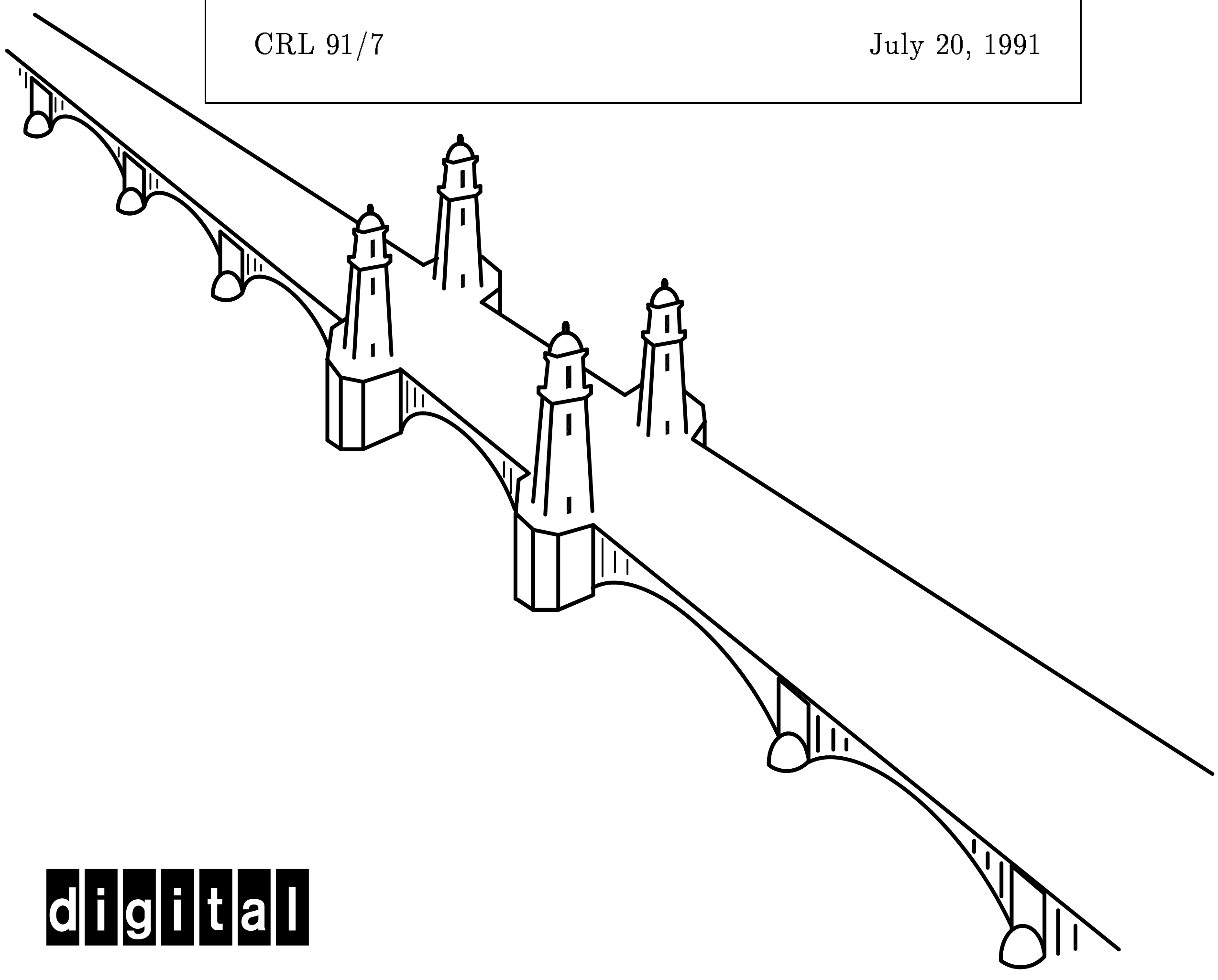
MLR: A Recovery Method for Multi-level Systems

David B. Lomet

Digital Equipment Corporation
Cambridge Research Lab

CRL 91/7 July 20, 1991



digital

CAMBRIDGE RESEARCH LABORATORY
Technical Report Series

digital™

Cambridge Research Laboratory
One Kendall Square
Cambridge, Massachusetts 02139

MLR: A Recovery Method for Multi-level Systems

David B. Lomet

Digital Equipment Corporation
Cambridge Research Lab

CRL 91/7 July 20, 1991

Abstract

To achieve high concurrency in a database system has meant building a system that copes well with important special cases. Recent work on multi-level systems suggests a systematic way of providing high concurrency. A multi-level system using locks permits restrictive low level locks of a sub-transaction to be replaced with less restrictive high level locks when a sub-transaction commits, hence enhancing concurrency. This is possible because sub-transactions can be undone by executing high level compensation actions rather than by restoring a prior lower level state. We describe a recovery scheme, called Multi-Level Recovery (**MLR**) that logs this high level undo operation with the commit record for the subtransaction that it compensates, posting log records to only a single log. A variant of the method copes with nested transactions, and both nested and multi-level transactions can be treated in a unified fashion.

Keywords: multi-level transactions, logging, recovery, hierarchical systems

1 Introduction

1.1 Precursor Multi-level Methods

Most of our understanding of concurrency control and recovery, e.g. two phase locking [5], serializability theory [3], and before-image/after-image recovery [9] is based on treating disjoint resources in a uniform "single level" way. But, when examined in detail, database systems support multiple levels of abstraction and exploit the levels to improve concurrency. Below are a few of the multi-level cases exploited by some existing systems.

- Mutual exclusion via a semaphore guarantees atomicity of page writes.
- B-tree concurrency methods hold locks on index nodes only for the duration of a tree structure change [15, 12].
- The ARIES recovery method [14] logs operations that update disk pages, but are re-interpreted (if necessary) during recovery to undo a record update even if the record has moved to a different page.

1.2 Explicit Multi-Level Systems

Multi-level transactions were made explicit in [18, 19, 2]. Theoretical work was reported in [1, 10, 13], and special cases were described in [6, 9, 17, 14]. An implementation of multi-level system recovery, restricted to two levels, is described in [20]. Here we briefly characterize multi-level systems.

We begin by describing "layers" of a multi-layer system, and then distinguish two kinds of layers, a "multi-level" layer and a nested transaction layer. Each layer of a multi-layer system realizes a set of abstract states, each represented by a number of lower layer states. Each layer sees state transitions in the form of atomic operations provided by the next lower layer, and uses them to provide atomic operations to the next higher layer. A layer thus transforms the sequence of operations that were supplied to it into operations that it supplies to its users.

A multi-level **ML** system exploits the layers of abstraction common in most systems to enhance concurrency. Concurrency control and recovery are concerned ultimately with ensuring that the system behavior at the highest layer is correct. By defining a layer to be a **level** of a multi-level system, an ML system can exploit the flexibility of choosing among the lower layer states that realize a desired high layer state. ML concurrency control need only ensure that committed transactions finish in one of the low level states that realizes a high level state providing high level serializability. More importantly, on abort, a level of an ML system need only be returned to the starting high level state, but not necessarily the starting low level state.

The lowest layer of an ML system directly provides abstractions upon which succeeding layers can be built. The system also provides a set of services to the implementor of a level. If an implementor desires to define his layer as a level, these enable the tailoring of concurrency control and recovery to the needs of the level. An implementor can also ignore these services and use nested transactions, thus exploiting the capabilities provided by the recoverable abstractions of the lower levels. Such a nested transaction layer is not a level.

The level-oriented services provided an implementor permit, e.g., the definition of lock modes, the requesting of locks, and the specification of compensation operations. The ML system framework is responsible for the interactions between layers, for controlling implicit release or retention of locks, for staging the recovery process that aborts transactions or recovers from system crashes, for assuring that these operations are applied only to an "operation consistent" database state and that idempotency of recovery is assured, etc.

An operation's level number is determined when it is defined. A multilevel layer's (see section 2.3.1) level is L_{i+1}, where L_i is the level of the layer for the operations that it uses. We denote operations at a level L_i as OP_is. We require here that all operations of a layer be at the same level, i.e. that the leveling be uniform, as it would be in a hierarchy of "virtual machines" as described in [18]. (In section 6, we discuss briefly a "layers of abstraction" generalization that does not require this uniformity of level.) Lowest level operations are defined to be L_0. (Table 1 contains a list of the notation used in this paper.)

1.3 Our Effort

In this paper we present an algorithm for recovery in multi-level systems. Our recovery method, called MLR, is reminiscent of ARIES/NT [16] and enjoys many of the same desirable features of that scheme. MLR copes

Table 1: List of Acronyms

L_i	level i of a multi-level system
OP_i	normal forward database operation at L_i
OP_i^{-1}	inverse(undo) operation for OP_i
ML	multi-level
MLT	multi-level transaction
NT	nested transaction (transaction itself or system)
CT	compensation transaction
$LOCK_i$	a lock at L_i
CLR	compensation log record (for recording undo progress)
LL_i	level list (of undo actions for L_i)
$TransT$	Transaction Table

correctly with both nested transactions(**NT**s) and with multi-level transactions(**MLT**s). It uses a single log for all levels, and thus presents a unified way of dealing with all levels.

Multi-level systems are discussed in section 2. Section 3 introduces the fundamentals of layered recovery. Section 4 shows the way that the log is used to prepare for MLR recovery, while section 5 describes the rolling back of transactions both during normal operation and as a result of system crash recovery. Section 6 discusses our results and suggests some generalizations.

2 Multi-Level Systems

2.1 High Level Compensation

The essence of multi-level recovery for a new level is that a completed OP_{i+1} is "undone" by the execution of its inverse operation (OP_{i+1}^{-1}) called a **compensation operation**. The OP_{i+1}^{-1} returns a system to a high level state in which the effect of the original OP_{i+1} has disappeared. OP_{i+1}^{-1}s require the re-acquisition of $LOCK_j$s, where $j < i+1$, and some of these may conflict with (possibly implicit) $LOCK_j$s held by currently active OP_js. MLR recovery must be prepared to deal with this. It does this by executing each OP_{i+1}^{-1}

within a compensation transaction(**CT**). The CT holds the locks required and can be rolled back should deadlocks occur. It is this use of CTs that clearly distinguishes MLR recovery from prior methods.

It is also possible to undo a OP_{i+1} by execution in reverse order of inverse low level operations OP_i^{-1}s for the OP_is that constituted its original execution. This also returns the system to the same high level state; i.e. it does this by returning the system to the original low level state. Low level recovery requires retention of $LOCK_i$s for the duration of the higher level subtransaction to guarantee that the OP_i^{-1}s can be executed without deadlock. In an ML system, $LOCK_i$s are retained until their containing OP_{i+1} completes. An interrupted OP_{i+1} cannot be compensated by its OP_{i+1}^{-1} because only complete executions of a forward operation can be so compensated. Hence, an abort within an OP_{i+1} operation can and must be recovered by execution of lower level compensation operations.

2.2 Concurrency Control

Concurrency control requirements for recovery are nicely characterized in [2] as: "Recovery actions must participate in concurrency control protocols, just like ordinary actions." We assume that locking is used so as to ensure serializability and recoverability. Thus, operations executed during recovery must honor all locks held

- explicitly by other transactions during explicit rollback;
- implicitly by other interrupted transactions during crash recovery.

Redo recovery can repeat history, i.e. execute the original operations in an order equivalent to their original order. These operations will honor both implicit and explicit locks because their original execution honored them. Undo operations are newly executed during recovery. Ensuring that they correctly obey the concurrency control protocol is more difficult.

Locks must guarantee not only serializability and but also absence from deadlock during rollback. Completing an abort of a subtransaction is not optional. An "abort of the abort" is not an acceptable outcome. Absence of deadlock during rollback ordinarily requires that locks acquired for an operation cover both its original execution and its undoing.

In traditional single level systems, the recovery process executes inverse (undo) operations in reverse order of the original sequence of operations. The locks for the forward operations are sufficient for the execution of the inverse operations, and hence recovery requires no additional locking. Hence, single level system recovery satisfies the concurrency control protocols while guaranteeing that recovery will not deadlock.

For an ML system, we assume that each level performs strict two phase locking such that the locks acquired for the level are sufficient for both serializability and for execution of inverse operations at that level. At level L_{i+1}, high level locks($LOCK_{i+1}$**s**) will be acquired during OP_{i+1} execution. Once an OP_{i+1} is completed and its $LOCK_{i+1}$s acquired, the low level locks $LOCK_i$s used by its implementation OP_is are released. There should be fewer conflicts with the $LOCK_{i+1}$s than with the $LOCK_i$s.

The locking protocol above is an instance of an order preserving conflict based scheduler. In [2], these schedulers were shown to correctly serialize multi-level systems. The order preserving condition simply means that lower level transactions of a single higher level transaction must be scheduled in the order determined by the higher level transaction, not in an arbitrary serializable order.

2.3 Layers in a Multi-Level System

2.3.1 A New Level

When an abstraction implementor cannot achieve the concurrency he needs when using lower level operations, he can define a new multi-level transaction(**MLT**) layer which is a new level. This involves:

1. defining for each OP_{i+1} an OP_{i+1}^{-1} that compensates the OP_{i+1};
2. specifying the $LOCK_{i+1}$s that are needed by OP_{i+1}s to ensure that subtransactions using OP_{i+1}s can be serialized and recovered. This includes defining the lock mode conflict matrix for the $LOCK_{i+1}$s. Locks are compatible exactly when the operations which use the locks on the same resource commute;
3. implementing each OP_{i+1} as a subtransaction over the OP_is which acquires the appropriate $LOCK_{i+1}$s during its execution. (The ML system will then release all acquired $LOCK_i$s when an OP_{i+1} "commits".)

2.3.2 A Nested Transaction Layer

An implementor need not introduce a new MLT layer in the ML system to export atomic and recoverable operations. If the concurrency achieved by using $LOCK_i$s is satisfactory, the definer can realize his operations using traditional nested transactions. A nested transaction retains $LOCK_i$s across the operations that it defines instead of replacing them with newly defined $LOCK_{i+1}$s. And a nested transaction is recovered by undoing each of the OP_is that was used to realize it. Thus, completed NTs that are parts of aborted transactions are rolled back in a similar way to the rollback of incomplete subtransactions, i.e. using lower level inverse operations. The level of an NT layer is the same as the level of the operations that it uses.

Implementing an NT layer is easier than introducing a new MLT layer. An implementor must balance his concurrency needs against the increased cost of an MLT layer. What is important here is that ML systems provide the choice between NT and MLT at every level of abstraction, independently of how other layers of the system have made the choice.

3 Recovery Fundamentals

3.1 Recovery Predicates

Recovery applies the actions specified in log records to whatever happens to be the available system state. Crashes can occur at arbitrary times, independent of operation boundaries. In addition, recovery must cope with the fact that the preserved system state (in stable storage) is not usually the same as the system state at the time of the crash. The stable system state after a crash may include the effects of some operations while not including the effects of others. And those included or excluded may be in no particular order on the log. All recovery schemes must be prepared to deal with these difficulties. To help in organizing the recovery process, we introduce two properties that, once established, greatly simplify further recovery. These properties are essential when non-idempotent operations need recovery. These are described below.

3.1.1 Operation Consistency

The operation consistency property (**OC**) states that the system state is such that an operation has either been

- executed and the system state reflects all of the results;
- not executed and the system state reflects none of the results.

Essentially, an OC state is one in which no logged operation is currently active, i.e. partially executed. In an ML system, a level L_i is said to be OC if no logged OP_i is partially executed.

The OC property solves two problems in ML recovery.

Partial Results: Correct redo requires that the part of the system state seen by the operation be the same as seen by the original execution. Correct undo requires that the results from the original operation all be present. With partially executed operations, recovery can leave the system state undefined.

Implicit Locks: If an ML system is not OC at L_i when undo recovery attempts to execute OP_i^{-1} operations, there may be $LOCK_i$s held (implicitly) by interrupted OP_is that conflict with locks needed to execute these inverse operations.

3.1.2 Determinable Execution

In order to guarantee recovery idempotence for non-idempotent operations, it must be known which operations have been executed and are reflected in the system state. We call this property **determinable execution** or (**DE**).

If a logged operation must be executed because it is part of a committed transaction and it does not have its effects reflected in the available system state, then the DE property enables us to detect this and to schedule the redo of that operation. If an operation has been done and it is part of a transaction that needs to be aborted, DE permits us to detect its execution and to schedule an undo operation that purges the system state of the effects of the operation.

3.2 Higher Level Undo Recovery

At levels above L_0, hardware atomicity will not normally guarantee OC. Further, it is very difficult and/or expensive to guarantee the DE property by direct examination of system state. For these reasons, lower levels of the system are recovered before higher levels. Recovery prior to L_i will be responsible for guaranteeing that

1. all logged operations have been executed, hence that operations are DE;
2. all lower level interrupted transactions have been rolled back, and hence the system is L_i OC;
3. only OP_js, for $j > i$ are in need of undo and those are indicated on the log as incompletely executed subtransactions.

L_i recovery undoes (compensates) the OP_is as indicated on the log that are part of incomplete OP_{i+1}s and logs what it has accomplished. This ensures that the system becomes OC at level L_{i+1}. The DE property is also preserved and the set of operations in need of undo is now restricted to operations in incomplete transactions at levels above L_{i+1}.

Logging undo recovery operations is essential because it permits redo recovery (see below) to re-establish the system state as reflected in what has been written to the log, and hence to re-establish OC and DE should a crash interrupt recovery. And it does this while not constraining the posting of updated system state to stable storage.

In the detailed description of section 4, we choose to write a compensation log record(**CLR**) that documents that an OP_i^{-1} has been executed that compensates for a specific prior OP_i. When we have completed the compensation of all OP_is within an OP_{i+1}, we mark the OP_{i+1} operation as aborted. This records the fact that this operation is complete and no longer needs undo recovery.

3.3 Level L_0 Recovery

Recovery at L_0 is responsible for "breaking the recursion" in which higher levels assume that the lower levels provide serializable and recoverable operations and which guarantees the OC and DE properties. Lowest level recovery must be prepared to deal with the part of the database state that is stable at the time of a system crash. It's goals are to transform whatever is in the stable system state into an a system state that is L_1 OC, to establish the DE property, and to make sure that only the operations in incomplete transactions above L_0 need undoing.

We perform redo recovery at L_0. Redo is responsible for making sure that all logged operations are installed in the system state. It establishes the DE property. The paradigm of performing redo first and repeating history, as in ARIES, works well with L_0 logging. When this is done, the L_0 undo pre-condition is the same as the pre-condition for higher level undo. Hence, it can simply remove the effects of incomplete L_1 operations by compensating all their constituent OP_0s, hence making the state L_1 OC.

ML systems can be built on top of a number of different L_0 recovery methods. We describe two possible L_0 techniques here, one which requires both OC and DE, and the other which does not.

3.3.1 (Non-idempotent) Operation Logging

The non-idempotent operations that cause state transitions are logged. In order for this to be effective, we must be able to guarantee the OC and DE properties at the start of recovery, before L_0 recovery begins.

L_0 is the only level for which OC is needed as of the time of a crash. Several approaches have been used. For example, System R [9] achieved RSS operation consistency by installing RSS operation consistent shadows during checkpoints. However, maximum flexibility in checkpointing and buffer management is achieved by having the OP_0s act on single blocks that can be atomicly written to disk. Blocks in the cache can be written to disk at any time, and in any order. Only the need to observe the write-ahead-log (WAL) protocol constrains the writing to stable storage.

The DE property also needs to be guaranteed. This is usually done by a form of testable state (**TS**). The TS property states that whether an operation has been executed or not can be determined by testing the system state that is available (this can be done in conjunction with information that is stored in the log). An easy way to do this is to write a state identifier [11] in each block, write the state identifier seen by an operation in the log record for the operation, and increase the value of the state identifier to a new unique value as a consequence of executing the operation. Then, by comparing the state identifier in a block to the state identifier in a log record, one can determine whether or not the block includes the effects of the execution of the operation.

3.3.2 (Idempotent) Before/After Image Logging

When before and after images of the portions of the state modified by an operation are logged, applying these "operations" to the designated parts of the system state is idempotent. That is, one can "execute" these "state installation" operations more than once without changing the result. No effort need be made to ensure that the system state is OC and DE at time of crash with this state based logging.

Only the order of installation of before and after images is important, and this is determined from the placement of log records on the log. During redo recovery, the after images of completed operations, as recorded in the log, are installed, independent of the stable system state at time of crash. Again, once redo recovery is completed via repeating history, the system state meets the necessary precondition for our standard undo recovery. Undo by before image installation can be treated just like any other undo operation.

An L_0 abstraction might be persistent virtual memory. Such an abstraction is very flexible, and perhaps is ideally suited for extendible or object-oriented systems. The operations on persistent virtual memory are reads or writes of byte strings, without concern for disk block boundaries. Blocks are written to disk freely, without regard to the boundaries of write operations. Hence, should the system crash, it is possible that parts of a logged write operation are reflected in the stable database while other parts are not. Using before and after image logging permits us to support this abstraction without the need for the testable state property. This is important because it permits the entire disk block to be available to support the virtual memory abstraction. We do not need any part of it to store a state identifier.

4 Logging for MLR Recovery

4.1 Forward Operations

When a transaction at any level is initiated, it is entered into the Transaction Table ($TransT$) used to keep track of system operation. In the $TransT$, its

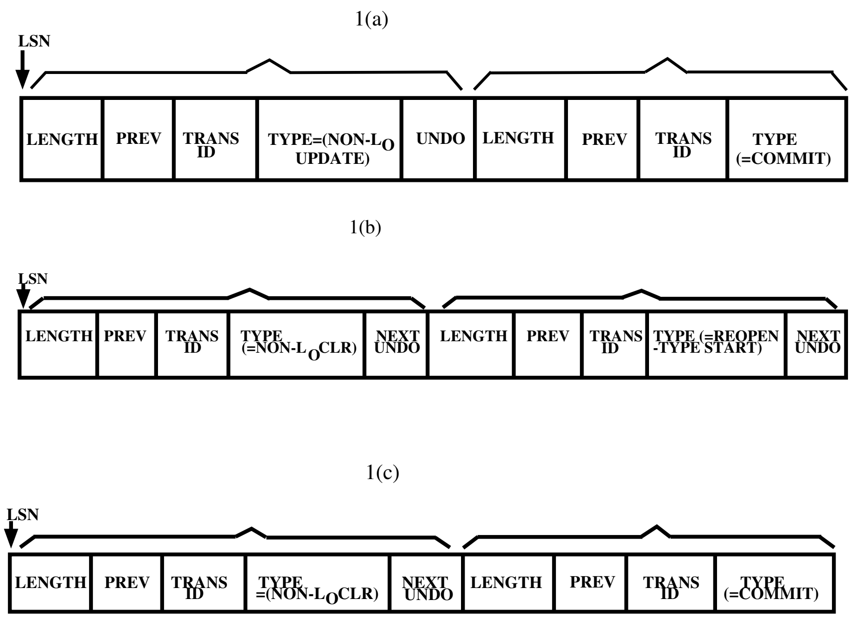


Figure 1: (a) MLR operation log record and subtransaction commit record, which handles both NT and MLT subtransactions. (b) MLR CLR and CT subtransaction start record for NTs. (c) MLR CLR and CT subtransaction commit record for MLTs.

parent transaction and its level are recorded. Its initiation is documented durably by writing a start-transaction record to the log.

MLR logs forward operations of subtransactions as if they were operations of an independent transaction. As each of its operations completes, a log record is added to the transaction's chain of log records to document its execution. When a subtransaction commits, its commit record links it to the chain of log records of its parent transaction, similar to ARIES/NT. MLR does this by writing a log record that both commits the subtransaction and that causes the committed subtransaction to be logged as an operation in the parent. The format of this record is as a pair consisting of an operation log record for the parent transaction and a commit record for the child subtransaction. This is illustrated in Figure 1(a).

Whether a subtransaction is an NT or an MLT is only distinguished

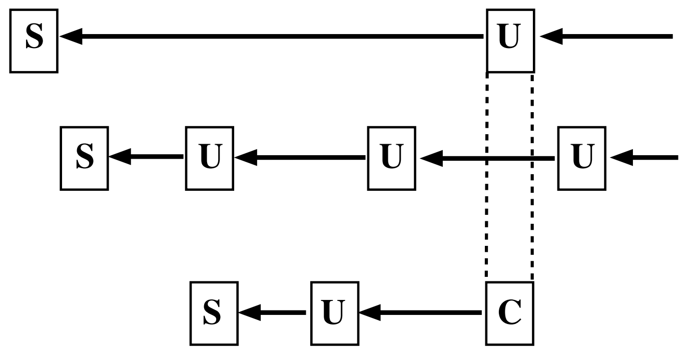


Figure 2: Normal operation logging for subtransactions. **S** denotes a start record, **U** an update (operation) record, and **C** a commit record. Log records are back linked within (sub)transactions and are shown from left to right as recorded on the log, left being earlier than right. When completed, a subtransaction is linked to its parent by its commit record, which is paired with an update record for its parent. These two components of a single log record are shown linked by dotted lines. Prior to commit, one cannot distinguish, based on log records, whether a transaction is an incomplete subtransaction or an incomplete top level transaction.

by whether the commit record for the subtransaction contains an inverse operation in the UNDO field of the operation part of the log record pair. This inverse operation is used to execute high level compensation should the completed MLT need to be rolled back. NTs have no such operation as they will be rolled back by compensation of each of their constituent operations.

As an example, the log records for two subtransactions executing in parallel during normal system operation are shown in Figure 2. This will serve as a running example. One subtransaction has committed and becomes linked with the parent (top level) transaction. The other has not yet committed and hence is indistinguishable from an active top level transaction.

4.2 Interrupted Transaction Undo Logging

We call a transaction that cannot finish normally and hence must be rolled back an interrupted transaction. MLR rolls back interrupted transactions of any level in exactly the same way, whether they are top level transactions or orphaned subtransactions of a parent transaction that is itself interrupted. Each such interrupted transaction must have its constituent operations (completed subtransactions or operations) compensated.

MLR durably records undo recovery progress by writing compensation log records. During rollback, a transaction's chain of log records is scanned backwards. Each operation encountered on the chain has its compensation operation, stored as the UNDO operation in the log record, executed, and logged with a CLR. Once compensation is complete, an abort record is written for the interrupted transaction. The abort record documents that the transaction is complete and needs no further undo recovery. Its effects have been completely purged from the system state.

As in the ARIES method, we distinguish CLRs from other log records, and indicate in them the next logged operation of a transaction on which to perform undo. This is useful (but not essential) in that CLRs are never undone, and undo recovery can continue from where it was interrupted should the system crash during recovery. The pointer to the next operation to be rolled back is copied into the CLR. This permits undo recovery to proceed from the point where it left off should a crash occur during recovery. CLRs do not need to be themselves undone. Thus, undo recovery progress always advances.

We do not need the ARIES ability to support so-called "nested top level actions" via distinguished CLRs whose pointer structure bypasses their logged operations during undo. This can be achieved via explicit multi-level subtransactions.

4.3 Undoing Completed Subtransactions

For each completed subtransaction, we initiate a separate compensation transaction that undoes its effects. The CT is added to the *TransT*, as if it were a normal subtransaction. Compensation for a subtransaction occurs entirely in its CT, with the added proviso that once compensation is assured, a CLR describing the compensation performed by the CT is written as a log record of the parent transaction. The form of this CLR differs depending on whether a subtransaction is an NT or an MLT.

4.3.1 Nested Transaction Compensation

A nested transaction CT executes in the "backward" direction, and executes only compensation operations, which are logged as CLRs. CLRs are written as CT log records as compensation operations are executed. Each compensation operation is performed when its forward operation from the original completed NT is encountered in the backward undo scan of the log. CTs for nested transactions contain only CLRs, and these CLRs always point to a next operation to be undone, which is in the original completed NT.

Nested transaction CTs can always be completed. They never deadlock since all needed locks have been acquired during forward operation. Should recovery be interrupted, the last CLR written for each transaction points to the next operation that is to be undone. Hence, the CT can be continued after a crash, without needing rollback.

Because a nested transaction's CT is guaranteed to complete, an NT is effectively compensated once its CT is initiated. Hence, we pair the CLR for the NT with the start record for its CT. This permits parent transaction rollback to proceed to its next operation. This record is shown in Figure 1(b).

By initiating a CT, handling multiple pointers in CLRs, as is done in ARIES/NT, is avoided. Each transaction consists of only a single threaded chain of log records keeping track of undo progress in a single level. The next CLR of the parent points to the the paired record CLR as if it were a simple atomic compensation operation that was part of flat transaction recovery. In the start subtransaction part of the above log record is posted a Next-undo pointer to the first (next) operation of the NT to be undone. This is the last operation logged for the completed NT.

The NT chain of log records is scanned backwards, and CLRs are written for each encountered operation. When the start record for an NT is reached, a commit record for the CT is written. This commit record need not be paired with a parent transaction record since the start subtransaction record for the CT is already paired with the CLR for the original NT.

Figure 3 illustrates nested transaction recovery using the MLR approach, applied to our running example.

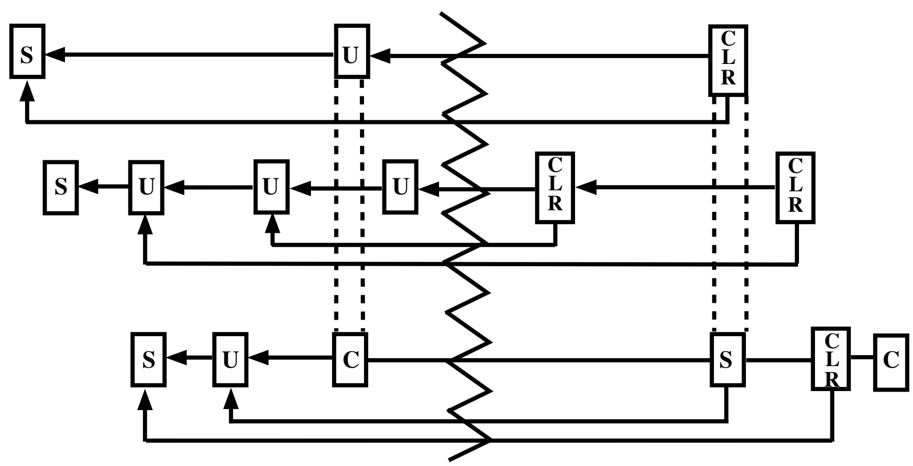


Figure 3: MLR log records for nested transaction recovery. The rollback of both interrupted and completed NTs is illustrated. A system crash is indicated by the jagged solid line. In addition to the normal back linking of log records, compensation log records(**CLR**s also contain "next undo" pointers, denoted by the arrows out of the bottom of these records.

4.3.2 Multi-level Transaction Compensation

When a completed MLT subtransaction, representing the execution of an OP_{i+1}, needs to be rolled back, the OP_{i+1}^{-1} stored as the UNDO field of its commit record pair is executed as a FORWARD operation in a CT. It is not rolled back by compensating each of its OP_is or included subtransactions.

An OP_{i+1}^{-1} CT executes in the forward direction during rollback, must observe concurrency control protocols, and hence may deadlock. Further, it cannot be continued across a system crash. In both these cases, it must be rolled back. An OP_{i+1}^{-1} CT is made recoverable via the logging of its constituent OP_is as forward operations or lower level subtransactions. The effects of an interrupted OP_{i+1}^{-1} CT are undone by compensating each of its OP_is, and logging undo progress via CLRs, just as when undoing a forward subtransaction. We need not know that it is a CT.

Because OP_{i+1}^{-1} CTs are not guaranteed to complete, an MLT's CT does not compensate an OP_{i+1} until it commits. As we do with forward operations, we pair the OP_{i+1}^{-1} operation log record with the L_i CT commit record. Since this is a CT, its L_{i+1} log record is a CLR. See Figure 1(c).

Until its CT commits, a CLR for the original MLT subtransaction has not been written. Hence, we will start another CT to compensate this MLT when we again undo operations at L_{i+1}. This limits special processing for MLTs to that required for log records describing their commit, whether they be forward subtransactions or CTs. Handling of their other log records is as if they were independent and flat.

Figure 4 illustrates the appearance of the log for our running example, interpreted now as having multi-level transactions instead of nested transactions.

4.4 Another CLR Logging Strategy

The logging we described for CLRs is consistent with how ARIES handles CLRs. That is, CLRs are distinguished from forward operation log records and are never undone. Recovery is made slightly simpler when CLRs are indistinguishable from forward operation log records. This has three effects.

1. CLRs can then be treated during recovery just like forward operations, reducing the number of cases to be handled.

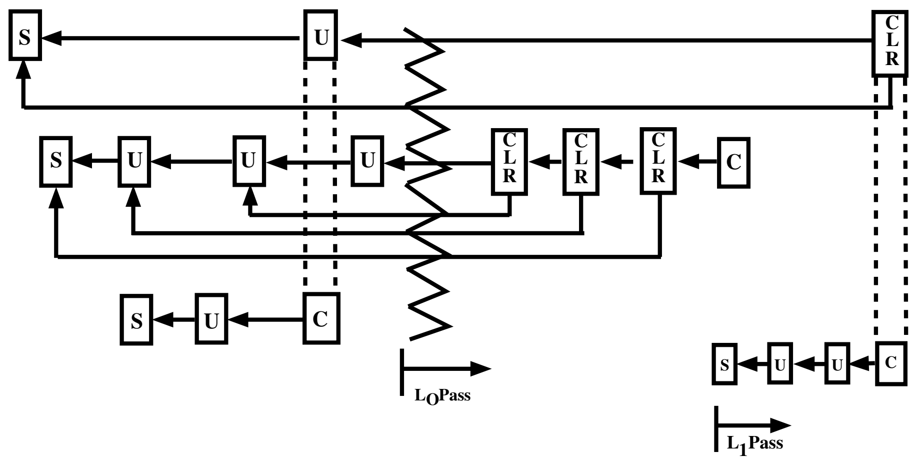


Figure 4: MLR log records for multi-level transaction recovery. The rollback of both interrupted and completed MLTs is illustrated.

2. Compensation operations for an NT can be logged as operations of the higher level interrupted transaction, rather than as operations of a CT.

3. Failures during recovery require the undo of already compensated operations and subtransactions. It can be argued that crashes during recovery do not occur sufficiently often to justify trying to avoid the rollback of compensation operations.

The unique property of MLR recovery, namely the initiation of a compensation transaction for a completed multi-level subtransaction within an interrupted higher level transaction, works regardless of which CLR logging method is used.

5 MLR Rollback

We describe here both the rollback that can occur via user or system initiated aborts, e.g., because of deadlock, during normal execution, and the rollback necessary when the system crashes and the transactions interrupted by the crash need to be aborted. The important issue for both forms of rollback is that rollback must succeed, despite deadlocks or system crashes and while honoring the normal concurrency control protocol. Completion of rollback, including completion of inverse operations, is not optional. Inverse operations that are halted by deadlock or system crash can be rolled back, but they must then be re-executed to ensure that recovery of the original forward transaction completes.

5.1 Rollback During Normal System Operation

In an ML system, rollback for incomplete subtransactions at any level is achieved by executing inverse operations at the next lower level.

While any subtransaction is active, it retains the $LOCK_i$s needed by its OP_is, and these $LOCK_i$s permit the corresponding OP_i^{-1}s to be executed. Deadlock during rollback is not possible **AT THIS LEVEL**. But, the execution of an OP_i^{-1} can require the acquisition of new $LOCK_j$s for $j < i$, that enforce serialization at that lower levels of the system. The attempt to acquire these lower level locks can result in deadlock during rollback. Of course, deadlocks can arise during normal forward operation as well.

Rolling back to subtransaction start can be exploited to overcome deadlocks, whether for forward or CT subtransactions. Once a subtransaction is rolled back, all locks that it holds can be released. Subtransaction start is a "savepoint" permitting partial rollback of a transaction. The subtransaction can be re-executed once the partial effects of its prior execution have been removed.

This technique has different implications depending on whether a subtransaction is an NT or an MLT.

- **NT:** Rollback of an L_j subtransaction does not guarantee the release of all $LOCK_j$s held by its top level transaction. Previously completed subtransactions may have passed their $LOCK_j$s to the parent. Another subtransaction, holding and requesting $LOCK_j$s might remain blocked waiting for this parent to release these $LOCK_js$. Hence, clearing of the deadlock is not guaranteed. Overcoming the deadlock might require rollback at even higher levels. Rollback is, however, always possible without deadlock since locks are not acquired during rollback. NT deadlocks occur only during forward operation.

- **MLT:** Rollback of an L_j subtransaction guarantees the release of all $LOCK_j$s held by its top level transaction. (This is not quite true, but can be made true by waiting for any concurrently executing subtransactions at L_j to complete.) Hence, any other subtransactions, holding and requesting $LOCK_j$s will no longer be blocked, thus clearing the deadlock. The rolled back MLT can then be re-executed. While it may deadlock again, repetitions of this process will ensure that the MLT will eventually complete if the scheduling is fair.

Systems where transactions need to acquire locks during rollback and that lack a subtransaction capability must make sure that a rolling back transaction is not itself subject to further rollback. System R [9] is an example. In special cases, it releases "low level" locks before commit, and then needs to re-acquire them should the transaction roll back. It designates the rolling back transaction as GOLDEN, constrains the system to permit only one GOLDEN transaction at a time, and never victimizes GOLDEN transactions in a deadlock. Rollback is hence single threaded. With an MLT subtransaction capability, one can abort a CT subtransaction and clear the deadlock. Hence multiple transactions can be permitted to rollback concurrently.

5.2 Rollback for System Crash Recovery

After a system crash, redo recovery is performed first and only at L_0. We then perform undo recovery level by level. To do this, we need to know the level of each logged operation. One simple way to recover the level of an operation is to record an operation's level in its log record. Another, since all operations of a subtransaction are at the same level, is to record the level in the $TransT$ during redo, and whether it is an NT or an MLT subtransaction. NTs are at the same level as their highest level operations while MLTs are at a level one more than the level of their highest level operations.

5.2.1 Multiple Scan Multi-Level Undo

The simplest multi-level undo by levels is to scan the log backward multiple times, once for each level. After redo recovery has completed, we first do L_0 undo by scanning the log backwards performing undo for L_0 operations that occur within interrupted L_1 subtransactions. On each subsequent backward pass of the log, we increment the level by one and perform undo for that level. For L_i undo, we begin at the tail of the log as of the time of the crash, and scan the log until there are no more interrupted L_{i+1} subtransactions in the $TransT$. There are no more undo scans when the $TransT$ is empty of interrupted transactions.

We distinguish three sets of log records.

1. **Log records for operations in Completed Transactions and MLT Subtransactions:** Ignore these. These never require compensation.

2. OP_j**s,** $j \geq i$**, Active Transaction:** Ignore these. These operations will be undone in subsequent undo scans.

3. OP_i**s, Active Transaction:** Perform undo recovery for these log records.

The undo recovery in case 3. includes recovery for OP_is that occur within completed NTs of active transactions as well as OP_is that are directly in active transactions. Those within an NT are compensated within a CT for the NT. Above L_0, OP_i undo requires execution of a CT. These CTs execute when no lower level locks are held, since lower level undo recovery has already been performed, making the system state OC at L_i. Further, the OP_i^{-1}s execute in reverse order of the original OP_i execution order, thus forming the required "palindromic" sequence. Each CT executes serially to completion, and its operations are logged as indicated previously.

Should the system crash during recovery, redo and undo phases of recovery are repeated. The state of the log is different, however. Already recovered subtransactions will not be active. However, lower level operations logged as part of MLT CTs must be rolled back, so, in general, it is not possible to skip levels during undo recovery. However, the previously completed levels of undo can be greatly shortened (see below).

5.2.2 Reducing Multiple Scans to One

We can avoid multiple passes over the log by recording during each undo pass what it is that still needs undo (case 2) but that is at a higher level. Instead, we write these log records in main memory into separate lists for each level that we call Level Lists or LLs). LLs must be linked in the same direction that the log records are encountered during the backward scan of the log. That means that each log record encountered at L_i that needs compensation is appended to LL_i. Undo recovery for L_i begins by scanning LL_i, initiating CTs for each operation in the order encountered on LL_i, just as if it had been encountered while scanning the log.

For each interrupted transaction at L_i, there will usually be an interrupted parent transaction at L_{i+1} that started before it. Thus, we expect to usually finish the undo of all L_i interrupted transactions before all the log records for interrupted L_{i+1} transactions have been seen. During the L_i undo phase, we may exhaust LL_i without having compensated all active L_{i+1} transactions. The backward scan of the log is then resumed where it left off during the L_{i-1} undo level scan and continues until all active L_{i+1} transactions are aborted. When continuing the log scan, we once again add log records to the higher level LLs. Since we can discard an operation on an LL_i as soon as it is undone, this "staggered" backward undo scan of the log reduces the sizes of the LLs simultaneously needed in main memory.

6 Discussion

6.1 Characterization of MLR Recovery Method

MLR is an integrated industrial strength multi-level recovery algorithm that works for systems with arbitrary numbers of levels. Like ARIES, it supports operation logging and provides flexible cache management, etc. During normal forward system operation, the additional MLR requirements are modest. Essentially, extra overhead is incurred to acquire higher level LOCKs, and, as with ARIES/NT, to link subtransactions with their parents upon subtransaction commit. It is these, of course, that enable the substantial increase in concurrency.

Rolling back a (sub)transaction is more complex in MLR than with flat or NT systems because one must deal with multi-level concurrency control and operation consistency. Undo recovery must be done level by level. MLR minimizes the impact of this by performing only a single backward scan of the log. We speculate that undo recovery is less than a factor of two longer than for flat transaction undo. When the cost of the redo phase, which is usually much more expensive and is essentially unchanged in MLR, is factored in, the relative performance difference should be much smaller than that.

An ML layer implementor does not need a detailed understanding of MLR recovery. He need only supply a set of OPs and OP^{-1}s, a set of LOCK definitions, and a compatibility matrix for these locks, and then implement his operations. He need not be concerned with whether an operation is executing as a forward or rollback operation. The ML system can determine this. Hence, variations in logging and locking needs between normal and rollback execution can be made invisible to the layer implementor.

6.2 Application to General Multi-level Systems

MLR recovery is useful in a wider setting than when only top level transactions are durable and contain non-durable subtransactions.

- MLR is not sensitive to the constraints being enforced by the concurrency control protocol. Only locking needed for compensatability is required and the MLR framework holds these locks until subtransaction completion. Whether the locks involved enforce other constraints is at the discretion of the implementor, including whether they enforce serializability.

- While the write-ahead log protocol must be observed, whether the log is forced at (sub)transaction commit is optional. An ML layer implementor can determine the durability of the layer's transactions. Durability can be a declarative part of each layer's definition and can be enforced by the ML framework.

For example, MLR can be used in a system that supports sagas [8, 7]. Transactions of a saga must be durable. But sagas need not be serializable. Compensatability is sufficient, and this may need substantially weaker locks. Traditionally, no locks are held between constituent transactions. Saga compensation is assumed to be possible regardless of other system activity.

In general, some locks may be needed to ensure compensatability. Consider a possible insurance company claims saga. The company wishes to expose intermediate states of a claim saga to its headquarters employees. Hence, locks held by the saga are compatible with locks used by headquarters operations. On the other hand, agents should not know these internal states, and hence locks of agent operations conflict. Possible rationales are

(i) compensation, where all agents are informed that a saga has been rolled back, is difficult and expensive because of the large number of agents and the cost of communication and (ii) agents might leak information to claimants, which the company views with alarm. MLR recovery can accommodate concurrency control of this form.

6.3 Extension to Layered Abstraction Systems

The ML system discussed thus far is structured as a strictly layered hierarchical system. Each layer must use only the operations provided by the layer immediately below it. More flexibility is possible. A system may permit the definer of an abstraction (which exports atomic and recoverable operations) to use instances of any abstraction previously defined. The abstractions so used might not all be at the same level. Hence, the lower level OPs used in the new abstraction's OPs have different levels. Can MLR recovery work for such a system?

The answer is yes, MLR recovery will work, provided that the specific instances of abstractions used to realize a new abstraction are not accessible outside of the new abstraction. This constraint is frequently satisfied as a natural result of our desire to hide the representation of an abstraction. For example, if some instances of a disk block abstraction are used to support a record abstraction, the constraint requires that those disk blocks only be accessed in operations supporting the accessing of records. Then, the lower level operations can be recovered first even though this may not be a palindromic ordering of forward and undo operations within a transaction. The ordering can be made palindromic by commutativity, i.e. operations on disjoint resources commute.

One example of the use of the layered abstraction technique is for building recoverable queues [4] on top of a database system. MLR solves two significant problems that arise in this.

1. Once an element is enqueued/dequeued from the queue, the low level database locks for these operations can be released. Only enqueue or dequeue locks need be retained, which can be defined to commute. That is, e.g., a dequeue operation for one transaction need not conflict with the dequeue operation on the same queue by another transaction. This avoids the head of queue (tail of queue) bottleneck that arises

if only single level database locks are used. This exploits the natural capabilities of multi-level locking. MLR.

2. The abort of a containing transaction will usually cause a dequeued element to be returned to the queue via the defined compensation operation. However, this compensation operation can do more, e.g., the number of times the queue element was dequeued and its transaction failed can be counted. After some number of failures, the queue element can be placed on an error queue. This exploits user defined compensation operations that do not need to exactly undo their forward operations.

Layered abstraction functionality, coupled with a persistent virtual memory abstraction for level L_0 may make MLR recovery ideal for extendible and object-oriented transaction systems. Persistent virtual memory is a very flexible basis from which to work. Layered abstractions with MLR recovery permit implementors to achieve high concurrency while preserving transactionality.

7 Acknowledgements

In 1976, Michael Melliar-Smith suggested to me that recovery for high level operations might be accomplished in a hierarchically structured system by executing inverses of higher level operations rather than inverses of the low level operations that implement them. Recently, in conversations with Butler Lampson, I became convinced that the low level redo, high level undo model of recovery was the correct model to provide operation consistency and maximum concurrency.

References

[1] Beeri, C., Bernstein, P., Goodman, N., Lai, M., and Shasha, D. A concurrency control theory for nested transactions. Proc. PODC (August 1983) 45-62.

[2] Beeri, C., Schek, H.-J., and Weikum, G. Multi-level transaction management, theoretical art or practical need? Lecture Notes in Computer Science, vol 303, Springer-Verlag, 1988, 135-154.

[3] Bernstein, P., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems*, Addison Wesley (Reading, MA) 1987.

[4] Bernstein, P., Hsu, M. and Mann, B. Implementing Recoverable Requests Using Queues. Proc. ACM SIGMOD Conf. (June 1989) 112-122.

[5] Eswaran, K., Gray, J., Lorie, R., Traiger, I. The notions of consistency and predicate locks in a database system. Comm ACM 19,11 (Nov 1975) 624-633.

[6] Garcia-Molina, H. Using semantic knowledge for transaction processing in a distributed database. ACM TODS 8,2 (June 1983) 186-213.

[7] Garcia-Molina, H. and Salem, K. Sagas. Proc. ACM SIGMOD Conf. (June, 1987), San Francisco, CA, 249-259.

[8] Gray, J. The transaction concept: virtues and limitations. Proc. VLDB Conf. (Sept. 1981) Cannes, France, 144-154.

[9] Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzulo, F., Traiger, I. The recovery manager of the System R database manager. ACM Computing Surveys 13,2 (June 1981) 223-242.

[10] Korth, H., Levy, E. and Silberschatz, A. A formal approach to recovery by compensating transactions. Proc. VLDB Conf. (August, 1990) Brisbane, 95-106.

[11] Lomet, D. Recovery for shared disk systems using multiple redo logs. DEC Tech Report CRL90/4 (Oct 1990), Cambridge Research Lab, Cambridge, MA.

[12] Lomet, D. and Salzberg, B. Concurrency and recovery for index trees. draft report (Dec 1990).

[13] Lynch, N. Multilevel atomicity- a new correctness criterion for database concurrency control. ACM TODS 8,4 (December 1983) 484-502.

[14] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM TODS (to appear) and IBM Research Report RJ6649 (Jan 1989), IBM Almaden Research Center, San Jose, CA.

[15] Mohan, C. and Levine, F. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. IBM Research Report RJ 6846 (Aug 1989), IBM Almaden Research Center, San Jose, CA.

[16] Rothermel, K. and Mohan, C. ARIES/NT: A recovery method based on write-ahead logging for nested transactions. Proc. VLDB Conf. (August, 1989) Amsterdam, Netherlands, 337-346.

[17] Traiger, I. Trends in systems aspects of database management. Proc. ICOD Conf., Cambridge, MA (1983)

[18] Weikum, G. and Schek, H.-J. Architectural issues of transaction management in multi-layered systems. Proc. VLDB Conf. (August, 1984), Singapore, 454-465.

[19] Weikum, G. A theoretical foundation of multi-level concurrency control. Proc. ACM PODS Conf. (March 1986) Cambridge, MA. 31-42.

[20] Weikum, G., Hasse, C., Broessler, P., and Muth, P. Multi-level recovery. Proc. ACM PODS Conf., Nashville, Tenn (April, 1990).