[Alo93] G. Alonso. Integrated Protocols for Unified Concurrency Control and Recovery. Technical report, Department of Computer Science, University of California at Santa Barbara, 1993. In preparation.

[Bad79] D. Z. Badal. Correctness of Concurrency Control and Implications in Distributed Databases. In *IEEE Proceedings of COMPSAC Conference*, pages 588–593, November 1979.

[BBG89] C. Beeri, P. A. Bernstein, and N. Goodman. A Model for Concurrency in Nested Transactions Systems. *Journal of the ACM*, 36(2):230–269, April 1989.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, Massachusetts, 1987.

[BSW79] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering*, 5(5):203–216, May 1979.

[Cas81] M. A. Casanova. *The Concurrency Control Problem of Database Systems*, volume 116 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1981.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, November 1976.

[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.

[SR81] R. E. Stearns and D. J. Rosenkrantz. Distributed database concurrency control using before-values. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 74–83, June 1981.

[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.

[Ver78] J. S. M. Verhofstad. Recovery Techniques for Database Systems. *ACM Computing Surveys*, 10(2):167–196, 1978.

particular interpretation of the the transaction manager. Correctness criteria depend in many subtle ways on the actual implementation. The large number of underlying assumptions, often overlooked, constrain the correct executions. One of these assumptions is that the recovery and concurrency control mechanisms are implemented as independent modules (the lock manager and the log manager). Schek, Weikum and Ye [SWY93] have challenged this assumption and, as a result, have been able to propose a unified theory for concurrency control and recovery. However, they do so from a strictly theoretical standpoint and no tools or protocols are proposed for recognizing these correctness classes. In this paper, we extend this theory by providing efficient dynamic schedulers to recognize classes of histories based on the unified theory. The efficiency of the schedulers is crucial for the application of the theory in practice. Conventional serializability theory, in general, uses serialization graphs to check the correctness of a given schedule. Thus, a natural implementation of any serializability criteria is serialization graph testing. However, this approach is inefficient from a practical standpoint. A better approach, and as a matter of fact the *de facto* standard in transaction management is to use locks. As Figure 1 shows, the use of locks restricts the class of correct schedulers even further. Two Phase Locking accepts only those schedules that are order preserving serializable. Similarly, the proposed unified criteria could be implemented based on an expanded serialization graph testing. To implement efficient schedulers, however, we use locking mechanisms and obtain similar results. The classes of histories accepted by the schedulers are more restrictive than the theoretical concept, but the histories that are not accepted are only those that are non order preserving serializable, exactly the same restriction imposed by two phase locking. For practical purposes we further restrict our criteria to avoid cascading aborts. Previous results show that cascading aborts can seriously jeopardize the performance in a system.

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.

[AEL92] D. Agrawal, A. El Abbadi, and A. E. Lang. Performance Characteristics of Protocols with Ordered Shared Locks. *in IEEE Transactions on Knowledge and Data Engineering*, 1992. To appear.

However, if desired it can be exploited to provide more concurrency by allowing non-strict executions.

4.3 Locking Protocols

We now describe how the proposed recovery mechanism can be integrated with the different concurrency control mechanism proposed in Section 3. Recall that SOT was defined in terms of the termination of transactions. Since the proposed implementation of the inverse write operation relaxes the constraints on the aborts of transactions, we need to redefine SOT as follows.

Definition 3 Expanded SOT

A history, H, is SOT for expanded PRED if it is serializable and for every pair of transactions T_i, T_j:

1. if $w_i[x] < r_j[x]$ then T_i must commit before T_j commits
2. if $w_i[x] < w_j[x]$ then T_i commits before T_j terminates.

Note that the expanded definition of SOT maintains the restriction on the reads-from relation between transactions. On the other hand, the restriction for concurrent writes has been relaxed and only constraint is imposed to the log ordering property. In particular, if $w_1[x] < w_2[x]$ then expanded SOT disallows $c_2 < c_1$ and $c_2 < a_1$. With respect to reads-from relation the only added restriction is due to recoverability, i.e., a committed transaction cannot read non-existent data.

The serialization graph testing protocol can be easily modified to weaken the constraints on the termination of transactions. Similarly, the locking analogue, which uses ordered sharing, can be adapted to use the proposed recovery mechanism. In particular, in this case, there are no restrictions placed on transaction aborts of concurrent writers. Finally, to avoid cascading aborts, instead of requiring strict executions, i.e., writers hold locks with non-shared relationships with respect to both readers and writers, now writers hold locks with ordered sharing with respect to readers. In [AEL92], we show that such a relaxation results in significant performance gains in terms of both throughput and response time. If the blocking from writers to readers is eliminated by using before images [SR81], the performance can be further improved significantly [AEJ92].

5 Discussion

Concurrency control and recovery theory is often presented as a theoretically sound approach to the problem of transaction management. However, this theory has been built around a

entries in the lock and log tables are manipulated.

When a transaction T_i commits it begins by scanning its sequence of log records. The transaction flushes the log record $w_i[x]$ and checks the corresponding lock table entry $wl_i[x]$ using the links established between the log record and the lock entry. All entries preceding $wl_i[x]$ as well as $wl_i[x]$ in the lock table are deleted. This is because at the time when T_i commits all preceding transactions in the lock queue of x must have terminated (committed or aborted). This process is repeated for all log records of the committing transaction T_i.

When a transaction T_i aborts it scans the log records to execute inverse write operations. A simple approach would be for the transaction to follow the theoretical definition of the $inverse(w_i[x])$ from Section 4.1, i.e., restores its before-image, then the before-image of the previous entry in the queue and so on. This is unnecessary, and the implementation can be optimized as follows. For each log record $w_i[x]$, the transaction identifies the corresponding lock entry $wl_i[x]$ and the following two cases are considered:

- If there exists an active lock entry $wl_j[x]$ entry following $wl_i[x]$, the transactions marks its entry as aborted and continues to execute its next inverse operation. The rationale for not restoring the 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 to find an active entry $wl_k[x]$ or reaches the head of the lock queue. The entry following $wl_k[x]$ or the first entry in the queue is used to restore the before-image. Furthermore, all lock entries that follow $wl_k[x]$ or the head of the queue correspond to aborted transactions and are no longer needed. Hence, all such entries are deleted from the lock queue maintaining the invariant that an aborted lock entry must be followed by an active lock entry.

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 and deletes its entry from the lock table (in the above description all 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 the log record to restore the before-image. Hence, the above implementation requires nominal changes and does not incur any additional overhead for traditional strict executions for transaction processing.

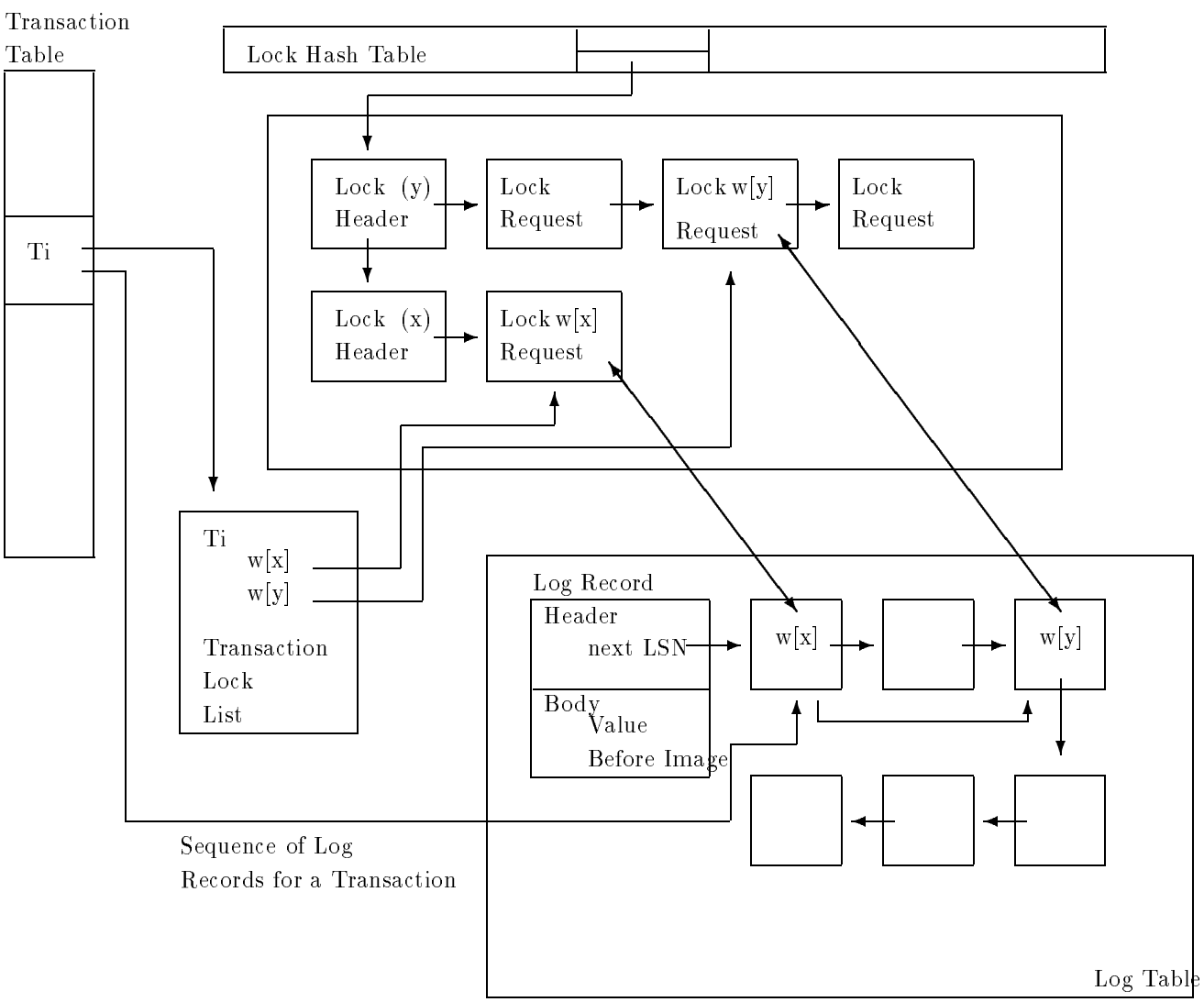

Figure 3: Modified data structures linking log records with lock entries

of a transaction processing system. In particular, there are three major components: the transaction table, the lock table, and the log table. 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. 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 2 aborts it will restore the before image for $w_i[x]$ and for $w_i[y]$, the two log records in its sequence. Similarly, when a transaction commits, the log records in its sequence are forced to the stable log and its lock entries are removed from the lock table.

We now extend the above structures to implement the inverse write operations specified above. We assume that we are still using a locking protocol such as ordered sharing [AE90] that permits concurrent write operations. Hence, the lock structure used to enqueue waiting entries can easily be adopted to implement ordered sharing. Note that in the original lock structure all lock entries except for the first one in a lock queue for an object are requests that are delayed. If locks with ordered sharing are used, these entries no longer correspond to pending requests rather correspond to lock requests that are granted. Furthermore, the ordering in the queue also depicts the order in which the locks have been granted, i.e., they capture the ordered shared relation.

Unlike in Figure 2, we assume that there exist links from a write lock entry to its corresponding log record and vice-versa. Generally, an empty log record is created at the time when a transaction executes its operation. In order to establish links between lock and log entries, we assume that the log record is created at the time the request for a write lock is granted to the transaction. There is no significant overhead associated with this modification and Figure 3 illustrates the new structure of a transaction processing system. 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 the status of each entry which is either: *aborted*, or *active*. An invariant that is maintained is that any aborted entry must be succeeded by an active entry. We maintain an aborted entry $wl_i[x]$, if at the time T_i aborted there existed a lock entry $wl_j[x]$ that follows $wl_i[x]$ and T_j is still active. Later if T_j is aborted it may need to restore the before-image of x stored in the log record $w_i[x]$. On the other hand, if T_j commits, $wl_i[x]$ will be deleted from the lock queue since the corresponding before-image is no longer needed. We now describe the termination of a transaction and discuss how the

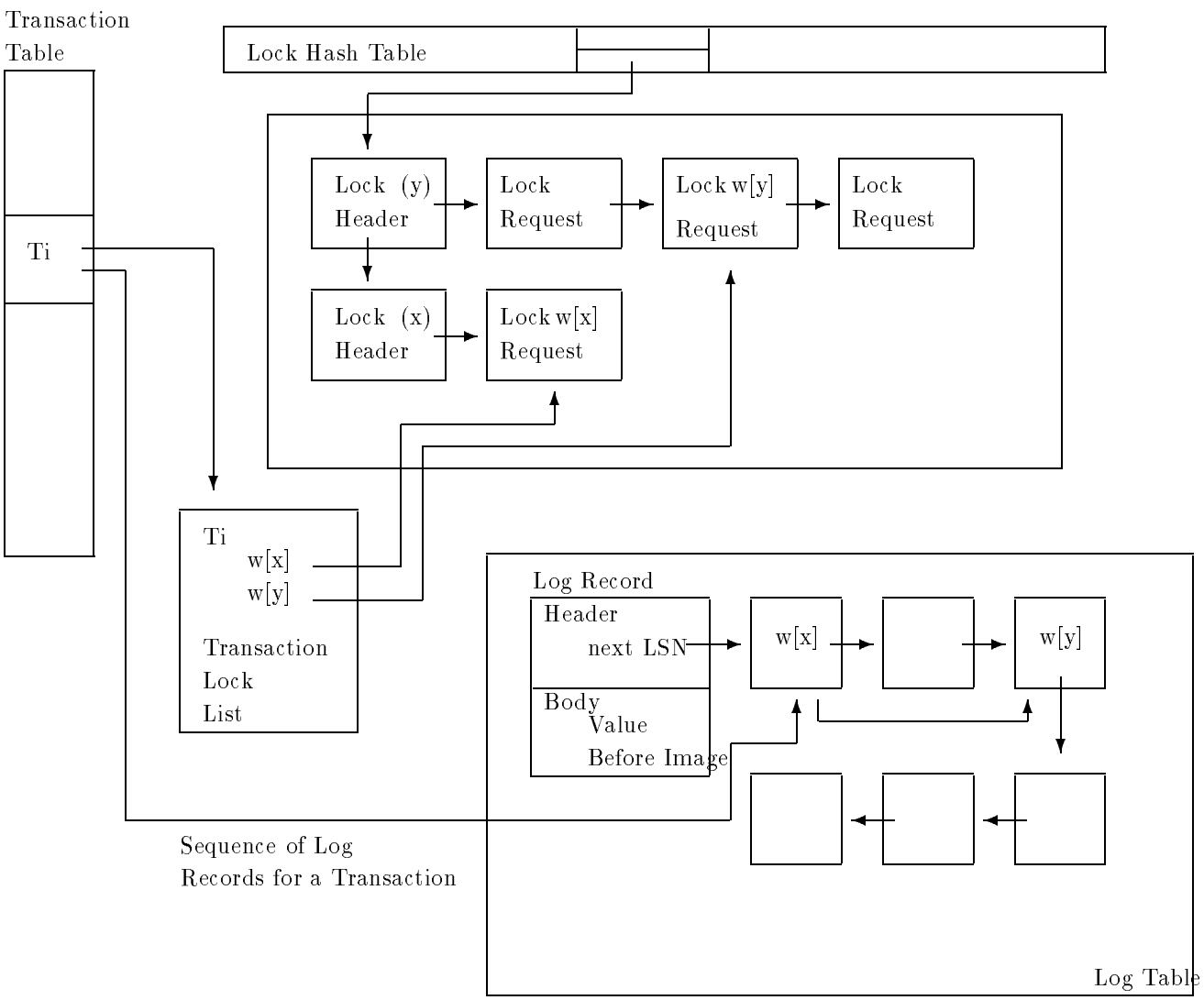

Figure 2: Data structures for the lock and log managers [GR93]

	Original History	Expanded History
1.	$w_1[x]w_2[x]c_1c_2$	$w_1[x]w_2[x]c_1c_2$
2.	$w_1[x]w_2[x]c_2c_1$	Violates Log Ordering
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.	$w_1[x]w_2[x]c_2a_1$	Violates Log Ordering
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.	$w_1[x]w_2[x]a_2a_1$	$w_1[x]w_2[x]\underline{w_2^{-1}[x]c_2w_1^{-1}[x]c_1}$

In the above expanded histories, the undo pairs have been redefined based on the new definition of inverse write operations. For example, in history 7, *UNDO* contains the two ordered pairs $(w_1[x], \epsilon_1[x])$ and $(w_2[x], w_2^{-1}[x]w_1^{-1}[x])$. Thus, the undo operations of aborted transactions are context dependent. In particular, the undo pair for $w_1[x]$ in history 8, is $(w_1[x], w_1^{-1}[x])$, i.e., the traditional before-image. Another distinction is that the termination of a transaction may be performed after the termination has been notified to the users. In history 7, $w_1^{-1}[x]$ is executed by the system following the actual abort.

The above expanded histories are valid under the original definition of PRED. However due to the new inverse write operations, the set of histories is a strict superset of the original class. The advantage of the proposed inverse write operation is that it permits concurrent write without imposing any restrictions on the aborts of transactions. In particular, the strange phenomenon of cascading aborts due to concurrent write operations no longer exists in the proposed approach.

4.2 Extending the Log-based Recovery Architecture

We now discuss how the specifications above can be realized by using the log-based recovery mechanism as well as a locking based concurrency control protocol. 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. The Lock Manager maintains the sequence of active operations on a given data item but the lock entries do not contain any recovery related information (viz. the before images or the values written by the transactions). The Log Manager has the advantage that it 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 new inverse writes.

In order to describe the new implementation, we briefly summarize the recovery implementation which is borrowed from [GR93]. Figure 2 illustrates the different components

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 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 the before-image of x from its log record 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 $T'_k, \ldots, T'_i$. More formally, $inverse(w_i[x])$ is defined as follows:

$$inverse(w_i[x]) = \begin{cases} \epsilon_i[x] & \text{if } \exists \text{ active } T_j \text{ such that } w_i[x] < w_j[x] \\ w_i^{-1}[x] w_{i'}^{-1}[x] \ldots w_{k'}^{-1}[x] & \text{otherwise} \end{cases}$$

Note that as was the case in [SWY93] $w_i^{-1}[x]$ is the before-image write in the log record for $w_i[x]$. Furthermore, in [SWY93], $inverse(w_i[x])$ is the same as $w_i^{-1}[x]$ and the set *UNDO* contains ordered pairs $(w_i[x], inverse(w_i[x]))$. Finally, the restriction that the inverse write operations precede the commit operation of the corresponding transaction is not enforced. In a sense, the termination of an aborted transaction may be announced early to the user whereas the complete termination of the aborted transaction may take place later in the system. The examples below illustrate this phenomenon.

We start by considering the histories with two concurrent write operations and their expansion under PRED.

	Original History	Expanded History
1.	$w_1[x]w_2[x]c_1c_2$	$w_1[x]w_2[x]c_1c_2$
2.	$w_1[x]w_2[x]c_2c_1$	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.	$w_1[x]w_2[x]c_1a_2$	$w_1[x]w_2[x]c_1\underline{w_2^{-1}[x]c_2}$
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.	$w_1[x]w_2[x]a_1a_2$	not in PRED
8.	$w_1[x]w_2[x]a_2a_1$	$w_1[x]w_2[x]\underline{w_2^{-1}[x]c_2w_1^{-1}[x]c_1}$

It can be easily verified that histories 2, 3, 4, and 7 will not be accepted by PRED as defined in [SWY93]. For example, the expanded history of case 3 is:

$$w_1[x]w_2[x]\underline{w_1^{-1}[x]c_1}c_2$$

This history not reducible to a serial history. However, using the proposed implementation of inverse write operations the expanded histories are as follows:

4.1 Expanding PRED

As discussed in the previous section, PRED places restrictions on the aborting of transactions. For example, 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 aborts. 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. For example, consider the following expanded history:

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

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 $w_2[x]$. Schek, Weikum, and Ye [SWY93] developed a unified theory of concurrency control and recovery assuming the above implementation of inverse write operations.

We start with an overview of the extended implementation of the inverse write operations. We propose to implement an inverse of a write operation $w_i[x]$ as follows:

- If there exists a write operation $w_j[x]$ of transaction T_j that succeeds $w_i[x]$ and T_j has not yet terminated 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[2].

In order to use the traditional restart mechanism after failure, we impose the following restriction on the underlying concurrency control mechanism:

- Log Ordering. The order of write operations in the log corresponds to the commitment order of the corresponding transaction.

Thus if $w_i[x] < w_j[x]$ then T_i must commit before T_j commits.

In this section, we propose a new implementation of inverse write operations that can be integrated with the unified theory of concurrency and recovery. We define the semantics of

[2]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.

of the history. The delayed commitment and abortion rule ensures that the expanded history is serializable. This protocol accepts a strict subset of PRED. However, we note that the PRED histories that are not accepted by this protocol are not ordered preserving serializable histories [Alo93][1].

Although the class PRED seems to be the weakest class of executions that can be allowed using the traditional recovery mechanisms, it has several undesirable properties. In particular, PRED executions may result in cascading aborts, which significantly reduce overall performance. Traditionally, cascading aborts occur when a transactions are allowed to read uncommitted data. However, in the unified theory, PRED executions may cause another type of cascading aborts that arise due the underlying recovery mechanism. In particular, consider the following execution:

$$w_1[x]w_2[x]a_1$$

The abort of T_1 forces the abort of T_2. In order to avoid these problems, a restricted form of ordered sharing can be used where non-shared locks are used from writers to both readers and writers and locks are released only when transactions terminate. Hence, a non-shared relationship from write locks to both read and locks is used in this protocol. For write after read conflicts, locks with ordered sharing are used and transactions must adhere to the ordered sharing rule. Since write locks are held until commit, transactions cannot read or write uncommitted data. Hence all executions resulting from this protocol avoid cascading aborts. Note that this restriction, i.e., non-shared locks and releasing write locks at commit results in strict executions. In the next subsection, we explore techniques to eliminate this restriction and still avoid cascading aborts.

4 Extending the Log-based Recovery Architecture

We now explore alternate implementation of the inverse write operations and evaluate its impact on the traditional log-based recovery architecture. The major motivation for extending the semantics of inverse write operations is to permit non-strict executions. Unlike the notions of recoverability and avoids-cascading-aborts (which are based on the execution semantics of transactions), the notion of strict executions is developed due to operational considerations [Had88]. That is, it is presumed that inverse write operations (UNDO) can only be supported by restoring before-images and hence transactions executions must be strict. We propose to allow non-strict executions thus permitting concurrent write operations in the system. This results in an expansion of the class PRED not by modifying the definition rather by modifying the underlying recovery mechanism.

[1]Order preserving serializable histories maintain the order of non-interleaved transactions in the equivalent serial history[BSW79, BBG89].

3.2 Efficient Locking Protocols for PRED

Although a serialization graph testing approach recognizes, in general, all correct executions, it is inefficient. Many widely used concurrency control protocols use *locking* as a basic primitive for synchronization. Traditionally, there are two types of relationships between locks: shared and non-shared. For example, read locks can be shared but a write lock cannot be shared with any other lock. Two phase locking [EGLT76] is the most widely accepted concurrency control protocol. Recently in [AE90], a new locking primitive was introduced that allows a new relationship, referred to as *ordered sharing*. Ordered sharing can be used with two phase locking to eliminate the blocking between read and write operations. For example, in order to eliminate read-write blocking, a transaction T_j can be granted a write lock on an object even if a transaction T_i holds a read lock on the same object. We say that there is an *ordered shared relationship* from T_i's read lock to T_j's write lock. The advantage of using ordered sharing is that it eliminates blocking between read and write operations, but can be used to restrict the execution of commit or abort operations. The advantage of using SOT instead of PRED is that the definition of SOT is intuitively closer to locking protocols than that of PRED. To ensure serializability with ordered termination, the two phase locking with ordered sharing must observe the following rules:

- Lock Rule. A transaction must obtain a read (write) lock on an object before executing a read (write) operation on that object.
- Two Phase Rule. A transaction may not acquire any locks once it has released a single lock.
- Lock Acquisition Rule. If T_j acquires a lock with an ordered shared relationship with respect to a lock held by another transaction T_i, the corresponding operation of T_j must be executed after that of T_i.
- Lock Relinquish Rule. If transaction T_j acquired a lock with an ordered shared relationship with respect to a lock held by transaction T_i and T_i has not released *any* of its locks, then T_j cannot release any of its locks.
- Delayed commitment and abortion. If T_j reads from T_i then T_j can commit only if T_i has committed and if T_j overwrites a value written by T_i then T_j does not commit before T_i terminates and T_i does not abort before T_j terminates.

The ordered sharing of locks is necessary because PRED and SOT do not impose any restriction in the order in which the operations are executed, as long as the overall history is serializable. With basic two phase locking, histories such as $w_1[x]w_2[x]w_1[y]w_2[y]$ are not possible, however they are correct and valid according to PRED and SOT. The rules of two-phase locking with ordered sharing guarantee the serializability of the committed projection

1. If $w[x]$ or $r[x]$ are added to H_{n-1}, H_n is PRED. This is because if the added operation causes a cycle in $SG(H_n)$, then H_n will no longer be SOT. Furthermore, a read operation of an active transaction can never create a cycle in $SG(\tilde{H}_n)$, since they are dropped from the expanded history. A write operation of an active transaction at the end of H_n is followed, in the expanded history $\tilde{H}$, by its corresponding undo operation, $w^{-1}[x]$, and both are then dropped from the expanded history. Therefore, adding a read or a write operation to PRED history following SOT rules always results in a PRED history.

2. If c_i is added to H_{n-1} and a cycle appears in $SG(\tilde{H}_n)$, this can only be because for some active transaction, T_j, there is an undo operation, $w_j^{-1}[x]$, that can not be reduced any more. For this to happen, $w_j^{-1}[x]$ must conflict with some $o_i[x]$, with $o_i[x] < w_j^{-1}[x]$ but if this is the case, then there must be a $w_j[x]$ such that $w_j[x] < o_i[x]$. Since T_j is still active, SOT rules specify that T_i can not commit yet, thus c_i could have not been added to H_{n-1}.

3. If a_i is added to H_{n-1} and a cycle appears in $SG(\tilde{H}_n)$, this can only be because for some undo operation, $w_i^{-1}[x]$, there is a conflict with some $o_j[x]$, with $o_j[x] < w_i^{-1}[x]$. But in this case there must be a $w_i[x]$ such that $w_i[x] < o_j[x]$. We consider the two cases when $o_j[x]$ is $r_j[x]$ or $w_j[x]$.

   Assume $w_i[x] < r_j[x]$. T_j must have not committed by definition of SOT histories. If T_j is aborted or active, the expanded history would not include $r_j[x]$, hence the history must be PRED.

   Assume $w_i[x] < w_j[x]$. Again, T_j could not have committed or be active when T_i aborts from the definition of SOT histories. If T_j is aborted, its undo operations can be reduced in H_{n-1} and therefore they can also be reduced in H_n. Hence, adding a_i can not create new conflicts.

Hence, we conclude that SOT is equivalent to PRED. □

This characterization of PRED in terms of termination rules allows us to derive a serialization graph-based testing protocol [Bad79, Cas81] for recognizing PRED executions. In particular, a scheduler would maintain the serialization graph and each time an operation is submitted, the appropriate nodes and edges are added to the graph. When a transaction T_j wants to commit, the graph is tested for any active transactions T_i such that T_j either read-from T_i or T_j overwrote the value written by T_i. In both cases T_j must wait for T_i to terminate. If, on the other hand, T_j wants to abort, the graph is is tested for any active transactions T_k such that T_k overwrote the value written by T_j. In this case T_j must wait for T_k to terminate. It is easy to show that this scheduler recognizes all PRED executions. Furthermore, it can be shown that no deadlock can occur [Alo93].

the order of termination is irrelevant, since the final value after both terminate will always be the value written by T_1. Finally, it is impossible to allow T_1 to abort, and T_2 to commit (recall, we are assuming that T_1 had not terminated when T_2 executed its write operation). Either order of termination would results in x containing the value of x before both write operations and not the value written by T_2. Third, we note that no termination restrictions are imposed if an active read operation precedes a write operation.

Based on the above, it is clear that to use before images in the traditional way, if two transactions T_1 and T_2 execute $w_1[x] < r_2[x]$ or $w_1[x] < w_2[x]$, then if both transactions commit, then $c_1 < c_2$. In addition, if $w_1[x] < w_2[x]$ then if both transactions abort, $a_2 < a_1$; if T_1 aborts then T_2 cannot commit and finally, if T_1 commits and T_2 aborts, then there is no restriction on the termination order. The following definition captures these constraints.

Definition 2 Serializability with Ordered Termination (SOT)

A history, H, is SOT 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 then T_i commits before T_j commits
2. if $w_i[x] < w_j[x]$ in H then either T_i commits before T_j terminates or T_j aborts before T_i terminates.

Note that in the definition of SOT, serializability is with respect to the committed projection of the history, while the constraints on the termination of transactions are in terms of both committed as well as aborted transactions in the history. We now show that SOT is equivalent to PRED.

Theorem 1 *The class of histories SOT is equivalent to the class of histories PRED.*
Proof: First we prove that PRED $\subset$ SOT. Assume this is not the case, i.e., there is a history H such that $H \in$ PRED but $H \notin$ SOT. There are three cases to consider. The first case if H is not serializable, then H can not be PRED by definition, hence a contradiction. In the second case, let H be serializable and there exist $w_i[x]r_j[x]$ but T_j commits before T_i commits. In this case, H is not PRED since the prefix $w_i[x]r_j[x]c_j$ is not PRED. Finally, H is serializable and there exist $w_i[x]w_j[x]$. There are two cases to consider: (a) if T_j commits before T_i terminates, then H is not PRED since the prefix $w_i[x]w_j[x]c_j$ is not PRED; (b) if T_i aborts before T_j terminates, then H is not PRED since the prefix $w_i[x]w_j[x]a_i$ is also not in PRED. Hence if a history H is in PRED, H must be in SOT.

Now we prove SOT $\subset$ PRED. This will be done by induction. The base case is an empty history. An empty history is both PRED and SOT. The hypothesis is that a history H_{n-1} that is SOT, is also PRED. In the induction step we add one of the operations: read, write, commit, or abort to the PRED history H_{n-1}. We show that the resulting history H_n is PRED as follows.

that the definition is not based on reductions or analysis of all prefixes, and hence is easier to use in the development of efficient methods for recognizing PRED. We then present the notion of ordered shared locks and describe a protocol that recognizes PRED executions. Finally, we point out that PRED allows cascading aborts, which are undesirable from a performance point of view, and describe a modified protocol that avoids cascading aborts.

3.1 Serializability with Ordered Termination

We start this section by describing in more detail the standard recovery mechanism [Ver78, HR83], which is also assumed in the model proposed by Schek, Weikum and Ye [SWY93]. In this model, a *log* is maintained on stable storage for recovery purposes. The log is composed of a sequential set of records, each corresponding to a write operation. The log has the following properties:

1. Each log record must contain enough information to either undo or redo the corresponding operation.

2. The log is a read only structure, except for the last entry. Access to the last entry is mutually exclusive.

The log is used for two main purposes. First, when a transaction aborts, it is used to retrieve the before-image for reinstallation in the database. Second, after recovery from failures, the log is traversed by a special restart routine that installs after-images for all committed transactions and before-images for all aborted or active transactions. Since the log is traversed sequentially, it is assumed that write operations are ordered in the same order in which transactions commit.

We now consider the various restrictions that are imposed by the log structure on the order in which transactions may terminate. In particular, we are interested in the restrictions, if any, imposed on the commitment or abortion of transactions when they execute conflicting operations, i.e., write read, write write and read write conflicts.

First, consider an active write operation $w_1[x]$ followed by a read operation $r_2[x]$. From the definition of recoverability, we know that T_2 cannot commit before T_1, otherwise, if T_1 aborts, then T_2 has read a nonexistent value. Hence, c_1 must precede c_2. This is also the case of cascading aborts, since it is impossible for T_1 to abort and for T_2 to commit. Second, consider an active write operation $w_1[x]$ followed by another write operation $w_2[x]$ before T_1 has terminated. Since, the restart operation assumes that write operations are ordered in the same order in which the corresponding transactions committed, it follows that c_1 must precede c_2. If both transactions abort, then to ensure that the final value of object x is the value before $w_1[x]$ was executed, a_2 must precede a_1 (otherwise, transaction T_2 would simply restore into x the value written by T_1, which is incorrect). If T_2 aborts and T_1 commits, then

(b) All inverse write operations follow the original operations of the transaction and precede the transaction's commit operation.

(c) The inverse write operations corresponding to active transactions in H, are added in $\tilde{H}$ at the end of the history.

(d) The inverse write operations are placed in $\tilde{H}$ in the reverse order of the corresponding original actions.

(e) *UNDO* contains all pairs $(w_i[x], w_i^{-1}[x])$ where w_i^{-1} is the inverse write operation of $w_i[x]$.

(f) *NULL* is the set of read operations of transactions that are aborted in H or not terminated in H.

A history, H, is an *expanded serializable* (XSR) history if its expansion $\tilde{H}$ is serializable. It is easy to see that XSR $\subset$ SR. However, the class of XSR histories can not be compared with the classes RC, ACA or ST. A further refinement of this definition yields a more elaborated criterion. A history, H, is *reducible* (RED) if its expansion $\tilde{H}$ can be transformed to a serial history by applying the following three transformation rules finitely many times.

1. **Commutativity Rule** if two operations o_i and o_j do not conflict, 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 $o_j \tilde{<} o_i$

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 Action Rule** read operations of aborted or active transactions in H can be omitted from $\tilde{H}$.

It can be shown that XSR $\subset$ RED. Unfortunately, the class of RED histories can not be compared to neither SR, RC, ACA nor ST. This is solved by further restricting this class of histories. In particular, a history, H, is *prefix reducible* (PRED) if every prefix of H is reducible.

If we use the term SR-RC for the class of histories that are both serializable and Recoverable, it can be shown that PRED $\subset$ SR-RC. Similarly SR-ST $\subset$ PRED. PRED has been proposed as a unified correctness criterion. Intuitively, PRED is the largest class of executions that is correct from both the concurrency control and the recovery points of view.

3 Efficient Recognition of PRED

In this section, we will explore efficient, locking-based protocols for recognizing PRED. We start by proposing a new, alternative correctness criterion called *Serializability with Ordered Termination* (SOT), and show that SOT is equivalent to PRED. The advantage of SOT is

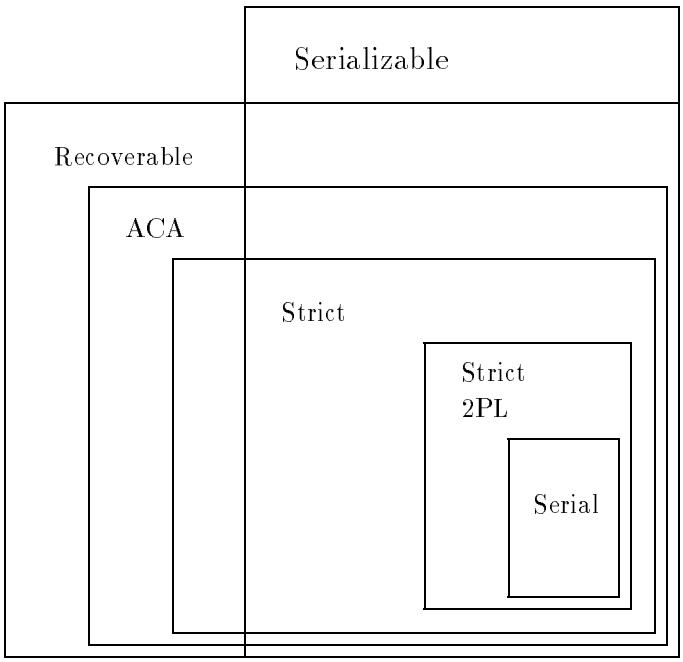

Figure 1: Relationships among various classes

To guarantee correct recovery, it is assumed that the active transactions will abort, thus they will be added to the history as aborted transactions. A history that contains all this information is referred to as an *expanded history*.

Definition 1 Expanded Histories [SWY93]

Let $H = (A, <)$ be a history. Its expansion $\tilde{H}$, or expanded history $\tilde{H}$, is a quadruple $(\tilde{A}, \tilde{<}, UNDO, NULL)$ where:

1. $\tilde{A}$ is a set of actions which is derived from A in the following way:

   (a) For all committed transaction $T_i \in H$, if $o_i \in T_i$ then $o_i \in \tilde{H}$

   (b) For all aborted transactions $T_j \in H$, if $o_j \in T_j$ then $o_j \in \tilde{H}$. For every write operation $w_j \in T_j$, there exists an inverse write $w_j^{-1} \in \tilde{H}$, which is used to undo the effects of the corresponding write operation. And $a_j \in H$ is changed to $c_j \in \tilde{H}$.

   (c) Active transactions are treated as aborted transactions, by adding the corresponding inverse write operations and a commit operation at the end.

   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 $\tilde{H}$.

2 Model

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 total order of read and write operations and 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 partial 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 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. In order to ensure that aborting a transaction does not influence previously committed transactions, we must require that for every transaction T that commits, its commit operation follows the commit of every other transaction from which T read. Such executions are called *recoverable* (RC) [Had88]. Recoverability, however, does not guarantee freedom from cascading aborts. Cascading aborts occur when a transaction reads from another uncommitted transaction that later aborts, forcing the former transaction to abort as well. Cascading aborts can be prevented by requiring transactions to read only committed values. Executions that satisfy this requirement are said to *avoid cascading aborts* (ACA) [Had88]. Finally, if the database uses *in-place* updating, it is convenient to implement the abort mechanism by restoring the *before-images* of all aborted writes [Ver78, HR83, BHG87]. In order to use before-images to eliminate the effects of aborted transactions, read and write operations on a data object x must be executed only on committed values. Executions with this property are called *strict* (ST) [Had88]. Figure 1 illustrates the relationship among the various serializability and recovery classes.

Instead of finding the intersection of two orthogonal criteria, e.g., serializability and strict or serializability and recoverability, Schek, Weikum and Ye [SWY93] propose to define a unified criterion for both concurrency control and recovery. In this way, the class of correct histories may be expanded. To do so, the histories must contain information about traditional conflicts (for concurrency control) and about the commit and abort operations that take place (for recovery). This means we have to add abort-related operations for all transactions that are aborted and all the transactions that are active (not yet terminated).

systems. For example, strict two phase locking and log based recovery with before images has become the de facto standard.

The reason for this orthogonality arises due to the different interpretations of transaction executions. In concurrency control theory, the criteria are defined over the committed projection of the execution, i.e. only those committed transactions are considered. Recovery theory takes care of the aborted transactions. Both criteria impose restrictions on the transactions, however they impose different restrictions: concurrency control concentrates on conflicting operations involving read and write operations while recovery focuses on the ordering of begin, abort and commit operations.

Recently, Schek, Weikum and Ye [SWY93] have developed a unified theory for concurrency control and recovery. In this theory, the traditional concurrency control theory is expanded to incorporate aborted transactions. As a result, the extended theory can be used to determine whether a given execution is correct both from a concurrency control as well as recovery points of view. A hierarchy of correctness classes is defined and the authors introduce a particular class, called *prefix-reducibility* (PRED). PRED is argued to be the most permissive class of executions that are intuitively correct accounting for both concurrency and recovery in the system. The authors state that an important research topic is to further explore the class of prefix-reducible executions, trying to identify a maximum subclass that can be recognized by a dynamic scheduler (with acceptable efficiency).

In this paper, we discuss the characteristics of such a scheduler and develop efficient schedulers based on a new class of locks. We start by proposing an equivalent definition for PRED. The original definition of PRED was stated in a recursive manner, which hinders its use for the development of dynamic schedulers. Our definition is procedural, and provides us with an easy, graph testing protocol for recognizing all PRED executions. However, serialization graph testing is known to be inefficient, and since locking is widely used for concurrency control, we propose a locking based protocol to recognize a subclass of PRED. The unified theory relied on a specific implementation of the recovery mechanism, viz., logs and before images. We develop an alternative recovery mechanism, which also uses logs and before images, but allows more concurrency among transactions. This recovery mechanism is integrated with efficient locking based protocols.

The paper is organized as follows: Section 2 introduces the model and notation that will be used throughout the paper. In Section 3 we present an efficient locking protocol implementing a subclass of PRED. In Section 4 we propose a new implementation of undo operations that can be integrated with the unified criteria discussed in Section 3. Section 5 concludes the paper.

A Unified Implementation of Concurrency Control and Recovery *

G. Alonso D. Agrawal A. El Abbadi
Department of Computer Science
University of California
Santa Barbara, CA 93106

Abstract

Transaction management in shared databases is generally viewed as a combination of two problems, concurrency control and recovery. These two problems are usually considered as orthogonal and, consequently, the correctness criteria derived from them are incomparable. Recently, a unified theory for concurrency control and recovery has been proposed along with a unified correctness criteria called *prefix-reducibility*. Developing efficient protocols for accepting prefix-reducible executions was left as an open problem. In this paper, we extend this theory to facilitate the design of efficient dynamic protocols that accept prefix-reducible executions. Furthermore, we develop a unified implementation of the log and lock managers required to support this new class of schedulers.

1 Introduction

Conventional transaction management in shared databases is traditionally viewed as a combination of two orthogonal problems. One is to guarantee correctness when several transactions are executed concurrently. This is ensured by the concurrency control component of a transaction processing system. The other is to ensure the correctness of the database in spite of transaction aborts and system failures. This is ensured by the recovery mechanism. In both cases there are several criteria that impose increasing restrictions on executions, resulting in a hierarchy in which only the most restrictive criteria are chosen for implementing concurrency control and recovery mechanisms. Since concurrency control and recovery have been considered orthogonally, solutions have been devised independently. The result is that the different criteria are incomparable except for the most restrictive criterion in each component. Hence the most restrictive solutions are integrated to design transaction management

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