Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions

C. Mohan, Hamid Pirahesh, Raymond Lorie

IBM Almaden Research Center, San Jose, CA 95120, USA
{mohan, pirahesh, lorie}@almaden.ibm.com

Abstract We present efficient and flexible methods which permit read-only transactions that do not mind reading a possibly slightly old, but still consistent, version of the data base to execute without acquiring locks. This approach avoids the undesirable interferences between such queries and the typically shorter update transactions that cause unnecessary and costly delays. Indexed access by such queries is also supported, unlike by the earlier methods. Old versions of records are maintained only in a *transient* fashion. Our methods are characterized by their flexibility (number of versions maintained and the timing of version switches, supporting partial rollbacks, and different recovery and buffering methods) and their efficiency (logging, garbage collection, version selection, and incremental, record-level versioning). Distributed data base environments are also supported, including commit protocols with the read-only optimization. We also describe efficient methods for garbage collecting unneeded older versions.

1. Introduction

It is a known problem in data base management systems (DBMSs) that it is difficult to run simultaneously transactions that update some data more or less randomly (call them transactions of ***type Tu***) and transactions that access a large quantity of data (***type Tr***) [PMCLS90]. If a Tr transaction writes data and there is a conflict with data accessed by a Tu transaction, then one of the transactions will have to wait. The conflict is logical and the system can only serialize the transactions. But, if a Tr transaction is a read-only transaction, then other solutions are possible. In the rest of the paper, we assume that the Tr transactions are read-only transactions. To avoid or reduce the undesirable interferences between the Tr and Tu transactions, it has been proposed, for example, to keep shadow pages around for use by the Tr transactions. Doing this could be quite expensive. R. Bayer et al. [BaHR80] have designed a locking scheme where two versions are kept but the method provides less concurrency and is quite complex (for a detailed critique of that algorithm, see the section "5. Related Work").

In those situations where the Tr transactions do not mind seeing an internally *inconsistent* version of the data base (e.g., a state in which referential constraints are observed to be violated), they can be shown the latest (committed) state as each piece of data is accessed and locks can be released as the scan moves forward. This is the case with use of isolation levels like *cursor stability* and so on. Cursor stability does not avoid the cost of locking. Nor does it avoid the waits that may be caused if the data being accessed is in the uncommitted state. For more detailed discussions about cursor stability and a simple method (called *Commit_LSN*) to reduce the locking costs, the reader is referred to [Moha90b]. Those ideas are applicable here also for the Tu transactions.

Here, we assume that every Tr transaction wants to see a *transaction consistent* state of the data base but it often does not matter if that consistent state is not the one that exists at the time the Tr transaction finishes (like in most DBMSs), but was the state at a *reasonable* time before. This can be done by maintaining a ***stable state*** of the data base for access by Tr transactions while Tu transactions are updating the data base. When no Tr transaction is active, the stable state can be updated to reflect a newer state. If a read-only transaction were to desire to see the most up-to-date version of the data, then it can always run as a Tu transaction and acquire locks on everything that it reads.

Figure 1 illustrates data base versions and transaction types. Transactions move the state of the data base along the time axis. The time axis is divided into version periods, which are numbered v-2, v-1, v, and so on. A Tu transaction has version v iff it **commits** during version period v. A Tr transaction has version v iff it **arrives** during version period v. A Tr transaction of version v sees the results of only Tu transactions with versions *less than* v. In other words, as shown in Figure 1, all Tr transactions arriving in version period v will see the state of committed data in the data base at the *beginning* of version period v. Periodically or on a command, the system switches to a new version period, hence allowing the subsequently arriving Tr transactions to see the more recently committed state of the data base. This is called a ***refresh*** operation.

Two algorithms are presented in this paper. The first algorithm, called ***two version algorithm***, allows only two kinds of transactions to be active at the same time: Tr transactions of the *current* version and Tu transactions. In this algorithm, as we will see later, the system can switch to a new version period only when no *Tr* transaction is active. The second algorithm, called ***N version algorithm***, removes this restriction, and allows the version period to be switched immediately and the old version periods to be phased out gradually. This algorithm allows Tr transactions of the last N-1 version periods, in addition to Tu transactions, to be active simultaneously. The N version algorithm subsumes the two version algorithm, and the overhead is minimal. However, in section 2, the two version algorithm is described first to explain the concepts in simpler terms. Next, in section 3, the N version algorithm is described and then, in section 4, it is enhanced for the distributed data base environment. Related work is discussed in section 5.

2. Two Version Algorithm

We assume that the data base is a series of records, each of which is identified by a key (logical key or any kind of record identifier). We also assume that the Tr transactions,

1992 ACM SIGMOD - 6/92/CA, USA
© 1992 ACM 0-89791-522-4/92/0005/0124...$1.50

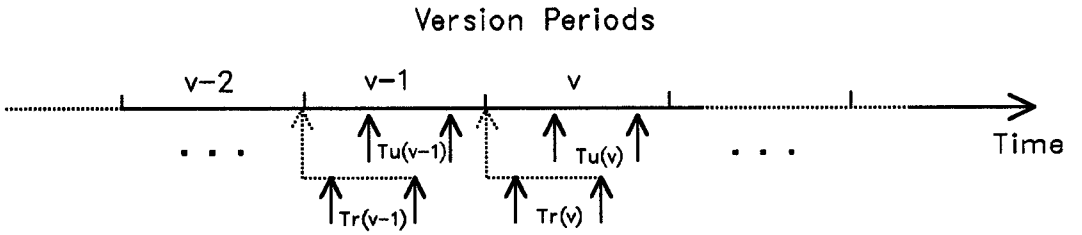

Figure 1: Version Periods and Association of Transactions to Version Periods

which are read only and which do not acquire locks, are identified as such when they are started and that the Tu transactions use locking to synchronize amongst themselves. We show that a maximum of only 3 versions of a record need to exist at any point in time.

Three versions may exist because one record may be needed to represent the stable state (for use by Tr transactions), another one to represent the last committed value that is not yet in the stable state, and a third one to represent the uncommitted state.

Assume that the data base system supports a direct access path from a key to the associated data. Generally, the data associated with a key is the record. This is the case with systems in which the data is stored in the leaf pages of the primary index. Every primary key's value is required to be unique. In DBMSs like System R, SQL/DS and DB2, the records are stored in separate areas of storage and each record is identified by a record identifier (***RID***). In the indexes of such systems, the data associated with a key is the RID of the record containing the key value. If an index is not a unique index, then multiple RIDs may be associated with a particular key value [Moha90a]. For the following discussion, the RID can be thought of as the primary key since it is also unique. We discuss *secondary* indexes in the section "2.2. Secondary Indexes". When transactions are started, they are assigned unique IDs. The IDs always increase in value.

2.1. Data Structures and Operations

In the proposed *transient* versioning scheme, the data associated with a primary key is a structure as shown in Figure 2. The structure is a stack of three substructures. Each substructure consists of 2 entries:

TRN the ID of the transaction which created this version
PTR pointer to the data (record)

A PTR value of 0 indicates that the corresponding TRN (> 0) had deleted the previous version of that record (i.e., the record does not exist) or that that version never existed (TRN = 0). If there is only one version, then only substructure 1 has nonzero values. If a newer version is added, then the contents of substructure 2 receive the data previously in substructure 1 and the information about the new version goes into substructure 1, and so on. We assume that initially there is a single version of each record and that the TRN associated with that version is 0.

In virtual storage, the DBMS maintains a list of uncommitted Tu transactions (call the list ***UL*** - uncommitted list), and a list of committed but not yet in stable state Tu transactions (call the list ***NSL*** - non-stable list). In terms of the version period diagram, NSL is a list of Tu transactions with version v (assuming v is the current version), and UL is a list of uncommitted Tu transactions which will have a version greater than or equal to v (because they may commit during version period v or later). Initially, both UL and NSL are empty. A ***version of a record*** is defined to be the version of the transaction (identified by TRN) that created that record.

In the following, we describe the actions taken at various stages of a transaction's execution.

Start a Tu transaction: Insert the transaction number in UL.

Read a record for a Tu transaction: Access the structure corresponding to the submitted key and return the version on the top of the stack (in substructure 1). This version would have been created by the current transaction or by a committed transaction (guaranteed by the normal locking protocol).

Update a record (Tu transactions only): Locate the record as in the Read operation above, and find TRN(1). If TRN(1) is the same as the current transaction's ID, then the update can be made in place in the most recent version of the record.

Otherwise (i.e., TRN(1) is not in UL), the version must have been created through an insert of a new record or update of a previously existing record by a committed transaction and the current transaction must create a new version. Before the current transaction inserts its version into substructure 1 and moves the other entries down the stack, it examines the currently existing versions to see if any of them should be discarded based on whether or not any Tr transaction could be reading it now or may need it for reading in the future. Note that locking ensures that the TRN(1) transaction cannot be in UL unless it is the same as the current transaction.

If TRN(1) is not in NSL also, then its version is the *stable* version (i.e., it belongs to a Tu transaction with version < = v-1) and all other (i.e., preceding) versions, if any, are discarded. If TRN(1) is in NSL, then that version is kept as it is the most recently committed value and it will be needed for the future Tr transactions if the system were to switch to the next version period (i.e., version period v + 1) before the commit of the current transaction. Then, the next most recent version is examined. If TRN(2) is in NSL, then that version is removed since it is an old committed value that is not used (and will not be used) in representing the stable state and the TRN(3) version is retained since that must be the stable version. If TRN(2) is not in NSL, then it must be the stable version and is kept and the TRN(3) version, if it exists, is discarded.

Note that it is enough if only the changed fields are recorded. We call this feature ***incremental versioning***. Removing an old version might therefore mean combining two versions together to get a full updated record. If incremental

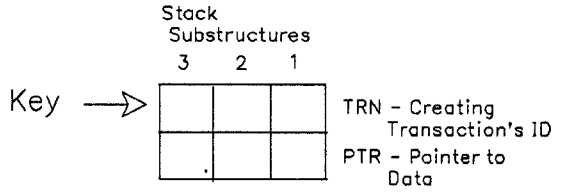

Figure 2: Data Structure Providing Information About Different Versions of a Record

versioning is done, then multiple versions may have to be consulted to get at the complete record in a given version during a read operation.

Commit a Tu transaction: First, force write the commit record to the log. Next, remove the transaction number from UL and insert it into NSL. Finally, release all the locks. Releasing all the locks and removing the transaction number from UL are the only operations that need to be performed for a transaction which did not modify the data base in anyway (i.e., a read-only *Tu* transaction).

Abort a Tu transaction: Write an abort record to the log. Remove the versions created by the transaction (i.e., pop the stacks) during the undo of the transaction and then remove the transaction number from UL. Release all the locks.

Read a record for a Tr transaction: Find the most recent version (substructure 1). If its TRN is in UL **or** in NSL consider the next most recent version, until the stable one (i.e., the first one whose TRN is neither in NSL nor in UL) is found. Note that the stable version might be the one in which this record is to be considered a nonexistent one (deleted or uncreated). If both TRN(1) and TRN(2) are in UL or NSL, then the version that is pointed to by PTR(3) is the version that should be read since it must be the stable version (i.e., there is no need to check if TRN(3) is in UL or NSL).

Refresh stable version (i.e., switch to a new version period): Reset NSL to empty when no Tr transaction is active (in the section "3. N Version Algorithm", we describe a very efficient method for determining when no Tr transaction is active). It is this action that atomically makes all the most recently committed versions of data become "visible" to new read-only transactions.

If PTR(1) is zero, and TRN(1) is neither in UL nor in NSL, then the existing versions, if any, pointed to by PTR(2) and PTR(3) can be garbage collected and the stack substructures can be freed by setting all entries to zeroes.

Note that the value of TRN(3) is never examined and hence there is no need to store TRN(3). This way, we can reduce the amount of storage consumed due to versioning. For simplicity of presentation, we continue to show TRN(3) in the following.

Figure 3 provides a self-explanatory example scenario to understand the functioning of the algorithm. It illustrates the reading and modification of a single record with key 1234 using several transactions. It should be clear from this example that more than 3 versions are not needed.

2.2. Secondary Indexes

The method presented above is readily applicable to secondary index information. A secondary index is simply con-

Tran Type/ Tran# ACTION	Primary Key or Rec ID	Versions (3) (2) (1)	UL	NSL
(Stable state)	1234	0 0 1 **TRN** 0 0 9 **PTR**	()	()
Tu/2 starts			(2)	()
Tu/2 updates	1234	0 1 2 0 9 8	(2)	()
Tu/2 updates	1234	0 1 2 0 9 7	(2)	()
Tr/1 starts	...		(2)	()
Tr/1 reads TRN(2) version (Pointed to by 9)	1234			
Tu/2 commits	...		()	(2)
Tr/1 ends	...			
Refresh	1234	0 1 2 0 9 7	()	()
Tu/3 starts	...		(3)	()
Tu/3 updates	1234	0 2 3 0 7 6	(3)	()
Tu/3 commits	...		()	(3)
Tu/4 starts	...		(4)	(3)
Tu/4 updates	1234	2 3 4 7 6 5	(4)	(3)
Tr/2 starts	...			
Tr/2 reads TRN(3) version (Pointed to by 7)	1234			
Tr/2 ends	...			
Refresh	...		(4)	()
Tr/3 starts	...			
Tr/3 reads TRN(2) version (Pointed to by 6)	1234			
Tr/3 ends	...			
Tu/4 ends	...		()	(4)
Tu/5 starts	...		(5)	(4)
Tu/5 updates	1234	3 4 5 6 5 4	(5)	(4)
Tu/5 aborts	1234	0 3 4 0 6 5	()	(4)
Tu/6 starts	...		(6)	(4)
Tu/6 deletes	1234	3 4 6 6 5 0	(6)	(4)
...	...	...	...	...

Figure 3: An Example Scenario

sidered as a unary relation where the only column contains the concatenated key *secondary key, primary key* and there is no other data (hence incremental versioning does not make sense for secondary indexes). The only operations that need to be supported are insert and delete. Additionally, the differences between versions of a key is only the information about whether the key exists or not. Therefore, the substructure becomes

```
          (3) (2) (1)
9|1234     0   0   1  TRN
          0   0   1  Flag - Information about
                            key's existence
```

In the above example, 9 is the secondary key and 1234 is the primary key (or RID). Since there is no other data associated with a secondary index entry, the existence flag says whether in a particular version the index entry should

Tran Type/ Tran#/ ACTION	Index Entry	Versions (3) (2) (1)	UL	NSL
(Stable state)	9|1234	0 0 1 **TRN** 0 0 1 **Flag**	()	()
Tu/2 starts			(2)	()
Tu/2 updates (Secondary key changes from 9 to 8)	9|1234 8|1234	0 1 2 0 1 0 0 0 2 0 0 1	(2)	()
...	...	...	...	...

Figure 4: An Example Scenario for a Secondary Index

be considered to exist (Flag = 1) or not (Flag = 0). As before, TRN(3) need not be stored since its value will not be examined. Assuming that there is a secondary index, Figure 4 illustrates the functioning of our algorithm for a portion of the scenario illustrated in Figure 3.

This approach to dealing with index updates has the unexpected side benefit of avoiding the need for *next key locking* during key delete operations to handle the *phantom* problem. Next key locking is discussed in great detail in [Moha90a, MoLe92]. The approach of using *logical* key deletions, rather than *physical* deletions, was proposed in [Moha90b] as a way to avoid next key locking during key deletions. Applying the transient versioning approach to indexes gives us that capability automatically since, for deleted keys, index entries exist at least until the deletion is committed. The reader is referred to [Moha90b] for the rationale on why, under these circumstances, there is no need to do next key locking.

Note that the index entries do not directly refer to any particular version of the referenced records and hence creation or deletion of a version does not always require updating the indexes. For example, if a new record version were to be created due to an update operation, an index update will not be needed if the update did not cause a secondary key to change.

2.3. Implementation

UL must be implemented in a compact way to provide efficient support for the existence test. One such way is to keep a bit map to indicate the status of transactions with TRN > M. Over time, M's value is advanced. The bit vector is made large enough to ensure that all transactions preceding M + 1 are neither in UL nor in NSL. A similar technique is used for representing NSL. A third bit vector, called ***NSUL***, representing the set union of UL and NSL can be maintained to make checking by Tu and Tr transactions very efficient. The idea is to make the mapping between the same bit position in all of the 3 vectors and a TRN number be the same. When NSL (the second vector) is set to zero (i.e., when the version period is being switched), the third vector is set to the first vector. When a transaction starts, it sets the corresponding bits in the first and third vectors. When a transaction commits, it resets the corresponding bit in the first vector and sets the corresponding bit in the second vector.

In fact, we do not need to use NSL at all! The only time we use NSL in the algorithms is in the update record logic, where we check if a transaction is a member of NSL, given that it is not a member of UL. This can be answered by looking at NSUL. Hence, we only need to keep UL and NSUL. A *latch* (i.e., a semaphore) is associated with each of these lists. Each of the latches is acquired in the S or the X mode depending on the kind of synchronization needed for the operation being performed on the associated list.

The proposed structure has a positive impact on logging and locking:

- Tr transactions do not set locks and do not write log records.
- If no intermediate save points (i.e., partial rollbacks) are supported, old values ("before images") of records themselves do not need to be logged by Tu transactions when they modify the data base; then, it is only necessary to log the key or RID of a modified record, and the new values ("after images") of the modified fields. Even if intermediate savepoints are supported, it is necessary to log the before image of an updated record only when a transaction is updating a record a second or a subsequent time (this should be a rare event and in any case it is very easy to detect when it is happening). After images are logged to support media recovery and to avoid having to force modified data to disk at commit time [MHLPS92].
- Tu transactions never get blocked by Tr transactions.

The method is amenable to on-line, asynchronous garbage collection. When transaction activity is low, if a page is modified, then all records on the page can be scanned for superfluous versions, taking into account the UL and NSL lists (more precisely, the NSUL list), as in an update operation. When transaction activity is very low, a garbage collection transaction can clean up a few pages at a time. This will bring down the average number of versions in the data base (this would be particularly appropriate for a dedicated back-end data base machine [PMCLS90]). Since a secondary index entry does not explicitly refer to versions of the associated data record, secondary index entries can be garbage collected *independently* of the corresponding data records.

Physical clustering of data is preserved as long as each page has enough free space for accommodating the updated record. Additional page access is performed by updaters only if the original page does not have enough space for the new version of the record. When such an overflow exists and incremental versioning is being done, garbage collection of the *overflow* record cannot be done independently. It must be done by going through the primary page (i.e., the page that contains the pointer to the overflowed record).

In this two version algorithm, when the version period is switched, no Tr transaction is allowed to be active. Either all the executing Tr transactions should be aborted or we have to wait until they are completed before the switching is done. While this is going on, *new* Tr transactions can either wait or be converted to *Tu* transactions. After the switching, Tr transactions are allowed to execute as usual. If some Tr transactions are very long, then it is desirable to allow gradual *phasing out* of these transactions when the version period is switched. This requires the system to allow Tr transactions of both the new version and the old versions to be active simultaneously. In fact, in general, we want many versions of Tr transactions to be active, so that we can switch the version period while the system is phasing out other version periods. The N version algorithm described in the next section allows this.

3. N Version Algorithm

The N version algorithm allows Tr transactions of the current version, the current version minus one, minus two, .., minus (N-2) and Tu transactions to be concurrently active in the system. Hence, the phasing out period for Tr transactions, after switching version periods, can be nested N-2 times. The two version algorithm allows no phasing out period, the three version algorithm allows one level of nesting of phasing out periods, and so on. The stack size for each record can range from 3 to N+1 with the N version algorithm. However, stack sizes less than N+1 may cause some Tu transactions to wait because there is no room in the stacks for them to create newer versions. For a given N, the higher the stack size, the lower is the probability of waiting.

The algorithm uses a data structure, called ***versionblock***:

```
versionblock
  cur_v      /* current version number      */
  oldest_v   /* version number of the oldest
                active Tr transaction       */
  N          /* max number of versions      */
```

Initially, cur_v and oldest_v are set to 1. The ID of the first transaction must be greater than TRN of any record in the data base. If all the records in the data base are initialized to contain TRN = 0, then the ID of the first transaction can be 1. The latch ***future_v*** is used to serialize refresh operations and assigning birth versions of Tr transactions. The NSUL list, as before, contains the committed Tu transactions of version v and the uncommitted transactions. We call this list ***NSUL(v)***. We add a new list, called ***NSUL(v-1)***, which contains the committed transactions of versions v-1, and v, and the uncommitted transactions. So, a Tr transaction of version v needs to check only NSUL(v), and a Tr transaction of version v-1 needs to check only NSUL(v-1). More generally, we need NSUL(v), NSUL(v-1), ..., NSUL(v-(N-2)). The bits in all the NSULs are initialized to contain zeroes assuming that all the records are initialized to contain TRN = 0.

In the following, we describe the actions taken at various stages of a transaction's execution.

Start a Tr transaction: The current version is assigned as the ***birth version*** of this transaction. The birth version is used for reading the record with the appropriate version. Also, the version period associated with the birth version is latched in share (S) mode. This allows us to phase out this version period later by latching this version period in exclusive (X) mode. The X latch will be granted only when all Tr transactions of this version are completed.

```
if N = 2 then latch (future_v,S);
/*wait if version period is changing & N=2*/
latch (versionblock,S);
if N = 2 then unlatch future_v;
birthversion = cur_v;
  /* remember the birth version */
latch (birthversion,S);
unlatch versionblock;
```

Note that while a version switch is in progress and N = 2, starting of new Tr transactions will be delayed due to the latch acquired on future_v.

Read a record for a Tr transaction: Read the most recently created version of the record whose version is *less than* the birth version of the Tr transaction That is, we need to find the version whose TRN is not in NSUL(birthversion). This lookup of NSUL must be done while holding an S latch on it.

Commit or abort a Tr transaction: Release the S latch on the birth version.

Start a Tu transaction: Add this transaction to UL and all NSULs (i.e., set the bit associated with this transaction in all these lists while holding an S latch on UL).

Read a record for a Tu transaction: Same as the one for the 2 version algorithm.

Update a record (Tu transactions only): The logic for updating a record is specified in the following pseudo code. The logic for delete and insert is similar, and is not shown below. The *first* update of a record within a transaction requires that a new record be pushed onto the stack. First, the stack is garbage collected, and then, if there is room, the updated record is pushed. Otherwise, the stack is full, which happens only if the stack size is less than N+1, and the transaction has to wait until the element at the bottom of the stack is not needed anymore. This happens when all the Tr transactions with versions up to and including the version of the element *before* the bottom of the stack (i.e., the last but one element) are finished.

Update transactions try to get rid of extra elements in the stack to keep the data base compact. This is done in two ways. The oldest_v pointer is moved to the oldest version period with active Tr transactions so that the unneeded elements can be deleted from the stack. Also, if there is more than one element in the stack belonging to the same version period, then only the *latest* one is kept.

```
if TRN(top) = my_transid then
  update the record on the top of the stack
 else
  Push_careful;
  insert the updated record on the top of the
        stack with TRN = my_transid;
 endif
```

Push_careful:

```
/* This routine does a garbage collection of the
   stack, and if the stack is full, it waits
   until a slot becomes available. Note that
   this routine is always called for the first
   update of each transaction. */
garbage_collect_stack;
if the stack is full then
  /* This happens only if stack size is <N+1.
     For N = 3 and a stack size of 3, at this
     point, the stack would look as follows:

     top:    a record with version v
     middle: a record with version v-1
     bottom: a record with version  < (v-1)

     bottom of the stack (version <(v-1)) is
     needed for Tr transactions of version
     v-1. We have to wait until those
     transactions terminate.              */
  tmpvbu = version of the element just above
           the bottom of the stack;
  latch (versionblock,X);
  tmp_oldest_v = oldest_v;
  unlatch versionblock;
  k = tmp_oldest_v;
  repeat
    if k > tmpvbu then exit;
```

```
    /*wait for completion of k version period*/
    latch (version period k,X);
    latch (versionblock,X);
    if k < oldest_v then
       last_v = oldest_v;
       unlatch versionblock;
       unlatch version period k;
       k = last_v;
     else /* k must equal oldest_v */
       oldest_v = k + 1;
       unlatch versionblock;
       unlatch version period k;
       k = k+1;
     endif
   endrepeat;
  garbage_collect_stack;
   /* now stack is guaranteed not to be full */
 endif
push the stack;

Garbage_collect_stack:
 latch (versionblock,X);
 call move_oldest_v;
 tmp = versionblock;
 unlatch versionblock;
 remove all elements of the stack except
  the latest record of version tmp.cur_v,
  the latest record of version tmp.cur_v-1,
     ...,
  the latest record of version tmp.oldest_v, and
  the latest record of version < tmp.oldest_v;
  /* note that these are done after unlatching
     the versionblock to prevent a hot spot on
     versionblock, particularly because an I/O
     may be involved to remove an overflow
     record */

Move_oldest_v:
 /* move oldest_v ahead if possible */
 for k = oldest_v to cur_v
   /* check if the version k is not needed */
   if k < cur_v then
     conditional latch (version period k,X);
     if latch granted then
        oldest_v = k+1;
        unlatch version period k;
      else exit;
      endif
    else /* k = cur_v */
      oldest_v = cur_v;
      exit;
    endif
  endfor
```

Commit a Tu transaction. Same as the one for the 2 version algorithm (since NSL is not being maintained, only UL needs to be changed and this is done while holding an S latch on UL).

Abort a Tu transaction: Same as the one for the 2 version algorithm.

Switch the version period (Refresh): This operation is used to increment the value of cur_v (current version number) it by 1. If the oldest *active* version period is N-2 versions behind the current version (i.e., oldest_v = cur_v - (N-2)), then it waits until all the Tr transactions of the oldest version period are completed before starting a new version. After switching the version, it moves the oldest_v pointer to the oldest active version period, thus allowing the data in the stack to be garbage collected. Note that attempts are made to advance oldest_v during updates by Tu transactions also. However, it is necessary to do it here too because not all Tu transactions are guaranteed to update the data base and thereby cause oldest_v to be moved ahead.

```
  latch (versionblock,X);
  if (cur_v - oldest_v) = (N-2) then
    /* Oldest version period must be
       phased out; all Tr transactions
       of the oldest version must end; */
    unlatch versionblock;
    latch (future_v,X);
    /* queue behind other refreshers */
    latch (versionblock,X);
    tmp = versionblock;
    unlatch versionblock;
    /* wait for Tr transactions with
      version = tmp.cur_v - (N-2) to finish */
    latch ((tmp.cur_v - (N-2)),X);
    latch (versionblock,X);
    unlatch version period (tmp.cur_v - (N-2))
    unlatch future_v;
   endif
  latch(UL,X);
  NSUL(cur_v+1) = UL;
  unlatch UL;
  cur_v = cur_v + 1;
  /* set oldest_v to oldest active version */
  call move_oldest_v;
  /* see the routine under update record */
  optionally, garbage collect unneeded bits
   in all the lists while holding X latches
   on all of them;
  unlatch versionblock;
```

An extreme case of this algorithm is to have a large N, and try to switch the version at the end of every update transaction. Under these conditions, Tr transactions will always get the most up-to-date data as of the start of such transactions. Note that, as N becomes larger, the number of NSUL lists will also become larger. Also, the width of NSULs will become larger because there must be one bit for each transaction from the oldest version to present, even if they are not active. As a result, the NSULs' memory consumption will grow nonlinearly (power of two). Also, commit processing will become more expensive, because all of these lists would have to be updated. However, the resource requirements do not change much for low values of N.

4. Distributed N Version Algorithm

The distributed N version algorithm is very similar to the nondistributed N version algorithm presented in the previous section. Here, all transactions are assigned global version numbers. However, this does not force the current version numbers of all the nodes to be synchronously changed. Different nodes can have different current versions at a given time. In fact, each node can decide individually what the value of N should be for that node. Switching the current version is autonomously decided by each node. However, to minimize transaction delays and aborts, the current versions of different nodes must be close to one another.

The version of a Tu transaction is decided by its *origin* node just before that node initiates the two phase commit

protocol (for details about the current industry standard *Presumed Abort* variant of the two phase commit protocol the reader is referred to [MBCS91, MoLo86]). All the other nodes where the transaction executed (i.e., the ***participant*** nodes) are informed of this value and those nodes use the same value for this transaction. If the version assigned is found to be *less than* the current version of a participant, then the participant votes for the abort of the transaction. The originator can try to assign a proper version to a Tu transaction to minimize this. This can be done by keeping track of the approximate versions of the participants during the course of interactions with other nodes, and assigning the maximum version number of the participant (or slightly higher) to minimize abort due to version number obsolescence. Note that a *future* version can be assigned to a Tu transaction. This is particularly necessary if the maximum version number of the other nodes is greater than the current version of a node. After commit of a future transaction, its locks are held until the current version is moved ahead to catch up with it.[1] Note that if the locks are released earlier, Tu transactions of current version may read the unlocked record whose values should be visible in future only. During recovery, the locks of the future committed transactions must be reacquired, the same way as for prepared transactions [MHLPS92].

The version of a Tr transaction is decided by its origin node at its start time. All the participants use this version (birth version) to read the correct record version for this transaction. A read transaction may come too early or too late to a node with respect to the current (*cur_v*) and last (*oldest_v*) version periods of the node. In this case, the Tr transaction is aborted. This can be given as a hint to the process that decides on version period switching at the involved nodes to bring their versions closer to each other. The timing of version switching (refresh) is the user's responsibility (although it can be made periodic with the system's assistance) and that aspect is not discussed here.

The nondistributed case is a special case of the distributed case. There is no significant overhead if the more generalized distributed algorithms are used for the nondistributed case also.

Each node has its own set of UL and NSUL lists which it maintains autonomously. It is possible for a node to commit a Tu transaction which has started in another node and has a version > cur_v. This transaction has a version in the future. The definition of the UL list is enhanced to include all the transactions that belong to the future and uncommitted transactions (which may also belong to the future). Also, a new list, called ***PL (Prepare List)***, is added. PL contains the list of transactions that are in the *prepared* state of two-phase commit or that have committed but have been assigned future version numbers. The format of this list is the same as that of the other lists.

In the following, we describe the actions taken at various stages of a transaction's execution.

Start a Tu transaction: Same as the one for the nondistributed case, both at the origin and the participant nodes.

Read a record for a Tu transaction: Same as the one for the nondistributed case.

Update a record (Tu transactions only): Same as the one for the nondistributed case.

During two phase commit of a Tu transaction:

- **Start of commit protocol at the origin node of a Tu transaction:** Assign a version > = cur_v to this transaction. Send the assigned number in the prepare messages to the participants.

- **Unilateral abort at a participant node of a Tu transaction:** Same as the one for the abort of a Tu transaction in the nondistributed case.

- **Processing of a prepare message at a participant node of a Tu transaction when participant is voting read-only:** Release locks. Remove the transaction number from UL while holding an S latch on UL.

- **Processing of a prepare message at a participant node of a Tu transaction when participant is considering voting yes (i.e., willing to commit):**

```
latch (versionblock,S);
if transaction version < cur_v
 then
  unlatch versionblock;
  follow abort logic and vote no;
 else
  if transaction version = cur_v then
     remove it from UL
   else /* transaction version > cur_v */
    remember its version in its transaction
      control block;
  endif
  set the corresponding bit in PL;
  unlatch versionblock;
  do any required 2PC logic (e.g., forcing
    prepare log record), remembering the
    version number as part of prepare
    record and vote yes;
endif
```

- **Processing during commit (phase two of commit protocol) at a participant node of a Tu transaction:**

```
do any required 2PC logic (including logging
  of commit record with version number),
  but do not release locks or delete the
  trans control block
latch versionblock;
if transaction version <= cur_v
   then
    remove the transaction from PL;
    unlatch versionblock;
    release locks;
    delete the transaction control block;
 else
  unlatch versionblock;
  remember the transaction locks and version
    number in log records;
  go to sleep;
   /* this process will be woken up by the
      version switching process when the
      version of this transaction is equal
      to the current version of the node.
```

1 Note that the read (S) locks of Tu transactions can be released during the prepare phase and hence the locks that would have to continue to be held for a *future* Tu transaction will be only the write (X) locks.

```
      The locks will be released then. */
   /* after wake up */
   latch(PL,S)
   remove the transaction from PL;
   unlatch PL;
   release locks;
   delete the trans control block;
  endif
```

- **Processing during abort (phase two of commit protocol) at a participant node of a Tu transaction:**

```
do any required 2PC logic;
remove transaction from PL and UL while
  holding S latches on them;
delete the transaction control block;
release locks;
```

Start a Tr transaction:

- **At the origin node:** Assign the birth version the same way as for the nondistributed case.

- **At a participant node** (this is done at the first visit of a Tr transaction to a participant node):

```
latch (versionblock,S);
if birthversion < oldest_v or
    birthversion > cur_v then
  abort; /* out of range version */
 else
   /* check to see if the oldest version
     period is being phased out */
   conditional latch (birthversion,S);
   if latch not granted then abort;
 endif
unlatch versionblock;
```

Read a record for a Tr transaction:

```
Find the first element in the stack with
  version < birthversion;
latch(PL,S);
check if transaction of version is in PL;
unlatch PL;
If transaction is in PL
    then abort
  /* An alternative is to wait until the
     prepared transaction ends, but this
     may take a long time, and we prefer
     that the read transactions not wait. */
 else read this version;
 endif
```

Commit or abort a Tr transaction: Release the S latch on the birth version at all the nodes visited by the transaction.

Switch the version period (Refresh):

```
do the same as in the nondistributed case
  except unlatching versionblock at the end;
for each transaction in PL
  if the transaction version = the new current
      version then
    /* i.e., this was a future transaction
       and now it is current */
    remove from UL;
    if transaction is committed then
      wake up its process;
       /* it will release its locks & return */
     endif
   endif
 endfor
take a checkpoint and log NSULs and PL
unlatch versionblock;
```

Restart Recovery: In the nondistributed case, the logic followed during recovery is quite simple. The NSULs and UL are initialized to zeroes, the transaction number counter is assigned the highest value seen in the log, and cur_v and oldest_v are initialized to 1. In the distributed case, we need to checkpoint Cur_v, NSULs and PL during a version switch operation. During the analysis pass of recovery, these checkpointed NSUL and PL lists are brought up to date as of the end of the log (see [MHLPS92] for a detailed description how this is done for bringing a similar data structure up to date). At the end of restart recovery, UL is made to have a 1 for each transaction in PL which has a future version and a 0 for the other transactions. Oldest_v and Cur_v are initialized to the value of Cur_v found in the last checkpoint record before system failure. The transaction number counter needs to be initialized carefully since, with Presumed Abort, [MBCS91, MoLO86] even though remote nodes know about a transaction the origin node of the transaction may not have any log records. Under those conditions, it would be incorrect to initialize the transaction number counter to be just one higher than the highest value found in the log records of transactions of the node's log. In R*, we made sure that we added a constant K to the highest value found in the log and initialized the counter with that sum. At checkpoint time, the value of the counter was logged and a checkpoint was taken if close to K transactions have begun since the last checkpoint.

5. Related Work

Other approaches for supporting transient versioning are presented in [AgSe89, BaHR80, BoCa92, CFLN82, ChGr85, Reed78, StRo81, Weih87]. None of these methods deal with versioning index data. [BaHR80] uses a complex mechanism. It requires the continuous and costly maintenance of a graph which tracks the dependencies among transactions to avoid nonserializable executions of transactions (of course, this graph turns out to be useful for detecting deadlocks also). Even read-only transactions are required to incur the overhead of acquiring locks on the objects that they read (of course, readers will be granted those locks without waiting) and the cost of analyzing the dependency graph to determine which version of an object should be read. Sometimes non-read-only transactions might be rolled back to preserve consistency of data. Read-only transactions are never rolled back. Since only at the most 2 versions of any data item are maintained, sometimes an updater will be delayed from committing by a read-only transaction that is reading the prior version of a record updated by the former. This approach also incurs additional lock related overhead for update transactions, compared to single version schemes. [BaHR80] does not discuss space management, the structures used to keep track of locations of different versions of an object, partial rollbacks, incremental versioning, etc.

[Reed78] uses timestamps to do synchronization. It requires even read-only transactions to perform updating of control information (timestamp) associated with data items. It allows any number of versions of an object to be created, thereby potentially causing space management problems. The garbage collection problem is not addressed. Reads may be delayed and update transactions may be aborted to avoid serializability violations.

[StRo81] may block or abort readers under some circumstances and may delay update transactions from committing until readers of previous versions of objects terminate.

[CFLN82] allows any number of versions of an object to be created, thereby potentially causing space management problems. It does versioning at the page level, thus necessitating copying of an entire page (to a slot in the "version pool") even if only a small part of it got changed. In addition to the pathlength overhead, this results in unnecessary wastage of buffer and disk space also. Furthermore, this guarantees that if a read-only transaction does a read of a logical page which has an uncommitted version then that transaction would have to look at at least one additional page before it can find the version of the page that it should read (this could cause an extra I/O). This happens because the different versions of a page are chained and each version is identified by the ID of the transaction which created it, and the read-only transaction is required to read the most recent version of the page that was created by a transaction that had committed by the time the read-only transaction had started its execution. For this reason, every time a read-only transaction starts a committed transaction list (CTL) is associated with it. The page level versioning using a "version pool" also guarantees that clustered access to physically contiguous pages for read-only transactions, especially long ones, is not possible. The way garbage collection is performed may cause wastage of space in the "version pool". Every update transaction is required to keep track of the slots that it has used in the "version pool".

As it should be clear, the method of [CFLN82] supports only page level locking. Page level locking even for index data would lead to an intolerable level of concurrency [MoLe92]. That method requires all modifications made by a transaction to be forced to disk at commit time. Since versioning is being done at the page level and since before images of modified records are not logged, before a modified version of a page with uncommitted changes is put on disk (due to a buffer "steal") the previous version of that page, which will be in the "version pool", must be forced to disk. This is a costly operation. Recently, major improvements to the above scheme were presented in [BoCa92].

[ChGr85] extends the scheme of [CFLN82] to the case of distributed read-only transactions. This algorithm causes the CTL of a given site to be transmitted in all the prepare and commit messages sent by that site. This means that read votes cannot be used to avoid the second phase of commit processing as in the efficient commit protocols of [MBCS92, MoLO86]. Sites that receive CTLs from other sites merge them with their own to create new versions of their CTLs. If one does not want a read-only transaction to be aborted due to the premature garbage collection of the data it needs at a remote site, the set of sites that the read-only transaction will visit needs to be known in advance and each of those sites needs to be queried first to get its CTL before the read-only transaction starts. The union of the received CTLs is transmitted to all the retrieval sites and is used by the transaction to determine the versions of data it should read. Once the retrieval sites communicate their CTLs they are prevented from garbage collecting the snapshot of the data defined by the CTLs. The algorithm has the overhead of CTLs having to be maintained on stable storage. CTLs could be pretty big. The paper does not say how premature garbage collection can even be detected.

The methods proposed by Weihl in [Weih87] are quite complex and they make read-only transactions become updaters since some information about those transactions is remembered at the objects that they read. They are expensive also because they require that update transactions revisit at commit time all the objects that they *accessed* in order to assign them the timestamps of the modifying transactions or to store some information about read-only transactions. For distributed transactions, this requirement rules out the important read-only optimizations of commit protocols like Presumed Abort and the releasing of read locks during the prepare phase (phase 1). Even as a read-only transaction begins its execution, one of the methods requires that the list of all the objects that the transaction might access be known. These objects must be visited at the beginning, during the execution and again at the end of the read-only transaction!

The method that probably comes closest to ours is the one described in [AgSe89]. An update transaction is assigned as its version number (and the transaction number) a unique number which represents its position in the serialization order. A read-only transaction is assigned, as its version number, a number such that *all* update transactions with version numbers *less than* that number have terminated. When it reads, it reads that version of an object which has the *highest* version number that is *less than* the query's version number. There are some significant differences between that method and our method. That method requires deferred updating since all versions created by a transaction are identified with the transaction's version number and the transaction does not get its number (for the locking approach) until it terminates. It does not provide user control over the degree of versioning. Nor does it have the concept of version periods and user control over the switching of version periods. No descriptions have been given about the unit of locking, the chaining of versions of an object, and garbage collection of unwanted versions.

Prime and Oracle are reported to have implemented, in their commercially available products, support for transient versioning. The details of the employed methods have not been published to our knowledge. A very brief description of a record-level transient versioning scheme employed by DEC is presented in [RaRe91].

6. Summary and Conclusions

Our method for *transient* versioning is characterized by its flexibility and its efficiency. The following items amplify these characteristics.

1. Read-only transactions that do not mind reading an old version of data do not acquire any locks, thereby not encountering any waits caused by concurrent updaters.
2. Distributed data access without locking is also supported efficiently.
3. Read-only transactions do not update any control information (e.g., timestamps) associated with data items read by them, unlike in some other multiversion schemes.
4. Read-only transactions are never forcibly rolled back by the system, except possibly in the distributed case in order to avoid long delays or because a read-only transaction's version is too old.
5. Record level locking by update transactions is supported.
6. Recovery can be performed by using shadow pages and logs, as in System R [GMBL81], or write-ahead logging, as in ARIES [MHLPS92].

7. In-place updating of data in the buffer pool and on disk is possible even before the end of a transaction. That is, there is no need for deferred updating or the use of the *no-steal* buffer management policy.
8. There is no need to force to disk all modified pages at commit time.
9. Partial rollbacks (i.e., intermediate savepoints) are also supported.
10. No transaction is required to predeclare the set of data items that it will read or write.
11. If at all non-read-only transactions are forcibly rolled back in the nondistributed case it will only be due to deadlocks and not due to the support of multiversions by the system. In the distributed case, a rollback could happen if the origin node were to assign a version number that is less than the current version in a participant node.
12. The maximum number of versions maintained by the system can be configured autonomously by the different nodes. In fact, the value for a given node could be changed even dynamically (for space reasons, we did not describe here how this is done).
13. Version switching can happen even when update transactions are in execution.
14. Incremental versioning can be performed to reduce increases in storage used due to versioning.
15. Support for record level and incremental versioning, and the tunability of the maximum number of versions should enable our method to better preserve intra- and inter-page clustering amongst related records compared to the other methods, especially the ones that support only page level versioning.
16. Reduction in log volume is possible since before image of updated fields (or of complete record during a deletion) need not be logged, except when the same record is being updated more than once by a transaction.
17. Asynchronous garbage collection can be performed by one or more background processes.
18. There is no need to assign timestamps to update transactions when they begin. The serialization order amongst conflicting such transactions is determined dynamically.
19. Transient versioning is supported for indexes also. This, in turn, allows next key locking to be avoided during key delete operations.
20. For Tu transactions, high concurrency approaches like cursor stability (degree 2 consistency of System R) and locking overhead reducing techniques like Commit_LSN [Moha90b] are still applicable.

Since a transaction ID is associated with each version of a record, the method does not permit a single record to contain the uncommitted updates of multiple transactions. Situations like the latter would arise if semantically-rich modes of locking (e.g., increment/decrement type locks) were to be used in the DBMS to increase concurrency even further by taking advantage of the commutativity properties of transactions' operations. This is a topic for future research.

7. References

AgSe89 Agrawal, D., Sengupta, S. *Modular Synchronization in Multiversion Databases: Version Control and Concurrency Control*, **Proc. SIGMOD International Conference on Management of Data**, Portland, May 1989.

BaHR80 Bayer, R., Heller, H., and Reiser, A. *Parallelism and Recovery in Data base Systems*, **Transactions on Database Systems**, Vol. 5, No. 2, June 1980.

BoCa92 Bober, P., Carey, M. *On Mixing Queries and Transactions Via Multiversion Locking*, **Proc. 8th International Conference on Data Engineering**, Tempe, February 1992.

ChGr85 Chan, A., Gray, R. *Implementing Distributed Read-Only Transactions*, **IEEE Transactions on Software Engineering**, Vol. SE-11, No. 2, February 1985.

CFLN82 Chan, A., Fox, S., Lin, W-T., Nori, A., and Ries, D. *The Implementation of An Integrated Concurrency Control and Recovery Scheme*, **Proc. SIGMOD International Conference on Management of Data**, Orlando, June 1982.

GMBL81 Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., Traiger, I. *The Recovery Manager of the System R Database Manager*, **ACM Computing Surveys**, Vol. 13, No. 2, June 1981.

MBCS91 Mohan, C., Britton, K., Citron, A., Samaras, G. *Generalized Presumed Abort: Marrying Presumed Abort and SNA's LU 6.2 Commit Protocols*, **IBM Research Report**, IBM Almaden Research Center, November 1991.

MHLPS92 Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P. *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*, **ACM Transactions on Database Systems**, Vol. 17, No. 1, March 1992. Also available as **IBM Research Report RJ6649**, IBM Almaden Research Center, January 1989.

Moha90a Mohan, C. *ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990. A different version of this paper is available as **IBM Research Report RJ7008**, IBM Almaden Research Center, September 1989.

Moha90b Mohan, C. *Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990. Also available as **IBM Research Report RJ7344**, IBM Almaden Research Center, February 1990.

MoLe92 Mohan, C., Levine, F. *ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging*, **Proc. SIGMOD International Conference on Management of Data**, San Diego, June 1992. A longer version is available as **IBM Research Report RJ6846**, IBM Almaden Research Center, August 1989.

MoLO86 Mohan, C., Lindsay, B., Obermarck, R. *Transaction Management in the R* Distributed Data Base Management System*, **ACM Transactions on Database Systems**, December 1986.

PMCLS90 Pirahesh, H., Mohan, C., Cheng, J., Liu, T.S., Selinger, P. *Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches*, **Proc. 2nd International Symposium on Databases in Parallel and Distributed Systems**, Dublin, July 1990. A longer version is available as **IBM Research Report RJ7724**, IBM Almaden Research Center, October 1990.

RaRe91 Raghavan, A., Rengarajan, T.K. *Database Availability for Transaction Processing*, **Digital Technical Journal**, Vol. 3, No. 1, Winter 1991.

Reed78 Reed, D. *Naming and Synchronization in a Decentralized Computer System*, **PhD Thesis**, Technical Report MIT/LCS/TR-205, MIT, September 1978.

StRo81 Stearns, R.E., Rosenkrantz, D.J. *Distributed Database Concurrency Controls Using Before-Values*, **Proc. SIGMOD International Conference on Management of Data**, Ann Arbor, April 1981.

Weih87 Weihl, W. *Distributed Version Management for Read-Only Actions*, **IEEE Transactions on Software Engineering**, Vol. SE-13, No. 1, January 1987.