Log-Based Recovery for Nested Transactions

J. Eliot B. Moss

COINS Technical Report 86–61
December 1986

Submitted to ACM SIGMOD '87 (International Conference on Management of Data).

Abstract. Techniques similar to *shadow pages* have been suggested for use in rollback and crash recovery for *nested transactions*. However, *undo/redo log* methods have not been presented, though undo/redo logs are widely used for transaction recovery, and perhaps preferable to shadow methods. We develop a scheme of log-based recovery for nested transactions. The resulting design is promising because it requires a relatively small number of extensions to a similar scheme of recovery for single-level transactions.

Overview

Our goal is to develop algorithms for log-based rollback and recovery of nested transactions. We assume general familiarity with transaction oriented concurrency control and recovery; [Gray 78] and [Haerder and Reuter 83] are good introductions to the subject. We also assume some familiarity with nested transactions [Moss 81, Moss 82, Moss 85, Moss 86].

In presenting the new design, we first review transaction commit semantics, for both single-level and nested transactions. We next describe a scheme of recovery for single-level transactions, which is then extended to nested transactions. Finally, we offer our conclusions concerning the results.

Transaction Semantics

Single-level transactions consist of some number of *actions* which are to be performed against a database in an atomic fashion. In particular, transactions should have the following characteristics:

- Transactions should be integral (atomic): eventually, either all of a transaction's actions are done, or none of them. This should be true regardless of failures, up to the overall resiliency required of the system. If a transaction's actions are performed, it is called *committed*; if they are not performed, *aborted*. Prior to that a transaction may be *in progress*.
- Transactions should not interfere with one another's execution so as to produce anomalous results. Preventing such interference is the job of concurrency control. The usual goal is *serializability*: the overall effect is to be as if transactions executed serially, one at a time. We will not be especially concerned with the details of concurrency control. However, it should be noted that concurrency control will sometimes abort transactions to prevent or resolve conflicts.
- Finally, transactions, once committed, should be *durable*: their effects should not spontaneously disappear, even if there are failures.

Nested Transaction Commit Semantics

A transaction provides a protected environment in which it performs its actions, free from interference, and with guarantees of integrity and durability. Nested transactions extend this notion to allow such environments to be nested in one another, similar to the nesting of lexical scopes in programming languages such as Pascal, though dynamic rather than static. A *top-level* transaction may have nested within it any number of *child* (or *sub-*) transactions, which may also have children, etc. In fact, each top-level transaction is the root of a *tree* of transactions. This tree evolves, by adding and pruning leaves, as the transactions execute.

When it is a leaf, a transaction performs actions, and then may commit or abort. An abort always discards the work performed by the transaction, *and* any subtransactions that ever existed in the sub-tree for which it is the root. However, a commit of a non-top-level leaf transaction is *not* a total commit in the usual sense. Rather, a commit indicates that the committing substransaction's results are available in the parent transaction's scope. Such a commit is *relative*, rather than absolute, in the sense that the parent (or any ancestor, in fact) might still abort. The only commit that is *absolute* is that of a top-level transaction; such a commit has the usual durability semantics.

A little example may clarify these ideas. Suppose top-level transaction a creates two child transactions b and c. Transaction b creates two children of its own, b_1 and b_2. Transaction b_1 runs and commits, releasing its results (only) in the environment provided by b. Thus b_2 can now access b_1's results. For some reason b_2 aborts. This does not affect the commit of b_1.

Consider now two cases. First, suppose that b can get by without the effects of b_2. In this case, b may be committed, which will release b_1's results more widely: into a's environment. If a commits, then the effects become permanent. The other case is that b aborts. If that happens, then b_1 must be undone, *even though it was previously committed* (in the relative sense). Likewise, if b had committed but a aborted, b_1 would still have to be undone.

Single Level Transaction Recovery

A taxonomy of recovery schemes is presented in [Haerder and Reuter 83]. While it should be possible to extend any of the schemes discussed there from single-level transactions to nested transactions, we will present a particular one, for clarity, simplicity, and brevity. We have chosen to consider the case described as ¬ATOMIC, STEAL, FORCE, TOC in [Haerder and Reuter 83], which corresponds to the approach taken in some important commercial database systems, including IBM's IMS [IMS 76]. Again, for simplicity of discussion, we will assume that pages (disk blocks) are the units read and written, and hence the units of logging and recovery. Specifically, we will use physical state logging, implying that we record before- and/or after-image of pages. This is described in detail later.

The database system is subject to the following sorts of failures, which have corresponding recovery methods (as in [Haerder and Reuter 83] and [Gray 78]):

- Failure of an individual transaction (abort), recovered via *transaction undo.*
- System crash, recovered via *global undo* (to remove those transactions which were interrupted and cannot complete) and *partial redo* (to insure that the effects of recently committed transactions are reflected in the database).
- Media failure, recovered via *global redo* (to restore the effects of all committed transactions on the destroyed part of the database).

Any given scheme might obviate some of the above mentioned recovery techniques, by preventing the bad situations in the first place.

The database system can be broken down into the following storage components (after [Haerder and Reuter 83]):

- The physical database, a collection of pages on mass storage devices (e.g., disks). A page being written might be corrupted by a crash, and occasional media failures might destroy one or more pages of the physical database.
- The main memory database buffers, a collection of pages that are lost in system crashes.[1]
- The temporary log, used to support transaction undo, global undo, and partial redo (i.e., recovery from transaction failure and system crashes). This may need to have more than one copy written, to guarantee necessary resiliency.
- The main memory log buffer, which is lost in a crash.
- The archival log, which supports global redo (recovery from media failure). This may need to have more than one copy written, to guarantee necessary resiliency.
- The archival database, which helps support global redo. This may need to have more than one copy, depending on how much of the archival log is retained.

Now we describe what ¬ATOMIC, STEAL, FORCE, and TOC mean. First, ¬ATOMIC means that collections of pages are not written to the database in a fashion that is atomic with respect to crashes. That is, if we need to update two or more pages, and a crash happens in the middle, we may be in a situation where some pages have been updated and some have not. ¬ATOMIC is interesting because it includes update-in-place techniques; some ATOMIC techniques include *shadow pages* [Lorie 77], timestamped pages [Reuter 80], and "differential files" [Severance and Lohman 76] or intentions lists [Sturgis, et al. 80].

The main implication of ¬ATOMIC is that after a crash the database is not necessary in a usable state. This is because its *internal* data structures may not be consistent. For

[1]We are not considering the possibility of main memory whose contents survives crashes, though that is an area of current investigation and interest.

example, when splitting nodes in a B-tree, several pages have to be updated to accomplish the split. If a crash occurs when only some of the pages have been written to disk, the B-tree pages may not represent any valid B-tree state.

STEAL means that database buffer pages may be written back to the database at any time; Thus, the database can be affected *before* a transaction commits. This implies that global undo will have to be implemented, to remove effects effects of failed transactions from the database.

FORCE means that modified pages are written to the database before a transaction commits. This has the nice property that partial redo is not required. However, global redo (to recover from media failures) will still require logging of information. TOC means *transaction oriented checkpointing*: FORCE implies that each transaction commit is a kind of checkpoint, bounding the amount of redo required.

Since we are assuming physical logging of page data, undo information will consist of a *before-image* of the page (the contents before the transaction modified the page), and redo information will be an *after-image* (the contents after the transaction modified it). Two rules that must be followed by any undo/redo logging scheme are:

- An undo record must be written to the temporary log before writing a page to the database. This guarantees that the change to the database can be undone if there is a crash. This rule is called the *write ahead log* principle [Gray 78].
- Redo records for a transaction must be written to the archival log (and in general, though not in our case, to the temporary log) before writing a *commit* record to either log. Likewise, the commit records must be written before acknowledging success to the user. The writing of the commit records is the act that commits a transaction and makes it permanent. After a transaction's commit records are written, its undo records can be discarded, and then its commit record in the temporary log can be reclaimed.

We will assume that the above rules are followed, and, as previously stated, a transaction will write all its modified pages to the database before it commits (FORCE). Thus, commit processing will be done as follows:

- All unwritten, changed pages are written to the database. Because of the write ahead log principle, this implies that undo records are written to the temporary log, too.
- Redo records are written to the archival log.
- A commit record is written to the temporary log. This commits the transaction.
- At any later time, the transaction may be acknowledged as complete; likewise a commit record is written to the archival log.
- Once the commit record is written to the archival log, the temporary log information for the transaction (both undo and commit records) can be discarded.

The temporary log is a single, ordered stream of log records, merging the logging requests of all transactions, in the order they are made. This is important because it reflects any serialization performed by concurrency control, and thus avoids certain bad situations. In particular, the actual completion order of transactions is reflected in the temporary log.

Here is how recovery from the various sorts of failures proceeds:

- Transaction abort: Process the undo records for the aborting transaction, in reverse chronological order: for each modified page, restore the before-image data; this may be done to the main memory copy of a page, if there is one, or otherwise to the database page itself. Discard the redo records of the transaction, perhaps by posting an *abort* record to the archival log. Also write an abort record to the temporary log. At this point, the transaction's resources can be released (e.g., locks on pages, if locking is being used for concurrency control) and the abort is logically complete. At any later time, the undo records, and then the abort record, can be discarded. This last step might be omitting if the temporary log is append only (e.g., tape or optical disk).

- System crash: Process the temporary log in reverse chronological order. It may contain undo, commit, and abort records. For those transactions having *neither* commit nor abort records, perform their undos, in the order encountered in the reverse scan. Once this has been done, write abort records in the temporary log for the undone transactions. At this point, the undone transactions' records in the temporary log may be discarded, if the logging scheme supports log storage reclamation. The redo records in the archival log should also be discarded (or an abort record written, so as to indicate their irrelevance). Committed transactions' temporary log records can always be discarded. Hence, since abort and commit records are not treated differently in the temporary log, they need not be distinguished: *end of transaction* records (EOT) suffice.

- Media failure: For a total failure, restore the archival database and process the archival log, in chronological order. For partial failure, restore only the appropriate portion of the archival database, and process the archival log in chronological order, applying only those redo records pertaining to the portion being recovered. In either case, one must be careful not to redo effects of an uncommitted transaction. Uncommitted transactions can have redo records if there was a system crash before the transaction's commit record was written. Media failure can be simplified if we require recovery from system crashes to clean out from the archival log any redo records for failed transactions. We will not discuss the details.

It should be clear that there are interesting storage management and data structures issues in implementing the logs – issues that we will not explore here.

Nested Transaction Recovery

The approach we will take in providing recovery for nested transactions is to transform the problem into essentially that of recovery for single-level transactions. The overall effect is that recovery from system crashes and media failures will be very similar to the single-level transaction case. This is reasonable because durability for nested transactions is a property only of top-level transactions. Transaction failure will need the most extension, since we need to deal with failure of child transactions. In particular, we must be able to abort a child without disturbing its parent, siblings, etc. Also, we must be able to abort the parent of a (relatively) committed transaction.

To explain the scheme, we will first describe the log information and the log records written in various cases. Then we will describe how to recover from the various modes of failure.

Each transaction can have the following kinds of temporary log records associated with it:

- A *beginning of transaction* (BOT) record, which gives the new transaction's id and its parent's id. Top-level transactions can be identified by a special parent id (e.g., 0).
- A *commit* record, again giving both the committing transaction's id and its parent's id.
- An *abort* record, analogous to a commit record. Clearly, a transaction can have at most one of these, but not both.
- *Undo* records, each identifying a transaction, a page it modified, and the page's before-image.

The only case in which the log buffer needs to be forced to disk is the commit of a *top-level* transaction. Here is how the various sorts of failure are handled.

Transaction Failure

Here we need to abort some transaction, be it top-level or not. The first step is to abort any of the transaction's children that are incomplete (have not yet committed or aborted). Next, we will process an appropriate set of undo records (which ones will be discussed in a moment). We process the undo records in a very particular way: we make it appear as if *the aborting transaction itself* is making *forward* changes, which "coincidentally" reverse the transaction's former effects. In our case, this will result in no additional undo records, since we are changing only pages that the transaction has changed before.[2] However, this "pseudo-forward" processing will perhaps cause redo records to be logged. There are a number of fairly obvious optimizations to reduce the number of these redo records actually written to disk, a subject we will not pursue.

[2]If we were logging state *transitions* rather than pre-images, then additional undo records would have to be produced, but we would have to ignore them.

The undo records that should be processed are those of the transaction and any of its committed children, and their committed children, etc. That is, we back out all of the successful (not yet aborted) actions of the transaction's subtree.

Here is an informal but detailed description of the algorithm for aborting nested transactions. The algorithm uses a local data structure called the *aborting set* of transactions: those transactions that are in the process of being undone. The contents of the aborting set changes as we encounter previously unseen transactions that must be aborted, etc. When requested to abort a given transaction T, the algorithm proceeds as follows. First, we abort the as yet unresolved children of T. The aborting set is then initialized to be $\{T\}$. Next, we proceed to read the temporary log in reverse chronological order, one record at a time. At any point in time we ignore records belonging to transactions not in the current aborting set. Here is how we handle the remaining records:

- A BOT record. This marks the beginning of a transaction. Since we are scanning backwards, we are finished undoing it, so we remove it from the aborting set. If the aborting set is then empty, we are finished scanning the log.
- A commit record. This marks the end of a committed child of an aborting transaction. The child needs to be undone, so add it to the aborting set.
- An abort record. This marks the end of an aborted child of an aborting transaction. Since the child has already been aborted, do *not* add it to the aborting set. Its undo records (and those of its descendants) will be ignored.
- An undo record. Restore the page contents to the indicated before-image. This will, in general, cause a redo record to be written. If so, the record record should be associated with T, *not* with the transaction associated with the undo record.

Once the scan is complete, if T is a top-level transaction, then proceed as if committing it, but substitute an abort record for the commit record. It is important to the crash recovery algorithms that modified pages be written back to the database. If T is not top-level, simply write an abort record for T to the temporary log buffer; the log need not be forced.

System Crashes

The processing in this case is similar to that of transaction aborts in many respects. However, we must in theory process the entire temporary log, at least back to the previous crash. This is because we cannot know how long ago it was that we started any transaction that was interrupted by the crash from which we are recovering. However, if the lifetime of individual transactions is bounded, then we can bound the scan. *Checkpoints* can further reduce the numbe of records scanned. A number of checkpointing techniques are described in [Haerder and Reuter 83]. In our case, all a checkpoint need consist of is a list of the currently active transactions. Let us assume that such *checkpoint* records are provided.

The general idea of the crash recovery algorithm is to undo, at least back to the last checkpoint, all incomplete top-level transactions (and by implication their descendants). Checkpoints let us know about additional transactions that may have been incomplete at the time of the crash. The algorithm will use the following local data structures:

- The *completed set*: those transactions known to have completed (committed or aborted).
- The *incomplete set*: those transactions in the process of being undone.
- The *newly aborted set*: those transactions whose abort we have accomplished during crash recovery.

All the sets start empty. We process the temporary log backwards, handling each record as described below:

- A BOT record: If the transaction is in the completed set, remove it. If the transaction is in the incomplete set, remove it from the incomplete set and add it to the newly aborted set; if the incomplete set becomes empty, and we have processed at least one checkpoint record, then terminate the backwards scan of the temporary log. If the transaction is in none of the sets, enter it in the newly aborted set.
- A commit or abort record: If the transaction is top-level, or its parent is in the completed set, enter the transaction in the completed set. Otherwise, enter the transaction in the incomplete set.
- An undo record: If the transaction is in none of the sets, enter it in the incomplete set. If the transaction is then in the incomplete set, process the undo.
- A checkpoint record: Note that we have seen a checkpoint record. Also, add to the incomplete set every transaction in the checkpoint record's lists of then active transactions that is not in the completed set. If the incomplete set is empty after this, then terminate the log scan.

After scanning the temporary log as described (and writing back to the database all pages changed by processing undo's), write an abort record to the temporary log for each transaction in the newly aborted set. If a transaction and its parent are both in the set, write the abort record for the child first. Since we turned aborted transactions into committed ones (in effect), media recovery needs no changes from the single-level transaction case, assuming that crash recovery effectively removes from the archival log any redo records of aborted transactions.

Speeding Up Transaction Abort

An obvious way to speed up transaction abort in a single-level transaction system is to back chain all of a transaction's records in the temporary log, so that they can be processed in reverse order directly, without scanning all the other transactions' log records. This technique does not quite directly extend to nested transactions. We now describe a reasonable extension of the back chaining scheme to nested transactions. Other techniques are possible; this one has the virtue of simplicity and space efficiency in the log.

The temporary log records will maintain chains as follows:

- Commit and abort records: These are considered to be log records of the parent transaction (if any), and are hence threaded on its chain. However, in an additional slot they point to the previous log record of the child transaction.
- BOT records: These are log records of the parent (if any), and are threaded on its chain. However, the next log record of the child transaction will refer back to the BOT record.
- Checkpoint records: These are not used by the transaction abort algorithm, so their chaining is not relevant.
- Undo records: These are threaded on their transaction's chain, just as for single-level transactions.

When running the algorithm to process a transaction abort, rather than scanning every log record, we maintain a log pointer for each transaction in the aborting set. At the beginning, the pointer to the last log record of the transaction to be aborted is used to get started. After that, there are three ways in which the pointers will be updated:

- When an undo record is processed for a given transaction, that transaction's log pointer is updated to be the pointer to the next earlier log record for that transaction – namely the chain value in the undo record.
- When a commit record is processed, the parent's log pointer is updated according to the chain value, and the child's log pointer is initialized according the the child pointer in the commit record.
- When a BOT record is processed, the parent's log pointer is updated according the chain pointer. Since the child transaction is removed from the aborting set, the child transaction's log pointer can also be discarded.

After each record is processed and the necessary log pointers updated, the next record to be processed is the chronologically latest of all the log pointers. By keeping the log pointers in a priority queue data structure (e.g., a heap as used in heap sort, or an appropriate balanced tree), we can efficiently update the pointers and calculate the address of the next record to be processed. The time to fetch the log record will almost certainly dominate the time required to determine which log record is required, so the additional calculation should have negligible impact on the time required to abort a transaction.

Summary and Conclusions

We have briefly described the recovery properties of nested transactions in a centralized database system, and reviewed an undo/redo log-based scheme for single-level transaction recovery from three classes of failures: transaction aborts, systems crashes, and media failures. We then extended that scheme to handle nested transactions.

The resulting recovery algorithms add little complexity to those used for single-level transactions. In fact, processing required for system crashes and media failures is essentially the same for nested transactions as for single-level ones. Nested transactions require a few additional log records to delimit the subtransactions, and these must be processed during recovery from systems crashes, but the additional work is negligible (assuming that undo records substantially outnumber BOT, commit, and abort records). Distributed commit protocols, such as two-phase commit ([Gray 78], for example) would require virtually no adjustment (but see [Moss 81], [Moss 85], or [Moss 86]): they can merely invoke the nested transaction commit/abort algorithms rather than the ones for single-level transactions.

Handling transaction aborts also requires processing the subtransaction log records, but the additional work can be assumed to be negligible. Finally, finding an individual transaction's records, along with those of its descendant transactions, is a little more complicated than following a simple back chain, but is still relatively simple and efficient, and almost certainly dominated by I/O and actual undo processing costs.

The conclusion we draw is that nested transactions should not be difficult to incorporate in undo/redo logging schemes for recovery, and should have little impact on system performance. Clearly, measurements from an actual implementation would strengthen this conclusion. A second point that can be made is that, except in terms of design and coding effort, a nested transaction mechanism would impose no significant overhead in those cases where it is not used. The only overhead might be the extra slot for the parent transaction id in BOT, commit, and abort records. However, special versions of these records could be used for top-level transactions, which would gain back the space, and the space is trivial anyway (when compared with before-images, for example).

These conclusions suggest that, provided efficient nested transaction concurrency control mechanisms can be designed, it is viable to offer nested transactions in production quality database systems. We hope this encourages design of nested transaction concurrency control schemes, and eventually, the widespread offering and use of nested transactions in database management.

References

[Gray 78] Jim Gray, "Notes on Database Operating Systems", in *Lecture Notes in Computer Science*, Volume 60, R. Bayer, R. N. Graham, and G. Seegmueller, eds., Springer-Verlag, New York, 1978.

[Haerder and Reuter 83] Theo Haerder and Andreas Reuter, "Principles of Transaction-Oriented Database Recovery", *ACM Computing Surveys*, Volume 15, Number 4, December 1983, pp. 287-317.

[IMS 76] IMS/VS-DB Primer, IBM World Trade Center, Palo Alto, CA, July 1976.

[Lorie 77] R. A. Lorie, "Physical Integrity in a Large Segmented Database", *ACM Transactions on Database Systems*, Volume 2, Number 1, March 1977, pp. 91-104.

[Moss 81] J. Eliot B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing", PhD thesis, Massachusetts Institute of Technology, available as Laboratory for Computer Science Technical Report 260, April 1981.

[Moss 82] J. Eliot B. Moss, "Nested Transaction and Reliable Distributed Computing", *Second IEEE Symposium on Reliability in Distributed Software and Database Systems*, Pittburgh, PA, August 1982, pp. 33-39.

[Moss 85] J. Eliot B. Moss, *Nested Transaction: An Approach to Reliable Distributed Computing*, MIT Press, 1985.

[Moss 86] J. Eliot B. Moss, "An Introduction to Nested Transactions", University of Massachusetts (Amherst), Department of Computer and Information Science Technical Report 86-41, September 1986.

[Reuter 80] A. Reuter, "A Fast Transaction-Oriented Logging Scheme for UNDO-Recovery", *IEEE Transactions on Software Engineering*, Volume SE-6, Number 4, July 1980, pp. 348-356.

[Severance and Lohman 76] D. G. Severance and G. M. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases", *ACM Transactions on Database Systems*, Volume 1, Number 3, September 1976, pp. 256-267.

[Sturgis, et al. 80] H. Sturgis, J. Mitchell, and J. Israel, "Issues in the Design and Use of a Distributed File System", *ACM Operating Systems Review*, Volume 14, Number 3, July 1980, pp. 55-69.