)*. When the transaction is committed, we set *trx.late=trx.early+1*. InnoDB's existing transaction state tracks if the transaction is committed or not.

The timestamp range attributes we added to InnoDB are given in Figure 3.

```
struct trx_struct { ...
  dulint early; /* earliest time a tran can commit */
  dulint late; /*latest time before which a tran must commit */
  unsigned end:1; /* indicates whether end is set */
    ... }
```

Fig. 3. Transaction Attributes for Timestamp Ranges

B. Adjusting Timestamp Ranges

The code used for RW conflicts is shown in Procedure 1 in Appendix A. There are two cases where we have chosen to abort as there is no way to adjust the timestamps to satisfy the situation. In both cases, we abort the requestor, who is the writer.

The code used for WR conflicts is shown in Procedure 2 in Appendix A. This is interesting due to the need to abort the lock holder, which is achieved by adapting the code for aborting transactions when deadlocks are found. Additionally, as we wished to be able to tell the difference between deadlocks and TCM aborts, we have added an error message to MySQL which displays when this situation occurs. Finally, on lines 11 and 27, it is not necessary to abort as blocking the requestor suffices. However, abort is both simple and correct.

The code for WW conflicts is given in Procedure 3 of Appendix A. Similarly to RW conflicts, we have chosen to abort the requester on lines 21 and 33.

If the timestamp range is empty, it is impossible to satisfy the constraint, and one of the transactions is aborted. In our implementation, for simplicity we abort the transaction which made the request.

If a transaction's early bound on the timestamp is moved backwards, this means that transaction which conflicted with it is blocking the transaction from continuing. Due to reasons described in subsection E.1 below, our implementation chooses to abort transactions which could be blocked.

C. Record locking

In order to integrate these rules into the lock manager, we first needed to understand how the lock manager handled lock requests. When a read or write operation occurs on a record, the *lock_rec_lock* function is called, which attempts to lock the given record in the requested mode. As shown in Figure 4, this function calls out to two other functions, first to *lock_rec_lock_fast*, which handles the more common case of no other locks on the record. If there are already locks on the record, *lock_rec_lock_slow* is called, which itself checks to see if the new lock request can be satisfied (or if the request needs to be blocked or aborted).

The InnoDB lock manager was re-implemented to include the pseudo-code shown in Procedures 1, 2 and 3. The changes were localized to *lock_rec_lock_slow*, which handles conflicting accesses. The new parts of the lock manager are shown in "gray" blocks. The white blocks are not modified.

D. Reading correct record versions

A transaction needs to read the correct version of the record. For instance, if it has previously read one version of the record, it should always see that version unless it has modified the version itself. Under S2PL, if T_1 reads a record, and then T_2 writes the same record, T_2 will block until T_1 is committed (or rolled back). However, with TCM, T_2 is allowed to write the new record, and T_1 continues to read the old version.

InnoDB already can read previous versions of records. It takes the most recent version, and uses the rollback log to undo the change made to it, until it finds a version which the transaction is allowed to see. The only part needing changes is using the timestamp range of the transaction to specify what versions the record is allowed to see. Given a transaction T_1 and the transaction T_2 which created some version of a record, the rules for conflict resolution will adjust the timestamp ranges such that T_1's timestamp range will never overlap T_2's timestamp range. T_1 reads the version of a record written by a committed transaction with the largest timestamp earlier than $T_1.early$. The only exception to this is when the version was also generated by T_1.

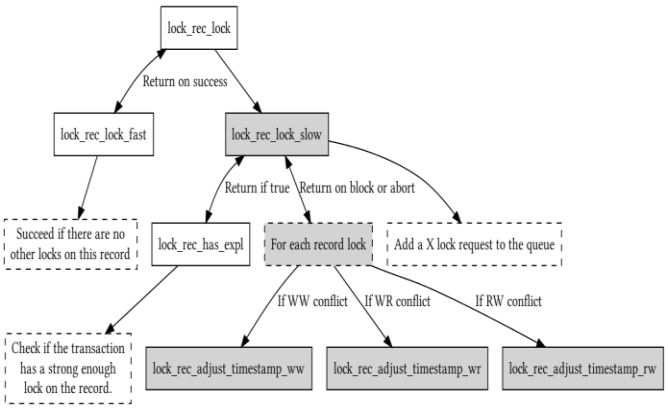

Fig. 4. Lock Manager Logic. Shaded boxes are timestamp range related changes made to InnoDB.

E. Other Modifications

1) Blocking and aborting transactions: When resolving conflicts, concurrent access may not be possible. In such cases, either at least one of the transactions must be aborted, or one of the transactions will need to block until a conflicting lock is released.

Under S2PL, a transaction is resumed from the blocking state when all the conflicting locks on which it is waiting have been released. However, with TCM, a record may simultaneously have conflicting locks, so the rules for when to resume a transaction need to be changed.

We have not implemented the rules for resuming blocked transactions, instead opting to abort any transactions which are blocked. When a transaction is aborted, we return a DB_TCM_ABORT error added to InnoDB as a new error code. This allows us to differentiate between errors from conflict resolution and other lock problems.

2) Disabling Implicit Locking: InnoDB has the concept of implicit and explicit locks. Implicit locks belong to the creator of a record version. An implicit lock exists simply by creation of a record, and reduces the number of lock objects (if a transaction inserts 10000 records, there won't be any lock objects created). When another transaction accesses a version with an implicit lock (creating transaction is not committed), a conflict occurs and the implicit lock is converted to an explicit lock, which is then used in the conflict resolution process. For simplicity and to ensure correctness, we have disabled implicit locking (requiring all locks to be explicit) in our implementation.

3) Range Locks: To handle key range locks, which InnoDB refers to as "gap" locks, we propagated the locks to adjacent keys as described in Section IV.A. Lock propagation uses additional lock block space and extra time copying the locks. However neither of these issues is significant. During implementation, lock propagation only required a single line of code to be added!

VII. PERFORMANCE

Here we describe an initial performance evaluation of a TCM implementation described in Section VI, based on InnoDB, which maintains versions (but they are only used to support weak isolation levels; InnoDB's serializable transactions use S2PL). We exploit InnoDB versioning and add timestamp-bound management to its lock manager, so our TCM provides serializable transactions, with timestamps consistent with the serial schedule. Our evaluation considers two factors. First we examine the overhead of performing timestamp adjustments during record locking. Second, we describe throughput and abort rate results. For both, we describe the testing procedure and results. Since time travel queries work identically for our design, as in a traditional S2PL system with versions, we only measure examples of current-time queries.

All testing was done using MySQL 5.1.48, run on Debian (Sid) Linux, kernel version 2.6.35, with a Intel Core2 Duo P8700 CPU, 3GB RAM, glibc version 2.10 and gcc 4.4.3. All queries to the database were run with autocommit disabled and at serializable isolation (using either the InnoDB S2PL lock manager, or our novel TCM). InnoDB does not support SI (though it can be added[7]) so we haven't compared to this.

A. Timings

We first checked the timestamp range overhead during record locking. In InnoDB, record locking is protected by the kernel mutex (a global lock on InnoDB's data structures). If the added overhead is significant, other parts of the system will wait to acquire the mutex, hence losing concurrency and decreasing performance.

The C function *clock_gettime* is the appropriate method for timing record locking (*lock_rec_lock*), with microsecond-resolution resolution and measuring only the computation time of the current thread (using the *Clock_Thread_Cputime_Id* timer). This is important. While record locking is protected by a kernel mutex, other threads may be concurrently active.

The test load on the database for queries we ran using *mysqltest* is shown in Figure 5, and was repeated 3000 times. The S2PL lock manager had a mean execution time of 2145.44 microseconds, and a standard deviation of 1231.97 microseconds. The TCM had a mean of 2252.87 microseconds, and a standard deviation of 1232.55. Figure 6 shows the time distribution for the record locking function. So, while our TCM's record locking is slightly slower, on average than InnoDB's original S2PL lock manager's, adjusting timestamp ranges adds very little to the cost of record locking.

```
T1> SELECT value FROM t1 WHERE id = 3;
T2> SELECT * FROM t1;
T2> UPDATE t1 SET value = 3 WHERE id = 1;
T2> SELECT value FROM t1 WHERE id = 1;
T1> SELECT value FROM t1 WHERE id = 3;
T1> UPDATE t1 SET value = 9 WHERE id = 3;
T2> COMMIT;
T1> COMMIT;
```

Fig. 5. Procedure to generate timing data.

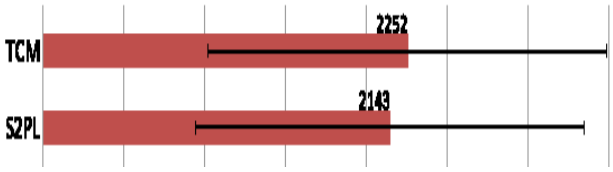

Fig. 6. Timing comparison: TCM vs. S2PL in μsec. Red bars show mean, error bars show standard deviation.

```
CREATE PROCEDURE read1(IN x INTEGER)
BEGIN
        SELECT SUM(value) FROM t1 WHERE id IN (
          SELECT value   FROM t1
          WHERE id = x);
END
-----------------------------------------------------------------------------
CREATE PROCEDURE write1(IN x INTEGER)
BEGIN
        UPDATE t1
          SET value = value - 10
          WHERE id = x;
END
```

Fig. 7. Reading and Writing Benchmark Procedures

B. Benchmarking

We built a benchmarking system where each experiment, cycles through a sequence of states, the usual order being: stopped, warm-up, measurement, stopped. Using it, we measured transactions/second (throughput), and percentage of transactions aborted (abort rate) for a simple benchmark

Each experiment connects to a database, runs setup queries, and then starts a configurable number of clients on separate threads. Each client is given a distribution of stored procedures to run, and chooses procedures randomly with that distribution. Transactions are run continuously, and when the experiment enters the measurement state, the client starts tracking transactions executed and aborted. When the experiment enters the stopped state, clients stop and the experiment reports results.

For our initial benchmarking, we created a key/value table (both integers) and populated it with data (100 rows with keys picked from a uniform distribution between 0 and 200). We ran 20 clients for a warm-up period of 30 seconds, and then measured for one minute, running the procedures in Figure 7 with equal probability. The input parameter is a random number between 0 and 200, generated by the clients.

During our benchmark test, the S2PL system processed 3305 transactions/sec while the TCM system executed 3656 transactions/sec, a difference of about 10%. Figure 8 illustrates this comparative throughput. Figure 9 shows that the TCM system had a 0.428% abort rate while the S2PL system had a 1.018 % abort rate, over twice as high.

Our experiments indicate that our TCM is capable of delivering results as good as, if not better than a S2PL lock manager. Further, our TCM has room for improvement. Transactions that could block are always aborted (see Section VI) and timestamp generation could also be improved. These should further increase throughput and decrease the abort rate.

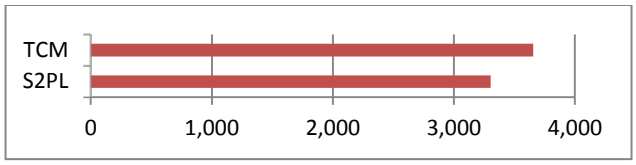

Fig. 8. Transactions/sec for S2PL and TCM.

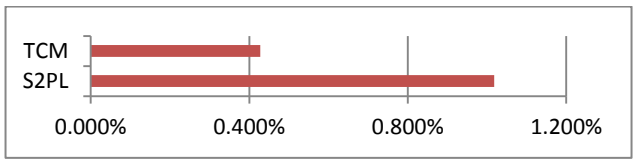

Fig. 9. Abort rates for S2PL and TCM.

VIII. DISCUSSION

There are other issues worth discussing: alternative ways to handle conflicts, certain isolation levels, and how to extend this to other settings. We briefly discuss these here.

A. Other Conflict Policies

Our illustrated conflict policy always favors readers. Other policies are worth exploring, with a final choice based on an analysis of system throughput, abort rates, and perhaps considerations of fairness. Some alternative timestamp range policies are: 1) make the impact on reader and writer equal; 2) favor the writer at the expense of the reader; 3) favor the current holder at the expense of the requestor; 4) favor the transaction with the earliest start time at the expense of a later transaction. As with timestamp ranges, there are alternative abort victim policies. We explored choosing the writer as victim. Some other choices are: 1) the reader; 2) the requestor; 3) the holder; 4) the later transaction, based on start time; 5) the transaction with the fewest or most conflicts.

B. Snapshot Isolation

Snapshot isolation (SI) appears to work best when all transactions are running at SI. An SI transaction read time is then its start time and its updates are stamped with its commit time. An SI transaction need not visit a lock manager for reads. It certifies its writes at commit time, aborting if another transaction's writes conflict with its writes.

With a TCM, a transaction can have a timestamp that is earlier than its commit time and hence earlier than a newly started SI transaction. We need to prevent those transactions from writing (creating) versions of objects read by the SI transaction that would change the set of versions it reads. To enforce this requires some extra effort. One simple approach (others are possible) is to set the start time of the SI transaction to before the earliest early time of any active transactions. Then one can access the versions exactly as SI is currently implemented, albeit with an earlier start time. The negative here is not performance, but rather a perhaps subtle change in the state being read.

We have to separate read timestamp from write timestamp. We can do that by noting that a transaction is SI in the transaction control block and separating read time from timestamp range. An SI transaction's write timestamp range is handled like the timestamp ranges of other isolation levels. However, unlike other isolation levels, SI's definition requires abort when there is a write/write conflict. Our conclusion is that SI support, while possible in the context of a TCM, loses some of its appeal. Serializability achieves so much of the concurrency for which SI is now used that SI isolation level has a much smaller advantage.

C. Read Committed

A read committed transaction RC sets only short term read (S) locks. However, with multi-version support, the read committed definition is not crisp (see [28]). There are two possible versions RC might read: (1) RC might immediately read the latest already committed version, or (2) RC might wait until an X lock holder commits, and then read its newer version. In any event, we want the timestamp for RC to be later than any of the versions that it reads or writes over.

Unlike reads by serializable or repeatable read transactions, a read by RC need not constrain timestamp ranges of other transactions. RC may invoke the TCM to ensure that when it reads, it reads a committed version. But the TCM need not include the read in its conflict matrix. And, if RC can determine when a version is committed and its timestamp by examining the version itself, it can avoid read locks entirely.

D. Optimizing Read-only Transactions

When a transaction RO declares itself to be read-only, supported commercially in, e.g., Rdb [11], RO need not set any "locks" if it is sufficiently early. Here such a declaration means that RO does not ask the TCM for permission to access resources if it identifies a recent time at which there are no concurrent transactions, i.e., earlier than the A.early of any active read/write transaction A. Thus, RO runs an as-of query for this time, just like any historical as-of query.

If RO is concurrent with active updaters, then it needs to participate in keeping the overall schedule serializable, even though it will never be aborted. We can set *RO.late* ← *RO.early* to minimize its conflicts with concurrent read/write transactions. Possible techniques for handling RO include those used for SI (section VIII.B). Further, once RO's time is earlier than A.early, for any active R/W transaction A, RO can continue without TCM visits, like an "as of" historical query.

E. Distributed Transactions

Dealing with distributed transactions using timestamping has been described in the past, e.g. [16]. The idea is that each cohort (local sub-transaction), when it enters the prepare phase (phase 1) of two phase commit, responds to the transaction manager with the timestamp range that bounds the acceptable timestamp for the transaction. The transaction manager can commit the transaction when the intersection of all timestamp ranges from all cohorts is non-null as it can choose any timestamp in this intersection as the transaction's timestamp. Otherwise, the transaction must abort. Our TCM provides the required timestamp range. Thus, our TCM approach can easily deal with distributed transactions as well as local transactions.

F. Single Version Data

Systems may support both multi-version and single version data. Because a TCM can provide both R/W concurrency and conventional LM blocking behavior, it can be used for both.

For single version data, the TCM acts like a conventional LM, blocking when conflicts occur. A TCM needs to retain locks on single version data of an earlier committed transaction until it can no longer impact timestamping of active transactions, as with multi-version data. But garbage collection can be prompt, as a single version data lock can be dropped once its data is overwritten by a subsequent committed transaction.

A TCM could be used in the place of an LM even when no multi-version data is present. One gets the same LM blocking behavior. Handling timestamp ranges increases TCM code path when conflicts occur, but the common "no conflict" case is the same. Deadlock detection can use the timestamp range technique instead of checking for wait cycles, a simplification. Aborts increase somewhat as null timestamp ranges occur more frequently than real deadlocks, but this should be rare.

G. Transaction Time Database Systems

Transaction time database systems provide multi-version support, including support for queries "as of" some past time in order to read a transaction consistent version of the database at a prior time. Immortal DB [19] went to great effort to reduce the penalty of supporting versions on current database access performance, reducing the penalty to a few per cent. Adding timestamp range conflict management to a transaction time database turns things around. With more concurrency for multiple versions, current database access can be improved. Supporting multiple versions thus turns into a performance plus, not a penalty to be minimized.

Highly concurrent read access is possible using a transaction time database with a conventional lock manager, but only for transactions declared read-only. Given a TCM, a read-only declaration is not needed to increase concurrency. With a read-only declaration, we can reduce lock overhead, however, sometimes while reading recent versions.

Using a TCM with a transaction time database has a very limited implementation impact. No versioning or "as of" query functionality need be changed. A TCM impacts only the way in which timestamps are selected. The timestamping process itself need not change.

H. Replication

Multi-master replication using snapshot isolation versions has been frequently discussed [8][9][14][15]. There are several flavors of snapshot isolation, with replication trade-offs in terms of strength of guarantee and efficiency of support. Replication that provides a variant of snapshot isolation is not truly serializable, and proposals that are serializable [6] do not allow read/write concurrency.

Our work does not deal directly with multi-master replication. However, primary copy replication (all updates go to the primary) with read-only secondaries [23] is readily provided via the transaction-time support enabled by our method. Readers at a secondary can see a historical transaction's serialized version with some small time delay. That is, an historical query at time t_1 cannot be asked until we can guarantee that no active transaction can commit with a timestamp earlier than t_1. Like historical queries in general, no locking is needed for these queries.

I. Conclusion

We have described our TCM timestamp range conflict manager, and discussed how it enables R/W concurrent access while providing all SQL isolation levels, including serializability. The timestamp range technique can also replace cycle detection as a way of detecting deadlocks. Our InnoDB implementation demonstrates that using a TCM improves performance while reducing the number of deadlocks. Finally, TCM use leads to timestamps for versions that are consistent with serialization order and so enables the TCM to be used as part of a transaction time database system.

REFERENCES

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

[2] P. Bernstein, N. Goodman: Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems. VLDB 1980: 285-300

[3] P. Bernstein, A. Fekete, H. Guo, R. Ramakrishnan, and P. Tamma: Relaxed-currency serializability for middle-tier caching and replication. SIGMOD 2006: 599-610

[4] P. Bober, M. Carey: Indexing for Multiversion Locking: Alternatives and Performance Evaluation. *IEEE TKDE* 9(1): 68-84 (1997)

[5] P. Bober, M. Carey: On Mixing Queries and Transactions via Multiversion Locking. ICDE 1992: 535-545

[6] M. Bornea, O. Hudson, S. Elnikety, and A. Fekete: One-Copy Serializability with Snapshot Isolation under the Hood. ICDE 2011: 625-636.

[7] M. Cahill, U. Röhm, A. Fekete: Serializable isolation for snapshot databases. *ACM TODS* 34(4):article 20 (2009)

[8] K. Daudjee, K. Salem: Lazy Database Replication with Snapshot Isolation. VLDB 2006: 715-726

[9] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. SRDS 2005: 73–84

[10] J. Gray, A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann 1993

[11] L. Hobbs, I. Smith, K. England: *Rdb: A Comprehensive Guide*. Digital Press (1999)

[12] C. S. Jensen and R. T. Snodgrass. Temporal data management. *IEEE TKDE*, 111(1):36--44, 1999.

[13] S. Jorwekar, A. Fekete, K. Ramamritham, S. Sudarshan. Automating the detection of snapshot isolation anomalies. VLDB 2007: 1263-1274.

[14] B. Kemme and G. Alonso. A New Approach to Developing and Implementing Eager Database Replication Protocols. *ACM TODS*,25(3):333–379, 2000.

[15] Y. Lin, B. Kemme, R. Jiménez-Peris, M. Patiño-Martínez, J. Armendáriz-Iñigo: Snapshot isolation and integrity constraints in replicated databases. *ACM TODS* 34(2): article 11, 2009
[16] D. Lomet: Using Timestamping to Optimize Two Phase Commit. PDIS 1993: 48-55
[17] D. Lomet, R. Barga, M. Mokbel, G. Shegalov, R. Wang, Y. Zhu: Transaction Time Support Inside a Database Engine. ICDE 2006: 35
[18] D. Lomet, M. Hong, R. Nehme, R. Zhang: Transaction time indexing with version compression. PVLDB 1(1): 870-881 (2008)
[19] D. Lomet, F. Li: Improving Transaction-Time DBMS Performance and Functionality. ICDE 2009: 581-591
[20] D. Lomet, R. Snodgrass, and C. Jensen: Using the Lock Manager to Choose Timestamps. IDEAS 2005: 357-368
[21] Oracle: Oracle Flashback Technology. http://www.oracle.com/technology/deploy/availability/htdocs/FlashbackOverview.htm, 2005
[22] Oracle:Total Recall http://www.oracle.com/technology/products/database/oracle11g/pdf/flashback-data-archive-whitepaper.pdf, 2008.
[23] C. Plattner, G. Alonso, and M. T. Özsu. Extending DBMSs with satellite databases. *VLDB J*, 17(4):657–682, 2008.
[24] S. Revilak, P. O'Neil, E. O'Neil: Precisely serializable snapshot isolation (PSSI). ICDE, 2011: 482-493.
[25] B. Salzberg: Timestamping After Commit. PDIS 1994:160-167.
[26] M. Stonebraker: The Design of the POSTGRES Storage System. VLDB, 289 – 300, 1987.
[27] M. Sinha, P. Nanadikar, S. Mehndiratta: Timestamp Based Certification Schemes for Transactions in Distributed Database Systems. SIGMOD 1985: 402-411.
[28] G. Weikum, G. Vossen: *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann 2002.

APPENDIX A

Procedure 1: for RW conflicts

```
1 function adjust_timestamps_rw(holder, requester):
2   if not requester.end and not holder.end:
3     new_time = get current time
4     holder.late = new_time
5     holder.end = True
6     requester.early = new_time
8   else if not requester.end and holder.end:
9     if holder.late > requester.early:
10        requester.early = holder.late
12  else if requester.end and not holder.end:
13    if requester.late > holder.early:
14      requester.early = requester.late - 1
15      holder.late = requester.late - 1
16      holder.end = True
17    else:  # cannot be solved via blocking
19      ABORT(requester)
21  else:
22    if requester.early > holder.late:
23      # no adjustment needed
24    else if requester.early <= holder.late <= requester.late:
25      requester.early = holder.late
26    else if holder.early <= requester.late <= holder.late:
27      if not holder.committed:
28        new_time = requester.late - 1
29        holder.late = new_time
30        requester.early = new_time
31    else:  # cannot be solved via blocking
33      ABORT(requester)
34  return SUCCESS
```

Procedure 2: for WR conflicts

```
1 function lock_rec_adjust_timestamps_wr(holder, requester):
2   if not requester.end and not holder.end:
3     new_time = get current time
4     holder.early = new_time
5     requester.late = new_time
6     requester.end = True
8   else if not requester.end and holder.end:
9     if requester.early > holder.late:
10      if not holder.committed:
11        ABORT(holder) # COULD block requestor
13    else:
14        new_time = holder.late - 1
15        requester.late = new_time
16        requester.end = True
17        holder.early = new_time
19  else if requester.end and not holder.end:
20    if requester.late > holder.early:
21      holder.early = requester.late
23  else:
24    if requester.early > holder.late:
25      if not holder.committed:
26        # COULD block requestor
27        ABORT(holder)
29    else if requester.early <= holder.late <= requester.late
30      new_time = holder.late - 1
31      requester.late = new_time
32      holder.early = new_time
34    else if holder.early <= requester.late <= holder.late:
35      holder.early = requester.late
36  return SUCCESS
```

Procedure 3: for WW Conflicts

```
1 function lock_rec_adjust_timestamps_ww(holder, requester):
2   if not requester.end and not holder.end:
3     new_time = get current time
4     requester.early = new_time
5     holder.late = new_time
6     holder.end = True
7     ABORT(requester) # COULD block requester
8   else if not requester.end and holder.end:
9     if requester.early < holder.late:
10      requester.early = holder.late
11    if not holder.committed:
12      ABORT(requester) # COULD block requester
13  else if requester.end and not holder.end:
14    if requester.late > holder.early:
15      new_time = requester.late - 1
16      holder.late = new_time
17      holder.end = True
18      requester.early = new_time
19      ABORT(requester) # COULD block requester
20    else: # we can abort either transaction here
21      ABORT(requester)
22  else:
23    if requester.early > holder.late:
24      ABORT(requester) # COULD block requester
25    else if requester.early <= holder.late <= requester.late:
26      requester.early = holder.late
27      ABORT(requester) # COULD block requester
28    else if holder.early <= requester.late <= holder.late:
29      requester.early = requester.late - 1
30      holder.late = requester.late – 1
31      ABORT(requester) # COULD block requester
32    else: # we can abort either transaction here
33      ABORT(requester)
34  return SUCCESS
```