Reducing Recovery Constraints on Locking based Protocols*

(Extended Abstract)

G. Alonso D. Agrawal A. El Abbadi

Department of Computer Science
University of California
Santa Barbara, CA 93106
{gustavo,agrawal,amr}@cs.ucsb.edu

Abstract

Serializability is the standard correctness criterion for concurrency control. To ensure correctness in the presence of failures, recoverability is also imposed. Pragmatic considerations result in further constraints, for instance, the existing log-based recovery implementations that use before-images warrant that transaction executions be strict. Strict executions are restrictive, thus sacrificing concurrency and throughput. In this paper we identify the relation between the recovery mechanism and the restrictions imposed by concurrency control protocols. In particular, we propose a new inverse operation that can be integrated with the underlying recovery mechanism. In order to establish the viability of our approach, we demonstrate the new implementation by making minor modifications to the conventional recovery architecture. This inverse operation is also designed to avoid the undesirable phenomenon of cascading aborts when transactions execute conflicting write operations.

1 Introduction

Serializability is the standard correctness criterion for executing transactions concurrently. However, in order to ensure correctness in the presence of failures, a restriction that must be imposed in addition to serializability is *recoverability* [Had88]. Recoverability mandates that the commitment of transactions that read uncommitted values must be delayed until these values are committed. Otherwise the semantics of commitment may be altered if there are failures, i.e., crash failures or transaction aborts. Pragmatic considerations, however, result in additional constraints on the concurrent execution of transactions to ensure failure atomicity of transactions. For instance, in order to avoid the problem of *cascading aborts*, transactions are not allowed to read uncommitted data. Performance studies [AEJ92] have shown that cascading aborts are undesirable since they may result in degraded throughput in the system. Similarly, the existing log-based recovery implementations that use before-images warrant that transaction executions be *strict* [BHG87], i.e., transactions are not allowed to read or write uncommitted data. From a theoretical perspective there is no reason for strict execution, however most transaction management protocols enforce this constraint. Strict executions are known to be quite restrictive, resulting in a significant sacrifice of concurrency and throughput in the system [AEJ92]. There are many applications where transaction or object semantics can be used to achieve more concurrency. However, protocols that try to exploit this property do so only by enhancing the concurrency control protocol used by the scheduler while ignoring the underlying recovery mechanism. In particular, it is assumed that the traditional recovery mechanism based on logs and before-images will be used. As a result, the schedulers produce classes of histories that are unnecessarily restrictive. Another reason for this fragmented development is the lack of an integrated model for reasoning about concurrency control and recovery protocols. Recently, Schek, Weikum and Ye [SWY93] proposed a unified theory for concurrency control and recovery. However, the theory is closely linked with the traditional recovery mechanism based on logs and before-images.

In this paper we identify the relation between the recovery mechanism and the restrictions imposed by concurrency control protocols. In particular, we propose a new *inverse* operation that can be integrated with the underlying recovery mechanism. This is in contrast to the traditional inverse, or undo, operation based on the straightforward restoration of before-images. In order to establish the viability of our approach, we demonstrate the new implementation by making minor modifications to the current recovery architecture [GR93]. This inverse operation is also designed to avoid the undesirable phenomenon of cascading aborts when transactions execute conflicting write operations. Although the concept of cascading aborts is related

*This research was partially supported by the NSF under grant number IRI-9117904.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

SIGMOD/PODS 94 - 5/94 Minneapolis, Minnesota USA
© 1994 ACM 0-89791-639-5/94/0005..$3.50

to write-read conflicts, the traditional undo operation creates a similar situation when two transactions have a write-write conflict and the first of them aborts. Our new formulation of the undo operation to recover from aborted transactions, referred to as an inverse operation, does not suffer from this problem. Hence, the proposed recovery mechanism can be used to design more efficient protocols for transaction management.

The rest of the paper is organized as follows. In the next section we explore the role of recovery in serializability theory. Section 3 introduces the new inverse operation. Section 4 describes the implementation of the proposed inverse operation for recovery. Section 5 contains a brief discussion of the use of the inverse operation and its implications on transaction management.

2 The role of recovery in serializability theory

The notations and concepts we use are borrowed from those used in traditional serializability theory [BHG87]. A database is a collection of data objects. These objects are manipulated through read and write operations. Users interact with the database by issuing transactions. A transaction is a sequence of read and write operations followed by either an abort or a commit operation. For a transaction T_i, a write operation on data object x is denoted as $w_i[x]$, a read operation on x is denoted as $r_i[x]$ and its commitment or abortion as c_i and a_i respectively. Note that a transaction either commits or aborts, but not both. Two operations conflict if they are on the same data object and one of them is a write. A history captures the order of execution of the transactions. It is defined as a partial order of all the operations in the transactions that are part of the history. This partial order must be in agreement with the order of the transactions and it must specify an ordering for all conflicting operations. A complete history is a history in which all transactions are either committed or aborted. The committed projection of a complete history is obtained from the complete history by removing all aborted transactions. Traditional concurrency control and recovery criteria deal only with the committed projection of a history. A history H is *serializable* (SR) if there is a serial history, H_s such that both H_s and the committed projection of H, $C(H)$, have the same order for all conflicting operations.

The traditional criterion for the correctness of concurrency control protocols is serializability. However, serializability is not enough to guarantee correctness [BHG87]. *Failure atomicity* is required to ensure that transactions are atomic in the presence of failures. The most common way to ensure failure atomicity is to use before-images and logging [HR83, MHL+92]. Before a value is written to the database, the old value is logged in the stable storage and it is referred to as the before-image associated with this write operation. This old value is the before-image for that particular write operation. If the write operation is aborted, its effects can be eliminated by restoring the before-image from the log to the database.

To ensure that restoring before-images always leaves the database in a consistent state, *strict* execution is enforced. A history is strict if no transaction reads or writes an uncommitted value. Of this restriction, only recoverability is really necessary. A history is *recoverable* when a transaction that reads an uncommitted value does not commit until that value is committed [Had88]. Note that strict executions are more restrictive than recoverable executions, even considering only write-read conflicts. For write-write conflicts, strict executions reject the following type of histories:

$$w_1[x]w_2[x]c_2c_1$$

The above history is correct, but it is not strict. However, the reason why this history is not accepted is that it is *potentially* problematic. If after T_2 commits T_1 decides to abort instead of committing, then the wrong before-image is restored leaving the database in an inconsistent state. This notion of being "potentially problematic" plays an important role in the dependency of concurrency control on recovery. Note, however, that this is because we are assuming a very specific way of undoing writes of aborted transactions by restoring before-images. If this mechanism is changed, then we might be able to relax the requirements for correctness.

A very interesting example of the influence of this approach of undo operations on a correctness criterion appears in [SWY93]. There a unified theory is proposed, providing a correctness criterion that accounts for both concurrency control and recovery. This is done by expanding the history to incorporate the actions that take place in the transaction manager when a transaction is aborted. In particular, an abort operation is replaced with a sequence of undo operations corresponding to the writes executed by the aborted transaction and the resulting execution is called an expanded history [SWY93]. A correctness criterion is defined over this *expanded* history: *prefix reducibility* (PRED). PRED is described in more detail in the appendix. For our purposes, we consider the following definition, which is more intuitive and has been shown to be equivalent to PRED [AVA+94].

Definition 1: *Serializability with Ordered Termination: Alternative definition of PRED*

A history, H, is PRED if it is serializable and for every pair of transactions T_i, T_j such that

1. if $w_i[x] < r_j[x]$ in H and $a_i \not< r_j[x]$ then T_i commits before T_j commits
2. if $w_i[x] < w_j[x]$ in H and $a_i \not< w_j[x]$ then either T_i

	Original History	Class
1.	$w_1[x]w_2[x]c_1c_2$	PRED
2.	$w_1[x]w_2[x]c_2c_1$	not in PRED
3.	$w_1[x]w_2[x]a_1c_2$	not in PRED
4.	$w_1[x]w_2[x]c_2a_1$	not in PRED
5.	$w_1[x]w_2[x]c_1a_2$	PRED
6.	$w_1[x]w_2[x]a_2c_1$	PRED
7.	$w_1[x]w_2[x]a_1a_2$	not in PRED
8.	$w_1[x]w_2[x]a_2a_1$	PRED

Table 1: Serializable and recoverable histories, and their inclusion within PRED

commits before T_j terminates or T_j aborts before T_i terminates.

□

The constraints imposed by PRED are two. The first corresponds to the traditional notion of recoverability. The second forbids certain orderings in the termination of transactions that have write-write conflicts. Without this latter constraint, the class becomes that of serializable and recoverable histories. The differences between these two classes of histories are summarized in Table 1.

History 2 is deemed incorrect because if T_1 decides to abort instead of committing, the state of the database is incorrect. This is because when T_1 aborts, the recovery mechanism performs the corresponding undo operation, restoring the before image of $w_1[x]$. As a result, the update made by T_2 is lost. In history 4, when T_2 commits it can not be assumed that T_1 will also commit, which may lead to history 2. Histories 3 and 7 are also similar. In history 3, if T_1 is undone, the effects of T_2 on the database are lost in spite of it being committed. In history 7, if T_1 is undone before T_2, then the effects of T_1 would appear in the database in spite of T_1 being aborted. Hence, PRED, that was intended as a unified criterion for concurrency control and recovery, maintains constraints that are due to a specific recovery mechanism. If this mechanism is changed, we can expand the unified theory to accept the class of serializable and recoverable histories.

This phenomenon of limiting the class of correct histories because of the underlying recovery mechanism is, unfortunately, very common. To our knowledge, it has not been addressed convincingly and, as a result, protocols that enhance concurrency remain theoretical constructions due to the lack of an appropriate recovery mechanism. Consider, for instance, *strict two phase locking* (strict 2PL). 2PL is enough to guarantee serializable executions. However, the most implemented version of 2PL is strict 2PL. In strict 2PL, a transaction does not release the locks until it commits. Hence, transactions can only read or overwrite committed values, which is the restriction imposed by strict executions. This is done primarily to simplify the recovery mechanism, i.e. so that before-images can be easily restored [BHG87] without giving rise to the anomalies of Table 1. The class of histories accepted by strict 2PL is known as *rigorous* histories [BGRS91]. Rigorous histories exclude many correct executions. For this reason many authors have proposed modified locking protocols that accept supersets of rigorous histories.

Ordered shared locks [AE90], were devised to allow transactions to perform conflicting accesses concurrently as long as they are done in an orderly manner and serializability is guaranteed. However, the family of protocols proposed ensure only serializability and no solution is proposed to deal with the problem of recovery. Similarly, altruistic locking [SGA87] was proposed to avoid the long delays introduced by long lived transactions. Altruistic locking allows these transactions to grant access to locked items according to certain rules. The result is that two transactions may write to the same object concurrently. However, the protocol does not take recovery into consideration. For pragmatic reasons, write locks must be held until the transactions commit and no concurrent access is allowed. In both cases, altruistic and ordered shared locking, more concurrency is achieved by allowing transactions to write concurrently to the same object. However, if the traditional approach for recovery is assumed, this is not possible. The restrictions are imposed due to the recovery architecture and not from the concurrency control point of view, thus, a protocol that only deals with concurrency control can not possibly eliminate these constraints.

This issue is not only found in modified locking protocols, but also in many attempts at allowing more concurrency. For instance, in [SSV92] it is proposed to divide transactions to allow access to data items before a transaction that uses them commits. One of the assumptions made in this model is that a transaction can not write uncommitted data. This is the same restriction imposed by strict execution and hence suffers from the same problems. *Split-transactions*, proposed in [PKH88] also have the same drawbacks. Finally, several concurrency control protocols that use transaction serializability to ensure correctness in advanced database applications have similar problems. In particular, [Gar83, Lyn83, FÖ89] have proposed the notion of *relative atomicity* to break up transactions into several atomic units based on the application semantics. Unfortunately, this theoretical development ignores the issue of recovery. In particular, if a transaction, T allows another transaction T' to be interleaved with it such that T' overwrites the value written by T, the standard

recovery mechanism fails to maintain data consistency if T aborts.

3 A new recovery model

The influence that recovery exercises over concurrency control is related to the particular way in which before-images are restored. In this section we introduce a more advanced form of *undo* operation (an undo operation *usually* just restores the before image) that will allow us to implement a recovery mechanism to accept the class of recoverable histories. Assuming that this mechanism is in place, the restriction of strict execution, which arises due to operational considerations, can be eliminated from the design of the concurrency control protocols. We motivate the new inverse operation by analyzing PRED and the different restrictions it places on the abortion of transactions. For example (see Table 1), in the execution $w_1[x]w_2[x]$, T_1 cannot decide to abort, without forcing the abort of T_2, and T_1 must wait for T_2 to abort first. Note that this type of "cascading aborts" is not necessary from a semantic or recoverability point of view, rather, it is imposed by the particular properties of the recovery mechanism. This partially explains the popularity of strict 2PL, which avoids cascading aborts and does not interfere with a transaction's "right to abort". In this section, we explore the possibility of extending the log-based recovery in order to allow non-strict executions and eliminate the restrictions on transaction termination. The current recovery architectures employ the notion of before-image of an object when a write operation is executed. The effects of the write operations of an aborted transaction on the database are eliminated by restoring the before-images. We represent an operation that restores the before image of $w_i[x]$ as $w_i^{-1}[x]$. In an expanded history [SWY93], the abort operation of a transaction is replaced with these inverse operations, and the transaction is considered committed. For example, consider the following history:

$$w_1[x]c_1w_2[x]a_2$$

When the abort operation a_2 is expanded, the result is:

$$w_1[x]c_1w_2[x]\underline{w_2^{-1}[x]c_2}$$

In the above expanded history, the inverse write operation $w_2^{-1}[x]$ restores the before-image of x that existed prior to executing the write operation $w_2[x]$. The traditional approach is to perform an undo operation for every write that is aborted. We now define the notion of inverse operation to take into account concurrent transactions executing conflicting write operations. Note that write operations on a given object are atomic and, thus, must be totally ordered. We propose to implement an inverse of a write operation $w_i[x]$ as follows:

	Original History	Expanded History
1.	$w_1[x]w_2[x]c_1c_2$	$w_1[x]w_2[x]c_1c_2$
2.	$w_1[x]w_2[x]c_2c_1$	$w_1[x]w_2[x]c_2c_1$
3.	$w_1[x]w_2[x]a_1c_2$	$w_1[x]w_2[x]\underline{\epsilon_1[x]c_1}c_2$
4.	$w_1[x]w_2[x]c_2a_1$	$w_1[x]w_2[x]c_2\epsilon_1[x]c_1$
5.	$w_1[x]w_2[x]c_1a_2$	$w_1[x]w_2[x]c_1\underline{w_2^{-1}[x]c_2}$
6.	$w_1[x]w_2[x]a_2c_1$	$w_1[x]w_2[x]\underline{w_2^{-1}[x]c_2}c_1$
7.	$w_1[x]w_2[x]a_1a_2$	$w_1[x]w_2[x]\underline{\epsilon_1[x]c_1w_2^{-1}[x]w_1^{-1}[x]c_2}$
8.	$w_1[x]w_2[x]a_2a_1$	$w_1[x]w_2[x]\underline{w_2^{-1}[x]c_2w_1^{-1}[x]c_1}$

Table 2: Histories expanded using the new inverse operation

- If there exists a write operation $w_j[x]$ of transaction T_j that succeeds $w_i[x]$ and T_j is active or committed then the inverse of $w_i[x]$ is a null operation.

- If there does not exist a succeeding write operation of $w_i[x]$ then the inverse is executed by restoring the value $w_k[x]$ that corresponds to the first active or committed transaction T_k that precedes T_i[1].

We define the semantics of the inverse write operation based on the resulting actions that will appear in the expanded history. Consider a transaction T_i that writes x and later aborts. If there exists an active or committed transaction which wrote x after T_i, $inverse(w_i[x])$ is simply a null operation and is denoted by $\epsilon_i[x]$ in the expanded history. Otherwise, T_i restores its before image of x and then recursively restores the before-images of all preceding aborted transaction which wrote x. The process terminates when an active transaction T_k is encountered that wrote x and has not yet committed or there are no more transactions. We label all aborted transactions that wrote x between the write actions of T_k and T_i as $T'_k, \ldots, T'_i$. A more formal definition of this operation, $inverse(w_i[x])$, can be found in Figure 1.

Note that as was the case in [SWY93] $w_i^{-1}[x]$ is the before image written in the log record for $w_i[x]$. Using the proposed implementation of inverse write operations the expanded histories of Table 1 are shown in Table 2.

Hence with the new inverse write operation, concurrent writes are allowed to execute without imposing any restrictions on the aborts of transactions. The recovery mechanism imposes no unnecessary restrictions on the concurrency control protocol. If the inverse operation is incorporated with PRED, the resulting class corresponds to all serializable and recoverable histories.

[1] The reader perhaps will be alarmed at this point as to how can such an inverse write operation be implemented using logs. We will address this issue in the implementation section.

$$inverse(w_i[x]) = \begin{cases} \epsilon_i[x] & \text{if } \exists \text{ active or committed } T_j \text{ such that } w_i[x] < w_j[x] \\ w_i^{-1}[x] w_{i'}^{-1}[x] \ldots w_{k'}^{-1}[x] & \text{otherwise, where } T_k \text{ is the first active transaction} \\ & \text{such that } w_k[x] < w_{k'}[x] \ldots w_{i'}[x] < w_i[x] \end{cases}$$

Figure 1: Formal definition of the inverse operation

4 Implementation of the inverse operation

We now discuss how the specifications of the inverse operation can be realized by using the log-based recovery mechanism. For the above specifications, it is necessary to keep a list with all the active writes to an object, plus the last committed value. This is not feasible on a per-object basis due to the storage overhead so it must be integrated with some pre-existing data structures which are used by the Lock Manager and the Log Manager. In order to discuss their implementation, we need to establish a common understanding of the transaction processing mechanisms involved. For reasons of space we can not elaborate on them here, so we briefly summarize the most important aspects, borrowed from [GR93], which contains a detailed description of a transaction processing system.

There are three major components in a transaction processing system: the transaction table, the lock table, and the log table (see Figure 2). The lock and transaction tables reside in memory while the log table is persistent. Among other things, the transaction table contains a list of the operations performed by the transaction, its identifier, status, and pointers to the lock and log entries related to this particular transaction. The lock table is a hash table in which the operations holding locks on data items are recorded. This table also contains the operations that are waiting to acquire a lock on a data item. The log table contains the log entries corresponding to the operations of the transactions. For each active transaction T_i, there are two links: one to a lock list of the transaction and the other is a pointer to the sequence of log records created by this transaction. When a transaction issues a lock request, its entry is enqueued in the lock table and a pointer is established from the transaction's lock list to this entry. When a transaction performs a write operation, the corresponding log entry is inserted in the log table and the entry is also appended to the transaction's sequence of log records. Each entry in the log table has a *log sequence number* associated with it [BHG87, MHL+92, GR93]. When a transaction aborts, it executes the inverse of all its write operations by restoring the before-images stored in its log sequence. Thus, for example, if T_i in Figure 1 aborts it will restore the before-images of $w_i[x]$ and $w_i[y]$, the two log records in its sequence. Similarly, when a transaction commits, the log records in its sequence are flushed to the stable log, if they are not already there, and its lock entries are removed from the lock table.

The Lock Manager (the code that maintains the table) maintains the sequence of active operations on a given data item. This sequence is basically the list of lock requests for that data item. These lock entries do not contain any recovery related information (viz. the before-images or the values written by the transactions). The Log Manager (or code that maintains the log table and log structures), on the other hand, keeps track of the values written by transactions and their before-images but it does not contain the ordering information of active operations on a given object. We propose to integrate the two data-structures for implementing the new inverse writes. This is done by extending these data structures to implement the inverse write operations previously specified.

The data structures commonly used to implement the lock manager are general enough to support other locking protocols such as altruistic locking [SGA87] or ordered shared locks [AE90]. The main difference between these protocols is when and what entries are kept in the lock table. Strict 2PL, for instance, allows deleting an entry as soon as its transaction terminates. Other protocols may require to keep entries corresponding to committed transactions until all conflicts are resolved. Similarly, entries may correspond to requests or simultaneous accesses depending on the protocol. In what follows we assume any locking based concurrency control protocol. The lock manager data structures are used in a very general framework, where entries are kept as long as needed (without requiring the lock table to be persistent).

In our implementation (see Figure 3) we add a link from a write lock entry to its corresponding log record and vice-versa. The creation of this link requires nominal changes to the lock manager and the log manager, since in a conventional system the lock and logs entries are allocated and treated independently. A solution to this problem is, for instance, the following: given that the lock entry is created earlier than the log entry, once a log sequence number is associated with the operation, the lock manager can go back and store this number in the lock entry. There is no significant overhead associated with this modification, although experimental evaluation is needed to determine its exact

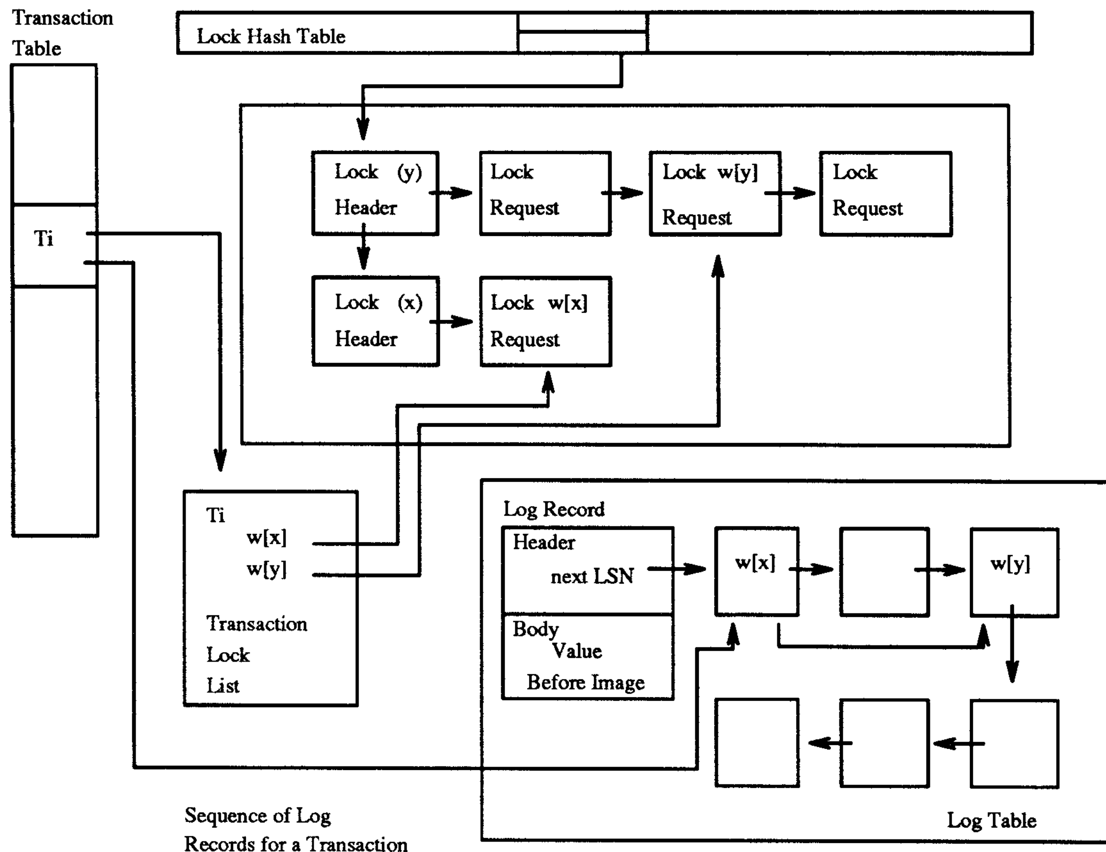

Figure 2: High level description of the main data structures of a transaction manager

impact on performance. With this arrangement, when a transaction is aborted, the sequence of log records related to this transaction are examined. For each log record corresponding to an update, the corresponding entry in the lock table is checked to determine which action to take: no-op or restore the value written by the last active or committed transaction.

In the lock queue of x we associate with each entry its status, which is either: *undo*, or *no-undo*. The traditional approach requires the undo of every aborted operation. With our new inverse operation, some operations do not need to be undone, hence the no-undo status. The entries also contain information regarding the status of the transaction: *active*, *committed* or *aborted*. An undo entry implies that the corresponding operation may require to be undone. A no-undo entry implies that the operation does not need to be undone if the transaction aborts. This entry codifies the status of other transactions accessing the same object. Note that some combinations are not possible. For instance it does not make sense to have an undo entry corresponding to a committed transaction.

When a lock entry is created its status is undo. When a transaction commits, it marks its lock entries as no-undo. It also marks all previous entries in the lock queue as no-undo. If some of the entries that are marked as no-undo belong to aborted transactions, they can be removed from the queue at this time. The entry corresponding to the committed transaction can also be removed. Any log entries corresponding to the transaction are flushed to stable storage if they are not already there.

When a transaction aborts, it first checks the status of its lock entries. If the entry is no-undo, the entry is simply discarded without further actions. This is because the value written by its operation has already been overwritten by a committed transaction and therefore, the inverse operation becomes a null operation. If the entry is undo, then a simple approach would be for the transaction to follow the theoretical definition of the inverse operation, i.e., check whether it is followed by an active lock entry and, if so, do nothing, otherwise restore all the previous before-images until an active transaction is found. This is unnecessary and the implementation can be optimized as follows. Assume the entry is $wl_i[x]$; there are two cases to consider:

- If there exists an active lock entry $wl_j[x]$ entry following $wl_i[x]$, the transaction marks its entry as aborted and continues to execute its next inverse operation. The rationale for not restoring the

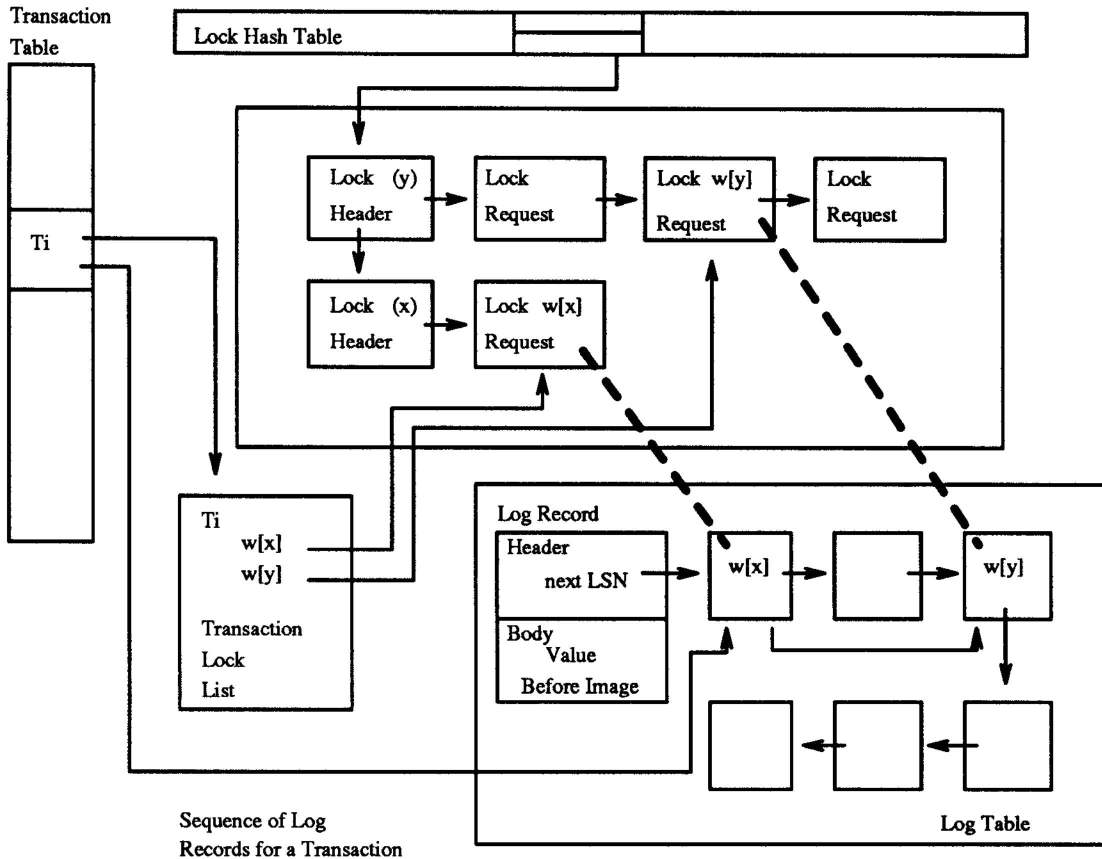

Figure 3: Modified data structures linking log records with lock entries

before-image and leaving the entry in the lock table is as follows: if T_j commits then the correct value of x is already in the database. On the other hand if T_j aborts, it may need to restore the before-image of x that is available in the log record of T_i.

- If $wl_i[x]$ is not followed by any active lock entries then T_i scans the lock queue backwards. The scan skips any entry that is aborted, until a committed or active entry is found (or the head of the queue is reached). Assume this entry is $wl_k[x]$, the entry following $wl_k[x]$ (or the head of the queue) is used to restore the correct before-image (i.e., restoring the value written by $w_k[x]$). This effectively undoes all aborted operations at the end of the lock queue, which can be deleted from it. Note that, during the backward scan, it is not possible to find an entry that is both aborted and no-undo since it would have been removed from the queue by the committed transaction that marked it as no-undo.

Note that this mechanism does not require to make the lock table persistent. The links are used for undoing aborted transactions as long as the system is operational. In the event of a system crash, standard recovery techniques can be used since the logical structure of the log has not been altered [BHG87, GR93].

This recovery mechanism allows concurrent write operations with no restriction on aborts. Furthermore, it is a generalization of the traditional recovery mechanism based on logs and before-images [GR93]. In particular, when there are at most one active write on any object, transactions terminate as follows. When a transaction commits, it simply flushes its log entry to the stable log, if this has not been done already, and deletes its entry from the lock table (in the above description all aborted entries from the head of the lock queue up to the committed transaction were deleted). Similarly, when a transaction aborts, it scans the lock queue to the right which is empty and to the left which is the head of the queue. Hence, it uses its own log record to restore the before-image. The above implementation requires nominal changes and does not incur significant overhead for traditional strict executions for transaction processing. However, if desired it can be exploited to provide more concurrency by allowing non-strict executions.

5 Discussion

The proposed inverse operation and the corresponding additions to the lock and log managers minimize the

restrictions on locking based concurrency control protocols, thus potentially allowing them to accept all serializable and recoverable executions. In particular, for write-write conflicts recovery is always possible, regardless of the order in which the transactions involved terminate. This implies that now the restrictions on an execution are imposed by the concurrency control protocol and not by the underlying recovery mechanism. For applications that require more concurrency this is an important advantage, since the flexibility provided by our new inverse operation does not restrict concurrency. There is, however, another point to consider. Concurrency control protocols, such as strict 2PL, often discard information about a transaction as soon as it commits. Using this new approach, information about committed transactions may need to be retained [HY86] in the lock table. However, this information is also kept in traditional protocols because the transaction's commitment would have been delayed. The only difference is that we allow the transaction to commit, which can be notified to the user, and keep the relevant information around until all conflicts have been resolved. Concurrency control protocols may not use all the potential offered by the recovery mechanism, however, now the restrictions are imposed from the concurrency control point of view and not from the underlying recovery mechanism. Finally, it is important to mention that with the new inverse operation, abort operations are not delayed, and there are no cascading aborts due to write-write conflicts, which avoids potential performance problems.

References

[AE90] D. Agrawal and A. El Abbadi. Locks with Constrained Sharing. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 85–93, April 1990. To appear in Journal of Computer and System Sciences.

[AEJ92] D. Agrawal, A. El Abbadi, and R. Jeffers. An Approach to Eliminate Transaction Blocking in Locking Protocols. In *Proceedings of the ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, pages 223–235, June 1992.

[AVA+94] G. Alonso, R. Vingralek, D. Agrawal, Y. Breitbart, A. El Abbadi, H.J. Schek, and G. Weikum. A Unified Approach to Concurrency Control and Transaction Recovery. *Proceedings of the 4th International Conference on Extending Database Technology, EDBT'94. An extended version will appear in Information Systems.*, 1994.

[BGRS91] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewisz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Transaction on Software Engineering*, 17(9), 1991.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley, 1987.

[FÖ89] A. A. Farrag and M. T. Özsu. Using Semantic Knowledge of Transactions to Increase Concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 1989.

[Gar83] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufman, 1993.

[Had88] V. Hadzilacos. A Theory of Reliability in Database Systems. *Journal of the ACM*, 35(1):121–145, January 1988.

[HR83] T. Härder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[HY86] T. Hadzilacos and M. Yannakakis. Deleting Completed Transactions. *Journal of Computer and System Sciences*, 38(2):360–379, April 1986.

[Lyn83] N. A. Lynch. Multilevel Atomicity – A New Correctness Criterion for Database Concurrency Control. *ACM Transactions on Database Systems*, 8(4):485–502, December 1983.

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method supporting fine-granularity Locking and Partial Rollbacks using Write-ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[PKH88] C. Pu, G.E. Kaiser, and N. Hutchinson. Split-Transactions for Open-Ended Activities. *Proceedings of the 14th Conference on Very Large Databases, Los Angeles, California, USA*, pages 26–37, 1988.

[SGA87] K. Salem, H. Garcia-Molina, and R. Alonso. Altruistic locking: A Strategy for Coping with Long-Lived Transactions. *In D.Gawlick, M. Hayne and A. Reuter, editors. Proceedings of the Workshop on High Performance Transaction Processing Systems, Springer-Verlag.*, pages 175–199, 1987.

[SSV92] D. Shasha, E. Simon, and P. Valduriez. Simple Rational Guidance for Chopping Up Transactions. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 298–307, June 1992.

[SWY93] H. J. Schek, G. Weikum, and H. Ye. Towards a Unified Theory of Concurrency Control and Recovery. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 300–311, June 1993.

6 Appendix

In conventional serializability theory, only committed projections of a history are considered. In recovery theory, the recovery classes are defined by also considering commit operations. However, schedulers execute many more operations than those that actually appear in the committed projection of a history. For instance, when a transaction is aborted, the scheduler performs undo operations that, in write-read model, are also write operations. The unified theory of [SWY93] is based on expanding a history by adding the undo operations corresponding to aborted an active transactions. The correctness criterion is defined over this expanded history, which allows to account both for concurrency control and recovery.

Aborted operations are handled by replacing them by undo operations to eliminate their partial effects. Each $w_i[x]$ operation, corresponding to an aborted or active transaction, is complemented in the expanded history with its inverse operation denoted by w_i^{-1}. Aborted transactions must be undone in any case. Active transactions are not necessarily undone but in case of failures the system should be able to undo any active operation. In a dynamic scheduler, active transactions are considered as potential aborted transactions and as such are treated to guarantee correct recovery. Read operations of aborted and active transactions are discarded from the expanded history since they do not affect the overall state of the database. This approach allows the expanded history to cover concurrency control (by monitoring interactions between normal operations) and recovery (by monitoring interactions with undo operations). To guarantee correct recovery it is assumed that active transactions will abort. The following definition of expanded histories is taken from [AVA+94], which is based on that proposed in [SWY93].

Definition 1: Expanded History

Let $H = (A, <)$ be a history, where A is a set of operations and $<$ is a partial order over those operations. Its expansion $\tilde{H}$, or expanded history $\tilde{H}$, is a tuple $(\tilde{A}, \tilde{<})$ where:

1. $\tilde{A}$ is a set of actions which is derived from A in the following way:
   (a) For each transaction $T_i \in H$, if $o_i \in T_i$ and o_i is not an abort operation, then $o_i \in \tilde{H}$
   (b) Active transactions are treated as aborted transactions, by adding an abort operation a_i for each active transactions $T_i \in H$.
   (c) For each aborted transaction $T_j \in H$ and for every write operation $w_j[x] \in T_j$, there exists an inverse write $w_j^{-1}[x] \in \tilde{H}$, which is used to undo the effects of the corresponding write operation. An abort operation $a_j \in H$ is changed to $c_j \in \tilde{H}$.
2. The partial order, $\tilde{<}$, is determined as follows:
   (a) For every two operations, o_i and o_j, if $o_i < o_j$ in H then $o_i \tilde{<} o_j$ in $\tilde{H}$.
   (b) For every pair of active transactions T_i, $T_j \in H$, their conflicting undo operations are in $\tilde{H}$ in a reverse order of the two corresponding write operations in H.
   (c) All undo operations of every transaction T_i that does not commit in H follow the transaction's original read-write operations and must precede c_i in $\tilde{H}$.
   (d) Whenever $o_n < a_i < o_m$ and some undo operation o_i^{-1} conflicts with o_m (o_n), then it must be true that $o_i^{-1} \tilde{<} o_m$ ($o_n \tilde{<} o_j^{-1}$).
   (e) Whenever $a_i < a_j$ for some $i \neq j$, then for all conflicting undo operations of T_i and T_j, o_i^{-1} and o_j^{-1}, it must be true that $o_i^{-1} \tilde{<} o_j^{-1}$.

□

Once the history has been expanded, it is necessary to examine if there are incorrect interleavings of operations. This is done by eliminating certain operations from the expanded history and checking the serializability of the resulting history. We say that history H is *reducible* (RED) [SWY93] if its expanded history $\tilde{H}$ can be transformed into a serial history by applying the following three rules:

1. **Commutativity Rule** if two operations o_i and o_j do not conflict in $\tilde{H}$, and there is no o_k such that $o_i \tilde{<} o_k \tilde{<} o_j$, then the ordering $o_i \tilde{<} o_j$ can be replaced by $o_j \tilde{<} o_i$ in $\tilde{H}$.

2. **Undo Rule** If $w_i[x]$ and $w_i^{-1}[x]$ are in $\tilde{H}$ and $w_i[x] \tilde{<} w_i^{-1}[x]$ and there is no o_j such that $w_i[x] \tilde{<} o_j \tilde{<} w_i^{-1}[x]$, then $w_i[x]$ and $w_i^{-1}[x]$ can be omitted from the expanded history.

3. **Null rule:** If T_i does not commit in H then all operations $r_i[x]$ can be removed for all x from $\tilde{H}$.

The class RED is not prefix-closed. Instead we use a more restrictive class: a history H is *prefix reducible* (PRED) if every prefix of H is reducible [SWY93].