Log-Based Recovery for Nested Transactions

J. Eliot B. Moss

Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

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.

1. 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.

2. 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 is *running*.
- 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 assume that some locking scheme is used to provide atomicity as required, but we will not be especially concerned with the details. However, it should be noted that concurrency control will sometimes abort transactions to prevent or resolve conflicts.
- A transaction, once committed, should be *durable*: its effects should not spontaneously disappear, even if there are failures.

3. Nested Transaction Semantics

A transaction provides a protected environment within 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 subtransaction'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, now allowing c to access b's (and b_2's) results. 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.

4. Single-Level Transaction Recovery

Recovery schemes for single-level transaction systems abound. We expect that virtually any single-level logging scheme can be extended to nested transactions; it is the extension that we are concerned with more than the specific scheme. However, we need to start with *some* particular scheme to be concrete. A survey and taxonomy of recovery schemes is presented in [Haerder and Reuter 83]. We have chosen to consider the case described as ¬Atomic, Steal, ¬Force, Fuzzy in their taxonomy (these terms will be explained below. In their rating of different techniques, this would be one of the most favored.

There is still the issue of *what* to log. *Pages* (disk blocks) are the units read and written to the database. However, rather than doing page level logging (which may tend to imply page level locking), we will assume that low level operations (add, delete, or modify a tuple, index entry, etc.) are logged. In most cases, these entries will pertain to a single page. If an operation pertains to more than one page (e.g., adding an index entry may cause several pages to be affected when an update occurs), we will assume that the affected pages are written to the database in an order which preserves the correctness of the data structure, and makes the change atomic (as the result of one write). This has been called *careful replacement* [Verhofstadt 78]. In the case of tree structures, writing from the bottom up is sufficient. Note that careful replacement can be used for some kinds of in-place updates, in addition to supporting shadow page recovery (as in System R [Gray et al. 81]).

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 (as in [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 cause the contents of one or more pages of the physical database to be lost.

- The main memory database buffers, a collection of pages that are lost in system crashes.[1]

- The temporary log, which is 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 the taxonomic description ¬Atomic, Steal, ¬Force, Fuzzy means. 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, Gray et al. 81], timestamped pages [Reuter 80], and "differential files" [Severance and Lohman 76] or intentions lists [Sturgis, et al. 80].

In general, an implication of ¬Atomic is that after a crash the database is not necessarily in a meaningful state. This is because its *internal* data structures may not be consistent. However, we have assumed that logged changes affect only a single page, or that an appropriate careful replacement strategy is used.[2] Hence, the fundamental structures (access paths) of the database will be consistent in our case, in the sense that they can be used. There is no guarantee that the database is more broadly consistent, e.g., to the level of transactions, or tables consistent with their indices.

Steal means that main memory buffer pages may be "stolen" from a running transaction and written back to the database in the middle of a transaction. Thus, the database can be affected *before* a transaction commits. This implies that global undo will have to be implemented, to remove effects of failed transactions from the database after a crash.

¬Force means that modified pages are not written to

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

[2] One could argue that our scheme really is Atomic, because of careful replacement. However, our interpretation of [Haerder and Reuter 83] is that our method is more in the spirit of ¬Atomic.

the database as a transaction commits (i.e., the pages are not "forced out"). This means that partial redo will be required, to reconstruct a transaction's effects after a crash. Global redo (to recover from media failures) must be supported by any recovery technique.

Fuzzy, in ¬Atomic, Steal, ¬Force, Fuzzy, refers to the checkpointing technique employed. If some form of checkpointing is not used, then global undo does not know where to stop (it cannot be determined which transactions were running at the time of a crash). This is unacceptable; checkpointing is used to bound global undo. Checkpointing can also help bound the amount of partial redo required. If information is written to the database at checkpoint time, then we need only redo from that point on. A fuzzy checkpoint is one in which not all the pages in memory are written at every checkpoint. In particular, we will assume that pages are written at checkpoint time only if they are dirty (modified but not yet written to the database) and have been resident a long time since being dirtied. The time could be real time or the number of checkpoints. We will assume that it is possible to determine readily if a given checkpoint in the log is the earliest relevant one for partial redo.

Every operation logged consists of an undo record and a redo record. The following rule must be followed when writing undo records:

> An undo record relating to (i.e., affecting) a given page must be written to the temporary log before writing the 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]. In our scheme, committing a transaction is particularly simple: just write a commit record to the temporary log, and then force the temporary log to permanent storage. Note that 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.

Every undo action must possess the following properties. After performing the undo action, it should be as if the normal "do" action was not applied, whether or not the "do" occurred. Furthermore, undo actions should be *idempotent*: if applied more than once it should be as if they were applied exactly once. Every redo action has similar properties it must obey. After a redo is applied, it should be as if the "do" occurred, exactly once, whether or not it actually occurred before. Redos must also be idempotent: if applied more than once, the effect is the same as applying them exactly once.

Here is a list of the various normal actions that impact on recovery, and how they are handled in the single-level transaction system:

- Transaction begin: The transaction is assign a unique transaction identifier (*tid*). A *begin* record is written to the temporary log; it includes the tid.

- An action for transaction t: Appropriate redo and/or undo records are written to the temporary log. Each is marked with the identifier t. The undo record also includes a pointer to t's previous undo (if any) record in the log. (This chaining is supported by keeping track of the most recent undo log record for each running transaction.)

- Transaction commit: A commit record for the transaction is written to the temporary log, and the temporary log is forced (all records up to the commit record are written to a safe place). Then the transaction may release resources, and notify the user that it completed successfully. Note that buffer pages need *not* be forced to disk.

- Buffer page write: First it should be understood that a buffer page cannot be written during an action; this implies that some sort of short term locks are used on buffer pages during critical sections. Second, because we are using careful replacement, a page cannot be written unless there are no dirty pages that must precede it to disk. Finally, the temporary log must be forced up through the most recent action affecting the page. To support this last requirement, each page will have associated with it a *high water mark*: the position in the log of the most recent log record for an action that modified the page. When all the conditions stated are satisfied, a page can be written back to its home on disk.

- System checkpoint: Write a *checkpoint record* to the temporary log. This record indicates which transactions are running at the time of the checkpoint. It also includes information for bounding redo: the *log scan limit*, which is determined in the following manner. For each dirty page, we keep track of when it was dirtied, by noting the pointer in the log to the redo record for the action that dirtied the page. This information is called the *dirty mark* for the page. The log scan limit is simply the minimum of the dirty marks of the dirty pages, at the time of the checkpoint. All actions preceding the log scan limit are guaranteed to be reflected in the database, so redoing from the log scan limit will bring the database up to date. Note that our checkpoint scheme does not require that activity against the buffers be quiesced, since we are not writing the buffers out.

There are several ways in which "hot spot" pages (ones which are almost continuously in use, and hence are never replaced in the buffer) can be handled. Pages that have been resident a long time can be forced at system checkpoint time (this may require quiescing activity temporarily). Perhaps a more flexible scheme is to associate a timer or counter with each page so that after it has been dirty for some period of time (measured either in absolute time, log activity, checkpoints, or some other convenient units) to cause it to be written at the next possible opportunity. We will assume that some such technique is used, since it serves to bound the amount of the log that must be processed for partial redo.

Another background activity that we will not describe in detail is the maintenance of the archival log and archival database. Basically, the redo records of successful transactions (only) are copied from the temporary log to the archival log, as a background activity. Also, the archival log is processed against the archival database as background activity (possibly as an occasional batch job), to bring the archival database up to date and prevent the archival log from growing without bound. Once temporary log information has made it to the archival log, and also is no longer needed for global undo or partial redo, the temporary log information can be discarded. Thus, copying to the archival log helps to bound the growth of the temporary log. However, to achieve a strict bound on the temporary log it is necessary to bound page residency (which we have assumed above), and also to bound the length of a transaction (which should also be done).

Now that we have described normal operation, here is how recovery from the various sorts of failures proceeds:

- Transaction abort: The temporary log records for the transaction are scanned backwards, using the chain. As the records are encountered, the undo actions are performed. This procedure stops at the earliest undo record, which has a null chain. After all undos have been performed, an abort record is written to the log, as the transaction's last log record. The log need not be forced, since a crash will abort any apparently running transactions anyway. The transaction's resources may be released after the undos have been performed. The user can be notified of abort at any point, though it may not be useful to do so until the undos have been performed.

- System crash: First, process the temporary log in reverse chronological order. While doing so, maintain four sets: C, the committed transactions; R, the running transactions; T, the terminated transactions; and U, the transactions needing to be undone. Initially these sets are empty, and the log scan limit has not been set. The log may contain begin, undo, redo, commit, abort, and checkpoint records. Here is how each kind is handled. Begin - if the transaction is in U, remove it; otherwise do nothing. Undo - perform the undo action, and if the transaction is not in T, add it to U and R. Redo - ignore it. Abort - add the transaction to T. Commit - add the transaction to C and T. Checkpoint (first one only) - for each transaction active at the checkpoint and not in T, put the transaction in U and R; set the log scan limit to the value in the checkpoint record. Do nothing for checkpoints other than the most recent one.

Stop the backward scan when U is empty *and* the log scan limit is reached (we must go back at least one checkpoint, to determine the log scan limit). Note that we undo committed transaction as well as aborted ones; this insures that there are no semantic anomalies. Depending on the exact nature of the actions used in a given database system, it may be possible to avoid undoing actions of committed transactions.

Now scan the temporary log forwards, processing records as follows. Begin, undo, abort, commit, checkpoint - no action. Redo - if the transaction is in C, perform the redo action. When the end of the log is reached, write abort records for transactions in R. As a help in bounding later work, we can force all buffers and then take a checkpoint (which will show no active transactions and no dirty buffers), but this is not necessary for correctness.

- Media failure: For a total failure, restore the archival database and process the archival log, in chronological order, and then proceed as in a system crash, except process all of the temporary log that has not yet been moved to the archival log. 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. Then process the temporary log as for a total media failure.[3] Media recovery will work faster if the archival log is kept reasonably up to date. It is possible to use the temporary log as an archival log, though there may be advantages to reorganizing the data when building an archival log, since it is strictly for recovery of physical media.[4]

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

5. Nested Transaction Recovery

Extending the recovery schemes presented above to handle nested transactions is not extremely difficult. In handling transaction abort, the most difficult aspect is unwinding a transaction and its subtransactions at once. In the case of a system crash, the trickiest part is determining exactly which transactions should be undone and redone. To add nested transactions, we proceed as follows. To the existing log record types we add these: *sub-begin*: subtransaction begin; *sub-abort*: subtransaction abort; and *sub-commit*: subtransaction commit. The sub-abort and sub-commit records are on the *parent* transaction's undo chain, but also contain a pointer to the child's most recent undo record. They also contain the child's transaction identifier (as well as the parent's). Note that committing a subtransaction does *not* require forcing the log, since it is not an absolute commit.

Here is how recovery from the various kinds of failure works:

- Transaction abort: This procedure maintains a set A of log pointers for transactions and subtransactions being aborted. Initially A contains the pointer to the most recent undo record for the transaction to be undone, and the pointers for that transaction's running descendants. We iterate, processing undo records as follows. Among the records referred to by A, choose the most recent one, and delete its pointer from A. If the record has a non-null undo chain value, insert that chain value (i.e., the pointer to the previous record) back into A. Note

[3] It may be possible to restrict attention to just those portions being recovered, but that is not always easy, since some actions may affect more than one page.

[4] In particular, there are benefits to organizing the archive log and archive database as physical page images and deltas, rather than higher level semantic operations.

that we will process records in reverse order of their insertion into the log. If the record is a sub-abort, do nothing; if it is a sub-commit, add the child's undo chain pointer to A; if it is an undo record, perform the undo action. Stop this process when A becomes empty. Then write an abort record for the original transaction and each of its running descendants, from youngest (most deeply nested) to oldest (the transaction whose abort was requested). The overall effect is to undo the transaction and all it running or committed children, their running or committed children, etc. Any aborted descendants (and committed descendants under them) can be ignored, since they have already been undone.

- System crash: This proceeds much as for the single-level case. As before, we maintain the sets C (committed), R (running), T (terminated), and U (being undone). Here is how each kind of log record is handled. Begin - if the transaction is in U, remove it; otherwise do nothing. Undo - perform the undo action, and if the transaction is not in T, add it to U and R. Redo - ignore it. Abort - add the transaction to T. Commit - add the transaction to T and C. Sub-abort, sub-commit - add the transaction to T; add to U and R each ancestor of the transaction that is not in T; for sub-commit, add the transaction to C if its parent is in C. Checkpoint (first one only) - for each transaction active at the checkpoint and not in T, put the transaction in U and R; set the log scan limit to the value in the checkpoint record. Do nothing for checkpoints other than the most recent one.

As before, stop the backward scan when U is empty *and* the log scan limit is reached. In the forward scan, process log records as for the single-level case: redo records for transactions in C. When done, write abort records for the transactions in R, from youngest to oldest.

- Media failure: This proceeds as in the single-level case, substituting nested transaction system crash recovery for the single-level version.

6. Correctness

It is extremely difficult to formalize recovery procedures, and we will not attempt to do so here. However, some arguments as to why the above algorithms work are in order. We first turn our attention to the single-level transaction recovery procedures, since the correctness of the nested transaction recovery can be argued incrementally from that point.

6.1 Single-Level Correctness

Let us consider transaction abort first. There are two keys to the correctness of this procedure. The first is that concurrency control has insured that undo operations are still applicable. The second is that undoing in the exact reverse order from the original do operations, using undos that are true inverses of the do operations, will remove the effects of the transaction being aborted. All these points are well understood; still, it is not always trivial to implement the undos.

Crash recovery is more subtle and open to question. The first point to have in mind is that careful replacement causes each individual change to be atomic, and thus either to appear or not appear in the database after the crash (depending on whether the relevant pages made it to disk before the crash). It is also necessary that undos work when the do has not occurred (for operations that are logged but not in the database), and that undos work if applied more than once (for operations that were logged, and undone in the database already, or in case of a crash in the middle of recovery). Similar comments apply concerning redo in the forward scan. The key point in crash recovery is that we undo far enough that we bring the database and log into synchrony: when we stop the backward scan, the database reflects exactly the effects of transactions that committed (or will commit) as of where we stop in the scan. To see this, note that because of the log scan limit, we have gone back to a point where all previous effects have been propagated to the database pages. Also, at the point where we stop, all effects of running transactions have been undone, so they have in fact been aborted. This latter point requires a little care, because of the way in which the checkpoint record helps us determine what transactions were running when the system crashed, but it is not too difficult. Once the backward scan has synchronized the log and the database, the forward scan redoes the operations of the successful transactions. This works because the unsuccessful transactions cannot affect the successful ones in any way; that is, it crucially depends the notion of transactions.

Media recovery, since it works from a "cleaned up" log against a clean archival database, it not particularly difficult to understand. The subtleties come in when one substantially reorganizes the form of the data in the process of archiving it. Since we were not very detailed about the media recovery procedure, we will not argue its correctness further.

6.2 Nested Transaction Correctness

When we extended the single-level transaction abort algorithm to the nested transaction case we did not change the algorithm very much; we did not affect the correctness argument strongly either. To see that our algorithm works, it must first be realized which operations should be considered part of the transaction to be aborted. The correct set of operations is those of the transaction being aborted, less those of inferior transactions that have already been aborted (either directly, or by having some ancestor abort). Our algorithm does indeed process this set. The other fact necessary is to have the undos performed in the correct order; our algorithm also accomplishes that. The extension for crash recovery follows the same principles as the extension for transaction abort, so there is really nothing more to argue about it.

7. 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 begin, 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, 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 (maintain the set A of the transaction abort procedure as a priority queue), 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 identifier in begin, 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.

Since it is apparent that efficient nested transaction recovery is possible, we suggest if efficient nested transaction concurrency control mechanisms can be designed, nested transactions can be offered 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.

[Gray et al. 81] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. L. Traiger, "The Recovery Manager of the System R Database Manager", *AC Computing Surveys*, Volume 13, Number 2, June 1981, pp. 223-242.

[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 Transactions 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 Transactions: 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.

[Siemens] UDS, Universal Data Base Management System, UDS-V2 Reference Manual Package, Siemens AG, Munich, West Germany.

[Verhofstadt 78] J. M. Verhofstadt, "Recovery Techniques for Database Systems", *ACM Computing Surveys*, Volume 10, Number 2, June 1978, pp. 167-195.