8-10-1987

Database recovery

Kim L. Emanuel

Rochester Institute of Technology
School of Computer Science and Technology

DATABASE RECOVERY

by

Kim Emanuel

A thesis, submitted to
The Faculty of the
School of Computer Science and Technology
in partial fulfillment of the requirements for the
degree of Master of Science in Computer Science

Approved by:

Henry A. Etlinger
Henry Etlinger, Chairperson — 8/10/87 Date

Peter G. Anderson
Peter Anderson — 10 Aug 87 Date

Jeffrey Lasky
Jeffrey Lasky — 8-10-87 Date

August 1987

Title of Thesis: DATABASE RECOVERY

I, Kim Emanuel, prefer to be contacted each time a request for reproduction is made. I can be reached at the following address:

Signature: Kim L. Emanuel

Date: August 10, 1987

TABLE OF CONTENTS

INDEX OF FIGURES

I. ABSTRACT

Recovery techniques are an important aspect of database systems. They are essential to ensure that data integrity is maintained after any type of failure occurs. The recovery mechanism must be designed so that the availability and performance of the system are not unacceptably impacted by the recovery algorithms running during normal execution. On the other hand, enough information must be stored so that the database can be restored or transactions backed out in a reasonable amount of time.

Concepts, techniques, and problems associated with database recovery will be presented in this thesis. The recovery issues for both centralized and distributed systems will be discussed, along with the tradeoffs of different recovery tools. The database recovery schemes in IMS/VS, DB2 and SDD-1 will be described to show approaches in existing systems.

II. INTRODUCTION

In a database environment, data integrity and data availability are top priorities. Incorrect data poses a serious impediment to an installation. Loss of data may occur due to application failures, hardware problems, system problems, or user interaction. Recovery procedures are essential in order to recover the data after a failure.

Because of the variety of failures, there is always a limit to the amount of recovery which can be provided. If a situation corrupts the recovery data in addition to the normal data, complete recovery may be impossible. Rare failures may not be accounted for or may be considered too expensive to recover from. In addition, the overhead costs to maintain the recovery data are a big consideration when determining what types of failure must be recoverable.

Recovery data is basically historical data, consisting of previous copies of the database and changes which have occurred since those copies were made. Redundancy of recovery data is another safeguard against unrecoverability. Recovery provided by database management systems can be enormous time savers. In addition, products exist which monitor database activity and automate the recovery process. In large installations, such products become heroes in bad situations.

In the following sections, a survey of recovery techniques is presented. Centralized and distributed database concerns are addressed. In addition, several existing systems are examined in terms of their recovery subsystems. Case studies are included to highlight installation concerns for these products.

III. CONCEPTS OF DATABASE RECOVERY

A. TRANSACTIONS AND CONSISTENCY

In order to comprehend the concepts behind current recovery techniques, the notions of transactions and consistency must be understood. A transaction is an atomic action which represents one meaningful unit of work by a user. An important point regarding transactions is that all parts of a transaction must be executed as a whole. All changes should be reflected in the database or else nothing should be shown. Consider the standard example where an amount is being transferred from one account to another. The system would be left in an inconsistent state if the amount were subtracted from account A but not added to account B due to an abnormal termination. The actions just described should be considered as a single transaction.

To achieve the indivisibility described, a transaction must have four properties: [HAER83]

1. ATOMICITY. It must be of the all-or-nothing type described above. If a transaction performs some updates, and then a failure occurs, the updates must be undone. The transaction either completes entirely or no evidence remains of partial execution.
2. CONSISTENCY. A transaction reaching its normal end, thereby committing its results, preserves the consistency of the database. In other

words, each successful transaction, by definition, commits only legal results. This condition is necessary for the fourth property, durability.

3. ISOLATION. Events within a transaction must be hidden from other transactions running concurrently. The techniques that achieve isolation are known as synchronization.

4. DURABILITY. Once a transaction has been completed and has committed its results to the database, the system must GUARANTEE that its results are properly reflected in the database in spite of subsequent malfunctions.

It follows from the definition of a transaction that, if a failure occurs which prevents the successful completion of a transaction, the state of all objects that the transaction has modified must be restored to their state prior to the transaction. Otherwise, failures would allow intermediate states of objects resulting from partial execution to be observable by processes outside the atomic action. Synchronization techniques prohibit access to affected objects by other independent transactions until a transaction either commits or is recovered.

B. TYPES OF FAILURES

The recovery component of a system must consider several types of failures. Certain failures are extremely rare. The cost of redundancy needed to cope with them may be so high that it may be a sensible design decision to exclude these failures from consideration. If one of them does occur, however, the system will not be able to recover automatically, resulting in a corrupted database. This type of catastrophe is not considered in this paper.

The following types of failures must be handled by the recovery mechanism: [GRAY79, HAER83, DATE86]

1. TRANSACTION FAILURE. This failure has already been mentioned in the previous section. For various reasons, a particular transaction cannot proceed and is aborted. The transaction must then be backed out as if it never occurred. Examples of such errors are application program bugs, timeouts, and protection violations.
2. SYSTEM FAILURE (Soft Crashes). In this case, processing is terminated in an uncontrolled manner, and the contents of main memory are lost. All transactions which are currently being processed are affected. Database-related secondary (non-volatile) storage remains unaffected. An example of this type of failure is a loss of power.

3. MEDIA FAILURE (Hard Crashes). Besides transaction and system failures, the loss of some or all of the secondary storage holding the database must be anticipated. If the device contained recoverable data, the manager must reconstruct the data from an archive copy using the log and then place the result in an alternative device. Media failures are rare, since magnetic storage devices are usually very reliable. Parity error, head crashes, and dust on magnetic media are typical causes for media failure.

C. OVERVIEW OF RECOVERY ACTIONS

The recovery manager of a database system is an important component in providing for data integrity. Actions performed during the recovery process are initiated through the recognition of a problem. The type of failure must then be determined, and analysis is needed to detect the cause of the failure. The extent of the damage is assessed by evaluating the status of files and transactions. A recovery method is then selected. Programs are run to reconstruct a damaged database, backout bad transactions and possibly restart them, and perform other activities necessary to reestablish normal operation.

The highest emphasis on recovery goals should be data integrity and consistency. Hopefully, a consistent state of the database as it was immediately before the crash can be obtained. However, in some cases, the database can only be restored to a past state, with the loss of later data.

In distributed systems, recovery becomes even more complex. Local sites must have their own recovery facilities to restore their portion of a transaction, or handle their media problems, etc., during failures. However, distributed recovery facilities are also required to interface between the separate sites. Communication problems may arise which cause partitions in which sites or groups of sites become isolated from one another. To avoid losing database updates and endangering data integrity, procedures must exist to handle such situations.

Even normal operation becomes more complicated in distributed systems, since the recovery managers at different sites need to be coordinated to uniformly commit or reject the effects of multi-site transactions.

IV. RECOVERY METHODS

A. RESTORE/RERUN RECOVERY

This method of recovery involves restoring the database to a previous state, using the latest database copy. Then all the programs, jobs. or transactions that have executed since that last copy are rerun.

This is a very simple approach. For on-line systems, the only requirement is that all the transactions must be saved. For batch jobs, the input transactions are usually still available after the job completes.

This approach is very common in non-database applications. The output datasets of a job can be eliminated in case of an error, and the job can be rerun after the program or data has been corrected. This method has advantages of simplicity and no need for journalizing database changes.

However, there are two main disadvantages. First, the time required to rerun all the transactions since the last database copy may be unacceptable. Often, database copies cannot be made frequently, due to their size. This will have a big impact on recovery speed. A huge number of jobs may have to be rerun in the event of a failure.

Another disadvantage to this approach is the sequence of the transactions. If a database management system operates in a single-thread mode, then the order of transactions can be determined by unique date/time stamps.

This will allow transactions to be reprocessed in their original sequence. However, if the system allows transactions to interweave, then it will be impossible to duplicate the original order of the transactions. This may cause different results when a rerun is performed, which may be unacceptable [KING81].

These disadvantages are severe enough that the restore/rerun recovery approach is not often used in database systems.

B. FORWARD RECOVERY

Forward recovery involves restoring the database to a backup copy and reposting all changes which have been made since then. A log file must be maintained, indicating all database updates. The reposting is done by applying all afterimages of changed database records to the restored database. Figure 1 describes the forward recovery process. This approach is faster than the restore/rerun method. Only the database changes are redone, eliminating the time to reprocess all the other logic contained in the update jobs. The length of time since the last database copy is still an important concern, and greatly affects the recovery speed.

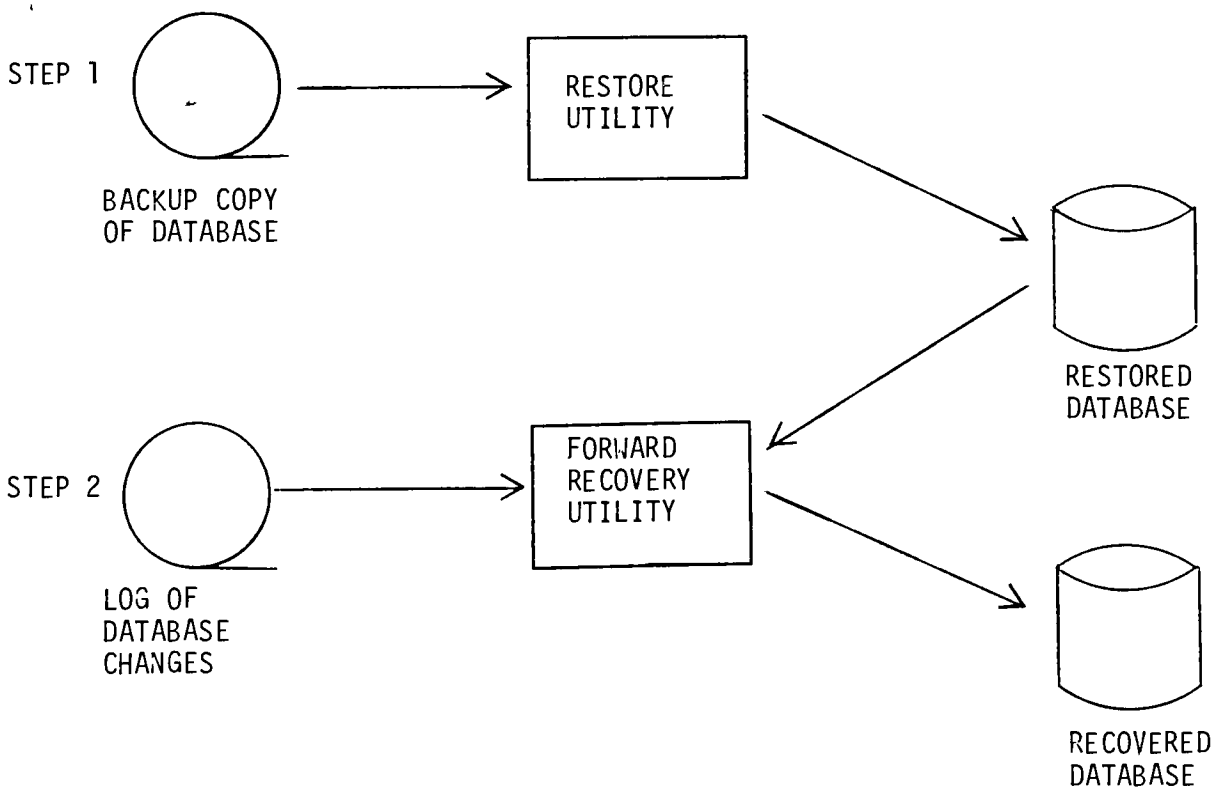


Figure 1. Forward Recovery.

In order to reduce the time required for forward recovery, the logs can be evaluated to combine records. The change accumulation facility of IMS/VS accomplishes this. If a record has been changed more than once since the last database copy, only the latest afterimage is needed. Processing earlier updates of the same record only wastes time.

Forward recovery is best used when the secondary storage containing the current database has been damaged. Backward recovery, discussed below, will not be successful because the current database is needed. However, many failures do not require forward recovery. If a program terminates abnormally after updates have occurred, any partial updates must be removed. As long as the database remains intact, backward recovery may be the best solution.

C. BACKWARD RECOVERY

Backward recovery is used to remove any effects of a transaction which has not completed successfully. Often, application programs are treated as one transaction. If the program is very time-consuming, then checkpoints may be added so that, after a certain time, all the program's updates are committed. Thus, backout recovery is usually used to back up to the beginning of a program, or to the last commit point.

Backward recovery is needed only if database updates have occurred since the program started. The before-image of all database records which were changed must be stored in a log. Backward recovery works by applying the before-image to the changed record, so that the update is basically undone. All the before-images for the aborted transaction will be processed.

The top portion of Figure 2 illustrates an application program which aborted. In this scenario, the entire application will be treated as one transaction. The backout utility processes the log to undo all the updates done by the application. At the end of this process, the contents of the "B" version of the database are equivalent to the values contained in Version A. The effects of the application program have been erased.

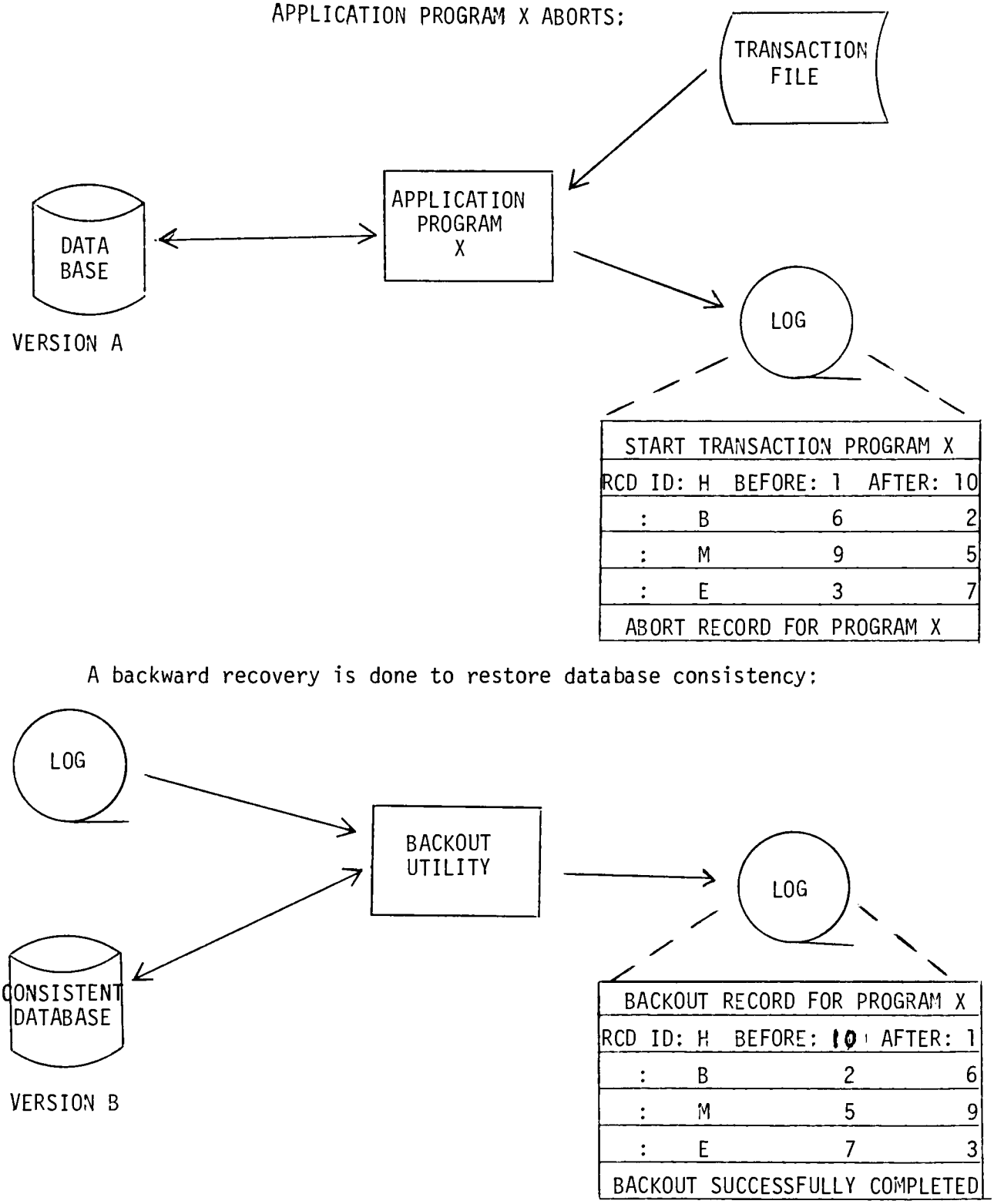
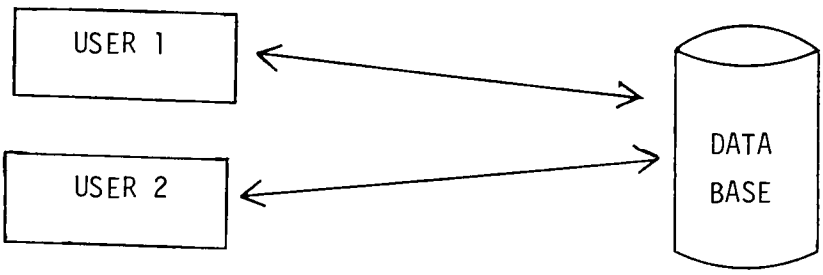


Figure 2. Backward Recovery.

Most installations use a lockout mechanism which spans a transaction. Any changes made by a transaction which has not completed are unavailable for use until the transaction has successfully completed. If this lockout procedure does not exist, backward recovery becomes more complicated. Transaction A may have updated a record, followed by Transaction B, which updated the same record. If Transaction A fails, and must be backed out, Transaction B must also be backed out. This is often referred to as a "domino effect." Identifying all the records updated by a failed transaction which were later updated by other transactions makes the recovery procedure much more involved. [RAND78] discusses the use of recovery lines to determine how to backout processes involved in a "domino effect."

Figure 3 describes the problem encountered when transactions can see the results of uncommitted transactions. Both User 1 and User 2 were concurrently updating the database. User 2 updated the record just modified by User 1. When User 1 fails, the normal system action would backout User 1's updates. If this backout is performed, the value of the database record will be restored to 40, and the effect of User 2's input is lost. Therefore, User 2 must also be forced to abort, and have his/her updates undone. The problem could have been avoided with record locking, since User 2 could not have read the record until User 1 released its lock.



JOURNAL INFORMATION:

TIME	USER	ACTION	OLD VALUE	NEW VALUE
T1	1	READ RECORD FOR UPDATE	40	
T2	1	UPDATE RECORD	40	15
T3	2	READ RECORD FOR UPDATE	15	
T4	2	UPDATE RECORD	15	3
T5	1	ABORT TRANSACTION		

Figure 3. Domino Effect.

The major advantage of backward recovery is the speed of recovery. Usually, the time since the last database copy is long, so backouts are much faster than forward recoveries. Backward recovery and forward recovery can both be used to restore the database after an application abend. Both methods restore the database to the point in time immediately preceding that when the application started. Usually, batch backout is preferred, since the image copy might have been taken a long time ago.

D. OTHER TYPES OF RECOVERY

Recovery schemes exist to allow partial recovery of databases. The area may be a certain key range, direct address, dataset, track, or other identifiable unit. The capability to restore the contents of a defective area may save considerable time in returning service to the users. In some systems, other portions of the database may be processed while the damaged portion is being repaired. Precautions must be taken so that the integrity of the database is not affected by a partial recovery. Support for these recoveries may be included in a database management system. The integrity of the database is the primary concern in all of these recovery cases.

Usually, full recoveries of database datasets are performed which bring the database back to the most current state. Similarly, backward recoveries are done to backout entire transactions. Time-stamp recoveries are different since they restore the contents of a database dataset to the values existing before the time specified. The indicated time may be in the middle of a transaction or a few days in the past, depending on the rules of the recovery utility. Time-stamp recoveries are useful for recovering databases to a time just prior to when a logical error occurred.

V. TRANSACTION RECOVERY PROTOCOLS

A. LOGGING CONCEPTS

If updates are to be performed directly on the physical database (update-in-place), the database management system needs to record all changes that are made. A dataset must be created containing information about changes made to the databases. These datasets are called logs, journals, or audit trails. The types of records written to the journal will depend on the scope of recovery supported. Logs may be used to satisfy system integrity, recovery, auditing, and security requirements. However, this paper analyzes logging in recovery terms only.

The purpose of the logs is to allow re-creation of files. Each time a transaction modifies a file which is being logged, a new log record is made. Read actions do not need to generate any log records. However, update actions on logged files must enter enough information in the log so that, given the log record at a later time, the action can be completely undone or redone. An update action must record the following in the log: file name, record identifier, old record value, and new record value. The log subsystem adds other fields to the log--transaction identifier, action identifier, timestamp, length of log record, and pointer to previous log record of this transaction [BLAS81].

One logging method only requires that the before images of pages are written in the log [BAYE84]. If a page which has been updated is chosen to be swapped out, then the before-image is written to the log before writing the changed page to the physical database. Before a transaction is allowed to commit, all updated pages must have their before-images reflected in the log. As part of the commit action, all changed pages are forced to the physical database.

A major disadvantage of the before-image-only technique is that the commit processing is slow. The forcing of changes to the database creates an I/O bottleneck. The pages are at random locations on the physical database, making the updating slow.

Another logging concept removes the need to force pages to the database before committing. The idea is to write both before and after images to the log. The before images are written when the page is updated in the buffer. The after image is written before an updated page is swapped out or when a commit occurs. In this approach, a page may be updated by a number of transactions without being written to the database. Although this saves I/O time, the physical database can quickly become obsolete. Restart is more complicated, since the log must be analyzed to determine the current versions of pages if they were not written to the database. In addition, since a system crash destroys the buffer, any updates not written to the physical database must be redone.

It is essential that the log data be reliable. If the log datasets are destroyed, it may render the system unrecoverable to an acceptable state. For this reason, many systems maintain two or three identical copies of the log tapes. The use of logs in a database recovery procedure is shown in Figure 4. An image copy and logs are common inputs to many recovery utilities and are needed to create a consistent version of the database.

There are differences of opinion regarding the best medium for log files. Tape is the popular choice because it is inexpensive and most installations can rarely spare disk space from online storage needs. Disk, on the other hand, is much faster and usually considered more reliable than tape. It can easily be read backward or randomly, can be read simultaneously by more than one process, and can be spooled to tape later.

The following options should be considered when deciding how a system's audit trail should be organized. Each program may have its own audit trail. This is useful to isolate changes made by a transaction when a backout is necessary. A cutoff time interval can exist where the log tapes are created to cover specific time periods. The logs may also be portioned on the logical database level. One audit trail may exist for all information in the database on a common subject. In this case, one audit trail would have a scope of one or more datasets. Since audit trails may be used for other means than recovery, the operations upon data is another way to break apart the logs. All update operations could be kept in a separate

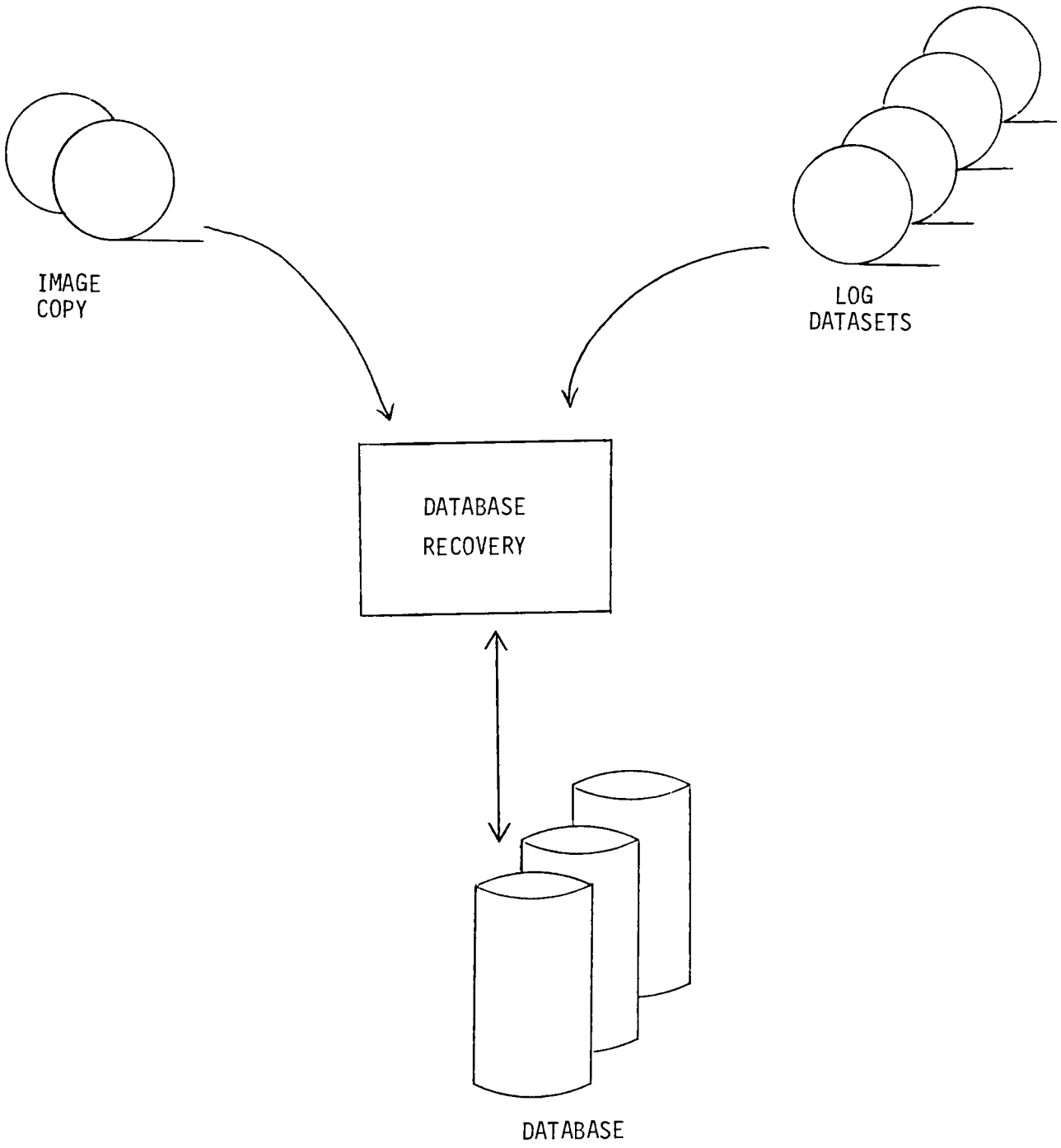


Figure 4. Logging.

log, since only those operations are needed for recovery. Other actions on the data may be kept in other logs [BJOR75].

In many database management systems, the page is the unit of log information. Even though only a field or record may be changed, the entire page is copied to the transaction log. This is done for two reasons: simpler restoration and better protection. If only the record or field to be changed is logged, the restoration requires a merge of old and new records. This merge can be complex if records vary in size, and could create block-overflow situations. Logging field changes are done if storage problems are anticipated.

Transaction termination, both commit and abort, are recorded in the log. Recovery management also makes various notations in the log to facilitate its recovery procedures. These records include checkpoint records designed to bound the amount of the log which has to be scanned during restart recovery.

The log is stored in volatile memory and in non-volatile storage. The non-volatile log may be duplexed for reliability. The writing of log records to non-volatile storage must be carefully synchronized with events affecting the database. Sometimes it is necessary to force log records to the non-volatile log in order to make some event recoverable. It is also important to control the order in which log records and the corresponding changed database pages reach the non-volatile storage.

Log Data Classification

One topic regarding logging is the type of objects to be logged. [HAER83, REUT84] If some part of the physical representation (the bit pattern) is written to the log, it is referred to as physical logging. If the operators and their arguments are recorded on a higher level, this is called logical logging. Logging on the logical level means that the INSERT, UPDATE, and DELETE operators, together with the record IDs and attribute values, are written to the log. Recovery is performed by re-executing the previously processed database commands. UNDO recovery is much more complicated, since the reverse commands must be derived. The amount of logged data is smaller, but recovery is much more expensive.

If the log data reflects the state of an object before or after a modification, the condition is state logging. This results in the actual before and after image being written to the log. In comparison, transition logging reflects the change of one state to the logically succeeding one. Transition logging writes the difference between the old and new page states to the log. The difference is determined through using the exclusive or boolean operation. If there are multiple changes applied to the same page during one transaction, transition logging can express these by either successive differences or one accumulated difference. Transition logging lends itself to compression, thus requiring less space for

logging than state logging. The limitations of transition logging come from the difficulty of constructing types with operations that are practical to invert. It may be difficult to know, at the time the log is written, exactly what information will be needed to invert the operation if a backout is necessary. Examining the log and undoing the operations may be expensive.

Write Ahead Log Protocol

The recovery system views memory as two types: volatile and non-volatile storage. Volatile storage does not survive a system crash, while non-volatile storage usually does survive.

Suppose the log records for an object are recorded in non-volatile storage after the object is recorded in non-volatile storage. If the system crashes at such a point, one cannot undo the update, since the log would not indicate that it had taken place. Similarly, if the new object is one of a set which are committed together, and if a media error occurs on the object, a mutually consistent version of the set of objects cannot be constructed from their non-volatile versions. Analysis of these two examples indicate that the log should be written to the non-volatile storage before the object is written.

The process of ensuring that recovery information is actually written to a log dataset before performing an operation that must be recoverable is referred to as the Write Ahead Log Protocol. This concept is discussed in [STRI82, GRAY80, GRAY78]. It is a key element in IMS/VS 1.3 logging.

At system restart, a transaction may be undone or redone. If an error in the restart occurs, the restart may be repeated. If so, an operation may be undone or redone more than once. Also, since the log is "ahead of" non-volatile storage, the first undo may apply to an already undone (not-yet-done) change. Likewise, the first redo may redo an already done change. This requires that the redo and undo operations must be repeatable so that doing the same operation more than once will produce the same result.

Log Compaction

Database recovery using logs often has the following inefficiencies:

1. All the records on the log are read even though certain record types are utilized.
2. Only a fraction of the database needs to be recovered, but the complete log must be reprocessed since the relevant information is scattered throughout the log.

3. A given record may have been updated many times, resulting in a number of log records. All the log records will be read and reapplied to the database even though later afterimages will overwrite earlier ones during recovery.

A bit-map technique [KAUN84] can produce a compacted, unsorted version of the log which contains only recovery information, that is, the latest afterimage of records updated during the time period. Compaction is directed by a bitmap, where each bit represents a physical database record or page. The algorithm requires that the log be read backwards after the bit map has initially been cleared. Suppose a database must be restored to its current state from the most recent backup. In this case, only afterimages need to be examined. If the bit corresponding to the database record is clear, then the afterimage is copied to the new log and the bit in the bitmap is set. If the corresponding bit in the bitmap is already set, the afterimage is ignored. Compaction brings significant improvements in situations where the ratio of irrelevant to relevant record types is high.

Undo Actions

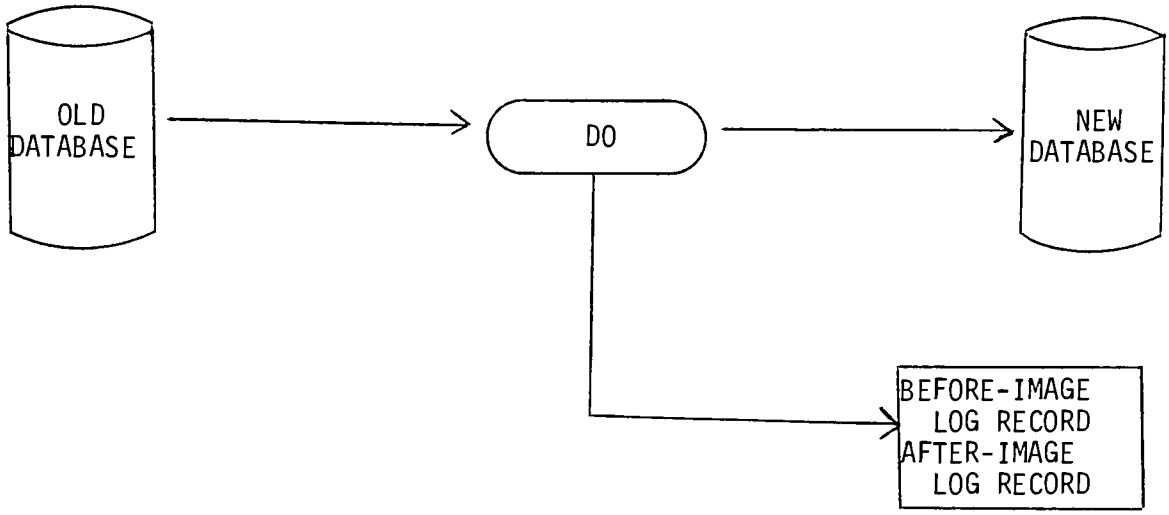
During normal operation, modified pages are written to disk by some replacement algorithm. Whether a page is swapped in or out is usually based solely on buffer management. In general, even dirty data (pages modified

by uncommitted transactions) may be written to the physical database. If a transaction is aborted, an UNDO action must remove all effects of the transaction from the database. This is done by applying the before-images stored in the log to any pages updated in the physical database and the buffer.

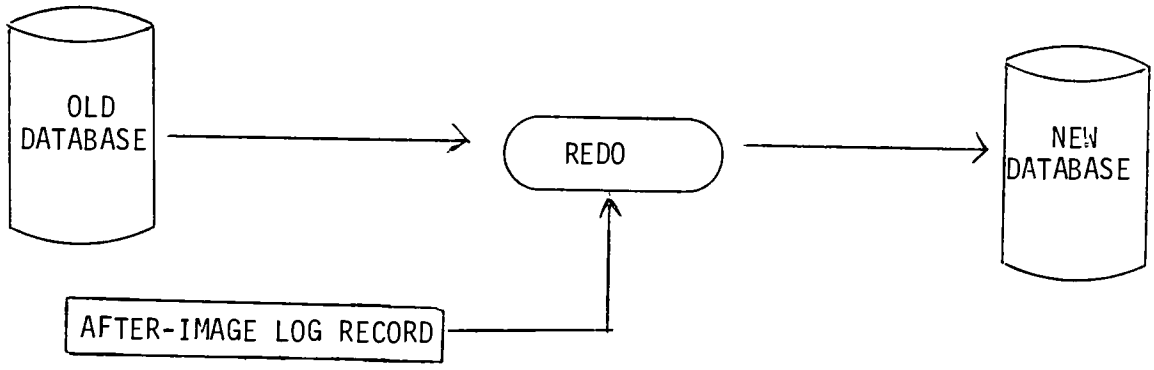
The buffer manager may be prevented from writing uncommitted pages to the physical database. In this case, the UNDO protocol only needs to be concerned with fixing main storage. However, very large database buffers would be required for long batch update transactions, making this scheme incompatible with existing systems.

Redo Actions

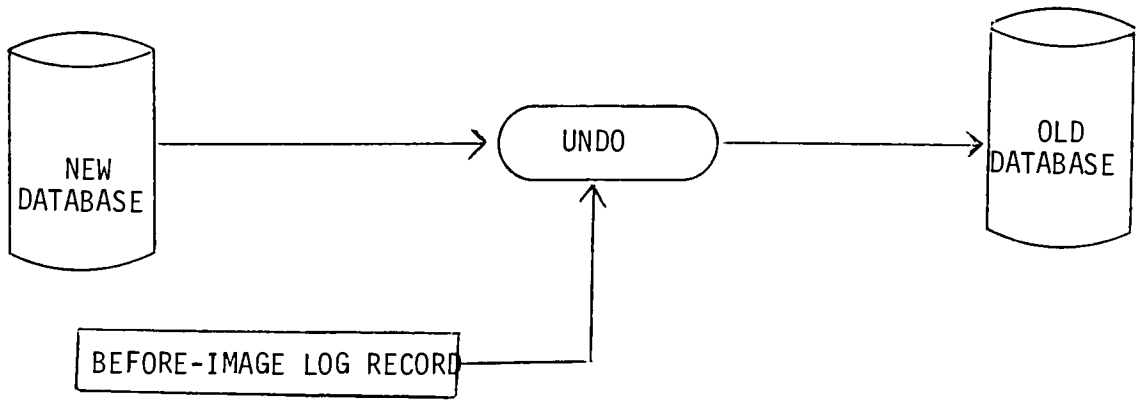
As soon as a transaction commits, all of its results must survive any subsequent failure (the durability property). Committed updates which had not yet been written to the physical database would be lost in a system crash. In this case, the after-image of the record would be applied to the physical database record. To avoid the need for REDO, all modified pages may be forced to non-volatile storage when a transaction commits. Refer to Figure 5 for an illustration of Undo and Redo actions.



*DO Action Generates a New State and Log Records.



*REDO Generates New State From Old State Using After-Image Log Records.



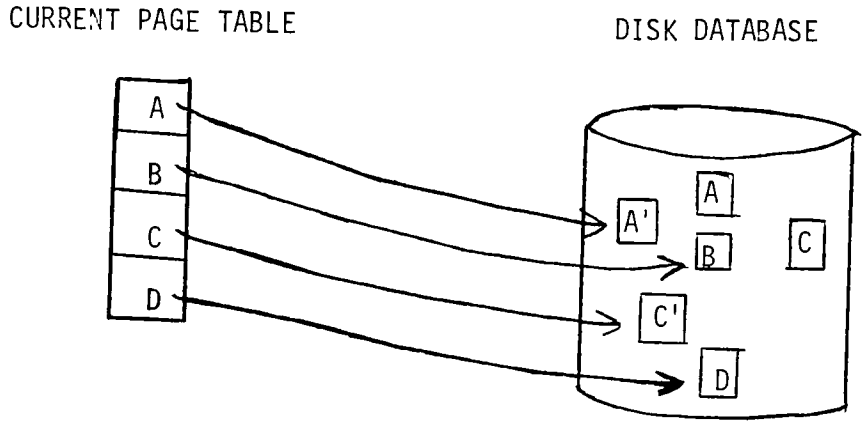
*UNDO Recreates Old State From New State Using Before-Image Log Records.

B. SHADOW FILES

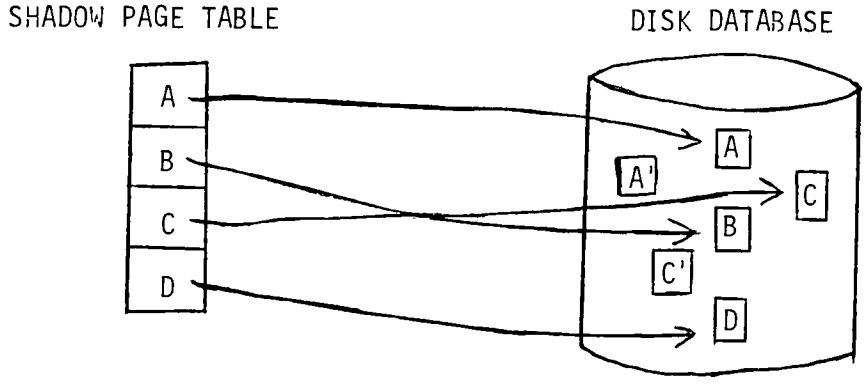
The fundamental idea of shadows is not to do in-place updating, but rather to keep two copies of an object being updated while the transaction is still active. When the object is changed, the old version is kept as the shadow copy, and the modified object will be used during the remainder of the transaction. When a commit occurs, the shadow copy is replaced by the updated copy.

The model of the shadow mechanism presented in [AGRA85b, AGRA85c] requires a shadow page-table which contains the physical addresses of the data pages in the file. When a transaction updates a data page, a new disk block is obtained for the updated copy, and its physical address is recorded in an incremental current page-table for the transaction. Any future retrievals and updates affect only the current version of a file and never alter the shadow version. At commit time, the shadow page-table is carefully updated, using the current page-table.

Recovery using shadow pages is straightforward. The data pages created due to updates by the transaction are freed. The current page-table is discarded, with the result that the shadow page-table contains the mapping to the pages. Figure 6 illustrates the differences between the current and shadow page tables.



Current Page Table Points to the Most Recently Updated Pages.



Shadow Page Table Only Points to Pages Updated by Committed Transactions.

Assume that a transaction is currently executing. Pages A and C were modified. The current page table will reflect the new pages, while the shadow page table points to the old pages until a commit occurs.

Figure 6. Shadow Pages.

The shadow concept does not appear to be advantageous for large files [BLAS81], due to the large amount of disk space needed for the shadow pages. In addition, the logic needed to find a requested page becomes more complicated. Often, more disk seeks are required in order to obtain a page, since the page location moves when it is first updated.

[AGRA85c] suggests two approaches that can be used to improve the performance of the shadow recovery mechanism:

1. reduce the penalty of indirection through the page table to access pages, and
2. avoid indirection altogether.

The first suggestion, reducing the penalty of indirection, may be done by keeping page tables on one or more page-table disks with separate page-table processors. If accesses are uniformly distributed across the page-table processors, the processors may work in parallel to reduce the indirection problem.

One technique to avoid indirection altogether is to alternately use two physically adjacent blocks on a disk to hold the original (shadow) and updated (current) copies of data pages and retrieve both copies in response to a read request. A version selection algorithm is used to determine the current copy. A timestamp may be stored with each page to indicate the page version. The cost of retrieving two versions of a page is likely to be less than the cost to perform the mapping function of the page table to obtain only the current copy.

The existence of shadow files can also be used to increase parallelism in database systems [BAYE80]. Usually, an update job gets an exclusive lock on any objects it modifies. This means that any job just wishing to read the same object is prevented from doing so. However, since shadows create two values for each updated object, the old (shadow) value could be used by any read job.

Another advantage of shadow files, which also applies to differential files (discussed below), is that there is always a way back to the old state. Propagation of an arbitrary set of pages can be made uninterruptable by system crashes. In comparison, update-in-place strategies can leave the database in an inconsistent state after a system crash.

C. DIFFERENTIAL FILES

Under the differential file scheme, a sequential file is used to store an identified and dated copy of each new or changed database page or record segment at the time that it is modified. With this file, transaction reprocessing is not required in recovery procedures. Once the latest dump is restored, the differential file is applied. The log is sorted by identifier and date, and the latest version of each modified record is selected and written directly to the database. Figure 7 demonstrates the use of differential files.

For large databases with moderate or naturally concentrated update activity, differential files offer an alternative strategy for reducing backup and recovery costs. A differential file isolates a database from the physical change by directing all new and modified records onto a separate and relatively small file of changes. Since the main database is never changed, it can always be recovered quickly from its dump in the event of a loss. Transaction reprocessing is required only in the event of damage to the differential file [AGHI82].

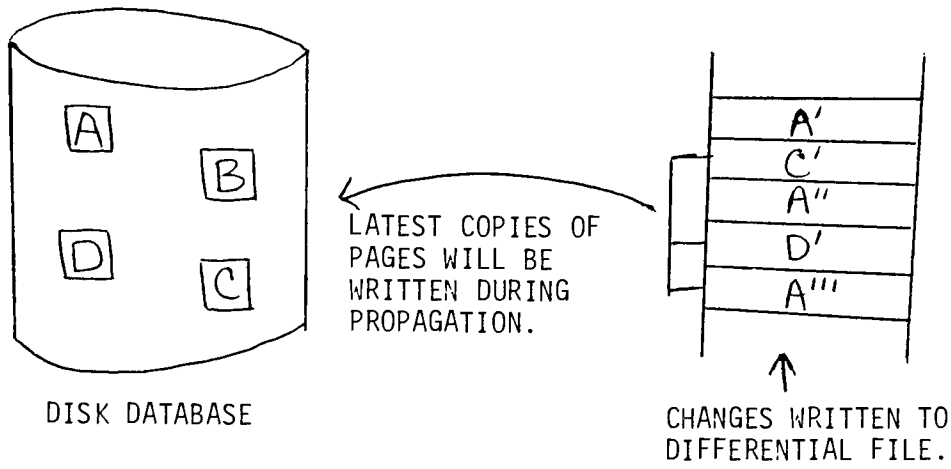


Figure 7. Differential Files.

There are different possible access strategies depending on how the data is indexed. If both the differential file and the database are accessed through a common index, one search through the common index will indicate what file and point to the correct record. If the files have separate indexes, the differential file is always searched first. If the record is not found, the data is retrieved from the main database. The most recent entry for a given record in the differential file must always be retrieved.

Lohman and Severance [LOHM76] describe these record-accessing techniques, and also a revised technique to reduce the number of unsuccessful searches in the differential file. A hashing function is presented to implement it. A small, associative memory, in the form of a bit map accessed by the hashing scheme, is checked to see if the bits for a record are set or not before accessing that record. If the bits are set, the record is probably, but not definitely, in the differential file. A filtering error occurs when the bit map suggests wrongly that the record is in the differential file. The probability of a filtering error at a point in time is a function of the number of main database records which have been changed and the proportion of bits in the bit map which are turned on. The hashing function maps the record address onto a number of bits in the bit map. The bits for a particular record may be set because each of them occurs in at least one set of bits associated with another record in the differential file.

A number of advantages are described in [LOHM76]. Backup costs, recovery time, and chances of serious data loss are reduced. Software can be simplified since the main database is read-only. The differential file technique makes incremental dumping very easy to implement. Another advantage claimed is the possibility of performing queries which do not need the exact values of all files; such queries access a suitable (current) view of the data without locking out the update transactions.

The disadvantages of the approach using differential files are [LORI77]:

--An access to a data element appears slow: if the element is not among the modifications when the differential file is searched, the desired element must be retrieved from the database.

--Eventually, a merge of the changes in the differential file and the main database will be needed, and this operation may be time-consuming. This may be a big problem if the system needs to be available without interruption.

--An update can affect an element which has already been modified. The organization of the differential file must account for this.

The idea of breaking a differential file into two files is presented in [AGRA85c]. One dataset will contain additions to the main file; the other dataset contains deletions. In this scheme, a database consists of the following view: the Base read-only portion combined with the Addition Differential file, and eliminating all records in the Deletion Differential file. The major cost overhead of this approach consists of the I/O cost of reading extra pages from two differential files and the extra CPU time needed to retrieve the records.

D. CHECKPOINTING

The purpose of checkpoints is to reduce the amount of log which must be scanned and the number of pages which must be fetched during restart. This must be balanced against the cost of taking frequent checkpoints. Whether checkpoint information is written in the log or elsewhere depends on specific implementations. Generating a checkpoint involves three steps [GRAY78]:

1. Write a BEGIN-CHECKPOINT record to the log.
2. Write checkpoint information to the log and/or database. Four different criteria, introduced in [HAER83], can be used to decide when to start checkpoint activities.
3. Write an END-CHECKPOINT record to the log. During restart, if the END-CHECKPOINT record is not found, the checkpoint is considered to have failed. The checkpoint record contains a list of active transactions and indicates whether or not any pages remained in the buffer pool.

Transaction-Oriented Checkpoints

One buffer-handling technique forces all modified database pages to be written to non-volatile storage before a transaction is allowed to end. In this scenario, the end of every transaction can be considered as a checkpoint, since the forcing of pages to disk limits the scope

of a REDO. Figure 8 illustrates transaction-oriented checkpoints. In this scenario, no REDO is necessary for completed transactions. When recovering from the crash in Figure 8, T_1 and T_3 are not affected. T_2 will have to be rolled back and resubmitted, since it had not yet committed. The major drawback of this approach is that hot-spot pages will be propagated each time they are modified by a transaction, even though they remain in the buffer for a long time. The cost factor of unnecessary write operations performed by forcing changes to non-volatile storage before a commit can be very high for large database buffers. For systems supporting large applications, transaction-oriented checkpointing is not a cost-effective choice.

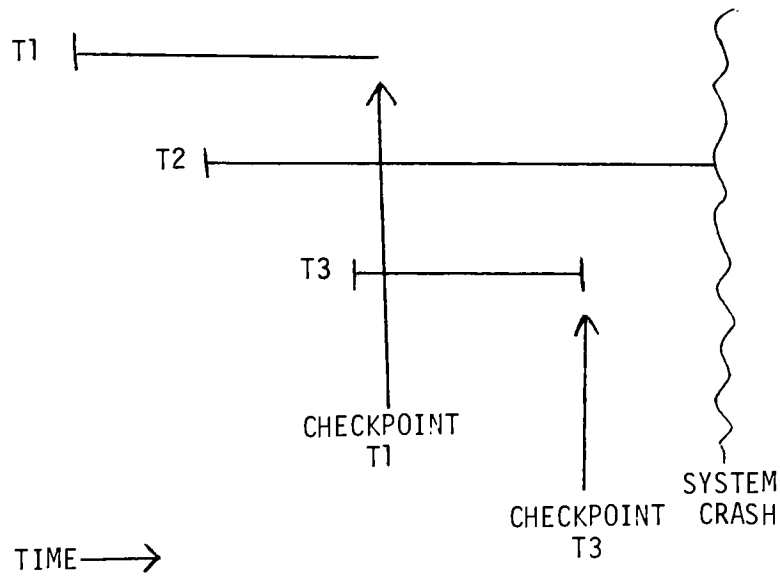


Figure 8. Transaction-Oriented Checkpoints.

Transaction-Consistent Checkpoints

This form of checkpointing ensures that the physical database will contain modifications of only completed transactions at the time of the checkpoint. As denoted in Figure 9, when the recovery component signals the checkpoint generation, all incoming transactions are suspended, and all active transactions must be allowed to complete. New transactions will be delayed until the checkpoint is completed. Checkpointing involves writing all modified buffer pages to the physical database and writing a checkpoint record to the log. When the checkpoint has been taken, a logically consistent state of the database has been determined, which can be used for recovery.

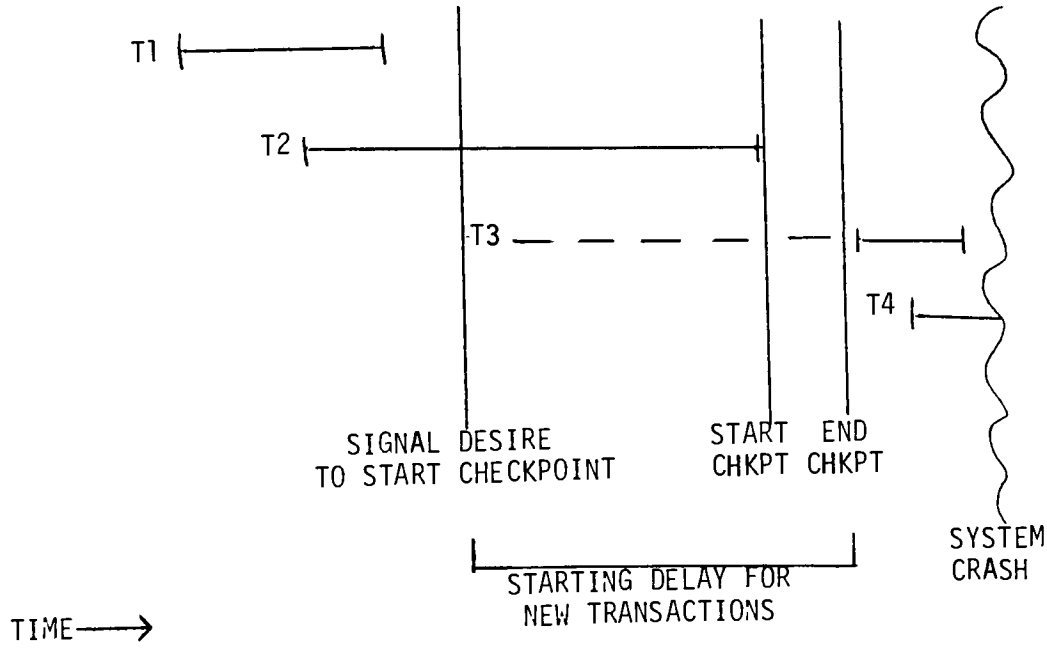


Figure 9. Transaction-Consistent Checkpoints.

The transaction-consistent approach is often unacceptable for large databases with high availability requirements. First, quiescing the system until no update transaction is active may cause an intolerable delay for incoming transactions. In addition, checkpoint costs will be high in the case of large buffers where many changed pages have accumulated, which makes the propagation time long. However, for small applications which can set aside time for these checkpoints, this scheme can be very useful.

Action-Consistent Checkpoints

Each transaction is considered a sequence of smaller actions affecting the database. Action-consistent checkpoints can be generated when no update action is being processed. The checkpoint itself is generated in the same way as was described for the transaction-consistent technique. Figure 10 points out the action-consistent approach, and illustrates how the total delay for all transactions becomes considerably less when compared with transaction-consistent checkpoints.

This technique is used in System R [BLAS81] and does not seem to adversely affect system availability. The transaction-consistent checkpoint approach creates a point where the database represents only committed transactions--no REDO is necessary. However, restoring to an Action-Consistent Checkpoint may involve REDO actions.

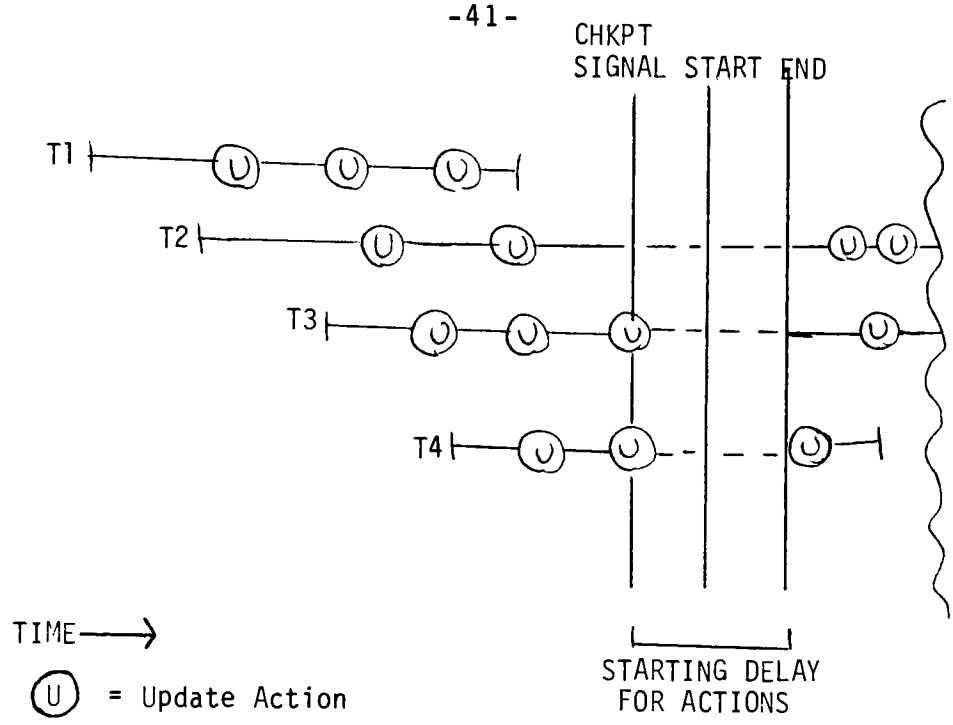


Figure 10. Action-Consistent Checkpoints.

Fuzzy Checkpoints

In order to further reduce checkpoint costs, propagation activity at checkpoint time has to be avoided whenever possible. Instead of writing the changed pages to disk, information about the buffer occupation is written to the log file. The information is used to determine which pages containing committed data were actually in the buffer at the moment of the crash. However, for frequently used pages, REDO information may be found very far back in the log.

In the checkpoint approach presented in [GALT79], the numbers of all pages (with an update indicator) currently in the buffer are written to the log file. If there are no hot spot pages, nothing else is done. However, if a modified page is found at two subsequent checkpoints without having been forced to disk, it will be propagated

VI. SYSTEM FAILURE

System failures interrupt the transactions currently in progress, but the physical database is intact. Basically, two sets of transactions must be identified. Transactions which were being executed cannot continue, so they must be undone. Other transactions which had committed may have updated the database in main storage only. These transactions must be redone in order for the physical database to contain the changed pages.

Checkpoints are crucial in determining what transactions must be undone or redone. Most checkpoint methods force changed database pages from the buffer to disk, write checkpoint records on the log, and create a list of all transactions being executed during the checkpoint [DATE86].

The following example will illustrate how the undo and redo lists are generated based on checkpoint information (see Figure 11).

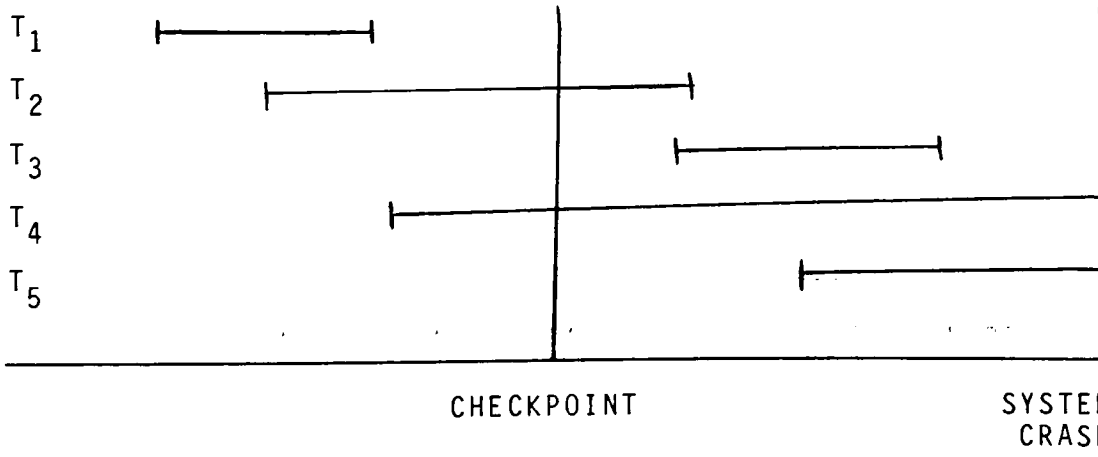


Figure 11. Transaction States After System Failure.

Five possible states of transactions are shown:

1. T_1 began and ended before the checkpoint.
2. T_2 began before the checkpoint and ended before the crash.
3. T_3 began after the checkpoint and ended before the crash.
4. T_4 began before the checkpoint, but no commit record appears on the log, since it never ended.
5. T_5 began after the checkpoint and never ended.

Transaction T_1 started and finished before the checkpoint. This implies that its updates are reflected in the database and no action is necessary. Transactions T_2 and T_3 finished after the checkpoint so their changes may not have propagated to the database. For this case, T_2 and T_3 must be redone, although T_2 only needs to be redone since the checkpoint. Conversely, T_4 and T_5 must be undone since they did not complete successfully before the crash. If only complete transactions are allowed to update the physical database, then T_4 and T_5 are already undone. If the checkpointing scheme forces pages to disk, then T_4 must be undone since the checkpoint.

The specific algorithm to decide how to treat each transaction type depends on the checkpoint and propagation strategies in the database system. However, a general algorithm follows [DATE86].

1. Begin with a REDO and an UNDO list. The UNDO list should contain all active transactions at the time of the checkpoint. The REDO list is initially empty. Search forward in the log.
2. If a "start transaction" record is found, put the transaction in the UNDO list.
3. If a "commit transaction" record is detected, move the transaction from the UNDO list to the REDO list.

Note that the first transaction, T_1, would not have been noticed at all in this application. Transactions T_2 and T_3 are the REDO list and Transactions T_4 and T_5 are in the UNDO list.

After the analysis is complete, the restart logic reads the log backwards from the checkpoint, undoing all actions of transactions in the UNDO list. Then the log is read forward to redo all actions of the REDO list. Once this is done, a new checkpoint is taken so that this restart work will not have to be repeated.

VII. MEDIA FAILURE

In the event of a failure which causes a loss of disk storage integrity, it must be possible to continue with a minimum amount of lost work. Forward recovery techniques, discussed earlier, are appropriate to recreate a complete version of the database. The idea is to reload the database from an archive copy and reapply all updates since the copy was made.

Media recovery is simplified if the archive log does not contain records for updates which were undone because the transaction was aborted. A forward-only log can be created containing only relevant log records.

Media recovery for a single page or isolated set of pages is an option, without needlessly recovering the entire database [GALT79]. First the page is locked exclusively to prevent transactions from accessing the page while it is being recovered. Next, the page is restored from the most recent image copy. All the logs for this database are now read and scanned for occurrences of the affected page(s). It is extremely important that the logs are read in chronological order. Whenever a change to one of the pages is found, the page is modified in the database. Logic must exist to ensure that only committed changes are applied. All aborted updates are skipped during media recovery. When all the logs have processed, the damaged page has been brought up to its most recent

transaction consistent state. To complete this recovery, the page is written to non-volatile storage and the lock is removed.

Systems may include recovery utilities which can specify damaged areas other than pages. For example, recovery could be limited to a certain key range or track.

VIII. RECOVERY IN DISTRIBUTED DATABASES

A. DESCRIPTION OF DISTRIBUTED SYSTEMS

A distributed database is stored by several computers in a network, and is scattered on multiple storage devices. A communication facility manages the information flowing between the different systems. An important aspect of a distributed system is that the database should be viewed the same by all users at the various locations. Even though parts of the database may be stored at separate locations, the database should appear logically centralized.

Each site of the distributed database may have its own recovery facility to assist in restoring database integrity in the event of a failure. However, if failures occur at one site, other sites may also be affected. The different locations need to communicate when a failure occurs to ensure that all sites handle the failure consistently.

One important property of many distributed databases is the replication of data. This improves the availability of data and reduces traffic on the communication network, resulting in better service for the customer. In addition, the workload of heavily accessed files can be reduced. Reliability increases, since each site usually stores its own backup copy, resulting in a better chance of recovery from media failures [GALT79]. However, data replication does make update and recovery strategies more

Management of multi-site transactions is as follows. Once a transaction has been initiated, it can access the local database and also migrate to other sites to pursue remote data. The local database manager creates a work request whenever access to a remote database is desired. The remote database system will perform the requested actions of the work request and send the results back to the original site. The work request and resulting messages carry overhead information used to coordinate multi-site transactions and detect site failures.

Certain applications require full-time operation of the computer system. Examples include airline reservations and automatic bank tellers. Communication failures in a distributed database system can cause network partitions. These partitions prevent access to certain databases or other outside information which is needed. Certain systems cannot be held up waiting for the problem to be identified and corrected. In these instances, processing must continue without a major impact on the business. In the case of replicated databases, continued processing may cause the copies of the database to diverge. In all instances, the databases must be brought up to a consistent state once communications resume.

Most distributed database systems restrict the operations which are allowed during a network partition. The restrictions prevent different users from updating different copies of replicated databases at the same time. Some systems just allow read-only access. Updates could be permitted only if the majority of the database copies are available. Various approaches are outlined in a later section on replicated data.

Figure 12 illustrates a distributed system. Communication failures have isolated Site A from the other areas. As a result, many questions and concerns must be addressed. How do Site B and Site C detect that Site A

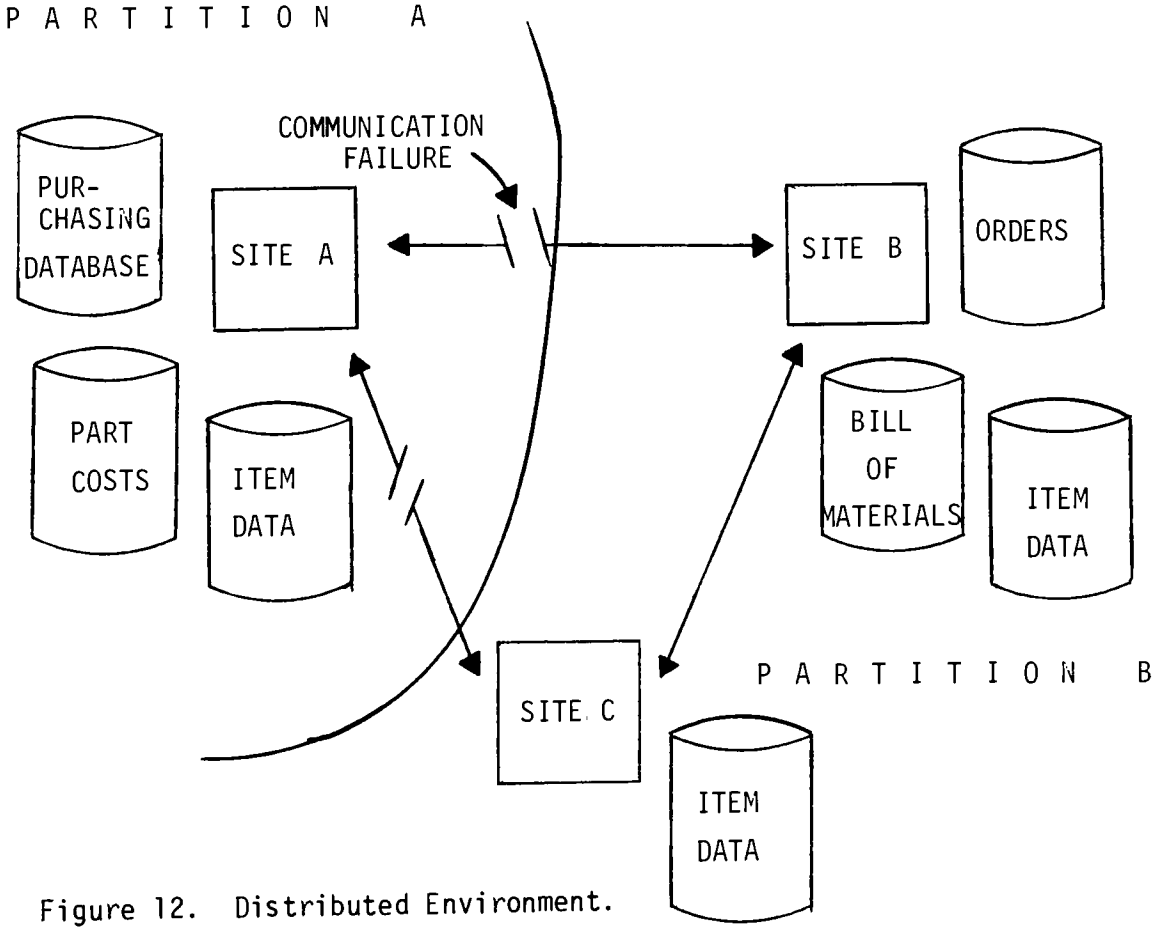


Figure 12. Distributed Environment.

cannot receive their messages? Can the other sites determine that Site A is only isolated, or must they treat that site as if it had crashed? Since the item database is replicated, which sites, if any, should be allowed to update it? If updates are permitted, how will other sites update their copies? How do you treat transactions at other sites that need to access data at Site A? These issues and others will be addressed in the following sections.

B. INTENTION LISTS

An intention list contains all the commands necessary to complete the update actions of a transaction [VERH78]. In comparison with log techniques, intention lists contain operations not yet performed, whereas logs record completed database updates (although, with write-ahead logic, the log may be written to before the update). Intention lists are useful in distributed networks when a coordinator of a transaction must send update activities to other participating sites. A site in the network may receive an intention list, containing all updates for a transaction which correspond to data held at that location. Once the intention list has been processed successfully, it is deleted.

If the system crashes before the intention list has been completely created, the transaction has not finished and no updates have taken place. In this case, the transaction will have to be aborted at recovery time. If the intention lists were received and carried out by all participating sites, the lists should not exist anymore, indicating that the transaction has completed and could be committed. However, if an intention list still exists at a participating site, the transaction has not yet completed and the recovery procedure must carry out the actions of the intention list. Intention lists must be stored on non-volatile storage so that they will not be lost during a site failure. Intention lists are written so that they may be re-executed from the beginning if an

C. STRATEGIES FOR HANDLING BLOCKED TRANSACTIONS

Transactions usually represent a user's task. Subtransactions are atomic actions which are initiated by the original transaction. The subtransaction is processed at just one site, which may be a different site than the one for the transaction. Because of the transaction/subtransaction concept, failures at one site of a distributed system can indirectly affect remote locations. If a crash stops one subtransaction, the entire transaction is affected. Other subtransactions may be able to continue, but eventually the whole transaction will be blocked. There are two strategies, which are discussed in [WALT80], to handle the recovery of blocked transactions.

The first strategy consists of making the transaction wait until the recovery completes. This wait strategy is involved since processing at both the local and remote sites is eligible to stop. In addition, all resources held by the waiting transactions remain locked up during the recovery period. In essence, all transactions requesting the same resources are also adversely affected. The cost of waiting consists of the price for holding certain resources during the recovery. This cost increases as the recovery time increases. Hence, the wait strategy seems to be appropriate for short term failures.

On the other hand, by using the backout strategy, all blocked transactions are rolled back and other transactions are able to use the released resources. The backout

strategy incurs its cost by undoing the effects of non-committed transactions and restarting them. This expense includes the time to back-out, the time to restart and again reach the point where the crash occurred, and the time to send coordinating messages to other sites. The backout strategy is most suited for failures of longer duration, where other transactions could make good use of the released resources during recovery.

The choice of what strategy is best depends on the expected length of the recovery process. Based on the observation that the wait strategy seems to be cheaper for short-time failures while the back-out strategy is cheaper for long-time failures, an algorithm for selecting the least costly strategy for each transaction is proposed in [WALT80].

The algorithm first identifies all affected transactions. The number of remote sites involved is determined and the restructuring overhead for sending messages around the network to coordinate restructuring is computed. If this restructuring time is greater than or equal to the expected recovery time, the wait strategy is selected. If not, the restructuring time is added to the backout time and restart time. If the sum is greater than or equal to the recovery time, the waiting strategy is selected. Otherwise, the backout strategy should be used. Determining the recovery time ahead can make this algorithm difficult to implement.

In the case of a media failure, the selection of the best strategy is done by the affected site. In failures where the site actually crashed, another protocol is needed. First, all other sites must recognize this crash. At this point, transactions exist where the originating site is up and also where the originating site is down. One site must be designated as the coordinator to determine the affected transactions and to proceed with an algorithm.

D. TWO-PHASE COMMIT PROTOCOL

A major concern in distributed database systems is that all sites must commit a transaction or else all sites must abort a transaction. When a transaction has been processed by multiple sites, the networkwide commit must be coordinated.

The two-phase commit protocol, described in [GRAY78, GALT79, KOHL81] requires a commit coordinator which has a communication path to all participants. The commit coordinator notifies the participants that a commit is being attempted. Each participant will either indicate that it must abort or that it can commit. Before the participant sends its answer to the coordinator, it must prepare to either abort or commit by recording recovery information on the log.

If any participant answers that it must abort, the coordinator sends the abort decision to all participants, which causes the transaction to be backed out by everyone. It may be, therefore, that a site which voted to commit would have to abort the transaction. However, if all participants are prepared to commit, the coordinator sends a commit response to all participants, allowing them to commit the transaction.

Two variations of the two-phase commit concept will be described--the linear two-phase protocol and the centralized two-phase protocol.

Linear Two-Phase Commit Protocol

This version of the two-phase commit requires that the sites of a multi-site transction are placed in a linear order. Each participant knows who is the next participant in the sequence, and also the last participant realizes that he is last. While the transaction is being processed, the origin site keeps track of all work requests. Once all work has been completed, the origin site will have a complete list of all sites used by the transaction. At this point, the origin site will initiate the commit process. The commit message will propagate down the sites in the linear order. As each site receives the message, it prepares to commit and then notifies the next site in the list by passing on the message. When the last site is notified to prepare, it can also make the commit decision and pass the decision back to its predecessor. As each site is notified to commit, it does commit and also sends the message along.

The following diagram presents a picture of a successful linear protocol. An unsuccessful commit is basically the same, with abort messages sent in place of commit messages.

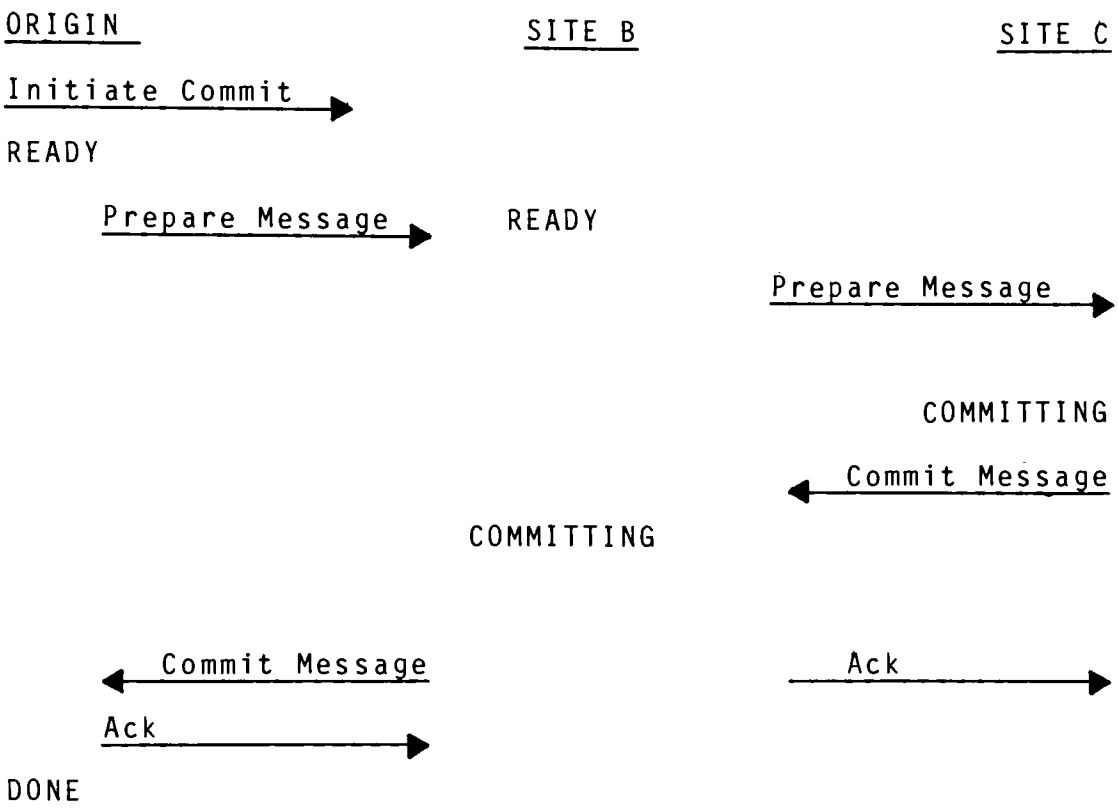


Figure 13. Linear Two-Phase Commit

To begin, the site of origin becomes recoverable so that it is ready to commit. The identifiers of all subtransactions which are part of the transaction being considered are written to non-volatile storage. The origin site sends a "prepare to commit" message to the next site in the list of sites used. The "prepare to commit" message contains the transaction identifier and the sites used list.

Any site which receives a "prepare to commit" message must check whether the transaction is active at the site. If the transaction has terminated or cannot be found, an abort message is returned to the sender. Otherwise, the receiver enters the ready-to-commit state by forcing its recovery data to non-volatile storage and sends the "prepare to commit" message to the next site. The last site in the used sites list is designated as the commit coordinator. This site decides whether to commit or abort based on responses from the other sites. If the transaction is active at the last site and all other sites voted to commit, the last site will bypass the ready-to-commit state and force a commit record to non-volatile storage. Once committed, the coordinator can release the local locks and resources held by the transaction.

After committing, the last site sends a commit message to the predecessor on the used sites list. This causes the predecessor to commit. Once the commit record is written to the log, an acknowledgement is sent to the successor and the commit message continues to propagate backwards through the used sites list. When an acknowledgement is received, the commit process is finished for that site. If it is not received, the commit message is resent. Once the site of origin receives the commit, the transaction has been successfully completed.

Centralized Two-Phase Commit Protocol

In contrast to moving the "prepare to commit" and "commit" messages along the used site list, a single site can send all the messages, as illustrated in the diagram on the following page. The site of origin is usually assigned as the coordinator. The coordinator sends a message to all participating sites asking if they are ready to commit the transaction. It also activates a timeout, which will force the transaction to be aborted if any sites do not respond. A site will answer as ready to commit only after the subtransaction's log records are written to non-volatile storage. If the transaction cannot commit, an abort message is returned to the coordinator.

If all sites answer that they are ready to commit, the coordinator will decide to commit the transaction. However, if one or more sites answer with an abort message, or if the timeout expires, the coordinator will decide to abort the transaction. At this point, phase one of the commit process is finished, resulting in a common decision.

The second phase involves the carrying out of the decision. The coordinator writes the decision in the log and then sends the decision to all participating sites. The sites write the message in their logs, execute it, and send an acknowledgement back to the origin site. When the coordinator receives all the acknowledgements, the transaction is marked in the log as completed.

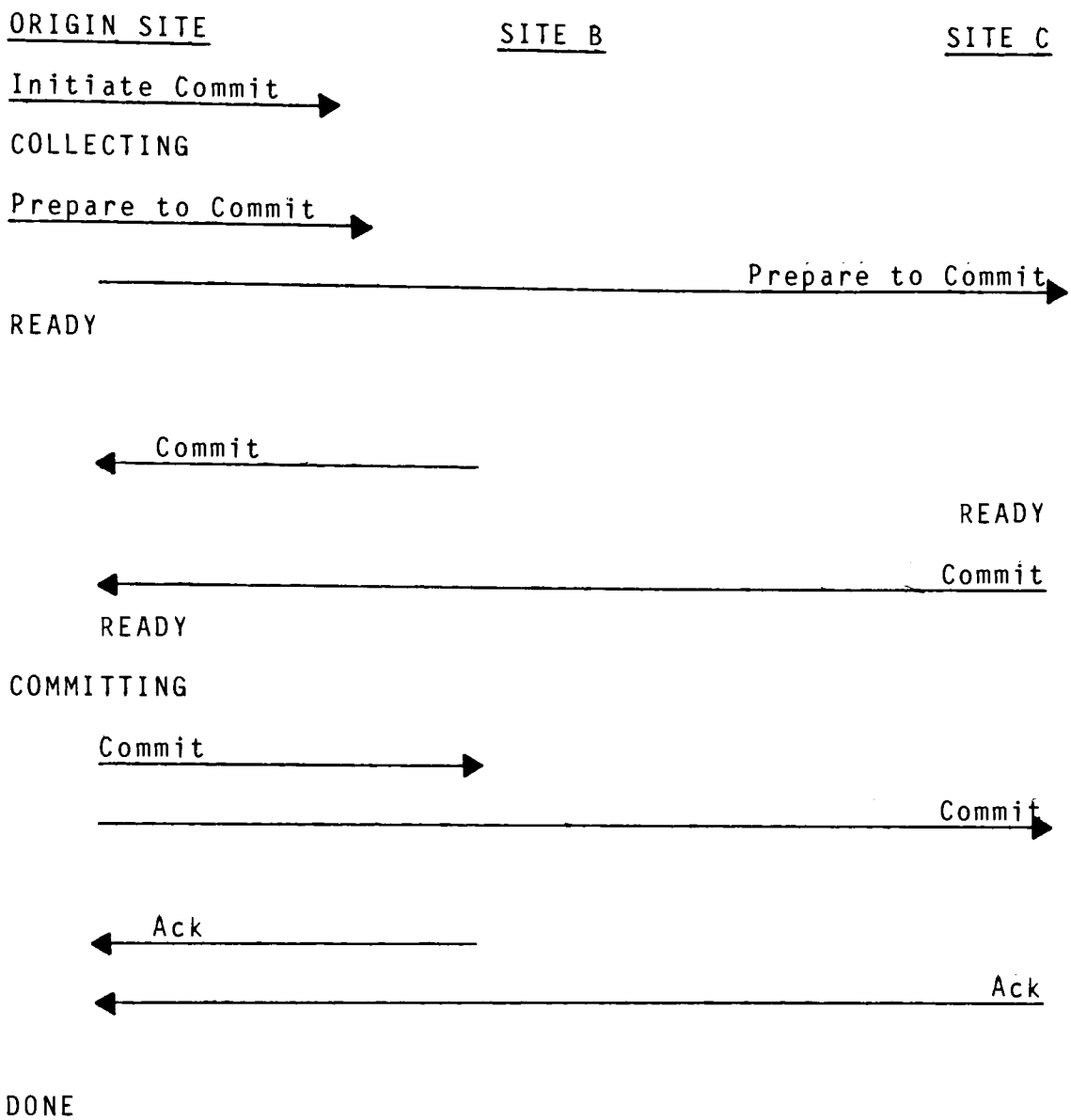


Figure 14. Centralized Two-Phase Commit

Comparison of Linear And Centralized Protocols

The linear commit protocol always requires fewer messages. This implies that a system which has a high message send-receive cost may benefit from this technique. There is no need for a broadcast mechanism, but the need for concurrency during protocol execution must be low.

The centralized commit protocol requires fewer message delays when three or more sites are involved. Broadcast should be the normal mode of interprocess communication. Also, the parallelism is a desirable feature in comparison to the linear commit protocol, which only has one active process at a time during commit processing. The system can choose which protocol to use for commit processing based on the number of sites involved. The centralized approach would be used for a large number of sites.

Failures During the Two-Phase Protocol

A failure can occur at any time during the two-phase commit procedure. The two-phase commit process can recover from all failures as long as the log information is available. If the coordinator site fails, the log records will indicate what phase the commit process was in at the time of the failure. If the coordinator crashed before the commit/abort record was recorded in the log, the coordinator was at phase one. After recovery, the coordinator must again ask the sites if they are ready to commit. If the log holds a commit or abort record, but no transaction complete message, the coordinator must resend its decision to all participants.

Another possibility is for a participant other than the site of origin to fail. If the participant crashes in phase one, the entire transaction will be aborted, since the coordinator will not receive a reply during the time-out interval. If a participant crashes in phase two, the coordinator must be prompted to retransmit its decision once the failed site recovers. If the coordinator also fails, another site could send the coordinator's decision. As long as one site in the group of used sites receives the decision, the transaction can complete.

When a message from the coordinator is lost, it will not receive an answer from a participant. Once the time-out expires, the transaction will be aborted. If the participating sites are expecting the decision, they should have a shorter timeout to cause the repetition of a message to be requested. This prevents a transaction from being aborted, after the coordinator has decided to commit, due to the loss of a message.

E. THREE-PHASE COMMIT PROTOCOL

The two-phase commit algorithm can cause a serious problem. Suppose the coordinator notifies the participants of the commit attempt, the answers are sent back to the coordinator, and then the coordinator fails. Since the participants have not been told whether the transaction should be committed or aborted, they must wait for the coordinator to recover. This implies that all the locks held by the transaction cannot be released until the coordinator recovers. If the coordinator is supervising a large number of transactions, major portions of the database may be locked and unavailable until the coordinator recovers.

This problem can be avoided if the coordinator sends backup information to other nodes. If the coordinator fails, one of its backups can take over its function. Each backup node maintains a commit list of all transactions attempting a commit.

The three-phase commit [BERN83] begins by sending the commit message to the participants and waiting for their responses.

Next, a precommit message is sent to each backup node. The backup node adds the transaction to its copy of the commit list and then sends an acknowledgement back to the coordinator. Once all the acknowledgements are received, the coordinator's commit decision is sent to all the participants. Essentially, the three-phase protocol is the same as the two-phase protocol with the addition of

Under this protocol, failures of a backup node may be ignored if the number of backups is still acceptable. Otherwise, a new backup may be assigned to replace the old one.

If the coordinator fails, one of the backups will be appointed as a replacement. All the backups will send their copy of the commit list to the new coordinator. The new coordinator takes the union of those copies and distributes the result to the other backups. This becomes everyone's copy of the commit list, which now ensures that if at least one of the precommit messages reached a backup node, the transaction can be committed. The new coordinator will notify the other nodes of its new role. If a participant wants to know what happened to a particular transaction, it will ask the new coordinator. If the transaction is in the commit list, the transaction should be committed; otherwise, the participant is told to abort the transaction.

F. CHECKPOINTS IN DISTRIBUTED SYSTEMS

The goal of checkpointing in database systems is to obtain a consistent state of the database for use in recovery. Since checkpointing is performed while the system is functioning normally, the overhead and interference caused must be kept to a minimum.

Distributed checkpointing can be classified into categories based on the amount of coordination needed among the sites. Fully synchronized checkpoints are transaction-consistent and require that no update transactions are processed at any site while a checkpoint is taken. Loosely synchronized checkpoints require all sites to take a local checkpoint within the same specified interval of time. Nonsynchronized checkpoints [DADA80] incur no costs for the synchronization of global checkpoints. However, recovery gets more complicated, since the consistent global state must be created based on local logfile information. This consistent database state does not exist until the database reconstruction process builds it.

A checkpointing scheme presented in [AGRA85a] creates a global consistent state while transactions continue to be processed. The approach consists of two types of processes: a checkpoint coordinator and checkpoint subordinate.

First, the coordinator broadcasts a checkpoint request message, which is accompanied by the current clock time. Each local site receives the message and calculates its own local checkpoint number. The local site increments its internal clock by one and then compares the resulting time with the coordinator's clocktime. The latest time becomes the local site's new clocktime, and also the local checkpoint number. The coordinator is sent all the local checkpoint numbers, and chooses the largest number as the global checkpoint number. The above steps complete the first phase of this checkpointing scheme.

Once the local sites return their local checkpoint number, all new transactions are marked as temporary, after-checkpoint transactions. If those transactions desire update capabilities, a new version of the data is created, avoiding updates of the actual data. When these temporary transactions commit, updated data objects are not stored in the database as usual, but are maintained as "committed temporary versions" of data objects. The data manager of each site maintains the permanent and temporary versions of the data. When a read request is made for data which has committed temporary versions, the value of the latest committed temporary version is returned. When a write request is made for a data item with committed temporary versions, the old version is not overwritten. Instead, a new committed temporary version is created.

Once a global checkpoint number is known by a local site, all temporary transactions which were initiated before the global checkpoint time are written to the actual database. This step ensures that the checkpoint is transaction-consistent for all sites based on the same time frame. Transactions started after the global checkpoint number are not included in the current checkpoint. After all the transactions which started before the global checkpoint time actually complete and are updated in the database, the checkpoint is taken by dumping data. Transactions may still execute, but they only affect the temporary database, not the permanent database. After the dumping operation is finished, all the committed temporary versions are stored on the actual database.

This checkpoint scheme does generate globally consistent checkpoints. It does not affect the processing of transactions, since none are delayed or aborted due to the checkpoint. This approach does require more storage for the transactions which arrive after the local and global checkpoint times are established. The resulting database copy will be easy to use in reconstructing a consistent copy.

G. DETECTING NODE AND COMMUNICATION LINK FAILURES

The function of detecting failures among network nodes is usually handled at a lower level in the recovery protocol than the function of transaction recovery. The higher level modules in the recovery mechanism are not involved with the delivery of individual messages or time-outs. An important part of a node failure protocol is that the operating nodes must agree on the status of down nodes. The status information must be accurate and consistent even during multiple failures. If not, recovery will only be initiated at some nodes.

The node failure protocol presented in [BRUS85] can be separated into two protocols: a Node-Down Protocol executed when a node crashes, and a Node-Up Protocol (NUP), executed when a node rejoins the network. The Node-Down Protocol must function despite multiple failures of network nodes or communication links. This is accomplished through redundancy to overcome lost messages. The protocol may be either centralized or decentralized. A centralized protocol appoints one specific node as the leader to coordinate messages among the nodes. The decentralized approach uses successive message rounds to achieve coordination between nodes. In comparison, a centralized protocol uses fewer messages, but complicates recovery if the leader node fails. Since the Node-Down protocol is the basic failure detection and notification mechanism, decentralization appears best.

Failures of nodes and links are detected by an acknowledgment/retry scheme. If a message has been sent to another node, the receiving node must reply within a certain time limit. If no response is received, the message is resent. After repeating the message a predetermined number of times, the node is assumed to be down or isolated from the sending node, and appropriate node-down steps are taken.

When a node is flagged as down, a node-down message is sent to all other network nodes. Each node receiving this message also begins the node-down protocol. The immediate result is that the network is flooded with messages as each node notifies all other nodes. This redundancy ensures that all nodes are notified of the failure. The node-down messages consist of a notify and acknowledgement. All information about the bad node is sent in the notify message. The reply is used only to ensure that the other node is up and has received the message.

The concurrent execution of the node-down protocol by each node ensures that every node in the network will be notified about the failure of a node, even if another node fails. If a node determines that it is unable to communicate with another node during execution of the node-down protocol, that information is stored. After the protocol for the first failed node is completed, the node-down protocol will be initiated for the second isolated node. The

node-down protocol is started only if a node's status changes from up to down. Once a node is flagged as down, only subsequent node-down notifications are replied to. No other action is taken, since the message is redundant. When a node goes down, the higher level modules which perform transaction recovery are also notified, so that the data integrity will be preserved.

The node-up protocol is usually centralized. When a node changes status from down to up, a message is sent to all other network nodes. The node-up protocol is performed one at a time by each node to avoid wasting work if the node crashes again. This is a different approach than the node-down protocol, where all nodes execute that protocol simultaneously. Each node maintains a table which indicates the status of the other nodes. Upon receipt of a node-up message, the entry is updated in the table to indicate the availability of the node.

Realistically, the node-down and node-up protocols are more complex, since both protocols may be executing at the same time for the same node. Suppose the following situation occurs. Node A is trying to send a message to Node B. However, Node B does not respond after successive tries, causing Node A to time out. Therefore, Node A recognizes Node B as a failed node and begins the node-down protocol. In the midst of accomplishing the protocol, Node B is revived, and sends a message to Node C. Node C may receive both node-down and node-up messages for

Node B, and Node B's status is not clear. The messages sent must indicate which information is more accurate.

Version numbers can be carried with the messages. Each node will have its own version number associated with it. All nodes start out at Version 1. Every time a node recovers from a failure, its version number is incremented. For example, if a node is at Version 3, it has crashed two times since the version number was set. Every node stores the current version number of all nodes in its status table. The status tables must be stored where they will survive in a system crash.

For the node-down protocol, the messages sent will contain the version number of the failed node. A node receiving the message will compare the version number sent to the version number stored in its status table. If the incoming version is less than the stored version, the message is ignored due to obsolescence. Otherwise, the node-down protocol is executed.

The node-up protocol also sends messages which contain the version number for the node that has just come up. If Node A receives a node-up message for Version 3 of a node, but has not received a node-down message for Version 2, Node A concludes that the node crashed and came up without Node A being informed. In this instance, Node A will initiate the node-down protocol for Version 2 and a node-up protocol for Version 3.

H. ENSURING CONSISTENCY AMONG REPLICATED DATA

A file may be made inaccessible to portions of a distributed system due to site failures or network partitions. These cases have different implications. During site failures, the failed site is inoperative, which implies that no updates are done to data stored at that site. Therefore, site failures are easier to handle regarding replicated data consistency than partitions. When the failed site recovers, it must obtain all updates done by other sites during the failure, and apply them to its own copies of the data. The integration needed only requires input in one direction--from the operating sites to the recovering sites. In comparison, during a network partition, updates may be performed to replicated data existing in the separate partitions, without any way to communicate those updates to the copies at other sites. In this case, when the network finally becomes connected, the updates processed at both sites must be combined in some manner so that the updates done in all partitions are either reflected or rejected in all copies of the data, to preserve data integrity and consistency.

The voting proposal [CHOW83] for handling partitions requires a majority agreement between sites to allow a file to be locked for updating. Obviously, at most one partition can contain the majority of sites, so a file can be available in no more than one partition. In addition,

it is possible that no partition will contain a majority of sites, which would not allow any updates to occur to that data. This scheme insures mutual consistency, but there is a risk of poor availability if partitions are frequent and compounding.

Another technique for dealing with partitions is the token concept [AGRA86]. A specific scheme is described later, but a brief outline is given here. Each file may have a token assigned to it which allows the bearer to update the file. If no token copies exist in a particular partition, that file cannot be updated. Implementations may allow every file to contain a token, or a limited number of tokens to be available.

The primary site approach relies on one specific site to coordinate a file's activities. If a partition occurs in which the primary site becomes inaccessible, the file may become unavailable in all partitions but the one containing the primary site. A preferred but more complicated approach involves appointing a backup site as the new primary site. This introduces potential consistency problems which would need to be resolved when the partitions rejoin. Distributed INGRES [STON79] is based on the primary site model. In its implementation, a primary copy can exist only if a majority of all copies are at sites which are currently up.

Optimistic protocols allow transactions to process without any restrictions during partition failures. Most

other protocols avoid conflicting transactions which endanger consistency by limiting availability of replicated data. Those limitations can severely degrade performance to an unacceptable degree. On the other hand, one optimistic approach [DAVI84] attempts to process all transactions. However, commitment is postponed until recovery is complete. Conflicting transactions will be backed out at that time to regain consistency. SDD-1 [HAMM78, HAMM80] is optimistic by collecting all messages (including update actions) meant for a site in a spooler, which is read and processed as part of the recovery process. All messages are guaranteed to reach their destination. [STRO85] uses a dependency tracking scheme to aid in reconstructing a consistent database after a period of failure where processing and commitments have occurred.

Two approaches, presented in [RIES82], allow updates to continue during a network partition and restore the consistency of the database after communications have been repaired.

Both methods require extra information to be recorded during the lifetime of a partition. When communications are re-established, the partitions exchange their respective extra pieces of information. This data, along with predefined and application specific rules, are used to restore database consistency. The approaches differ as to what extra information is recorded and how the consistency of the database is to be restored.

The first approach, called log transformations, records a history of the transactions that have executed during communication failures. After communications have been restored, the histories in the different partitions are merged. Special predefined rules are used to determine which transactions may have to be rerun or run differently. These rules specify which transactions overwrite other transactions, and special transactions that need to be run. For example, these rules would specify the corrective actions for overdrawn accounts or overbooked flights.

The second approach, called data patch, records initial database values when a network partition occurs. It also records which values have changed during the lifetime of a partition. When communications are reestablished, sites in the different partitions exchange their current values of the data items that have been updated. These values and the prepartition data values are then used to update the database according to the prespecified rules. In this approach, there are rules for each type of data item that can be updated. The rules specify whether to use the latest data value, apply an arithmetic function to the data values from the different partitions, take the intersection of the data values, or run a specific application program.

Semantic knowledge can be used to minimize or resolve conflicts. One example is to split the files into new

files unique to each partition. An example involving a reservation system would analyze the number of available seats when a partition occurred. A partition containing 40 percent of the sites would modify its available number of seats to be 40 percent of the previous total. The other partition would likewise handle 60 percent of the reservations. This scheme would prevent either partition from overscheduling the number of seat reservations.

[BHAR82] suggests that the system may allow transactions to commit in different partitions if an UNDO log is kept. When the partitions reconnect, graphs may be constructed to analyze the update processing performed in the partition. If a cycle occurs in the graph, the transactions must be analyzed to remove the conflict. Any semi-committed transactions would be the first candidates to abort and restart. If only committed transactions exist, and if the effects of the transactions have not yet been seen by another transaction or user, the committed transaction could be rolled back. Note that this approach contradicts the idea that committed transactions are permanent. Another method would be to issue a compensating transaction if one can be constructed. If no transaction can be rolled back or compensated, the system should avoid committing such transactions during a failure, or pay the consequences of potentially affecting data integrity.

I. TOKEN CONTROL SCHEME FOR REPLICATED DATABASES

Tokens are used in distributed database systems to provide consistency among replicated data [AGRA86]. Tokens may be assigned at any data level--files, pages, records, etc. However, in this discussion, the tokens are associated with specified files. A file which contains a token is designated as an update copy. There is a predetermined number of tokens per file; allowing more than one update copy when replicated files exist. Replicated files without tokens are read-only files.

When discussing replicated data, a differentiation is needed between a logical data object and physical data objects. Logical data is viewed by the users of a distributed database system. However, this data can have more than one physical copy in the replicated data scheme.

Each file has a directory which holds a list of all available token copies and available read-only copies. These directories are duplicated at each site containing a replicated copy. The directories are updated when failures and recoveries occur. The coordinator of a transaction must either have or request a copy of directories for all files which must be updated by a transaction. The data required by a transaction is divided into two sets--read-only data and data to be updated. In order to update a logical data object in a file, the tokens from all update copies of that file must be locked

successfully. In order to read a data object, a read-lock must be obtained from one of the read-only files containing that object. If no read-only copies are available, the coordinator can try to read from a token copy.

When a read-lock is obtained on a read-only replicated file, the copy of the data objects may not reflect the most up-to-date value. This inconsistency may occur since the read-only files are not written to when the data is updated in corresponding replicated files. Therefore, before a transaction can access data in a read-only file, the copy must be made mutually consistent. Actualization requests are sent out from a read-only site to a token copy when a read is requested. This message is treated the same as a read-lock at the token copy. The token site responds with the current value of the file requested. By using this procedure, inconsistency of read-only copies is avoided.

Remote copy actualization is also the technique used to restore a token copy to the most current values after a failure. During site recovery, the recovery mechanism makes actualization requests for its token copies to other available token files. Once the remote copy actualization is complete, the recovered token-copy will be included in the list of available update files.

In order for transactions to commit when tokens are used, all the available token sites of each file written to must have sent precommit messages to the transaction

coordinator. In addition, the files used in the transaction read list must have also sent precommit messages.

When a failure occurs which disables a copy from use, the network protocol which discovered the failure will send messages out to all other sites to update their directories by removing the file from the available list. If there are no other copies of that file available for use, any transactions needing to access that file must be aborted, and a total failure occurs.

To recover from a failure, read-only copies need to be included in the available read-only list at other sites. However, token copies must be updated to the current value. If a partial failure occurred, the copy actualization procedure must be followed for the token copy, and an update directory message is then sent to each replicated site. On the other hand, if all sites containing token copies failed, the first step is to determine the site which failed last. The token copy at that site is then included in the available list in all directories. All other token copies must follow the copy actualization procedure, as if a partial failure occurred, in order to get consistent with the most current copy at the last available site. The above procedures must be carried out by a recovering site for each file stored at that site.

The token scheme has flexibility which allows an administrator to alter the reliability capabilities

easily, by changing the number of tokens for a file. Two extremes may exist. First, all copies may have tokens which designate all files as read-write copies and actually eliminates the need for tokens. Second, only one copy may have a token, which reduces system reliability since the loss of one site may cripple the system. The concurrency of the system improves as the tokens become attached to smaller objects. However, the directory size will also increase.

IX. RECORD SEGMENTATION

A physical database design technique, called record segmentation, seeks to improve system performance by grouping data items that are usually required simultaneously by the users of the database into physical subfiles. Conversely, data items that are not retrieved together are stored separately. Thus, user requirements can be met with a minimal number of accesses to secondary memory and a minimal amount of data transfer. [MARC84]

In a database system supporting multiple users, determining the most efficient record segmentation scheme becomes more difficult. Two users may require the same key data, but different detailed information. A segmentation which is efficient for one user may be quite inefficient for another, creating a need to analyze trade-offs among users.

The motivation behind record segmentation is the concept that a large proportion of all activities to a database is directed at only a small portion of the stored data items.

Record segmentation is usually discussed in terms of increasing retrieval times. However, segmentation can also have a big effect on recovery. Modern large databases with a high degree of use and a large amount of changes face serious problems concerning recovery issues. Because of their size, the archival process tends to be

lengthy. If the system needs to be available continuously, then it will not be acceptable to reserve large blocks of time for backup procedures. In this type of environment, record segmentation can be very helpful.

Once the data is split into various subfiles, each portion can be archived independently with no detrimental effect on the subfile's performance. If properly applied, record segmentation would lead to the following advantages [KUST84]:

1. recovery and archiving can be handled easily for each reduced subfile;
2. operational reliability of the entire database increases because each subfile is independently supported by a hardware/software unit;
3. better overall performance, since a subfile's backup process will not affect another subfile's update and retrieval process.

The main disadvantage of record segmentation is the increased complexity of the database organization. Each subfile has a separate location which must be noted by the database management system.

In principle, there is no limit on the database size when the record segmentation technique is used. The decomposition process can be applied as often as necessary to achieve sufficiently small archivable subfiles.

Basically, the retrieval costs of a database are reduced by physically grouping items that are frequently

retrieved together. Backup and recovery costs are reduced when volatile data items are isolated. Thus, records must be segmented so that total system operating costs are minimized by balancing the retrieval and backup considerations. Equations to help evaluate the possible groupings are discussed in [MARC84].

One form of segmentation is the isolation of all data items which can be updated in a single segment, with the remaining data items stored in a second segment. The backup and recovery considerations for the volatile subfile are very different than those for the stable subfile. Substantial cost savings for backup and recovery activities may result by such an arrangement; however, the effects of this organization on retrieval and update performance must be weighed.

Another method of segmentation is frequency decomposition. The decomposition pattern is based on separating parts of the original database according to their inherent access frequencies [KOST84]. The archiving frequency is directly proportional to the update frequency.

Thirdly, when indexes are not available, data items which are used for selection tend to be isolated. The segment containing the selection data items will end up acting like an index [MARC84].

X. DESCRIPTION OF RECOVERY IN EXISTING SYSTEMS

A. IMS/VS

The basic approach to database recovery in IMS/VS is to make periodic copies of the datasets that contain the database and record database changes on the system log. In the event of failure in a dataset, the latest copy can be updated with changes logged since the copy was made, thus restoring the dataset to its condition at the point of failure. A database change is recorded in the system log by storing the before and after images of the segment.

Copying databases is done with a database image copy utility program. That program copies one dataset at a time, thus creating an image copy of the dataset on disk or tape. Databases are normally copied just after the database has been initially loaded and immediately after reorganization. If a database has been reorganized, any copies made before the reorganization cannot be used in recovery [IMS86a]. Copies may also be made at intermediate points, as determined by the update activity against the database.

When database damage is discovered, the affected datasets may be recovered by running a database recovery utility program. For each dataset to be recovered, the utility performs the following actions [MCGE77]:

1. It allocates space for a new version of the dataset and loads the latest image copy into this space.

2. The system log is read forward, looking for changes made to the dataset after the image copy. For each such entry log, the "after" image is used to replace the corresponding data in the database.

To protect databases against the effects of incorrect application programs, IMS/VS provides a Data Base Backout utility program. The backout program searches the system log in the backward direction for database change entries that were made by the aborted program. The before image from each such entry replaces the corresponding data in the databases. The utility ends when it encounters the scheduling entry for the program.

To prevent the rerunning of an entire job in the event of system failure, IMS/VS provides a program checkpoint/restart facility that permits programs to make periodic copies of selected program and system variables (checkpoints), and to be restarted from such checkpoints in the event of system failure [MCGE77]. Program checkpoints are taken by a Checkpoint (CHKP) call that specifies the program variables to be saved, and a checkpoint identification for subsequent reference to the checkpoint. The system responds to the call by flushing the database buffers to direct access storage and by creating a checkpoint entry in the system log that contains the user-specified checkpoint identification, the keys of the last database records to be processed, and the user-specified variables. The system also writes a checkpoint message to the operators, giving the checkpoint identification.

Before restarting a program, following a system failure, the Data Base Backout utility must be run to backout the changes made by the program since a specified checkpoint. The utility ends when it finds the specified checkpoint (or program scheduling entry, if that occurs first) in the system log. Data Base Backout creates a log of change activity for updates which were backed out in the run. When restarted, the system restores the specified program variables and position in the databases, using the log data at the last checkpoint.

The system logging technique was enhanced in IMS/VS 1.3. The full logging function, as provided by IMS/VS 1.2 and prior releases, uses magnetic tape as the primary logging media. The log write ahead approach is now mandatory. With the increasing transaction rates processed by IMS/VS systems, the present tape logging is proving inadequate due to [GEND82]:

1. concerns related to the volume of tape handling;
2. inability to share the tape log for multiple purposes in one pass;
3. long restart times if forced to use the tape log for restart;
4. reliability of tape media and tape drives when compared to direct access devices.

The logging enhancements of IMS/VS 1.3 [STRI82, GEND82] require users to allocate the IMS/VS log to multiple datasets on direct access devices. These datasets are referred to as On-line Log Datasets (OLDS). Log

records are initially written to the OLDS in a wraparound fashion and subsequently copied to another dataset known as the system log dataset (SLDS). The SLDS is the input to database recovery. The third required dataset in IMS logging is the Write Ahead Dataset (WADS), which contains a copy of committed log records which are in OLDS buffers, but have not yet been written to the OLDS. The WADS space is continually reused after the appropriate log data has been written to the OLDS. When Log Write Ahead requests are made to the logging manager, a partially filled OLDS buffer is written to the WADS, which ensures system recovery in the event of failure.

At execution of the Log Archive Utility, which produces the SLDS, a dataset is created that contains all of the log records needed for database recovery. This output dataset is referred to as a Recovery Log Dataset (RLDS).

IBM's on-line system, Customer Information Controls System (CICS) supports the use of IMS databases. CICS performs dynamic backouts which will use a dynamic log to automatically backout any changes made by incomplete transactions [YELA82]. The information which is recorded in the task's dynamic log is a subset of what is recorded in the system log. For instance, after images are not placed in the dynamic log, since the purpose of the dynamic log is backout and not forward recovery.

Overview of DBRC Recovery Features

Data Base Recovery Control (DBRC) was introduced in IMS/VS Release 1.2. It provides a high degree of control over the recovery of IMS/VS databases. This control consists of aid in the actual recovery action and assistance in many IMS/VS utility jobs necessary to enable recovery of databases.

Without DBRC, the manual effort needed to recover databases would be enormous, with an open invitation to error. If there were a write error on the database, for example, forward recovery would be needed. The data manager must locate all the jobs which updated the broken database since the last image copy. The log tapes from these jobs would need to be recorded and the JCL needed for the recovery related actions would have to be created manually. If any information were incorrect, missing, or in the wrong order, bad input could easily be supplied to a recovery operation. An invalid recovery would then result, causing loss of database integrity.

DBRC provides a number of recovery-related functions. First of all, it automatically records recovery-related information in DBRC datasets. In addition, it can generate JCL for many IMS/VS database dataset utilities. DBRC will also verify that the correct input and output datasets are specified when many of the IMS/VS database dataset and log tape recovery related utilities are run. Finally, a DBRC log tape processing utility, called the DSLOG utility, is provided by the product.

Recording of Recovery Information

RECON (REcovery CONtrol) datasets are used by DBRC to record recovery and data sharing information, and are the heart of DBRC operation. The RECON dataset consists of up to three physical datasets: the RECON dataset itself, a copy of the original, and a spare. The two recons comprising the pair are both written to, always in the same order. The spare is never written to unless there is a problem with one of the RECONs in the pair.

The RECON datasets should be spread out about your system. They should have different space allocations, be on different channels, controllers, devices, etc. One system problem should not cause all RECONs to be lost.

The information contained in the RECON is maintained through two sources. The first method is provided by a set of internal DBRC exits that are called by various IMS/VS modules during execution of a subsystem. The exits are used to automatically record the following information in the RECON dataset: [DBRC84]

1. the subsystems which access data bases and the logs which these subsystems create.
2. which database datasets have change records contained on each of the various logs.
3. what image copies of database datasets have been made.
4. what change accumulations of logs have been made.
5. what database dataset reorganizations have been made.
6. what database dataset recoveries have been made.

The second of the two sources of information for the RECON dataset is provided by the Recovery Control utility, which is supplied as part of DBRC. This utility can add, delete and change most of the information contained in the RECON dataset. One parameter in this utility will cause "old" data to be removed. What information should be considered old depends on the length of time you want to recover. The utility is also used to list the contents of the RECON, invoke the JCL generating capability of DBRC, and backup the RECON datasets.

JCL Generation

DBRC provides JCL generation support fpr the IMS/VS image copy, change accumulation, database recovery (forward recovery), and DBRC DSLOG utilities. For example, if JCL for a database dataset recovery is desired, DBRC will generate the JCL and insure that the correct image copy, change accumulation, and log tape volumes are used, and that the log tape volumes are specified in the right order. [KEEN82]

The use of the GENJCL process can significantly reduce the amount of time required to recover a database. In addition, it can eliminate many of the causes of invalid recoveries. DBRC knows what log tape volumes are required for a change accumulation or recovery, and will ensure that all required volumes are processed in the correct order. A significant amount of time is spent

during database recoveries to determine what inputs, in what order, should be provided to the Data Base Recovery utility. DBRC eliminates this manual, error-prone effort. Thus, the benefits provided by the GENJCL process are:

1. faster preparation of the utility JCL,
2. elimination of human indecision and intervention, and
3. elimination of human errors.

Image Copy

DBRC provides full support for standard image copies. DBRC will capture the dataset name, volume serial number, and file sequence number for the image copy output dataset and will record this information in the RECON dataset. If dual output image copy datasets are requested from the image copy utility, DBRC will record the same information for both duplicate datasets, so that either can be used in a recovery.

When a database is registered to DBRC, the maximum number of image copy dataset generations to be maintained by DBRC must be specified. When this number is exceeded, DBRC will discard the information associated with the oldest image copy and perform appropriate RECON maintenance.

Change Accumulation

DBRC introduces a concept of "grouping" database datasets for change accumulation processing. These groups of database datasets are called change accumulation groups. Change Accumulation is used to combine change records from the logs of many update jobs in order to have one collection of changes for the database datasets. The Change Accumulation utility is also used by DBRC to merge change records from logs created by multiple, concurrently executing subsystems in a data sharing environment. If DBRC knows that a log dataset does not contain any changes or I/O-error records for members of the change accumulation group, that log dataset will not be included as input, thus reducing unnecessary volume handling and processing.

Database Recovery

Three types of recovery supported for database datasets are controlled by DBRC. These are full recovery, time-stamp recovery, and track recovery. The GENJCL and utility JCL verification support is provided for all three recovery types. The inputs to recovery are the latest appropriate image copy data set, the latest appropriate change accumulation dataset, and all required logs concatenated in the correct sequence.

Full Recovery

A full recovery of a database set restores the database dataset to its current state, as indicated by the RECON dataset.

Time Stamp Recovery

A time stamp recovery of a database dataset allows the recovery of a database dataset to some point other than the current state of the database dataset. Time stamp recovery is typically used in a batch environment to restore the database(s) to the beginning of a batch run. For example, if the Nth step abended or had incorrect input, the entire batch network could be re-run without running batch backout for all N steps.

The time stamps to which you can perform recovery with this technique are the stop time of a log tape volume, the stop time of a log tape dataset, and the time at which an image copy dataset is created.

An image copy for the database dataset should be created after a time stamp recovery has been performed. Although not required, this image copy will simplify and further reduce the duration of future recoveries.

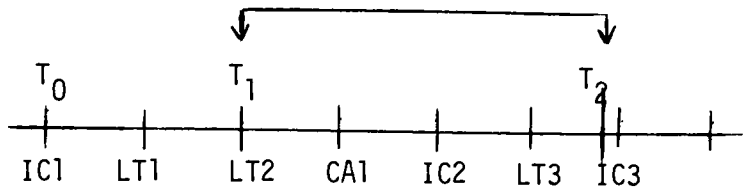


Figure 15. Time Stamp Recovery

Figure 15 shows a time line along which are shown various recovery related events for a database dataset. At time T_2, the user performs a time stamp recovery requesting that the database dataset be recovered to its time T_1 state. The GENJCL feature will produce a database recovery utility job step that would include image copy 1 (IC1) and log tape volumes 1 and 2 (LT1 and LT2). Following the successful execution of the Data Base Recovery utility, the Image Copy utility will be run, producing IC3. Following time T_2, the database dataset appears as it did at time T_1, with an image copy dataset (IC3) as backup.

Time stamp recovery must be used with caution. When performing a time stamp recovery for a database dataset, the same must be done for all related database datasets, thereby recovering them to the same "logical" time stamp. Examples of related database datasets would include not only those connected through logical relationships, but also those where multiple datasets form a single database, as is the case with databases containing indexes or multiple dataset groups. Since DBRC does not know how database datasets are related, it is the user's responsibility to ensure that all of these related time stamp recoveries are performed.

Track Recovery

DBRC also provides support for the track recovery option of the IMS/VS database recovery utility. Track recovery is used to recover one or more "bad" tracks in a VSAM database dataset to its most current state. When track damage is detected, the system creates search criteria for finding all records pertaining to the track in error. Since the entire database dataset is not being recovered, a time stamp track recovery cannot be performed.

DSLOG Processing

DBRC provides an optional facility to further reduce the tape handling associated with change accumulation and database recovery processing. This facility is called "DSLOG processing." This process extracts the database dataset change and I/O error records from the log tape volumes and stores them in database dataset related datasets called DSLOG datasets.

Conceptually, the DSLOG datasets appear to be log tapes that contain records for only one database dataset. DSLOG datasets are used as input to the recovery and change accumulation utilities in place of the original logs, thereby reducing tape handling and recovery times.

DBRC Operation

The communication between IMS, DBRC, and the various utilities is illustrated in Figure 16. Normal processing will be discussed first, and relates to the upper left-hand side of the diagram.

When a job which uses IMS/VS databases starts, a sign-on exit to DBRC is performed. This exit identifies the job to DBRC. DBRC checks to see if a SUBSYS record with the same job name exists in the RECON. If it does, the job will immediately fail. This condition indicates that the job had aborted previously after a database data-set had been updated, and no recovery has been done. If no SUBSYS record exists, one will be created and the job will proceed.

The next step is to request authority to access the databases. If authorization cannot be granted for all of the requested databases, the job will fail. The log tapes are now opened and database information is updated in the RECON when change records are written to the logs, so that the RECON knows which databases have been updated. PRILOG (PRImary LOG) records store log information in the RECON. If dual logging is used, a SECLOG (SECondary LOG) record will also be created.

The application program is now running. As each registered database is opened with an access attempt of update or exclusive use, a DBRC exit is invoked to ensure that recovery information is consistent. DBRC is also

invoked each time the first update to a database dataset occurs. Time stamps are kept for the time of each update. It is also noted in the RECON that the database has been changed.

Whenever a log tape volume becomes full and a new volume is mounted, the DBRC log exit is driven. DBRC will update the PRILOG or SECLOG record with the stoptime of the prior volume and add the new volume serial number to the record. When a log is closed (due either to normal or abnormal termination), the stoptime field for the last volume and the log dataset are set in the PRILOG or SECLOG record.

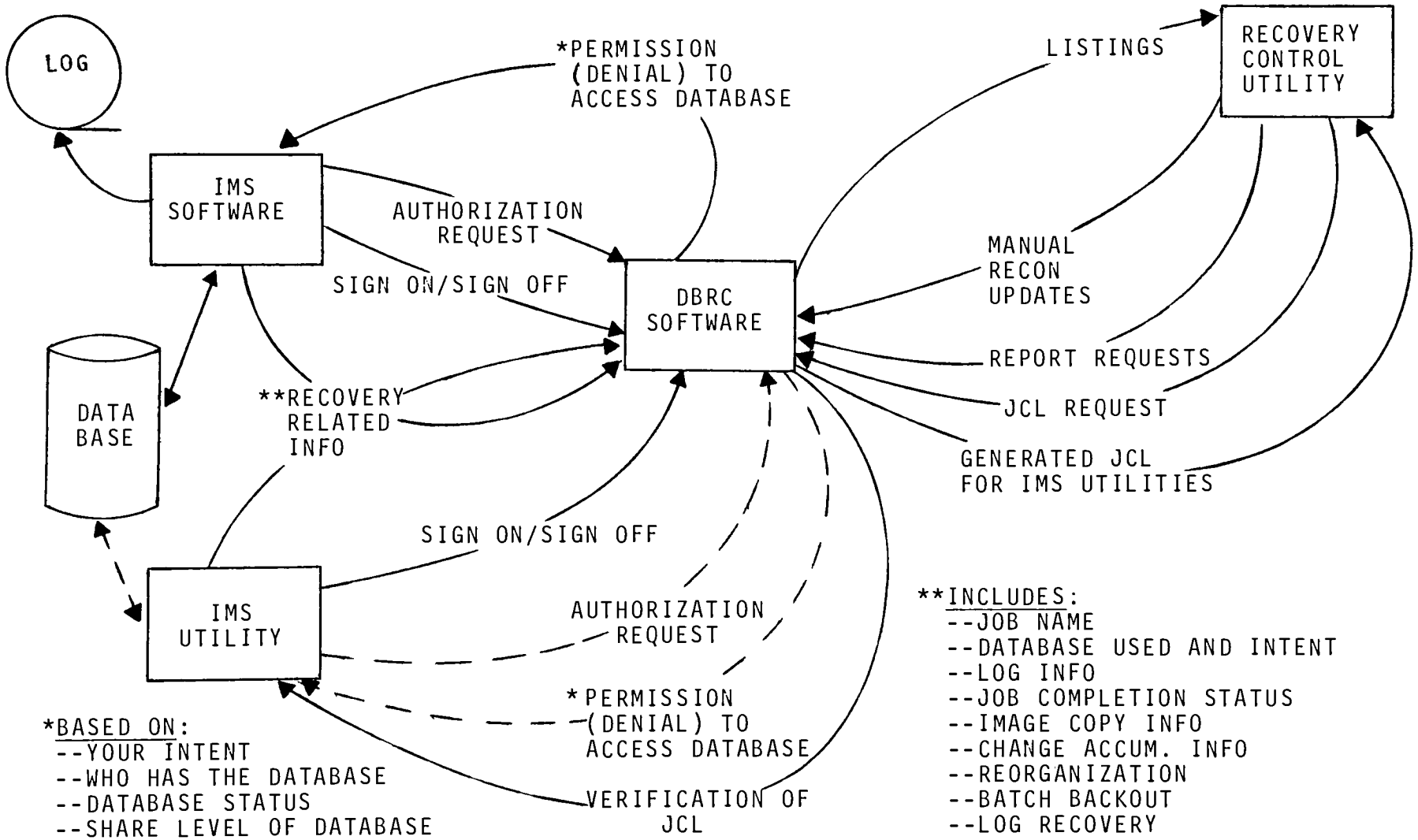
If a batch job normally terminates, then a "sign off normal" call is made to DBRC. Database authorizations are released and the SUBSYS record is deleted.

If a batch job step abnormally terminates, a "sign off abnormal" call is made to DBRC. Authorization is released for databases that have not been updated. However, databases which were modified will not have their authorization released, the SUBSYS record is not deleted, and it is marked as abnormally terminated. Batch backout must be run to release any databases still authorized and to delete the SUBSYS record. The operation of DBRC in an on-line environment is very similar to the batch system just described.

The lower left-hand side of Figure 16 shows the interaction between IMS utilities and DBRC. If a batch backout was submitted to undo the effects of an aborted transaction, the first operation done by the utility is through the sign-on exit. A SUBSYS record must exist for the job that is being backed out. If this record does not exist, DBRC will cause the backout to fail since it does not know the job aborted. Figure 16 shows the authorization request and access permission/denial lines as dotted lines, since these activities are not performed by all utilities. The batch backout utility does not request database authorization. Since the utility can only be run for failed jobs, there is no integrity exposure or need to authorize the databases. The JCL submitted in the utility is verified by DBRC. After the utility completes, a sign-off is done to DBRC.

The Recovery Control utility portion of DBRC is illustrated on the right side of Figure 16. Listings can be provided which print the contents of the RECONs, manual updates can be done, and JCL for certain recovery utilities can be automatically generated.

Figure 16. DBRC Communication.



B. DB2

IBM's relational database management system, DB2, incorporates many of the recovery techniques previously discussed in this paper. It uses a recovery log, redo/undo logic, an update-in-place strategy, and the two-phase commit protocol, when needed.

DB2's recovery log consists of three parts--an active log residing in main storage, an archive log on disk, tape, or drum, and a special dataset called the Bootstrap dataset. The Bootstrap dataset does the following: determines which log datasets contain the needed recovery data, provides pointers to checkpoint log records, and holds other information needed for restart and recovery. Log records are written to the active log unless they are specifically forced to disk. If all the available active log buffers become full, they are offloaded to an archive log dataset.

At the beginning of each transaction, a begin-transaction log record is written. The updates in turn generate log records, which are followed by an end-transaction log record. All the log records for a particular transaction are linked together. The various phases of the commit process and checkpoint records are also logged.

DB2 may be initiated through another system, such as IBM's online system, CICS. In this case, the two products have their own recovery procedures which must be coordinated with each other. This synchronization is done

through the two-phase commit procedure. Depending on which subsystem is used, either DB2 or the subsystem is assigned the role of coordinator. When CICS invokes DB2, for example, CICS assumes the coordinator responsibility. Both systems log the commit proceedings in their own log files.

In addition to the transaction control log records, the following information is also stored in the log files. Whenever a tablespace (holding one or more DB2 tables) or indexspace is opened or closed, this activity is recorded. This information enables DB2 to identify which portions of the log need to be processed when recovering a specific tablespace.

Checkpoints, which are taken at periodic intervals, are also noted in the log. The checkpoint frequency is proportional to the amount of log activity. The checkpoint process can occur at the same time as normal database activity. A checkpoint includes listing all the active transactions in the log and noting information about each open tablespace and indexspace on the log. DB2 always begins the restart process at the last complete checkpoint. The frequency of checkpoints has a direct bearing on how much log data will need to be processed for a system restart, thus affecting recovery time.

DB2 writes uncommitted updates back to the physical database in order to reclaim buffer space. This property makes it necessary for DB2 to undo uncommitted updates in

case of transaction or system failure. Once a page has been updated, it can be selected to be swapped out. When the system swaps an updated page out, the log up to that point is forced to non-volatile storage.

If the system fails before the updated pages are written, the recovery process must redo any committed changes which are not reflected in the database. In order to perform these undo/redo actions, appropriate information is written to the log with each update operation. The page number affected is included in the log record.

During the recovery process, compensation log records are written to track the undo log records which were acted upon during recovery. The compensation log records are needed in case of failure during the recovery process. If a crash occurred during an undo operation, the result of the undo may not yet have been recorded on disk. When the restart begins again, the compensation log records will indicate if previous recovery actions have not completed.

Each page header for DB2 stores a field which identifies the log record pertaining to the last update done to that page. This field is used to determine if an update described in a log record has actually been forced to the physical database. If the last update identifier in the database page is less than the location of the log record being examined, the update has not been applied to the actual database. On the other hand, if the last update identifier in the physical page is greater than or

equal to the location of the log record, the update was reflected in the database. The redo and undo operations work in a very similar fashion to the descriptions given earlier in this paper. However, extra analysis is required to process the compensation logs correctly.

DB2 supports two types of database image copies. One is a full copy of a tablespace. The other feature is an incremental copy including only those pages which have been modified since the last full copy.

To process logs more efficiently when recovery is needed, DB2 keeps track of when updates began and ended to a specific DB2 data object, as reflected in the log. If media recovery is required, only the portion of the log containing updates to the affected object is processed. Uncommitted updates contain compensation log records which are used to back out those updates. The media recovery process will act upon all log records, even those pertaining to uncommitted transactions. However, a compensation log record will undo the uncommitted update when it is processed.

System restart procedures follow very closely the actions discussed in the system failure section. REDO and UNDO lists are constructed by reading the log taken since the last checkpoint, to determine which transactions started and committed, versus which transactions are not complete. Also the tablespaces and indexspaces used since the last checkpoint are noted.

The earliest point containing update records for any open tablespace is found in the log. At this location, the log is processed in a forward direction to redo any uncompleted updates not written to disk, including those to be committed. The log is then processed in reverse order to undo any updates of incomplete transactions, or those which were in abort status. All updates are then forced to non-volatile storage, and a checkpoint is taken.

C. SDD-1

SDD-1 is a prototype distributed database system which was developed by Computer Corporation of America [HAMM80]. The reliability system is based upon RelNet (Reliable Network). RelNet employs site status monitoring, event timestamping, guaranteed delivery of messages, and the atomic control of transactions.

RelNet is built as a series of layers. Below the RelNet architecture is the message transmission layer. In SDD-1, ARPANET is used as the facility for sending and receiving messages between sites. At this layer, no guarantees are made that a message sent by one site will actually be received at the destination. However, higher levels of RelNet do expect responses for messages which have been sent. The lowest level in RelNet is the Global Time Layer, which maintains a global clock used by all layers. This layer synchronizes the actions performed at the other sites, and keeps track of all site statuses. The Guaranteed Delivery layer enables communication between sites, even when the receiving site is down. The Transaction Control layer is responsible for the correct execution of transactions, despite failures which may occur at the participating sites.

Global Time Layer

One responsibility of this layer is to make sure that events occurring within a single process which involve

other sites are perceived to occur in the same order when viewed by all sites. To achieve this goal, a global clock is used to analyze events. In order to simulate a global clock, each site maintains a local clock, which is used to provide the timestamps for events occurring in that site. The local clock is incremented each time a timestamp is requested. In order to support the idea of a global clock to order events among all sites, the timestamps attached to received messages are examined. If the timestamp on a received message is greater than the local clock value on the receiving site, the local clock value is increased to a value beyond the timestamp value. If this is not done, an action done at the receiving site which is sent to other sites may have a timestamp earlier than that of the previous message received. This would not be reasonable, since the action actually occurred after the message was received.

When a site recovers from a failure, the Global Time Layer initializes the local clock to a value of zero. Receipt of messages by other sites will restore the local clock to an appropriate value.

Each site maintains a local status table which is used to store the site's knowledge of the status of all sites in the network. A Watch facility exists which is used by a site if it wishes to be informed when another site is in a particular status. When the status is reached, the requesting process is notified.

If a site does not respond to a message within a certain timeframe, the site is assumed to be down. Procedures are followed to make sure that the site is really down. If so, the site is marked as down in the local status table. RelNet preassigns guardians for each site which handle the processing when a site crashes. The failure of a site is not broadcast to all sites in the network. Only the site discovering the failure and one of the site's guardians need to know. This approach assumes that only sites which attempt to interact with a site must know its status, and each of these can learn of a failure independently. This idea eliminates the need for synchronizing the communication of site failure and recovery among all other sites in the network, which can become very complicated. However, when a site recovers, the Global Time layer does notify all other sites of the up status.

When a message is to be sent to another site, the local status table entry for that site is interrogated. If the site is listed as down, the message is discarded and the sending process is informed that the message is undeliverable. If the site is listed as up, the message is sent.

When messages are received, the status of the sending site is also checked. If the status is marked as up, the message is either sent to the user process or passed on to a higher layer of RelNet. However, if the status is indicated to be down, the type of message becomes important.

Messages indicating that a site has just recovered and is now up are processed by marking the site's status as up in the local status table and issuing a response to the sender. Any other message received from a site marked as down implies that the site is not actually in a down state. However, actions may have already been performed based on the assumption that the site was truly down. In this case, a message is sent to the site telling it to perform as though it had indeed failed, followed by a recovery of that site. Any change to the local status table may cause sites which had issued a Watch against a recovering site to be notified of its change in status.

In RelNet, only two statuses are considered, up or down. In reality, the behavior of a node in a distributed system can be summarized by the following four states.

1. DOWN - Not operating

2. SLOW - Running, but slow in responding to messages, thus hitting the time-out interval.

3. FAULTY - Answering messages within the time-out interval; however, responses are incorrect.

4. UP - Running correctly and answering messages within the specified interval.

SDD-1 does not distinguish between all these possibilities. RelNet cannot determine whether another site fails to respond because it has crashed, or just because it is slow. Therefore, any site which does not respond to messages within a prespecified time frame is assumed to be

down. However, the Global Time layer recognizes the fact that the site is potentially only slow, and accommodates this by sending a "YOU'RE DOWN" message to the appropriate site. In addition, it is also not possible to distinguish beween faulty sites and those sites which are operating properly. In this case, both situations would consider the site to be up. Components outside RelNet can explicitly crash a site if it is found to be in error.

Guaranteed Delivery Layer

The fact that an intended recipient for a message may fail before a message has been delivered could result in loss of messages. However, this would jeopardize database consistency in a distributed system. A transaction which updates a distributed database must be certain that crucial messages, such as updates, will be eventually delivered to all destination sites. This is the duty of the guaranteed delivery layer.

Messages may be designated for "guaranteed delivery." In this case, the RelNet ensures that, if the destination site is currently down, it will receive the message when it recovers. As an example, suppose two messages are sent to Site A. If one message is flagged for guaranteed delivery, Site A will definitely receive the message when it recovers. However, the other message sent may not arrive at all.

The sender of a guaranteed message receives an acknowledgement from the Guaranteed Delivery layer when the message has been processed by this layer. At this point the sender can be assured that the message will reach its destination if the site eventually recovers. When a guaranteed message is received by the destination, an acknowledgement must also be sent to the Guaranteed Delivery Layer indicating that the message will be processed. After this last acknowledgement, the Guaranteed Delivery layer considers the message to be successfully delivered. If the destination fails again before it can return an acknowlegement, the message will be resent by the Guaranteed Delivery layer when the site again recovers.

Another function of this layer is the Check primitive which allows a site to determine if it has received all messages sent to it by a certain time. This facility is used by a recovery site to determine if it has received all messages sent to it while it was down and also to ensure that all messages sent by a crashed site before it went down have been received.

The promise that messages will be delivered is accomplished through use of the Reliable Buffer, which is located at all sites. This storage area holds all messages destined for a failed site which are flagged for guaranteed delivery. During the recovery process of a failed site, a request is made to receive all messages in its

Reliable Buffer. The recovering location will then store these messages in its own non-volatile storage, and process the messages in the correct order once recovery is complete.

The Reliable Buffer is internal to RelNet. It is implemented by replicating and coordinating each message at several sites in the network. Crashes may occur when the Reliable Buffer is being read or written to. Variations of these failures are all accounted for in the Guaranteed Delivery layer.

Transaction Control Layer

The Transaction Control Layer ensures that transactions remain atomic. The execution of a transaction typically involves sending update mesages from one site to a number of other sites. The sending site may crash before completing all the updates for a transaction. In order to maintain data integrity, all updates for a transaction must be processed by all associated sites. Although the unsent messages may be guaranteed delivery, it is unrealistic in the SDD-1 enviroment to wait indefinitely until a site recovers to complete a transaction. Other processes dependent upon completion of previous transactions would also be forced to wait.

The above problem is resolved by an atomic commit procedure. Whenever any update messages become held up due to a failed site, the transaction with which they are associated will be aborted, and none of its updates will be performed. Only when all of the update messages are issued will the transaction be committed. At that point, all of the updates will actually take place.

The commit procedure in RelNet is a four-phase process. It makes use of backup processes which take control if the committing process fails. The commit procedure consists of the following steps:

Phase One - The committing process, C, appoints a certain number of backup processes which are informed of all sites involved in the transaction, and all the other backup processes. The backup processes send a message to C to indicate that they know of their backup role.

Phase Two - C issues UPDATE messages to the other participating sites, causing the local updates to prepare to commit. C must receive acknowledgement from the Guaranteed Delivery layer, indicating that the messages were received.

Phase Three - C issues COMMIT messages to all the backup processes, and waits for acknowledgements for the messages.

Phase Four - C sends the COMMIT messages to the other participating sites, causing the commit to take place and the local databases to be updated. Once C has received

acknowledgement for the COMMIT messages, it destroys the backup processes and considers the commit procedure to be complete.

The role of the backups is as follows. The backup processes default to an abort state unless a COMMIT message is received. Each backup watches its predecessor. If the process it is watching fails, it first determines if it is watching C, the committing process. If so, the backup process assumes control of the transaction and issues a COMMIT or ABORT message based on its own state. Therefore, if Phase Three has not begun, a COMMIT will not be issued. If the process which failed was another backup process, a failure watch is issued against the failed backup's predecessor.

XI. CASE STUDY

The recovery methods employed in the computer installation of a manufacturing company were studied. This site utilizes large-scale IBM3083J and IBM3081G central processing units running MVS/XA (Extended Architecture) as the operating system in conjunction with JES3, the Job Entry and Scheduling subsystem. The main database product is IMS. DB2 is an adhoc end user tool providing inquiry and update capabilities for end user databases both in an on-line and batch environment. The IMS databases are copied to DB2 databases weekly for ease of inquiry. CICS is the major application system providing on-line access to the divisional IMS databases.

The online CICS environment is available during normal work days from 6 a.m. to 6 p.m. The remainder of the time comprises the batch window. Due to the enormous amount of data and jobs being processed, the batch window is very tight. Problems encountered during the evening processing must be handled very quickly. Thus, reliable and timely recovery procedures are essential.

The database administrator stated that a "never go back" strategy is a basic principle. Once a database update job has successfully completed, the job will not be rerun. If a problem, such as bad input data, is detected, the approach is to generate corrective transactions to adjust for the bad processing. A data sharing facility

exists in IMS/VS which allows multiple application programs to concurrently process common DL/1 databases. Data sharing complicates the recovery issue and virtually makes the restore database/rerun job approach extinct.

There are three layers of data protection--an image copy, change accumulation, and raw log datasets. Dual logging is used, which creates two logs for one update function. One actual change accumulation dataset is made, but a copy of that file is created. In addition, a remote copy of the image copy is made. This backup is stored in a different location at the plant than that of the computer center.

Three image copies, along with their associated change accumulations and log tapes, are kept to allow recovery from the past. Most image copies are regularly scheduled to run once a week. Adjustments are made to this default based on update frequency, database volume, and database volatility. Change accumulations are run twice a day--once after the on-line system, CICS, is brought down and next after the batch network has completed.

The frequency of image copies can be analyzed in a historical perspective. When IMS databases were first used, image copies were made after every update job. This was necessary in order to handle the manual effort involved in recovery. Later image copy frequency was changed to occur after several update jobs had completed.

In this case, if a media failure was detected during the last update job, the logs from all those update jobs would have to be applied to the previous image copy in order to recover.

Once DBRC was available and operating in the installation, the image copy frequency was changed to occur once a day, after CICS was brought down for the day. Due to increasing database sizes and a very tight batch window created by a newly installed production control application system, the image copies became too time consuming to fit into the network. As a result, the image copy was substituted with a change accumulation to collect all the log updates. The change accumulations occur after CICS and after all the batch jobs for that database group. The image copies are created every weekend.

DBRC involved a large-scale effort by many people to incorporate it into the operating environment. There were several problems along the migration path to full DBRC utilization. Many problems were due to the large learning curve associated with DBRC.

When incorporating any recovery product into an installation, care must be taken to insure that adequate recovery procedures are in place for the recovery tools themselves. The following example illustrates how hidden some clues may be which indicate a problem has occurred. The RECONs are the heart of DBRC recovery and the ability to recover exists only as long as these datasets remain intact.

In the adaptation of DBRC to this installation, a job was written to make RECON I/O errors more visible. In earlier releases of DBRC which had only two RECONs, if one RECON was lost, the jobs currently running would finish, but any new jobs would fail due to unavailable RECONs. This would signal the group supporting the production jobs that a new RECON must be created and copied from the good RECON. However, the current release of DBRC uses three RECONs--two currently used copies and a spare. If an I/O error occurs on one of the copies, DBRC will automatically write the contents of the good RECON to the spare, and tag the bad RECON as discarded. The spare RECON now becomes the second copy. This activity is completely transparent to the application. An error message is sent to the system log when RECON replacement occurs, which is easily missed. Therefore, a job was written to list the status of the RECONs. The job runs twice a day--at 4 a.m. and 4 p.m. The header record on the RECON is scanned and, if the status of a RECON is listed as "discarded," the job will bomb. The abortion of this job signals the operators that one of the RECONs had an I/O error and must be redefined.

Log datasets are put on a particular medium based on how many updates a certain job typically makes. For large jobs, both logs are sent to tape. For medium-size jobs, one log will go to tape and the second log is written on disk. Both logs are written to disk for small update jobs.

Media failures are very rare at this installation. Most recovery needs have arisen due to application abends. The vast majority of recovery uses are for backing out application updates. Forward recovery has only been used for application abends, after database integrity has been destroyed due to a mistake during the batch backout.

The next example illustrates how easy it is to provide bad input to a batch backout. Assume that an update job with two logs has aborted. A space problem occurred with the logs where both datasets ran out of space. A key point is that the primary log ran out of space before the secondary log. Therefore, the secondary log had updates recorded on it which were not represented in the primary log. In the majority of the batch backouts, the application program causes the job to abort, and the logs are fine. As a norm, people tend to bring in the primary log into the batch backout, although either log would suffice. The same procedure was followed for this backout. As a result, updates had been made, but not backed out. This situation did occur at this site, and the bad backout was identified only after a later job aborted due to database integrity problems identified by an application. Problems such as these could be avoided if products such as DBRC were extended to handle application recoveries in addition to database recoveries. The automatic generation of JCL for backouts would eliminate backout problems caused by bad input. Also, this example points out that the primary log should always have a bigger space allocation than the secondary log.

XII. ANALYZING DATABASE RECOVERY TECHNIQUES

During my research on database recovery, I have found a lack of information analyzing the various available techniques. The strengths and weaknesses of each technique should be carefully analyzed to determine which one is most suitable to a specific set of circumstances. The trade-offs between these different techniques have been summarized in Figure 17.

Important criteria which should be considered when analyzing recovery products include:

1. overhead during normal operation of the database system;
2. recovery speed after a failure;
3. degree of reliability;
4. additional system requirements needed;
5. cost of backup and recovery operations.

It seems essential that all recovery management systems must be able to recover from the three basic types of failures--transaction, system, and media failures. The techniques chosen may put unacceptable overhead on the system, making transaction processing very slow. Causes include complex page fetch algorithms, effects of certain logging techniques, and page update strategies. When choosing the appropriate recovery techniques, answers depend on tradeoffs. Longer transaction response time and greater cost is often a tradeoff if faster recovery from failures is desired. The degree of reliability is an important issue, since the recovery manager may only guarantee a consistent state which is out of date, causing

work to be lost. On the other hand, more overhead can allow the current database to be recreated. In addition, if an installation has limited storage, shadow or differential file techniques may present a problem.

Although dumps make the recovery process faster as they are taken more frequently, a heavier burden is placed on throughput. For example, the longer the time between dumps, the more before/after images are generated, thus the more I/O activity is required to perform a recovery operation. The same trade-off is true for checkpoints. It is important to minimize the time between dumps/checkpoints. During that process, however, no updates are allowed to the database. Depending on the size of the database, this restriction could significantly delay update processing. The ability to save or restore several sections of the database simultaneously can lessen the impact (at the expense of complicating the procedure).

Recovery overhead, recovery speed, and system requirements are all issues that are affected by the method of propagation used to update pages. A database transaction normally affects more than one physical page. In database systems which employ an update-in-place strategy, the physical database is usually inconsistent, since pages of related changes can be written only one at a time. The propagation order is likely determined by the buffer replacement algorithm. The recovery mechanism must erase these inconsistencies when restoring to a consistent database state. Logs are used to keep records of the updates.

On the other hand, the shadow page and differential file techniques write related pages which were updated to a different disk location than that used for storing the old pages. These algorithms are classified under atomic propagation, since either all updated pages or no updates will be reflected in the actual database. In the event of a system failure, the old, consistent version of the database is immediately available. Transaction aborts are easily handled by discarding the updated pages. The recovery manager has much less work to do when compared to the update-in-place strategy. However, periodically the updated pages must be used to update the actual database so that the pages will contain the new values even after a crash.

It is more expensive to perform undo transactions with logging when compared with shadows or differential files. However, logging puts a smaller burden on a successful transaction. Since most transactions succeed rather than abort, logging emerges as the best mechanism. The major disadvantage of shadows is the cost of indirection through the page table. The disadvantage of the differential file approach is the overhead of reading differential file pages and the extra processing required to obtain the correct version of a page.

[AGRA85c] evaluates logging, shadow pages, and differential files in terms of their performance impact. Parallel logging allows logging to occur in parallel at

more than one log disk, thus making the recording of recovery data more efficient. The parallel log architecture appears to be the best recovery architecture since the recovery data can be accumulated simultaneously with the processing of data pages. These findings are consistent with the design of the majority of recovery components, which incorporate logging to record changes.

When the differential file technique was studied, the size of the differential file was assumed to be only 10% of the base file. This architecture degrades the throughput of the database machine because of the extra disk I/Os required to access the differential file pages. If the size of the differential file is increased, the processors quickly become saturated.

For the shadow page architecture, the throughput does degrade somewhat for random transactions. Throughput can improve by increasing the buffer size or utilizing multiple page table processors until there is virtually no impact on performance. If logically adjacent pages can be stored physically close, the performance of sequential transactions is very good. If physical clustering is not feasible, performance becomes much worse due to the relatively large seek times. If the overwriting approach is used, which writes the new page over the shadow page upon transaction completion, the physical and logical data ordering can be maintained. However, the overwriting architecture performs poorly due to the extra accesses to the data disks.

A design issue related to logging is the choice of a method for storing the images. One approach is to keep the before-images of a transaction separate from the after-images. This is because the before-images are only used for backing out a transaction and for some read-only transactions. Before-images may be discarded once the transaction has committed and the read-only transactions initiated before that transaction has completed. On the other hand, the after-images are needed for forward recovery from media failures. Many systems keep before- and after-images on the same recovery log. I/O cost is reduced during normal writing to the logs when the images are combined. However, both backward and forward recovery will be slowed down due to skipping over the images ignored by that recovery technique.

Page replacement strategies can affect system requirements. One policy writes updated pages to the actual disk database before the transaction has committed. Another discipline prevents dirty pages from being written until the transaction has successfully completed or a commit has been issued. In this case, a sufficiently large database buffer is needed. This second policy avoids undoing updates if a transaction fails or system crashes.

The level of granularity used for the before and after images is another issue. The simplest approach is to make those images identical to the physical unit of

retrieval of a database system, called a block or page. With page-level granularity, the recovery process involves overwriting pages in the database without reading them first. If the after images are sorted into physical address order and reverse time sequence, only the most current after image of a given page must be written. This procedure significantly improves the forward recovery process; however, the cost of performing this sort may exceed the savings realized. Although this process can also be applied to backward recovery, the smaller time between checkpoints detracts from its cost effectiveness.

Another technique that reduces throughput overhead uses record granularity to write before and after images. The advantage of this approach is that the size of the log file is greatly reduced, lessening the I/O activity to this file. The cost in system resources to write a single element of a full page is still one I/O operation, however. To use this technique optimally, the log is usually buffered so that a block is not written until it contains enough images to fill the buffer. Record-level granularity can cause fewer I/O operations to the log file. The recovery mechanism is less efficient, however, because the affected pages must be read, a specific record altered, and written back to the database. All updates to a page--not just the most recent--must be performed.

As organizations rely more heavily on their computer systems, their ability to continue processing without

those systems is diminished. This poses two challenges to the data administration function: to develop and implement those steps necessary to ensure processing continuity of automated systems, and to develop procedures to aid individuals in running their business during periods when computerized systems are down. These two tasks are extremely important and may determine the success of the database installation.

The database administrator should estimate the cost of loss associated with various types of database failures so that the needed backup and recovery procedures can be installed. Users must identify how quickly the database must be recovered for their applications. That step will indicate the length of time users can survive without the computer system, and establishes recovery time requirements. Alternate processing procedures should be developed for the users during a failure, when necesary. Also, procedures must exist to verify the integrity of data immediately following recovery procedures.

Recovery products exist which aid in the recovery process. Automation in recording log data set information, image copies, and other recovery information is one function of such products. In addition, this data can be used to create job streams for recovery purposes. It is very desirable to be able to collect this information and reduce human intervention in the recovery area. The purchase of a recovery assistance product is recommended for large installations.

TECHNIQUE	DESCRIPTION	ADVANTAGES	DISADVANTAGES
Logging/ Update-In-Place	Information about changes made to the database is written to a log file. Updates are written to database immediately.	*Recovery information is stored separately from database. *If logs are on tape, they can be stored offsite. *If each application has its own log, recoveries for different jobs may be done in parallel. *Can recover to any point in time.	*Physical database is usually inconsistent, since pages of changes are written only one at a time. *Every operation on the database requires a log entry. *Can contain many different types of records. Much information is often skipped over during recovery, which slows down the process. *More expensive for UNDO actions than with shadows/differential files.
Shadow Files	Two copies of updated objects are kept until transaction commits.	*Can increase parallelism by taking advantage of two copies of the pages. *Database remains in a consistent state after transaction and system failures due to atomic propagation. *When compared to differential files, multiple copies of pages are kept only until transaction commits; duplicate pages exist much longer in differential files.	*Additional disk space is needed for shadow pages; becomes a problem for large files. *Increased overhead costs due to more disk seeks in order to find a page. *Technique becomes very complex if concurrent users are allowed to update the same data.
Differential Files	Accumulates update requests for the database in a sequential file.	*Database remains in a consistent state after transaction and system failures due to atomic propagation. *Quick recoverability when physical database is damaged. *Back-up costs and chances of serious data loss are reduced.	*Complicated page fetch algorithms. *Access of data can be slow. *Propagation of changes to physical database can be time-consuming and will interrupt system availability.

Figure 17. Trade-Off Analysis.

TECHNIQUE	DESCRIPTION	ADVANTAGES	DISADVANTAGES
Checkpoints	Updated pages are forced from the buffer to the physical database.	*Recovery time decreases as checkpoint frequency increases.	*No activity may be done during a checkpoint. Amount of processing delay varies among different checkpointing schemes.
Dumps	Periodically copy the direct access version of the database onto tape.	*Image copies are necessary for media failures and are a starting point for many recovery operations.	*If database is large, the copy jobs may be time consuming.
Restore/Rerun	If system fails, restore database to latest database copy and rerun all programs, jobs, and transactions that completed since then.	*Simple in terms of required DBMS architecture. *Can be added to a system where recovery is not a part of the original architecture.	*Very time consuming to rerun all transactions. *If database is large, preventing frequent copies from being created, the rerun time may be unacceptable. *It may be impossible to duplicate the order in which transactions were originally processed if tasks are allowed to intermingle.
Forward Recovery	Restoring the database from a backup copy, and reposting all changes that were made since the copy was created.	*Less time consuming than restore/rerun. *If there is a media failure, only restore/rerun or forward recovery will be successful. *Useful for recovery from system, transaction, and media failures.	*DBMS must log after-image records for all database changes.
Backward Recovery	Backs out changes created by an application program that has not terminated normally.	*Service is usually returned to end-users much faster by backing out a transaction rather than doing a forward recovery.	*Must log before-image records for all database changes. This increases amount of I/O during normal execution. *Not useful for media failures.

Figure 17. Trade-Off Analysis.

XIII. CONCLUSION

The majority of research efforts in the recovery area has been focused on the actual recovery algorithms needed in order to provide high reliability. However, little has been done to provide customers with an analysis of what they can expect from the database system in terms of availability and performance (response time and throughput). The goals of continuous service and performance are often conflicting ones. Recovery techniques and algorithms require additional system recovery resources in terms of both hardware requirements and data. In addition, recovery methods put an additional workload on the system, which generally results in decreased performance during fault-free operation. Also, the cost of designing and implementing fault-tolerant algorithms can be high, and the marginal gain obtained in availability through use of sophisticated techniques may not seem cost-effective. The same resources needed to provide fault-tolerant operation could be used instead to enhance system performance. Therefore, quantitative methods must be developed for evaluating the performance, availability, and cost of database systems.

The increasing use of distributed computing systems makes research in that area of recovery very important. Distributed databases need protocols which keep track of multi-site transactions. Communication among separate

sites must result in uniform and atomic transaction commits or rejections. Available versus failed sites must be recognized. Replicated data is another problem area since the data must be consistent in all locations.

The trend for the future of database recovery is to make it even more automatic and dynamic than it is today. The recovery system should assess the damage, perform the recovery, and advise users of possible effects of the recovery. Products are currently available which assist groups in recovering physical databases by basically providing a history of logs, updates, image copies, reorganizations, etc. The next step is to develop aid for handling application abends. This support would help eradicate human error involved in application recovery.

The subject of database recovery is critically important in order to ensure data integrity. Installations must be aware of the limitations of their recovery products and either make adjustments or accept those restrictions. This paper presents and analyzes many recovery techniques. As an aid to further explanation of these concepts, overall descriptions of different approaches to the design of recovery components in existing systems are given. Although some research in this area has been done, further investigation is mandated.

XIV. BIBLIOGRAPHY

ACHA85 Acharya, S. and Buckley, G. "Transaction Restarts in Prolog Database Systems." Proceedings of ACM-SIGMOD 1985: International Conference on Management of Data (1985), pp. 364-373.
An overview of the Prolog state maintenance mechanisms is given. Optimistic concurrency control schemes and a restart algorithm are described in detail.

AGHI82 Aghili, H., and Severance, D. "A Practical Guide to the Design of Differential Files for Recovery of On-line Databases." ACM Transactions on Database Systems, Vol. 7, No. 4 (December 1982), pp. 540-565.
This paper studies the problem of database backup and recovery for on-line systems using the concept of a differential file. Five key design decisions are identified and an optimization procedure for each is developed.

AGRA85a Agrawala, A.K. "A Non-Intrusive Checkpointing Scheme in Distributed Database Systems." IEEE 1985 Fault-Tolerant Computing, pp. 99-103.
A new scheme for checkpointing in distributed database systems is proposed and compared with other schemes. This technique is non-interfering with transaction processing and efficiently generates globally consistent checkpoints.

AGRA85b Agrawal, R. and Dewitt, D. "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation." ACM Transactions on Database Systems, Vol. 10, No. 4 (December 1985), pp. 529-564.
A summary of basic concurrency control mechanisms and recovery mechanisms is discussed. The interaction between concurrency and recovery and the comparison between the performances of these mechanisms is presented.

AGRA85c Agrawal, R. and Dewitt, D. "Recovery Architectures for Multiprocessor Database Machines." Proceedings of ACM-SIGMOD 1985: International Conference on Management of Data (1985), pp. 131-145.
In this paper, several parallel recovery architectures for multiprocessor database machines are proposed. The characteristics and performances of each are evaluated, as well as the impact on performance.

AGRA86 Agrawala, A., Hyukson, S. "A Token Based Resiliency Control Scheme in Replicated Database Systems," IEEE Fifth Symposium on Reliability in Distributed Software and Database Systems, (1986), pp. 199-206.
A token scheme for coordinating updates to replicated distributed databases is presented. Database copies are tagged as either read-write copies or read-only copies. Read-only copies are updated by a remote copy actualization process when a modified data object needs to be read. Recovery schemes are presented in terms of preserving database consistency.

ALAV83 Alavian, F., Avizienis, A., Cardenas, A. "Performance of Recovery Architectures in Parallel Associative Database Processors." ACM Transactions on Database Systems, Vol. 8, No. 3 (September 1983), pp. 291-323.
Three different types of recovery mechanisms that may be considered for parallel associative database processors are identified. The workload imposed by the recovery mechanisms on the execution of database operations and the workload involved in actual recovery is analyzed. The performance of the three architectures is also compared.

ANDE78 Anderson, T., Lee, P., Shrivastava, S. "A Model of Recoverability in Multilevel Systems." IEEE Transactions on Software Engineering (November 1978), pp. 486-493.
This paper discusses two distinct categories of multilevel systems and then examines in detail the issues involved in providing backward error recovery in both types of systems.

ARDI79 Arditi, J. "An Optimized Backout Mechanism for Sequential Updates." IEEE Very Large Data Bases: Fifth International Conference (1979), pp. 147-154.
A backout mechanism is described in this paper which uses the technique of periodic updating of a key-sorted data base from a key-sorted "transaction file" as an option for updating databases.

ATTA84 Attar, R., Bernstein, P., Goodman, N. "Site Initialization, Recovery, and Backup in a Distributed Database System." IEEE Transactions on Software Engineering, Vol. SE-10, No. 6 (November 1984), pp. 645-649.
The problem of merging a recovering site into a distributed database system is addressed. An algorithm for site initialization is presented.

BARI83 Barigazzi, G. and Strigini, L. "Application-Transparent Setting of Recovery Points." IEEE 1983 Fault-Tolerant Computing (June 1983), pp. 48-55.
An error recovery method for multiprocessor systems is presented, with emphasis on transparency to programmers and minimum backup information.

BAYE80 Bayer, R., Heller, H., Reiser, A. "Parallelism and Recovery in Database Systems." ACM Transactions on Database Systems, Vol. 5, No. 2 (June 1980), pp. 139-156.
For recovery reasons, two copies of an object often exist until a transaction is fully committed. Usually an update transaction blocks all other transactions until it has completed in order to avoid inconsistency. This paper describes a scheme to increase concurrency by allowing read-only processes to look at the old copy of an object while an update transaction is creating a new copy.

BAYE84 Bayer, R. and Elhardt, K. "A Database Cache for High Performance and Fast Restart in Database Systems." ACM Transactions on Database Systems, Vol. 9, No. 4 (December 1984), pp. 503-525.

Characteristics of existing recovery methods are described. Design principles of the database cache are presented and compared to buffer management. For typical applications requiring high throughput of short transactions, the database cache method leads to an essential performance enhancement.

BERN83 Bernstein, P., Goodman, N., Hadzilacos, V. "Recovery Algorithms for Database Systems." Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, 1983, pp. 799-807.

This paper is a survey of recovery algorithms for centralized and distributed database systems.

BERN84 Bernstein, P. and Goodman, N. "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases." ACM Transactions on Database Systems (June 1984), pp. 597-615.

This paper describes an algorithm for handling replicated data, which allows users to operate on data so long as one copy is "available." The algorithm handles clean, detectable site failures.

BHAR81 Bhargava, B. and Lilien, L. "Feature Analysis of Selected Database Recovery Techniques: AFIPS Conference Proceedings: 1981 National Computer Conference, Vol. 50 (1981), pp. 543-554.

Database recovery techniques in a real-time environment for single-division databases are investigated. The recovery techniques discussed center on an idea of a database recovery block.

BHAR82 Bhargava, B. "Evaluation of the Network Merging Protocol Based on the Optimistic Concurrency Control and Rollback Semantics." IEEE Computer Software and Applications Conference (COMPSAC), 1982, pp. 447-451.

The performance questions regarding the optimistic approach for network partitioning are discussed in this paper.

BHAR84 Bhargava, B. and Lilie, L. "A Scheme for Batch Verification of Integrity Assertions in a Database System." IEEE Transactions on Software Engineering (November 1984), pp. 664-680.

A technique for implementing an integrity control subsystem to ensure the semantic integrity of a database is proposed. After a database is updated by transactions, its integrity must be verified by evaluation of a set of semantic integrity assertions. Implementation issues and algorithms are included.

BHAR86 Bhargava, B., Ruan, Z. "Site Recovery in Replicated Distributed Database Systems." IEEE Distributed Computing Systems (1986), pp. 621-627.
The problem of bringing a recovering site back into the active system is discussed. Methods for handling transactions which were interrupted during a failure and also restoring the consistency of the database due to updates are presented. Control transactions are responsible for maintaining consistent views of site status across all sites. Specific recovery procedures are analyzed.

BJOR75 Bjork, L. "Generalized Audit Trail Requirements and Concepts for Data Base Applications." IBM Systems Journal, No. 3 (1975), pp. 229-245.
The aspects of database auditing are covered in this article, including the content and format tradeoffs of an audit trail, time domain addressing, and hardware implications.

BLAS81 Blasgen, M., et al. "The Recovery Manager of the System R Database Manager." Computing Surveys, Vol. 13, No. 2 (June 1981), pp. 223-242.
A description of the System R Recovery Manager is given. Implementation techniques are discussed including files, logs, transactions, checkpoints, system restart, media failure, recovery and locking. Evaluations of cost and time are also included.

BRUS85 Bruso, S. "A Failure Detection and Notification Protocol for Distributed Computing Systems." IEEE 1985 Distributed Computing Systems (1985), pp. 116-123.
The Failure Detection and Notification Protocol (FDNP) has the capability of detecting node crashes and communication link failures in distributed computing systems. This protocol coordinates recovery among the nodes, while the specific transaction recovery is done by the failing node.

CHAN75 Chandy, K. "A Survey of Analytic Models of Rollback and Recovery Strategies." IEEE Computer Magazine, Vol. 8, No. 5 (May 1975), pp. 40-47.
Following a brief discussion of the cost-effectiveness of redundancy schemes, attention is then restricted to the analysis of rollback and recovery strategies, using a data-based system and a process-control system as examples.

CHAN85 Chan, A., Gray. R. "Implementing Distributed Read-Only Transactions." IEEE Transactions on Software Engineering, Vol. SE-11, No. 2 (February 1985), pp. 205-212.
Conflicts between read-only transactions and update transactions are eliminated in a multiversion scheme for a distributed environment. Read-only transactions are guaranteed completion without blocking.

CHOW83 Chow, J. et. al. "Detection of Mutual Inconsistency in Distributed Systems." IEEE Transactions on Software Engineering, Vol. SE-9, No. 3 (May 1983), pp. 240-246.
An approach is presented to detect mutual inconsistency among replicated data through the use of version vectors and origin point constructs.

COHE82 Cohen, D. "Database Systems: Implementation of a Distributed Database Management System to Support Logical Subnetworks." The Bell System Technical Journal, Vol. 61, No. 9 (November 1982), pp. 2459-2474.
This paper describes the design and implementation of a special-purpose distributed data management system. The authorization model used to define logical subnetworks and the related transactions are discussed. Issues addressed in this paper, in the context of transaction processing, include multicopy updates, concurrency control, and crash recovery.

CRUS82 Crus, R.A. and Putzolu, F. "Data Base Allocation Table." IBM Technical Disclosure Bulletin, Vol. 25, No. 7B (December 1982), pp. 3722-3724.
A database management system is described involving a database allocation (DBA) table. This table contains only information about those objects which are in an "exception" state, and is used for restart purposes.

CRUS84 Crus, R.A. "Data Recovery in IBM Database 2." IBM Systems Journal, Vol. 23, No. 2 (1984), pp. 178-188.
This paper presents the various forms of data recovery provided by IBM Database 2 (DB2). It describes the DB2 recovery log, units of recovery, DB2 logging, checkpoints, and recovery processes.

DADA80 Dadam, P. and Schlageter, G. "Recovery in Distributed Databases Based on Non-Synchronized Local Checkpoints." Information Processing 80, Proceedings of 8th IFIP World Computer Congress, 1980, pp. 457-462.
In this paper, the issue of reconstruction in a distributed database is analyzed. Methods are presented which allow reconstruction to be based on purely local checkpoints.

DATE86 Date, C. An Introduction to Database Systems. Vol. 1 (1986). Addison-Wesley Publishing Company.
Chapter 18 of this text is concerned with recovery and concurrency issues. An overview of recovery concepts is given, breaking recovery problems down by transaction, system, and media failures.

DAVI84 Davidson, S. "Optimism and Consistency in Partioned Distributed Database Systems." ACM Transactions on Database Systems, Vol. 9, No. 3 (September 1984), pp. 456-481.
A protocol for transaction processing during partition failures is presented which guarantees mutual consistency between copies of data-items after repair is completed. The protocol is "optimistic" in that transactions are processed without restrictions during failure; conflicts are then detected at repair time using a precedence graph, and are resolved by backing out transactions according to some back-out strategy.

DBRC84 "DBRC and Datasharing for the CICS/VS User." IBM Document, Information Systems Center--Santa Teresa (February 1984).
This publication provides information on some of the IMS/VS features being supported by CICS/OS/VS Release 1.6.1. Included is data sharing DBRC for recovery control and IRLM as a lock manager.

ESWA76 Eswaran, K.P., et al. "The Notions of Consistency and Predicate Locks in a Database System." Communications of the ACM, Vol. 10, No. 11 (November 1976), pp. 624-633.
This paper defines the concepts of transaction consistency. It shows that a transaction cannot request new locks after releasing a lock. Then it is argued that a transaction needs to lock a logical rather than a physical subset of the database. These subsets may be specified by predicates.

FINK83 Finkelstein, S., Mohan, C., Strong, R. "Method for Distributed Transaction Commit and Recovery Using Byzantine Agreement Within Clusters of Processors." Operating Systems Review, Vol. 19, No. 3 (July 1985), pp. 29-41.
The Byzantine Agreement is applied in the second phase of a distributed commit algorithm. It presents tradeoffs at commit time, but speeds up recovery after a failure.

FISH77 Fisher, P.S. and Maryanski, F.J. "Rollback and Recovery in Distributed Database Management Systems." ACM Proceedings of the Annual Conference, October 17-19, 1977, pp. 33-38.
Recovery mechanisms of single machine database management systems are discusssed. The problem of recovery and a distributed database system is also described. Recovery alternatives are analyzed referring to a distributed system.

GALT79 Galtieri, C. et al. "Notes on Distributed Databases." IBM Research Report, IBM Research Lab, San Jose, California (July 1979).
This report discusses replicated data, update strategies, data consistency, distributed transaction management issues, recovery facilities, and transaction recovery methods for distributed systems.

GARC84 Garcia-Molina, H., Pittelli, F. "Is Byzantine Agreement Useful in a Distributed Database?" Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (1984), pp. 61-68.

This paper studies the Byzantine agreement and its application to the distributed database environment. The main use presented involves distributing input transactions to a replicated database structure.

GEND82 Gendry, R. "Information Management System/Virtual Storage (IMS/VS) Version 1, Release 3: Logging Planning Guide." IBM Technical Bulletin, Dallas Systems Center, G320-5920-0 (November 1982).

This document discusses a new architecture in IMS/VS 1.3 for logging and enhanced usage of the DBRC functions.

GRAH84 Graham, M., Griffeth, N., Smith-Thomas, B. "Reliable Scheduling of Database Transactions for Unreliable Systems." Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (1984), pp. 300-309.

Five different transaction scheduling policies are described and compared--optimism, pessimism, realism, deferred writing, and paranoia.

GRAY78 Gray, J. "Notes on Database Operating Systems." IBM Research Report, IBM Research Lab, San Jose, California (February 1978), pp. 85-111.

After an overview on database management and data communications, the report focuses on the problems of locking and recovery in a transaction environment. A detailed and unified design is presented and compared with alternative approaches.

GRAY80 Gray, J., Lindsay, B., Price, T., Putzolu, F., Tralger, I. "Database Recovery Using Write Ahead Log Protocol." IBM Technical Disclosure Bulletin, Vol. 22, No. 8A (January 1980), p. 3362.

A method for database recovery using a write ahead log (WAL) protocol in IMS which avoids logging the transaction UNDO activity is described.

GRAY81 Gray, J. "The Transaction Concept: Virtues and Limitations." IEEE Very Large Data Bases: Seventh International Conference (1981), pp. 144-154.

This paper restates the transaction concept and attempts to put time-domain addressing and logging approaches in perspective.

HAER79 Haerder, T. and Reuter, A. "Optimization of Logging and Recovery in a Database System." Proceedings IFIP Working Conference on Data Base Architecture. G. Bracchi and G. Nijssen, Eds., North-Holland Publishing Company: New York (1979) pp. 151-168.

A special log strategy based on an indirect mapping scheme from logical pages to slots on disk is introduced. In case of a system failure, the UNDO-processing of incomplete transactions is performed more efficiently, while REDO-operations of completed transactions are avoided.

HAER83 Haerder, T. and Reuter, A. "Principles of Transaction Oriented Database Recovery." ACM Computing Surveys, Vol. 15, No. 4 (December 1983), pp. 287-317.

In this paper, transactions and recovery actions are summarized. The concept of a mapping hierarchy for database updates is covered. Crash recovery is discussed by examples and evaluation of logging and recovery concepts.

HAMM78 Hammer, M., Shipman, D. "An Overview of Reliability Mechanisms for a Distributed Data Base System." Tutorial: Centralized on Distributed Data Base Systems (1978), pp. 645-647.

Goals and reliability techniques used in the SDD-1 recovery mechanism are presented.

HAMM80 Hammer, M. and Shipman, D. "Reliability Mechanisms for SDD-1: A System for Distributed Databases." ACM Transactions on Database Systems, Vol. 5, No. 1 (December 1980), pp. 431-466.

Mechanisms to create a reliable network that guarantees the eventual delivery of messages and the broadcast of messages to a number of sites are discussed. This approach is used to support the redundant update mechanisms of SDD-1, a distributed database system being developed by the Computer Corporation of America.

HOSS83 Hosseini, S., Kuhl, J., Reddy, S. "An Integrated Approach to Error Recovery in Distributed Computing Systems." IEEE 1983 Fault-Tolerant Computing (June 1983), pp. 56-63.

A distributed fault-diagnosis algorithm, a technique for logical isolation of failed facilities, and a backward error recovery procedure are combined into an overall algorithm to insure integrity in a distributed system.

IMS86a "IMS/VS Version 1 Data Base Recovery Control: Guide and Reference." IBM Program Product, SH35-0027-3, Copyright by IBM (June 1986), Fifth Edition.

This publication describes the facilities of IMS/VS DBRC. It provides a detailed description of the three environments of DBRC: log control, recovery control, and share control.

IMS86b "IMS/VS Data Base Recovery Control Feature: General Information." IBM Program Product, GH35-0010-2, Third Edition (September 1986), Copyright by IBM.
This document contains general information about the functional characteristics of the DBRC feature of IMS/VS and the responsibilities involved in installing and operating the feature.

IYER86 Iyer, B., Lee, Y., Yu, P. "Transaction Recovery in Distributed DB/DC Systems: A Progressive Approach." IEEE Fifth Symposium on Reliability in Distributed Software and Database Systems (1986), pp. 207-214.
A progressive approach to transaction recovery in distributed systems is discussed. The concept is based on tracking a transaction's progress through subsystems. Issues arising from applying progressive recovery are discussed with respect to logging, shared input/output queues among subsystems, and backup subsystems.

KAIS86 Kaiser, J., Kroger, R., Nett, E. "Providing Recoverability in a Transaction Oriented Distributed Operating System." IEEE Distributed Computing Systems (1986), pp. 590-596.
A recovery strategy is discussed which incorporates recovery graphs to define dependencies among tranactions. Algorithms are presented to illustrate how recovery lines are used for restoring database consistency.

KANT83a Kant, K. "A Global Checkpointing Model for Error Recovery." AFIPS Conference Proceedings, Vol. 52, 1983 National Computer Conference, pp. 81-88.
This paper describes an error recovery mechanism which combines the traditional global-checkpointing mechanism with the recovery-block concept. It involves smaller overhead in case of moderate to high process interaction.

KANT83b Kant, K. "Efficient Local Checkpointing for Software Fault Tolerance." Operating Systems Review, Vol. 17, No. 2 (April 1983), pp. 11-13.
Checkpointing methods are compared in terms of reducing the time and storage overhead required.

KATZ84a Katz, R.H. and Weiss, S. "Recovery of In-Memory Data Structures for Interactive Update Applications." IEEE 1984 Compcon Spring, pp. 335-338.
An operational log approach is presented which can deal with unformatted data and complex data structures. Performance problems which can arise at restart and checkpointing are discussed.

KATZ84b Katz, R., Lehman, T. "Database Support for Versions and Alternatives of Large Design Files." IEEE Transactions on Software Engineering, Vol. SE-10, No. 2 (March 1984), pp. 191-200.
A B-tree based storage structure is used to implement versions in hopes of keeping disk requirements small. Previous approaches, implementation details, and an evaluation are included.

KAUN84 Kaunitz, J. and Van Ekert, L. "Audit Trail Compaction for Database Recovery." Communications of the ACM, Vol. 27, No. 7 (July 1984). pp. 678-682.
Total elapsed recovery time from disk-based database corruption can be shortened by reprocessing the audit trail off line and thereby avoiding excessive resource utilization penalties. Using a bit map, the audit trail is compacted by eliminating irrelevant or superseded records. The compacted trail is then partitioned and the partitions are processed in parallel.

KAUN85 Kaunitz, J. "Database Backup and Recovery in Transaction Driven Information Systems." IEEE 1985 First International Conference on Supercomputing Systems (1985). pp. 265-272.
The characteristics of transaction driven systems are examined and the types of failures which may occur are categorized. The paper then focuses on media recovery and sets relevant recovery objectives. A technique based on incremental dumps is described in detail.

KEEN82 Keene, W. "Information Management System/Virtual Storage (IMS/VS), Version 1, Release 2, Data Base Recovery Control (DBRC) and Data Sharing User's Guide." IBM Technical Bulletin, Dallas Systems Center, #G320-5911-0 (August 1982).
This document contains an overview of IBM's Data Base Recovery Control product and the concept of datasharing. The functional operations of DBRC are described, including the different types of database errors, batch backout facility, image copies, change accumulation, and DBRC tips.

KING81 King, J.M. Evaluating Data Base Management Systems. Van Nostrand Reinhold Company, 1981, pp. 169-179.
Different recovery approaches are described. Features included are restore/rerun recovery, forward recovery, backward recovery, write ahead journal records and partial recovery.

KOHL81 Kohler, W.H. "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems." ACM Computer Surveys, Vol. 13, No. 2 (June 1981), pp. 149-183.
The problems of synchronizing access to shared objects while allowing concurrency and recovery of objects are introduced. The proposed solution techniques are discussed.

KOST84 Kostovetsky, A. "Frequency Decomposition Technique for Large Archivable Databases." International Journal of Computer and Information Sciences, Vol. 13, No. 4 (1984), pp. 237-249.
Modern large databases intended for use in a highly interactive and responsive environment face serious performance and availability problems caused by prolonging archiving and backup procedures. The suggested technique solves the problem by splitting the original database into a collection of smaller, independently archivable ones.

KUNG81 Kung, H.T., Robinson, J.R. "On Optimistic Methods for Concurrency Control." ACM Transactions on Database Systems, Vol. 6, No. 2 (June 1981), pp. 213-226.
Two families of nonlocking concurrency controls are presented. The methods used rely on transaction backup as a control mechanism, "hoping" that conflicts beween transactions will not occur.

LAND80 Landes, O. and Menasce, D. "On the Design of a Reliable Storage Component for Distributed Database Management Systems." IEEE Very Large Data Base: Sixth International Conference (1980), pp. 365-375.
A design of reliable storage component of a distributed database management system is presented in this paper. This design builds on several existing ideas and adds some other crash recovery strategies.

LEE78 Lee, P., Randall, B., Treleaven, P. "Reliability Issues in Computing Systems Design." ACM Computing Surveys, Vol. 10, No. 2 (June 1978), pp. 123-165.
This paper surveys the various problems involved in achieving very high reliability from complex computing systems and discusses the relationship between system structuring techniques and techniques of fault tolerance.

LEG079 LeGore, L.B. "Smoothing Data Base Recovery." Datamation (January 1979), pp. 177-180.
This article stresses the importance of database log and dump tape integrity. A set of guidelines is provided for recovery systems. The Central Management Information Center (CMIC) developed a recovery system which was used for examples.

LOHM76 Lohman, G. and Severance, D. "Differential Files: Their Application to the Maintenance of Large Databases." ACM Transactions on Database Systems. Vol. 1, No. 3 (September 1976), pp. 256-267.
This paper describes and analyzes the concept of differential files, along with the advantages of their use.

LORI77 Lorie, R.A. "Physical Integrity in a Large Segmented Database." ACM Transactions on Database Systems, Vol. 2, No. 1 (March 1977), pp. 91-104.
This paper is primarily concerned with damage to database storage. A recovery scheme is first proposed for system failure which causes the contents of main storage to be lost. Another section proposes a facility for protection against damage to the auxiliary storage.

MARC84 March, S.T. and Scudder, G.D. "On the Selection of Efficient Record Segmentations and Backup Strategies for Large Shared Databases." ACM Transactions on Database Systems, Vol. 9, No. 3 (September 1984), pp. 409-438.
The impacts of record segmentation, backup and recovery strategies on the process of database design are analyzed. A combined record segmentation/backup and recovery procedure is presented and an application of the procedure is discussed.

MCDE81 McDermid, J. "Checkpointing and Error Recovery in Distributed Systems." IEEE Distributed Computing Systems (1981), pp. 271-281.
Checkpoint usefulness is described. Strategies are compared and protocols are given to account for different checkpoint inconsistencies. Algorithms for automatically generating new checkpoints and discarding old ones are given.

MCGE77 McGee, W. "The Information Management System IMS/VS." IBM Systems Journal, Vol. 16, No. 2 (1977), pp. 84-168.
A complete description of the IMS/VS functions up to 1977 are presented in this series of papers. Included is database structure, application program options, evolution, and recovery.

MEAC84 Meacock, G. "Data Recovery for CICS Using Time Stamps." IBM Technical Disclosure Bulletin, Vol. 27, No. 5 (October 1984), pp. 3167-3169.
This paper shows that, in the IBM database, data communica-tion program product Customer Information Control System (CICS) queues may occupy more than one control interval with data recovery using a sequential time stamp.

MUNZ80 Munz, R. "Transaction Management in the Distributed Database System VDN." Information Processing 80, Proceedings of the 8th IFIP World Computer Congress (1980), pp. 481-486.
VDN is a distributed database system being developed at the Technical University, Berlin. This paper describes the realization of the concept of update transactions in a distributed environment. The robustness of the VDN transaction management is outlined.

RAND78 Randall, B. "Reliable Computing Systems." In Lecture Notes in Computer Science: Operating Systems. eds. (Bayer, Graham, Seegmuller) New York: Springer-Verlag (1978), pp. 287-388.
The following recovery schemes are discussed: backward error recovery, forward error recovery, multi-level error recovery. Examples and problems with those techniques are given.

RECO76 "Recovery in Database Systems." EDP Analyzer (November 1976), pp. 1-11.
The author discusses the types of failures, elements of a recovery system and complexities in recovery. Specific examples of recovery systems in organizations are given.

RIES82 Ries, D. "Continued and Correct Operation of Distributed Databases in the Presence of Network Partitions." IEEE Computer Software and Applications Conference (COMPSAC), 1982, p. 452.
Two approaches, log transformations and datapatch, are described which allow updates to continue during a network partition and restore the consistency of the database after communications have been repaired.

REUT79 Reuter, A. "Minimizing the I/O Operations for Undo-Logging in Database Systems." IEEE Very Large Data Bases: Fifth International Conference (1979), pp. 164-172.
A special Undo-Log algorithm combining the advantages of update-in-place strategies with the shadow page concept is introduced. It is to support applications demanding high rates of updating transactions as well as fast Undo-recovery.

REUT84 Reuter, A. "Performance Analysis of Recovery Techniques." ACM Transactions on Database Systems, Vol. 9, No. 4 (December 1984), pp. 526-559.
Various logging and recovery techniques for centralized transaction-centered database systems and their performance aspects are described and discussed.

SALU84 Saluja, K. and Upadhyaya, J. "A Hardware Supported General Rollback Technique." IEEE 1984 Fault-Tolerant Computing (June 1984), pp. 409-414.
This paper describes the watchdog processor. The hardware involved in merging the watchdog processor with the rollback technique for improving system reliability is presented.

SCHL76 Schlageter, G. "The Problem of Lock by Value in Large Data Bases." The Computer Journal, Vol. 19, No. 1 (February 1976), pp. 17-20.
The concept of lock by value in multiple-access database systems is developed, and a lock system to support this approach is proposed. The problems of implementing lock by value are discussed.

SCHW83 Schwarz, P., Spector, A. "Transactions: A Construct for Reliable Distributed Computing." Operating Systems Review, Vol. 17, No. 2 (April 1983), pp. 18-31.
Synchronization, deadlock, recovery, and communication issues with respect to distributed transaction processing are presented.

SKEE81 Skeen, D. "Nonblocking Commit Protocols." ACM-SIGMOD International Conference on Management of Data (1981), pp. 133-142.
The properties of nonblocking protocols are presented. A centralized nonblocking protocol and distributed nonblocked protocol are discussed for used during site failures.

SKEE83 Skeen, D., Stonebraker, M. "A Formal Model of Crash Recovery in a Distributed System." IEEE Transactions on Software Engineering, Vol. SE-9, No. 3 (May 1983), pp. 219-227.
Site failures and protocols are examined for situations which do and do not cause partitions. Commit protocols based on nondeterministic finite state machines are described.

SKEE85 Skeen, D. "Determining the Last Process to Fail." ACM Transactions on Computer Systems, Vol. 3, No. 1 (February 1985), pp. 15-30.
When all processes involved in a transaction fail, the recovery manager must determine the last process to fail. This paper discusses procedures and data which are needed in order to determine the last process that failed.

STON79 Stonebraker, M. "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES." IEEE Transactions on Software Engineering, Vol. SE-5, No. 3 (May 1979), pp. 188-194.
Distributed INGRES is a proposed distributed version of a relational database system designed and implemented by researchers at the University of California, Berkeley. This paper describes the design philosophy and presents rough outlines of the algorithms proposed for concurrency control, consistency of multiple copies of data, and system crash recovery.

STRI82 Strickland, J.P., Uhrowczik, P.P., Watts, V.L. "IMS/VS: An Evolving System." IBM System Journal, Vol. 21, No. 4 (1982), pp. 490-510.
Advances in the IMS/VS system are discussed, including data sharing, system logging, and database recovery control.

STRO85 Strom, R. and Yemini, S. "Optimistic Recovery in Distributed Systems." ACM Transactions on Computer Systems, Vol. 3, No. 3 (August 1985), pp. 204-226.
The replacement of synchronization by causal dependency tracking is proposed with the claim of better throughput and response whenever failures are infrequent.

TANT84 Tantawi, A. and Ruschitzka, M. "Performance Analysis of Checkpointing Strategies." ACM Transactions on Computer Systems, Vol. 2, No. 2 (May 1984), pp. 123-144.
This paper calculates checkpointing intervals based on the reprocessing time and failure distribution. A checkpointing strategy called equicost is proposed and compared to other methods.

VERH78 Verhofstad, J. "Recovery Techniques for Database Systems." Computing Surveys (June 1978), pp. 167-195.
A survey of techniques and tools used for recovery is given. Categorization of different recovery techniques and basic principles is presented, along with examples of their applications in existing systems.

WALT80 Walter, B. "Strategies for Handling Transactions in Distributed Data Base Systems During Recovery." IEEE Very Large Data Bases: Sixth International Conference (1980), pp. 384-389.

Strategies to handle blocked transactions due to failures in distributed database systems are discussed. An algorithm is presented which supports transaction handling in the case of media failures.

WIED83 Wiederhold, G. Database Design. McGraw-Hill Book Company (1983), pp. 555-576.

The idea of transaction management is analyzed in detail. Activity logging is discussed to protect the reliability of the database. A sequence of actions which could be employed when a system failure occurs is presented. Distributed systems are included in the discussion.

WULF75 Wulf, W.A. "Reliable Hardware/Software Architecture." IEEE Transactions on Software Engineering. Vol. SE-1, No. 2 (June 1975), pp. 233-240.

A strategy used to achieve reliability in a hardware/software system is discussed. The proposed approach is based on an extension of modular programming methodology to include dynamic error detection and recovery. Each module is responsible for maintaining the integrity of the abstraction it implements.

YELA82 Yelavich, B. "Recovery/Restart in Customer Information Control System/Operating System/Virtual Storage (CICS/OS/VS) Information Management System/Virtual Storage (IMS/VS) Data Base (DB) Data Language/One (DL/1) Environment." IBM Technical Bulletin, Dallas Systems Center, G320-5915-0 (July 1982).

This document focuses on recovery/restart in a CICS environment which uses DL/1 databases. Emphasis is placed on single transaction failure and system failure.