An open and safe nested transaction model: concurrency and recovery

by S Madria, S Maheshwari, B Chandra, B Bhargava
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

ELSEVIER

The Journal of Systems and Software 55 (2000) 151–165

The Journal of Systems and Software

www.elsevier.com/locate/jss

An open and safe nested transaction model: concurrency and recovery

Sanjay Kumar Madria [a,*], S.N. Maheshwari [b], B. Chandra [c], Bharat Bhargava [a]

[a] Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA
[b] Department of Computer Science and Engineering, Indian Institute of Technology, Hauz Khas, New Delhi, India
[c] Department of Mathematics, Indian Institute of Technology, Hauz Khas, New Delhi, India

Received 1 August 1999; received in revised form 1 November 1999; accepted 8 December 1999

Abstract

In this paper, we present an open and safe nested transaction model. We discuss the concurrency control and recovery algorithms for our model. Our nested transaction model uses the notion of a recovery point subtransaction in the nested transaction tree. It incorporates a prewrite operation before each write operation to increase the potential concurrency. Our transaction model is termed "open and safe" as prewrites allow early reads (before writes are performed on disk) without cascading aborts. The systems restart and buffer management operations are also modeled as nested transactions to exploit possible concurrency during restart. The concurrency control algorithm proposed for database operations is also used to control concurrent recovery operations. We have given a snapshot of complete transaction processing, data structures involved and, building the restart state in case of crash recovery. © 2000 Elsevier Science Inc. All rights reserved.

1. Introduction

1.1. Overview of nested transaction models and recovery algorithms

1.1.1. Close nested transaction model

In close nested transaction model (Moss, 1985), a subtransaction may contain operations to be performed concurrently, or operations that may be aborted independent of their invoking transaction. Such operations are considered as subtransactions of the original transaction. This parent–child relationship defines a nested transaction tree and transactions are termed as nested transactions (Moss, 1985). Failure of subtransactions may result in the invocation of alternate subtransactions that could replace the failed ones to accomplish the successful completion of the whole transaction. Each transaction has to acquire the respective lock before accessing a data object. A subtransaction's effect cannot be seen outside its parent's view (hence, called closed). A child transaction has access to the data locked by its parent. When a transaction writes a data object, a new version of the object is created. This version of the object is stored in volatile memory. When the subtransaction commits, the updated versions of the object are passed to its parent. If the transaction aborts, the new version of the object is discarded. Parent commits only after all its children are terminated. When the top-level transaction commits, the current version of each object is saved on stable storage.

In the closed nested transaction model, the availability is restricted as the scope of each subtransaction is restricted to its parent only. This forces a subtransaction to pass all its locks and versions of data objects updated to its parent on commit. The effect of a committed subtransaction is made permanent only when the top-level transaction commits. In many applications, it is unacceptable that the work of a longlived transaction is completely undone by using either of the above techniques in case the transaction eventually fails at finishing stage. The current strategy forces short-lived transactions to wait before they acquire locks until the top-level transactions commit and release their locks. Therefore, the model is not appropriate for the system that consists of long and short transactions.

[*] Corresponding author. Present address: Department of Computer Science, University of Missouri, Rolla, MO 65405, USA.

E-mail addresses: madrias@umr.edu, skm@cs.purdue.edu (S.K. Madria), snm@cse.iitd.ernet.in (S.N. Maheshwari), bchandra@maths.iitd.ernet.in (B. Chandra), bb@cs.purdue.edu (B. Bhargava).

0164-1212/00/$ - see front matter © 2000 Elsevier Science Inc. All rights reserved.
PII: S 0 1 6 4 - 1 2 1 2 (0 0) 0 0 0 6 7 - 4

1.1.2. Open nested transaction model

To exploit layer specific semantics at each level of operation nesting, Weikum presented a multilevel transaction model (Weikum, 1991; Weikum et al., 1990). The model provides non-strict execution by taking into account the commutative properties of the semantics of operations at each level of data abstraction, which achieves a higher degree of concurrency. A subtransaction is allowed to release locks before the commit of higher level transactions. The leaf level locks are released early only if the semantics of the operations are known and the corresponding compensatory actions defined. When a high level transaction aborts, its effect is undone by executing an inverse action which compensates the completed transaction. Recovery from system crashes is provided by executing undo actions at the upper levels and redo actions at the leaf level. Each level is provided with a level specific recovery mechanism. This model has also been studied in the framework of object oriented databases in Muth et al. (1993) and Resende et al. (1994).

In many applications, the semantics of transactions may not be known and hence, it is difficult to provide non-strict executions. In real time situations, there are other classes of operations that cannot be compensated. These are the operations that have an irreversible external effect, such as handing over huge amounts of money at an automatic teller machine. Such operations have to be deferred until top-level commits, which restricts availability (i.e., increases response time).

1.1.3. Nested transaction recovery algorithms

The intentions-list and undo-logging recovery algorithms given in Fekete et al. (1993) handle recovery from transaction aborts in the nested transaction environment by exploiting the commutative properties of the operations. The intentions-list algorithm works by maintaining a list of operations for each transaction. When a transaction commits, its list is appended to its parent; when it aborts, the intentions-list is discarded. When the top level transaction commits, its intentions-list is transferred to the log. This scheme provides recovery from transaction aborts only and does not handle system crashes. To increase concurrency during undo logging recovery, scheme allows some non-strict executions. It allows a transaction to share the uncommitted updates made by other transactions by exploiting commutativity of operations. On execution of an operation, the data object records change their states and the new state is transferred to the log. When a transaction aborts, in contrast to intentions-list algorithm, all operations executed by its descendants on the object are undone from its current state and are also subsequently removed from the log. This algorithm does not take care of recovery from system crashes.

In both intentions-list and undo-logging algorithms, an incomplete transaction is allowed to make uncommitted updates available to those transactions that perform a commutative operation. However, this is restricted to transactions at the same level of abstraction. This limits availability. In both algorithms, all the work done by descendent transactions are discarded in case of aborts at higher levels. This may not be possible or desirable in many real time applications. In undo-logging algorithm, when a transaction aborts, in contrast to intentions-list algorithm, all operations executed by its descendants on the object are undone from its current state and are subsequently removed from the log. In both intentions-list and undo-logging algorithms, an incomplete transaction is allowed to make uncommitted updates visible to those transactions that perform a commutative operation. This is restricted to the transactions at the same level of abstraction.

The above two recovery models consider semantics of operations at leaf level only. System R (Gray et al., 1981) exploits layer specific semantics but restricted to two level of transaction nesting. In System R, to perform recovery, updates are undone by performing inverse tuple-level operations. For this purpose, System R records tuple updates on a log. To recover from a system crash, before applying any tuple level log record, the database must first be restored to some tuple-level consistent state. In other words, a low-level recover mechanism is necessary to make tuple actions appear atomic.

In Moss (1987), a crash recovery technique similar to shadow page has been suggested in nested transaction environment based on undo/redo log methods. In terms of logging, both undo/redo logs are used. Mohan et al. (1992, 1989) has also discussed "write ahead logging" based crash recovery algorithm using conventional nested transaction model. This undo/redo type of recovery model exploits semantics of nested transactions. The actions of a transaction undone during previous abort have not been undone again in case of one more failure. This is an advantage over Weikum's multilevel recovery algorithm by which requires undo actions to be undone again in case of one more failure.

1.2. Our contributions

In this paper, we introduce an open and safe nested transaction model in the environment of normal read and write operations to remove the deficiencies stated above and to further improve availability and provide efficient crash recovery. Our model supports inter- and intra-transaction concurrency. We assume that semantics of transactions at various levels of nesting are not known. There are two basic motivations behind our model. First, it is desirable that long-lived transactions should be able to release their locks before top-level transactions commit. Second, it may not be desirable or possible to undo or compensate the effects of one or more of the important committed descendants after the failure of a higher level transaction due to abort or a system crash. We introduce the concept of a "recovery point subtransaction" of a top-level transaction in a nested transaction tree. It is essentially a subtransaction after the commit of which its ancestors are not allowed to rollback. In other words, once the recovery point subtransaction of a top-level transaction has committed, all its superior transactions are forced to commit. In case it aborts, its ancestors can choose an alternate path to complete their execution. Our nested transaction model uses a prewrite operation before an actual write operation to increase the concurrency. The nested transaction tree of our model consists of database operations, system recovery operations (such as analysis and redo operations) and buffer management operations specified as nested transactions. The read, prewrite and writes operations are modeled at leaf levels in the transaction hierarchy. The recovery operations are specified in terms of nested transactions to achieve higher concurrency during system restart. Our locking algorithm controls the execution of both the normal operations as well as recovery operations. We also discuss the data structures required for the implementation of the recovery algorithm. We have discussed a snapshot of the concurrency and recovery algorithm with the help of an example. A brief overview of our crash recovery algorithm has appeared in Madria et al. (1997c). The correctness of the concurrency control algorithm using I/O automation model (FLMW) has reported in Madria et al. (1997b).

The rest of the paper is organized as follows. In Section 2, we present motivating examples and the overview of our nested transaction and recovery model. In Section 3, we discuss nested transaction system model and implementation. Section 4 presents system restart operations. We present a snapshot of transaction processing, logging and recovery in Section 5. We conclude in Section 6.

2. Open and safe nested transaction model and recovery algorithm

In this section, we motivate the readers about the open and safe nested transaction model with some examples and provide an overview of our model and recovery algorithm.

Motivating examples: Consider a part of the nested transaction tree for fund transfer operation from a group of accounts to another account. In the transaction tree, let T_s be a transaction on whose behalf T_{s1} invokes various subtransactions to collect (access) funds from different accounts. Once T_{s1} is committed, T_s invokes T_{s2} to finally credit the funds collected into another account. Suppose after a subtransaction T_w has withdrawn all the amount, T_s commits. If any transaction situated above T_s aborts then it is desirable for the transaction to complete successfully on transaction revival. This is because it is not possible to undo or correct the failed transaction's action by some compensatory actions. Another possibility is to delay the actual commit of T_s until its top-level transaction commits which restricts availability. For example, a balance transaction has to wait until the commit of the top-level transaction.

Consider another scenario where in the nested transaction tree, a subtransaction determines the success (commit) or failure (abort) of the top level transaction. Suppose this nested transaction tree models various activities (modeled as subtransactions here) related to a business travel. Some of the activities are very crucial in determining the completion of the top level activity. For example, the commit of "fund" and "visa" subtransactions will determine whether the travel transaction will commit or not. That is, these subtransactions commit will determine the commitment of the top level transaction no matter what may be the fate of other subtransactions in the transaction tree. Note that once, the fund and visa subtransaction commits, its upper level (sub)transactions will be forced to commit (some delay or restart may involve).

Salient features of our model: Our model allows some particular subtransactions to release their locks before their ancestor transactions commit. This allows the other subtransactions to acquire required locks earlier. Our nested transaction model can handle the situations where a committed lower level subtransaction's effect cannot be undone or compensated in case of a higher level transaction's failure. A transaction's semantics may be such that beyond a certain point, it cannot rollback entirely or its effect should not be lost. We achieve this by introducing the concept of "recovery point subtransaction" of a top-level transaction in a nested transaction tree. It is essentially a subtransaction after whose commitment, its ancestors are not allowed to rollback. In case a superior transaction aborts or the system fails after the commit of its recovery point subtransaction, the failed transaction has to complete on system revival. Such a transaction execution permits a recovery point subtransaction to reveal its result to other transactions at any level of nesting before its superior transactions commit. A recovery point subtransaction's effect is made durable before its top-level transaction's commit. This results in the relaxation of the isolation property (Harder and Reuter, 1983) of the transaction.

To avoid undo actions and the consequent cascading aborts and to increase the availability, we assume that each transaction issues a prewrite operation (Madria, 1995; Madria et al., 1999; Madria and Bhargava, 1997a) before a write for the object it intends to write. Each prewrite operation contains the value that a user-visible transaction wants to write and precedes the associated final write. A prewrite operation actually does not change a data object's state but only announces the value the data object will have after the associated write is performed. The advantage of prewrite is that a read operation of another transaction can get the value before a data object's state is updated on stable storage and hence, results in increasing the availability of new data values. Prewrite operations are particularly helpful in the engineering design applications (Kim et al., 1984), CAD (Korth et al., 1990), large software design projects (Korth and Speegle, 1990) etc. where transactions are long. A subtransaction that initiates different prewrite access subtransactions at leaf level for different data objects is defined to be the recovery point subtransaction. These announced prewrite values are made visible to other subtransactions after the commit of recovery point subtransaction. The prewrite subtransactions release their locks before their ancestors commit. Discarding some of the prewrites before the commit of the recovery point subtransaction will not introduce cascading aborts (hence safe) since the prewrite values are made visible only after the commit of the recovery point subtransaction.

2.1. Crash recovery algorithm

2.1.1. Basic goals of system crash recovery algorithm

- Revive the database state of those data objects which do not contain their last committed values with respect to the execution up to the system failure.
- Revive the prewrite values (kept in prewrite-buffers) of the data objects which have been announced by the committed recovery point subtransaction before system failure.
- To identify such data objects, the dirty object table has to be revived. The dirty object table is used to keep track of those data objects whose finally written values are inconsistent with the stable database values. This table also keeps information about those data objects whose prewrite values, announced by the committed recovery point subtransactions, have not been subsequently written on the database before a system crash.
- A system crash creates an additional problem of accomplishing the completion of those top-level transactions whose recovery point subtransactions have been committed before system crash. They have to reacquire the locks held by them at the time of failure before new transactions acquire such locks.
- To handle above, the transaction and lock tables have to be revived. The transaction table keeps a list of all active transactions in the system at any time. The revived transaction table will recognize those active top-level transactions (and their active descendants) whose recovery point subtransactions have been committed before failures. The lock table contains the type of locks held by the transactions on different data objects at any time. The revived lock table will help in reacquiring the locks held by active top-level transactions and their descendants at the time of failure.
- To initiate new top-level transactions as soon as the dirty object, transaction, and lock tables and consistent states of prewrite- and write-buffers of dirty data objects are re-established.

2.1.2. System crash recovery steps

To achieve above recovery goals, we need the following steps to restart the system:

Revival of dirty object table: The dirty object table is required to be checkpointed periodically by transferring a copy of it to the stable storage during normal processing. The prewrite values and after-images are logged on stable storage during the execution of transactions to build the consistent dirty object table in case a system failure occurs before the next checkpoint is taken. A transaction is not permitted to complete its commit processing until the redo portion of that transaction has been written to stable storage. The redo portion of a log record provides information on how to redo changes performed by the committed transactions. During system restart, the dirty object table is recovered with the help of most recent checkpointed copy of the dirty object table and is modified with the help of log stored after the last checkpoint.

Revival of transaction and lock tables: The transaction and lock tables are checkpointed by transferring a copy of each of them to the stable storage periodically during normal processing. Whenever a subtransaction is made active or when any transaction acquires or releases a lock, the information is also logged to build a consistent state of these tables. However, such information may not be logged for read-only and prewrite access subtransactions as these are to be discarded in case of a system failure. If a checkpoint is taken during restart recovery then the contents of transaction and lock tables will also be included in checkpoint. The entries corresponding to all other transactions except those that are to be restarted (whose recovery point subtransactions have not been committed) are removed from the transaction and lock tables. To do so, we need to find whether the stable storage contains the commit-state of the recovery point subtransaction of each active top-level transaction. The commit states information is transferred to the stable storage during the commit of those subtransactions whose effects cannot be undone or lost in case of a failure. The commit-state information contains, besides associated variables, private data and other information, the identifier of the committed subtransaction as well as of its parent transaction.

A commit-state information of a subtransaction $T1$ defines the state of its parent transaction $T2$ at the time of commit of $T1$. The commit-state information helps in re-establishing the restart state of a top-level transaction in order to complete its remaining execution. No subtransaction whose effects cannot be undone in case of failure can be considered complete until its commit-state information and all its data are safely recorded on stable storage. If the stable storage does not contain the commit-state of the recovery point subtransaction of an active top-level transaction then all the entries corresponding to it and all its subtransactions are removed from the table. Otherwise, the top-level transaction has to complete its remaining execution on revival.

Revival of buffers: To revive the contents of write-buffer of a dirty data object, we copy the value of the data object from the stable-db to the write-buffer. However, the stable database version of the data object may not contain some or all the updates of committed transactions. It involves redoing those committed transactions after-images, which have not been transferred to the stable-db before the failure. The redo of after-images will re-establish the state of the database in the write-buffer at the time of failure. Similarly, to recover the prewrite-buffers corresponding to dirty data objects, we redo the prewrite values logged on stable storage after the last checkpoint, which have not subsequently been written. This re-establishes the states of prewrite-buffers of dirty data objects as they exist at the time of failure. The contents of prewrite- and write-buffers are recovered using the non-volatile storage version of the database, dirty object table and the log. There is at the most one prewrite log corresponding to a data object since once the associated write values are written, the corresponding prewrite log entry is removed from the stable storage as well as from the dirty object table.

Transaction logging and recovery: The log records written on behalf of subtransactions are always linked to the last record of their parents which reflects the transaction tree in the log. Whenever a prewrite access subtransaction commits, its commit information and prewrite values are passed to its parent transaction, which is the recovery point subtransaction. When the recovery point subtransaction decides to commit, its commit-state information and the associated prewrite values are required to be logged. This commit-state informs the scheduler that from this point of time, the committed subtransaction's effect cannot be lost under any circumstance. The commit of recovery point subtransaction occurs only after all its prewrite access subtransactions have been committed and therefore, the recovery point subtransaction's commit-state has the effect of all its committed prewrite access descendants. Hence, the commit-state of recovery point subtransaction is the first commit entry in the stable log. If system crashes immediately after the commit of its recovery point subtransaction, the scheduler will restart its active top-level transaction from the logged commit-state onwards.

Whenever a write access subtransaction at leaf level decides to commit, its commit-state and write value are logged. The commit information is passed to its parent transaction that helps in the termination of the parent transaction. The process of logging the commit-state will continue till the top-level transaction commits. In case of failure, these logs will help in completing an active top-level transaction on revival. The process of transferring a transaction's commit-state or prewrite or write values to the stable log is called transaction checkpointing (early writing). It is required in order to keep track of commit-states, prewrite and write values of various subtransactions, which helps in completing an active top-level transaction's re-execution. Transaction checkpointing will keep track of all logical committed states as well as prewrite and write values, which cannot be lost. Readonly transactions require no early writing as they do not change a data object's state.

To complete the active top-level transactions, whose recovery point subtransactions have been committed before system failure, the scheduler has to decide the restart states to re-initiate such transactions. If the recovery point subtransaction's commit is the only commit record in the stable log then its active top-level transaction restarts from this commit-state. Otherwise, the scheduler finds out the last commit-state logged after the commit of recovery point subtransaction prior to system crash in order to restart the transaction from the last commit-state. Once the restart-state is established, the scheduler reacquires the type of locks the active top-level transaction and all its active descendants were holding at the time of failure. Once the locks are reacquired, the execution of a top-level transaction restarts from the restart-state.

2.2. Data structures

Here we discuss the data structures used in the logical implementation of the recovery model. Most of these data structures are needed in the physical implementation as well. We first discuss some of the fields present in different types of log records, which are as follows.

LSN: This gives the address of the log record in the log address space. It is a monotonically increasing value. It is present in log records of the type "data". This may be included in other type of log records also but is not mandatory.

Transaction-id: Identifier of the transaction involved in the log record.

Object-id: Identifier of the object involved in the log record. It is present in log records of "data" and "lock" types.

Value: This is the redo data that describes the update that was performed. This also includes the committed prewrite value.

Active: Present in the log record written during the activation of a transaction.

Commit-state: Present in the log record written during the commit of a subtransaction. This includes the private data, local variables, etc. of the committed subtransaction.

Lock: This is present in the log record which is logged when a subtransaction acquires or releases any lock. This includes information whether the lock is "retained" or "held" by the transaction.

Log records can be of the following types:

"Data" type of log records of the form: ⟨**LSN**, **transaction-id**, **object-id**, **Prewrite or write value**⟩

"Transaction" type of log records are the form ⟨transaction-id, status⟩. Note that status takes either the value "Active" or "Commit".

"Lock" type is of the form ⟨**transaction-id**, **lock type**, **object-id**⟩

We refer the complete log record structure by ⟨log record⟩ along with its type information.

⟨**END-CHK-POINT**⟩ is a record to identify the end of a checkpointing activity.

2.2.1. Transaction and lock tables

To distinguish between different transactions and to know their status (active or not) in the system, we need to maintain the transaction-id and status of each transaction using a transaction table. Furthermore, to reflect the transaction tree, each transaction-id is such that it contains the identifier of its own as well as the identifier of its parent transaction. This type of transaction-id helps the scheduler in informing the commit or abort of a subtransaction to its parent. In addition, an abort request also contains identifiers of all its inferiors.

For the scheduler to know whether a transaction is active or not, it is sufficient that a transaction may keep only one status namely "active". Since the parent–child relationship of committed subtransactions are to be stored in the log separately by linking the log records of committed subtransactions to their parents, the transaction table need not keep information about committed subtransactions. A subtransaction enters the "active" status as soon as it is initiated and remains "active" until it commits or aborts. When a subtransaction commits, its entry is removed from the transaction table. After the commit of the recovery point subtransaction, the status of its upper level subtransactions remains "active" even in case of aborts at higher level because they have to complete their remaining execution on revival. On system revival, once such "active" top-level transactions are decided, all other "active" top-level transactions and their "active" descendants are removed from the table.

The lock table keeps the information about the locks held by all active transactions in the system at any time. Each entry of the table keeps information about the transaction-id, type of lock held and the object-id. When a transaction acquires a lock on a data object, an entry is made in the lock table. When a transaction commits or aborts, the corresponding transaction entry is removed from the table. A new entry is made about the transaction which inherits the lock from the committed or aborted transaction.

2.2.2. Dirty data object table

Each entry in the dirty data object table consists of field's object-id, RecLSN (Recovery log sequence number) of prewrite and write operations. The value of RecLSN of write operation indicates in the log there may be updates which are, possibly, not yet in the non-volatile version of the data objects. The minimum of RecLSN values in the table gives the starting point for redo activity. All the write log records whose LSNs are greater than the min RecLSN are, possibly, required to be redone as these log records effects might not have been transferred to the stable-db. The min RecLSN of all prewrite operations which is greater than the max RecLSN of all write operations gives the starting point for redoing the prewrite operations. All the prewrite log records whose LSNs are greater than the min RecLSN are required to be redone as these are the prewrite log records whose associated write operations are not performed. Therefore, these prewrite log records need to be redone. Whenever the write values are announced, the corresponding prewrite entries from the dirty object table are removed. Similarly, whenever the data objects are written back to the non-volatile storage, the corresponding entries are removed from the table.

3. Nested transaction system model and implementation

Our nested transaction database system model formally consists of transaction managers (TMs), recovery managers (RMs) and data managers (DMs). The data managers (DMs) model the data objects. Each data manager keeps a copy of the data object in the secondary storage, called stable-db. The prewrite and write values of each object are kept in the respective buffers at the corresponding DMs. These are called prewrite- and write-buffers, respectively. Physically, only a subset of these DMs will have prewrite and write values of the data objects in the corresponding buffers. A read operation gets the value of the referenced data object from the prewrite-buffer (if any) otherwise it gets the value from the write-buffer. If the DM does not have a copy of the data object in the write-buffer, the read operation gets the value from the stable-db copy of the data object. The write-buffer's contents of a data object are transferred periodically to stable storage. Also, each DM maintains a log corresponding to the data object. Each DM also shares a common log.

Considering the above configuration, our model has four different transaction managers (TMs) for performing read (read-TM), write (write-TM) and system restart's analysis (analysis-TM) and redo (redo-TM) operations. Of these, read- and write-TMs are initiated by the user-visible transactions. An external agent such as the operating system invokes analysis-TMs and redo-TMs. A hidden daemon transaction is associated with each write- and redo-TM transaction to coordinate the buffer management operations, i.e., the transfer of a data object's value to stable-db during the normal and system restart operations. To achieve the notion of spontaneity and transparency of buffer management operation, the daemon transaction wakes up and commits with respect to its associated transaction.

TMs are situated at one level below user-visible transactions. Next level of transaction hierarchy has six different recovery managers for co-ordinating read (read-RM), prewrite (prewrite-RM), write (write-RM), transfer (transfer-RM) and system restart's analysis (analysis-RM) and redo (redo-RM) operations. These RMs are made active by the corresponding TMs. During the span of daemon transaction, a daemon can initiate many transfer-RMs. This will help the transfer process to be made active during redo operations. These RMs initiate access subtransactions situated at the leaf level. Each read, prewrite and write-RM initiates read, prewrite and write access subtransactions, respectively. A read access reads the value either from the prewrite or the write-buffer of the data object whereas a write subtransaction accesses only the write-buffer component of the data object. A transfer-RM initiates a transfer access (like read access but returns no value) to transfer the contents of write-buffer of the data object to the stable-db. The analysis-RM initiates copy, read, write and read-analysis access subtransactions. A copy transaction re-initializes the tables in the volatile memory whereas read accesses read the log entries after the last checkpoint record and write accesses update tables to bring their state as existing at the time of failure. A read-analysis subtransaction returns information about the dirty objects at the time of system failure. It also returns a list of active transactions (and their restart states) to be restarted on system restart. A redo-RM initiates copy, read, prewrite and write access subtransactions. Here, a copy transaction places the stable-db copy of the object in its write-buffer. A read access initiated by redo-RM reads a log entry corresponding to a data object logged after the last checkpoint record. A prewrite (write) access makes a prewrite-buffer's (write-buffer's) value consistent. The nested transaction tree structure is shown in Fig. 1(a) (for normal operations) and Fig. 1(b) (for system restart operations).

We assume that each user transaction knows its write-set before initiating a write-TM to write all the

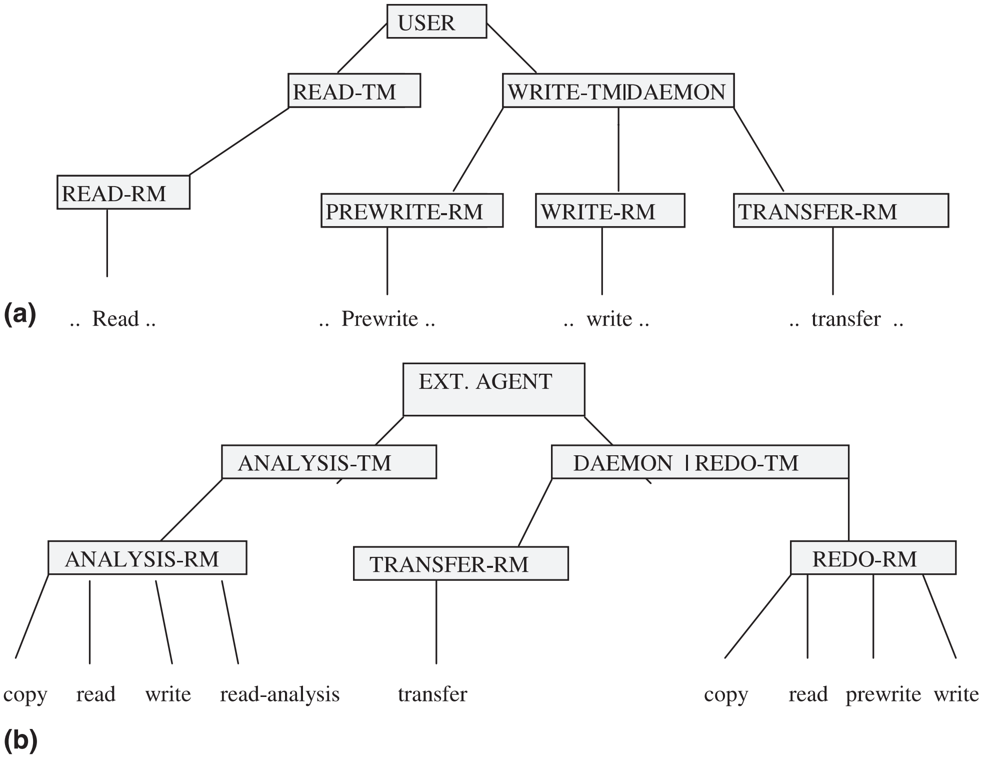

Fig. 1. (a) Nested transactions tree for normal operations and (b) nested transaction tree for system restart operations.

data objects. A write-TM first initiates a prewrite-RM, which further initiates prewrite access subtransactions in order to announce prewrites for all the data objects contained in the write-set. This value for each data object is written in the prewrite-buffer allocated in the volatile memory. Modeling prewrites at leaf level provides user transparency to the prewrite operations. We formally specify the prewrite-RM as the recovery point subtransaction of the top-level transaction. Once the prewrite-RM has committed, the prewrite values become visible outside its parent's view at any level of nesting without necessarily requiring the commit of all its superior transactions. After the prewrite-RM's commit, the write-TM initiates a write-RM to update all the data objects whose prewrite values have been announced before. The final updates are written in the write-buffers allocated in the volatile memory at each DM. With the invocation of each write-TM automaton, a daemon transaction is initiated automatically which further initiates transfer-RMs. A transfer-RM initiates a transfer access subtransaction to transfer the write-buffer's value to the stable-db. The write-buffer's contents can be transferred without the commit of the top-level transaction because write-values, once written, cannot be undone or lost.

To meet transaction and data recovery guarantees, the system maintains a log corresponding to each data object at the respective DM. The system maintains a common log (shared by all DMs) which keeps information about the progress of transactions and its associated data, their lock holding information etc. Our algorithm asserts that the log records representing changes to some data must already be on stable storage before the changed data is allowed to replace the previous version of that data on non-volatile storage. The log record corresponding to each data object is assigned a unique log sequence number (LSN) at the time the record is appended to the log. The LSNs are assigned in ascending order. Whenever a prewrite value is announced, the LSN of the prewrite log record to be written is placed in the LSN field of the prewrite value in the prewrite-buffer. Similarly, when the data object's value is updated in the write-buffer, the LSN of the log record is placed in the LSN field of the updated data object in the write-buffer. This value of LSN will be more than the value of LSN associated with the prewrite value of the same data object. The LSN with each prewrite and write value in the associated LSN field keeps track of the data object's state. Also, when a write-buffer's value is transferred to the stable-db, the after-images LSN is placed in the LSN field of the stable-db. This tagging of LSN allows precise tracking of the states of the object with respect to logged updates of the data object for system restart purpose. Each write-buffer is also associated with a stableLSN field. The stableLSN field is the LSN of the stable-db copy of the object. A write-buffer's value is transferred to the stable storage if the stableLSN of the write-buffer is greater than the write-buffer's LSN. This will avoid accessing the stable-db's LSN to check whether the transfer of the object's value to the stable storage is required or not.

3.1. Concurrency control algorithm

In this section, we discuss the type of conflicts which occur in our model during normal and recovery operations and the locks needed to control them. Note that we use the same concurrency control protocol to control the concurrent execution of recovery operations and database operations (see Fig. 2). The database operations are read, write and prewrite. The locks needed to control the concurrent executions are read-lock (RL) for read, write-lock (WL) and prewrite-lock (PL) for prewrite operations, respectively. During recovery operations, the access operations executed are copy, read, write, transfer, read-analysis and prewrite. The operations copy, read, transfer and read-analysis acquire read locks and hence, follows the read-locking protocols as in Fig. 2. Write and prewrite acquire write-lock and prewrite-locks, therefore, they also follow the locking protocols given below with respect to write-lock and prewrite-lock. A more detailed discussion on the correctness of concurrency control algorithm for database operations has appeared in Madria et al. (1997b).

Formally, we have the following locking algorithm as shown in Fig. 2.

4. System restart operations

System restart has to perform two passes of the log: analysis pass and the redo pass. After the analysis pass of log records, the transaction table will contain the list of transactions active at the time of failure, the lock table will have lock entries corresponding to active transactions, and the dirty object table will contain the list of data objects which were dirty at the time of failure. The redo activity is performed in second pass in order to restore the dirty data objects to the values consistent with the information kept in the stable log.

The analysis pass is modeled as an analysis-TM which initiates an analysis-RM. The analysis-RM further initiates a copy access, which re-initializes the lock, transaction and the dirty object tables by placing their stable storage copies in the buffer after the last checkpoint record. Next, it invokes read accesses to read the log records from the corresponding DMs after the last checkpoint record. The analysis-RM with the help of write accesses updates these tables as follows. If a log record corresponding to transaction table is encountered whose identity does not already appear in the table, then an entry is made in the table. The transaction table is

```
Begin
1. If the prewrite-lock (PL) and write-lock (WL) on the corresponding DM is retained by its ancestor transaction
   then a read access T can acquire a read-lock (RL) only. That is,
   If PL and WL $\in$ anscesters (T) then read-lock-set (RLS) = RLS $\cup$ {T}
2. If the prewrite-, write- and read-locks are retained by its ancestor transactions then a prewrite access
   subtransaction T can get the prewrite-lock only. That is,
   If PL, WL and RL $\in$ anscesters (T) then prewrite-lock-set (PLS) = PLS $\cup$ {T}
3. If read-, prewrite-, write-locks are held by its ancestors then a write access transaction T can get the write-lock
   only. That is,
   If PL, WL and RL $\in$ anscesters (T) then write-lock-set (WLS) = PLS $\cup$ {T}
4. If its ancestors hold the write-lock then a transfer access T can get a read-lock only. That is,
   If WL $\in$ anscesters (T) then read-lock-set (RLS) = RLS $\cup$ {T}
5. When a read access subtransaction T commits, it releases the lock to its parent. When its parent commits, it
   passes the lock to its parent and so on. That is,
   If T $\in$ COMMIT then RLS = RLS – {T} $\cup$ parent (T)
6. When a transfer access transaction T commits, it passes the lock to the daemon transaction $T_D$. When a
   daemon commits, it passes the lock to its associated transaction $T_k$. That is,
    If T $\in$ COMMIT then RLS = RLS – {T} $\cup$ $T_D$ ;
   If $T_D$ $\in$ COMMIT then RLS = RLS – {T} $\cup$ $T_K$ ;
7. When a prewrite access commits T, it passes the lock to its parent $T_P$. When the parent commits
   (prewrite-RM), it passes the lock to the least common ancestor of all other access descendants waiting for the
   lock. That is,
    If T $\in$ COMMIT then PLS = PLS – {T} $\cup$ parent(T) ;
    If parent(T) $\in$ COMMIT then PLS = PLS – {T} $\cup$ least-common-ancester (T) ;
8. When a write access transaction commits, it releases the lock to the least common ancestor of all other access
   descendants waiting for the lock. That is,
    If T $\in$ COMMIT then WLS = WLS – {T} $\cup$ least-common-ancester (T) ;
9. When a write access or prewrite-RM or any of its ancestor transaction T aborts, the aborted transaction's locks
   are passed to its parent transaction. That is,
   If T $\in$ ABORT then { WLS = WLS – {T} $\cup$ parent (T) ; RLS = RLS – {T} $\cup$ parent (T); PLS = PLS – {T}
$\cup$ parent (T) }
10. When a read-only transaction T aborts, its lock is released entirely. When a transaction in the hierarchy of
   daemon transaction aborts, its lock is released entirely. That is,
   If T $\in$ ABORT then RLS = RLS – {T};
   End;
```

Fig. 2. Locking algorithm.

modified to track the active transactions at the time of failure. Similarly, a log record corresponding to dirty object table is entered with the current LSN in the dirty object table if it is not already there. In a similar fashion, the lock table is also updated. Whenever a commit record is encountered, a list of commit records is made.

This helps in establishing the restart states of the transactions to be re-executed. A read-analysis returns a set of min RecLSN (min LSN of log records from where redo recovery has to restart, called RedoLSN): one for each dirty data object with respect to write operations and a set of min RecLSNs (greater then the max RecLSNs of the corresponding data object) for each dirty data object with respect to prewrite operations. It also returns a list of active transactions and their restart states. The above information will be the output of an analysis-TM. A checkpoint is taken at the end of the analysis pass.

The redo activity is performed using the prewrite values and after-images logged on the stable storage. The redo activity can be skipped if there is no dirty data object. The transaction hierarchy for redo activity consists of a redo-TM for each object, a redo-RM at the next level and access subtransactions at the leaf level. Each redo-TM initiates a redo-RM which further triggers copy, read, prewrite and write access subtransactions to redo the operations present in the stable log corresponding to the data objects after the last checkpoint record. A copy transaction re-initializes the write-buffer by copying the contents of the stable-db to the write-buffer. Read operations read the write and prewrite log entries one by one after the last checkpoint. A prewrite access will re-initialize the prewrite-buffer with the help of prewrite log record value if the value of LSN associated with the prewrite log record is greater than LSNs of all the after-images logged on stable storage. A write access substitutes the value (after-image) of the data object read from the log in the allocated write-buffer of the data object if the data object's LSN in the write-buffer is found to be less than the log record's LSN. These log records will be those whose effects are not yet in the non-volatile storage version of the data object. The daemon transaction associated with the redo-TM initiates transfer-RMs to transfer the contents of the write-buffer to the stable-db. After the commit of redo-TM, a checkpoint is taken.

The analysis and redo activities in the form of nested transactions provide faster recovery since a redo-TM for each data object can be initiated in parallel. Also, if a system crash occurs during the execution of an analysis- or redo-TM, the corresponding new TM can be triggered on system restart. All the actions of previously committed subtransactions of the failed TM are not required to be discarded in case of a system failure. Similarly, in case of a normal transaction abort of any of these TMs, RMs, or their subtransactions, a corresponding new transaction can be initiated without discarding the effects of the aborted transaction and their descendants (if any). This helps in relaxing the atomicity property of such subtransactions since neither the actions of previously committed subtransactions are undone nor the failed TM or RM is retried until their completion on revival.

4.1. Buffer management operations

The transfer of a data object's value from the write-buffer to the stable storage is initiated with the help of a daemon transaction associated with each write-TM and redo-TM. The daemon transaction invokes a transfer-RM that transfers the value of the data object from its write-buffer in the volatile memory to the stable-db on secondary storage. For the sake of uniformity, and in order to model correct atomicity requirements, transfer-RMs are modeled as transactions and placed as children of write- and redo-TMs in the transaction tree. One would like to permit transfer operations to take place with the intention that these TMs do not control their invocations. They are intended to run spontaneously and transparently. Therefore, we have associated a daemon transaction with each write- and redo-TM so that these TMs do not have to be aware of the invocations of transfer-RMs. A daemon transaction commits with the commit of its associated TM. Therefore, it is possible to invoke many transfer-RMs dur ing its span and these transfer-RMs may not have committed before their parent transaction's commit. However, we know that the sequence of transfer operations for a read-write data object from the write-buffer to stable-db is the same as transfer of last write. This enables the daemon transaction to initiate many transfer-RMs and to commit without the commit of all its transfer-RM subtransactions. The daemon transaction achieves the spontaneous and transparent behaviour of the periodic transfer of the data from the volatile memory to stable storage and is failure-atomic.

The transfer of write-buffer to the stable-db takes place if the stableLSN of the write-buffer is greater than the LSN of the write-buffer. Once the transfer of the value along with LSN is completed, the corresponding log entry from the stable log is removed and the stableLSN of the write-buffer is set to the LSN of the write-buffer.

5. Snapshot of transaction processing, logging and recovery

Consider the nested transaction tree structure as shown in Fig. 3, where U is a user-visible transaction. T_1 and T_2 are read- and write-TMs, respectively. T'_2 is the associated daemon transaction. T_{11}, T_{21} and T_{22} are read, prewrite and write-RMs, respectively. T'_{23} is the transfer-RM. T_{111} is a read access, T_{211} and T_{212} are prewrite access subtransactions whereas T_{221} and T_{222} are corresponding write accesses, respectively. T'_{231} and T'_{232} are

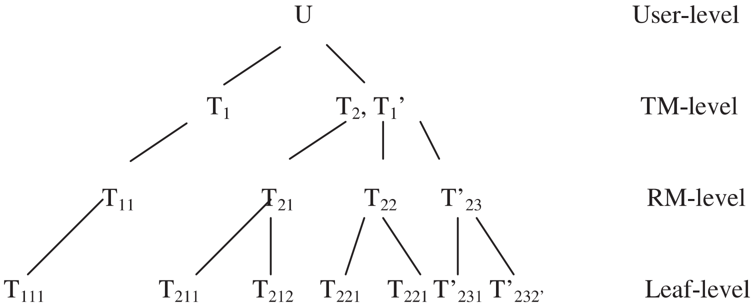

Fig. 3. Nested transaction tree.

the transfer access subtransactions. Data objects supposed to be accessed by these transactions are X and Y.

As soon as the top level transaction U is initiated, its state is set to be "active" in the transaction table. Also, this information is appended to the log on stable storage. As soon as transactions T_2, T_{21}, T_{211} and T_{212} are made active, the information is recorded in the transaction table as well as logged on stable storage. The state of the transaction table is as shown in Fig. 4(a). The transaction T_{21} has initiated the prewrite access subtransactions T_{211} and T_{212} to announce the prewrite value of the data object X and Y, respectively. As soon as the prewrite access subtransaction T_{211} (T_{212}) gets the prewrite-lock on the DM corresponding to the data object $X(Y)$, an entry in the lock table is made as shown in Fig. 4(b). When a prewrite value of a data object $X(Y)$ is announced by $T_{211}(T_{212})$, LSN of the log record to be written is placed in the LSN field associated with the prewrite value. When $T_{211}(T_{212})$ decides to commit, the commit information and prewrite value along with LSN are passed to their parent subtransaction T_{21}. The prewrite-lock held by T_{211} and T_{212} are passed to the prewrite-RM T_{21} on their commit and the corresponding entries are made in the lock table. The updating of dirty object table, logging of prewrite values and lock holding information associated with T_{211} and T_{212} are deferred until the commit of recovery point subtransaction T_{21}. This is because in case of system crash, the effect of T_{211} and T_{212} are to be discarded. On commit of T_{21}, the prewrite values are transferred to the respective logs as shown in Fig. 4(c) where each tuple in the database log is of the form ⟨LSN, transaction-id, object-id, prewrite value⟩.

To sustain the effect of recovery point subtransaction T_{21} in case of system crash, its commit-state information that includes all the variables, its private data and prewrite values along with the LSNs, are also logged. Logging of commit-state is repeated until the top level transaction commits. When the prewrite-RM T_{21} commits, the lock released by T_{21} is inherited by U (it is safe to release lock to U here since the ancestors of recovery point subtransaction, prewrite-RM, cannot be aborted in any case). This information is also recorded in the lock table as well as logged on stable storage. The state of the transaction, lock and dirty object tables after the commit of T_{21} is shown in Fig. 5(a)–(c). Note that these updated tables are also checkpointed periodically during normal processing which reduces the system restart's work in case of system crash.

Transaction-id	Status
U	Active
T_2	Active
T_{21}	Active
T_{211}	Active
T_{212}	Active

(a)

Transaction-Id	Locks (Hold/Retain)	Object-Id
T_{211}	Prewrite-Lock(Hold)	X
T_{212}	Prewrite-Lock(Hold)	Y

(b)

Event	**"Data" type Log Records**
Commit (T_{21})	w.r.t. DM_X : <100, T_{211}, X, $\underline{x}$>
	w.r.t. DM_Y : <101, T_{212}, Y, $\underline{x}$>
	where $\underline{x}$ and $\underline{y}$ are the prewrite values

(c)

Fig. 4. (a) Transaction table; (b) lock table; (c) event and log records.

Transaction-id	Status
U	Active

(a)

Transaction-Id	Locks (Hold/Retain)	Object-Id
U	Prewrite-Lock(Hold)	X
U	Prewrite-Lock(Hold)	Y

(b)

Object-Id	RecLSN (Prewrite)	RecLSN(write)
X	100	
Y	101	

(c)

Fig. 5. (a) Transaction table; (b) lock table; (c) dirty object table.

Event	**"Data" type Log Records**
Commit (T_{22})	w.r.t. DM_X : <102, T_{221}, X, x>
	w.r.t. DM_Y : <103, T_{212}, Y, y>
	where x and y are write values

Fig. 6. Event and log records.

Transaction-id	Status
U	Active

(a)

Transaction-Id	Locks (Hold/Retain)	Object-Id
U	Prewrite-Lock(Retain)	X
U	Prewrite-Lock(Retain)	Y
U	Write-Lock(Retain)	X
U	Write-Lock(Retain)	Y

(b)

Object-Id	RecLSN(Prewrite)	RecLSN (Write)
X	-	102
Y	-	103

(c)

Fig. 7. (a) Transaction table; (b) lock table; (c) dirty object table.

After the commit of prewrite-RM T_{21}, the write-TM T_2 initiates write-RM T_{22} to write data objects X and Y whose prewrite values have been announced before. Write-RM T_{22} further initiates write access subtransactions T_{221} and T_{222} which finally update data objects X and Y, respectively. When an update is performed, the log record is written along with LSN and this value of LSN is also placed in the LSN field of the updated value of the data object as shown in Fig. 6. The write-lock held by T_{221} (T_{222}) is also inherited by U and the information is recorded in the lock table as well as logged on the stable storage. The states of different tables after the commit of T_2 are shown in Fig. 7(a)–(c).

When the write-TM T_2 is initiated, the associated daemon transaction T'_2 is also invoked automatically. The daemon transaction initiates the transfer-RM T'_{23} which further initiates transfer access subtransactions T'_{231} and T'_{232} to transfer the updated values of data objects X and Y to the stable-db, respectively. The information that these transactions are initiated is also recorded in the transaction table. As soon as T'_{213} (T'_{232}) acquires a read-lock, the information is recorded in the lock table. When the transfer of the data object X's (Y's) value from the write-buffer to the stable-db is completed, the corresponding entry from the dirty object table as well as from the log is also removed. When any of the readonly transactions T_1, T_{11} and T_{111} are initiated, the information is recorded in the transaction table. Similarly, when a read-lock is acquired or released, the lock table is also updated. However, such information is not logged on stable storage as these transactions are to be discarded in case of system failure. When the top level transaction U commits, all the entries corresponding to U are removed from the tables and the logs.

5.1. System restart processing

5.1.1. Analysis pass processing

Given below is an example to show how analysis pass operations explained in Section 5 are performed. Suppose in the stable storage, the tables are in a state as shown in Fig. 7(a)–(c) and the log has the following entries after the last checkpoint record at the time of system failure.

"Data" type	At Log$_x$	$\langle 102, T_{221}, X, x\rangle$
	At Log$_y$	$\langle 101, T_{212}, Y, y\rangle$
"Transaction" type	At common Log	$\langle T_{221}$, commit$\rangle$
"Lock" type	At common Log	$\langle T_2$, write-lock (retain), $X\rangle$

On system restart, the external agent initiates an analysis-TM T_a to perform the analysis pass (see Fig. 8). The analysis-TM invokes the analysis-RM T_{aa} at the next level of transaction hierarchy. The analysis-RM further initiates copy, read, write and read-analysis access subtransactions. The copy transactions T_{c_i} $(i = 1, 3)$ copy all the tables from the stable storage into the volatile memory. Read subtransactions T_{r_i} $(i = 1, 4)$ read the log records one by one from the logs corresponding to each data object and the common log. The corresponding write accesses T_{w_i} $(i = 1, 4)$ update the tables as follows : since the entries of database type log records are not in the dirty object table, therefore, entries are made into the table. Similarly, other tables are also updated. Also, since T_{221} has committed, the corresponding entry in the transaction table is removed. The read-analysis returns min RecLSN (RedoLSN) of 102 for the write operation of the data object X and min RecLSN (RedoLSN) of 101 for the prewrite operation of the data object Y. It also returns the information that U and T_2 are active subtransactions along with lock holding information about such transactions. These transactions are to be restarted as their recovery point subtransaction T_{21} has committed.

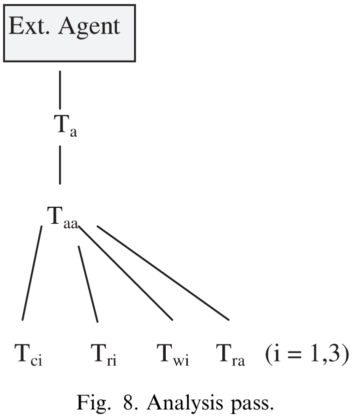

Fig. 8. Analysis pass.

5.1.1.1. Concurrency control for analysis pass. To explain concurrency control among access transactions of the analysis pass, we assume that only one copy, one read, one write and one read-analysis access subtransactions have to be initiated. The copy, read and read-analysis accesses require a read-lock before accessing a DM whereas a write access requires a write-lock before accessing a DM. When an access transaction commits, the lock is released to its parent. When a non-access transaction commits, its lock is released to the external agent. When a transaction aborts, its lock is released to the parent transaction without discarding the effects of the aborted transaction. In case of a system failure, analysis pass is started again. In case of a transaction abort, a new transaction can be initiated. The locking rules are similar as given for normal system operations in Section 3.1.

5.1.2. Redo pass processing

Here, we explain the redo pass operations given in Section 4 with the help of the following example. Consider the following database log records corresponding to the data object X at the time of system failure.

"Data" type log records		**Write-buffer of X**
Write Log		
	$\langle$END-CHK-POINT$\rangle$	stableLSN = 222
	$\langle T_1, X, 220, x_1\rangle$	LSN = 224
	$\langle T_2, X, 222, x_2\rangle$	stable-db of X
	$\langle T_3, X, 224, x_3\rangle$	x_2LSN = 222
Prewrite Log	$\langle T_4, X, 225, \underline{x}\rangle$	

Suppose RedoLSN of data object X with respect to the write operations in the reorganized dirty object table is 222. Let the min RecLSN (RedoLSN) of prewrite operations which is greater than max RecLSNs of write operations in the dirty object table be 225. Consider Fig. 9. During redo operations, redo-TM transaction T_s initiates a redo-RM transaction T_{s1} which in turn initiates a copy transaction T_c which re-initializes the write-buffer of the data object X by copying its value from the stable-db along with LSN. It sets its stableLSN and LSN fields to be equal to the LSN of the stable-db. After

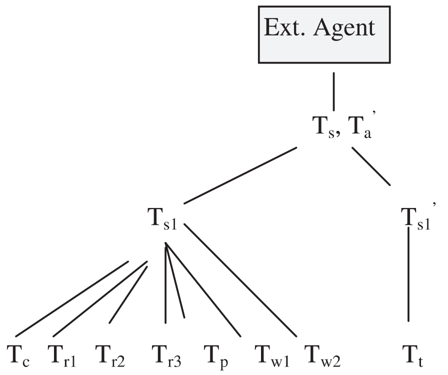

Fig. 9. Redo pass.

its commit, the value in the write-buffer will become x_2 with LSN 222. T_{s1} then triggers a read access subtransaction T_{r1} with RecLSN 222. It reads the database write log record whose LSN is greater than or equal to 222. Similarly, T_{r2} and T_{r3} are initiated to read the next write and prewrite log entries, respectively. The write access subtransaction T_{w1} invoked by T_{s1} replaces the write-buffer value of the data object X by x_2 if LSN of the log record is greater than stableLSN of the object in the write-buffer. Since LSN (222) of the log record is not greater than the stableLSN (222) of the object in write-buffer and therefore, T_w commits without updating the value in the write-buffer with the return message "not to be redone". T_{s1} will then initiate next write access subtransaction T_{w2} which replaces the write-buffer of the object by x_3 as its LSN (224) is greater than the stableLSN (222). T_{s1} also concurrently initiates a prewrite access subtransaction T_p to re-initialize the prewrite value in the prewrite-buffer if the LSN value of the prewrite log record is greater than the max RecLSN of all the write operations. T_p re-initializes the prewrite-buffer with the prewrite value $\underline{x}$, as its LSN value (225) is greater than the max RecLSN 224 of all the write operations. During the redo operation, the daemon transaction T_s' associated with T_s is also invoked automatically. T_s in turn initiate a transfer access subtransaction to transfer the contents of the write-buffer of the data object to the stable-db. T_t is the transaction which transfers the data object X's write-buffer value if its LSN is greater than the stable-LSN of the data object in the write-buffer.

5.1.2.1. Concurrency control for redo pass. Consider Fig. 9, where T_s is a redo-TM and T_s' is the associated daemon transaction. T_{s1} and T_{s1}' are redo- and transfer-RMs, respectively. T_c, $\{T_{r1}, T_{r2}\}$, T_p and T_t denotes copy, read, prewrite, and transfer access subtransactions whereas T_w is the write access subtransaction, respectively. In this example, for simplicity, we have assumed reading of only one write and one prewrite log record. As before, copy, read, and transfer accesses require read-locks whereas prewrite and write accesses require prewrite- and write-locks, respectively. All the conflict relations are the same as before except that a prewrite-lock does not conflict with a write-lock. When an access transaction commits, its lock is passed to its parent and so on except that write and transfer accesses release their locks to the external agent. When a transaction aborts, its lock is released to its parent. The locking rules are same as given in Section 3.1.

6. Conclusions

A nested transaction model, its concurrency control and recovery algorithms are presented in this paper. We have introduced the concept of a recovery point subtransaction and prewrite operation in our model to achieve higher concurrency. Our recovery algorithm consisting of system restart operations; analysis and redo operations and buffer management operations, which are modeled in terms of nested transactions. Modeling recovery operations in terms of subtransactions increases concurrency during system restart operations. As a future work, our transaction model need to extend in the context of orthogonally persistent programming languages where serious problems arise unless all computation exists within a transactional context, but this restricts concurrency. We are exploring this issue further. We are also looking into adapting our transaction recovery model in mobile computing environment (Madria et al., 1999). We are working on the verification of our recovery algorithm using I/O automaton model (Fekete et al., 1993). Each component of our recovery model is modeled as I/O automata and is specified with the help of some pre and post conditions to capture the operational semantics. We have proved some invariant that leads towards the correctness of the model. Thus, our model gives clear understanding of the recovery algorithm. This work will be reported as a separate paper (Madria and Bhargava, 1999), where main thrust is only on the proof of correctness.

References

Fekete, A., Lynch, N., Merrit, M., Whiel, W., 1993. Atomic Transactions. Morgan-Kaufmann, Los Altos, CA.

Gray, J., et al., 1981. The recovery manager of the System R database manager. ACM Comput. Surveys 13 (2), 223–244.

Harder, T., Reuter, A., 1983. Principles of transaction oriented database recovery. ACM Comput. Surveys 15 (4), 287–378.

Korth, H.F., Kim, W., Bancilhon, 1990. On long-duration CAD transactions. Inform. Sci. 46, 73–107.

Kim, W., Lorie, R., Mcnabb, D., Plouffe, W., 1984. A transaction mechanism for engineering design databases. In: Proceedings of the 10th International Conference on Very Large Databases. VLDB Endowment, pp. 355–362.

Korth, H.F., Speegle, G., 1990. Long duration transactions in software design projects. In: Proceedings of the Sixth IEEE International Conference on Data Engineering. New York, pp. 568 – 574.

Madria, S.K., 1995. Concurrency control and recovery algorithms in nested transaction environment and their proofs of correctness, Ph.D. thesis. Indian Institute of Technology, Delhi, India.

Moss, J.E.B., 1985. Nested Transactions: An Approach to Reliable Distributed Computing. MIT Press, Cambridge, MA.

Moss, J.E.B., 1987. Log-based recovery for nested transaction, COINS, Technical Report. University of Massachusetts at Amberest, pp. 87–98.

Madria, S.K., Bhargava, B., 1999. Improving Availability in Mobile Computing Using Prewrite Operations, CSD-TR-32. Department of Computer Sciences, Purdue University, IN, June 1997 (under revision in Distributed and Parallel Databases).

Madria, S.K., Bhargava, B., 1997a. System defined prewrites to increase concurrency in databases. In: Proceedings of the First East-European Symposium on Advances in Databases and Information Systems (in co-operation with ACM-SIGMOD). St.-Petersburg, Russia.

Mohan, C., Haderle, D., Landsay, B., Pirahesh, H., Scwartz, P., 1992. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans. Database Syst. 17 (1).

Madria, S.K., Maheshwari, S.N., Chandra, B., 1997b. Formalization and correctness of a concurrency control algorithm for an open and safe nested transaction model using I/O automaton model. In: Proceedings of the Eighth International Conference on Management of Data (COMAD'97). Madras, India.

Madria, S.K., Maheshwari, S.N., Chandra, B., Bhargava, B., 1997c. Crash recovery algorithm in an open and safe nested transaction model. In: Proceedings of the Eighth International Conference on Database and Expert System Applications (DEXA'97), France, Lecture Notes in Computer Science, vol. 1308. Springer, Berlin.

Madria, S.K., Maheshwari, S.N., Chandra, B., Bhargava, B., 1999. Crash Recovery in an Open and Safe Nested Transaction Model: Formalization and Correctness (under communication to Journal).

Mohan, C., Rothermel, K., 1989. Recovery protocol for nested transaction using write-ahead logging. IBM Technical Disclosure Bulletin 31(4), September 1988 (also appeared in the Proceedings of the 15th VLDB Conference, Amsterdam, 1989).

Muth, P., Rakow, T.C., Weikum, G., Brossler, P., Hasse, C., 1993. Semantic concurrency control in object-oriented database systems. In: Proceedings of the Ninth IEEE International Conference on Data Engineering. pp. 233–242.

Resende, R.F., Agrawal, D., Abbadi, A.E., 1994. Semantic locking in object oriented database systems, Technical Report TRCS 94-01. University of California at Santa Barbara.

Weikum, G., Hasse, C., Brossler, P., Muth, P., 1990. Multi-level recovery. In: Proceedings of the Ninth ACM Symposium on Principles of Database Systems. Nashville, pp. 109–123.

Weikum, G., 1991. Principles and realization strategies of multi-level transaction management. ACM Trans. Database Syst. 16 (1).

Sanjay Kumar Madria has obtained his Ph.D. from Indian Institute of Technology, Delhi, India in 1995. He has been a Visiting Assistant Professor in the Department of Computer Science, Purdue University, USA since 1999. He has earlier worked in Nanyang Technological University, Singapore and University Sains Malaysia, Penang, Malaysia. He has widely published in the area of nested transaction processing, mobile transaction processing and more recently on management of web data. He is a Guest-editor of WWW Journal. He is Program-Chair for EC&WEB 2000 conference to be held in UK. He is PC member of many leading international database conferences and reviewers of many database journals. He has given many invited seminars and tutorials in leading institutes and database conferences (EDBT'2000). He has served as invited Panelist in NSF (National Science Foundation) and in some other database conferences.

S.N. Maheshwari received his Ph.D. in Computer Science from Northwestern University, USA. He is currently a Professor of Computer Science and Engineering at the Indian Institute of Technology, Delhi, India, where he has held the appointments of Head, Department of Computer Science and Engineering, and Dean, Undergraduate Studies. His research interests range from graph algorithms, computational geometry, distributed and parallel computing, to theory of transaction processing.

B. Chandra is a Professor of Computer Applications at the Indian Institute of Technology for the past two decades. Her area of specializations include Distributed Databases with special reference to recovery, knowledge acquisition using Neural Networks. She has published a number of research papers in reputed journals in these two areas. She has held visiting faculty positions at the University of Pittsburgh, USA and the Penn State University, USA. She has also held positions at the World Bank at Washington D.C. and INRIA, Paris.

Bharat Bhargava is a full professor in the Department of Computer Science, Purdue University, West Lafayette, USA. His research involves both theoretical and experimental studies in distributed systems. Professor Bhargava is on the editorial board of three international journals. In the 1988 IEEE Data Engineering conference, he and John Riedl received the best paper award for their work on A Model for Adaptable Systems for Transaction Processing. Professor Bhargava is a fellow of Institute of Electrical and Electronics Engineers and Institute of Electronics and Telecommunication Engineers. He has been awarded the Gold Core Member distinction by IEEE Computer Society for his distinguished service. He has been awarded IEEE CS technological Achievement Award for Adaptability and fault-tolerance in communication network in 1999.