The Recovery Manager of the System R Database Manager

JIM GRAY

Tandem Computers, 19333 Vallco Parkway, Cupertino, Californıa 95014

PAUL McJONES

Xerox Corporatıon, 3333 Coyote Hıll Road, Palo Alto, Calıfornia 94304

MIKE BLASGEN, BRUCE LINDSAY, RAYMOND LORIE, TOM PRICE, FRANCO PUTZOLU, AND IRVING TRAIGER

IBM San Jose Research Laboratory, 5600 Cottle Road, San Jose, Calıfornıa 95193

The recovery subsystem of an experimental data management system is described and evaluated. The *transactıon* concept allows application programs to commit, abort, or partıally undo their effects. The *DO-UNDO-REDO protocol* allows new recoverable types and operations to be added to the recovery system Applıcation programs can record data ın the transaction log to facilitate application-specific recovery. Transaction undo and redo are based on records kept in a *transaction log*. The *checkpoint* mechanism is based on differential files (shadows). The recovery log is recorded on disk rather than tape.

Keywords and Phrases transactıons, database, recovery, reliability

CR Categorıes: 4.33

INTRODUCTION

Application Interface to System R

Making computers easier to use is the goal of most software. Database management systems, in particular, provide a programming interface to ease the task of writing electronic bookkeeping programs. The recovery manager of such a system in turn eases the task of writing fault-tolerant application programs.

System R [ASTR76] is a database system which supports the relational model of data. The SQL language [CHAM76] provides operators that manipulate the database. Typically, a user writes a PL/I or COBOL program which has imbedded SQL statements. A collection of such statements is required to make a consistent transformation of the database. To transfer funds from one account to another, for example, requires two SQL statements: one to debit the first account and one to credit the second account. In addition, the transaction probably records the transfer in a history file for later reporting and for auditing purposes. Figure 1 gives an example of such a program written in pseudo-PL/I.

The program effects a consistent transformation of the books of a hypothetical bank. Its actions are either to

- discover an error,
- accept the input message, and
- produce a failure message,

or to

- discover no errors,
- accept the input message,

CONTENTS

- debit the source account by AMOUNT,
- credit the destination account by AMOUNT,
- record the transaction in a history file, and
- produce a success message.

The programmer who writes such a program ensures its correctness by ensuring that it performs the desired transformation on both the database state and the outside world (via messages). The programmer and the user both want the execution to be

- *atomic:* either all actions are performed (the transaction has an effect) or the results of all actions are undone (the transaction has no effect);
- *durable:* once the transaction completes, its effects cannot be lost due to computer failure;
- *consistent:* the transaction occurs as though it had executed on a system which sequentially executes only one transaction at a time.

In order to state this intention, the SQL programmer brackets the transformations with the SQL statements, BEGIN_TRANSACTION to signal the beginning of the transaction and COMMIT_TRANSACTION to signal its completion. If the programmer wants to return to the beginning of the transaction, the command RESTORE_TRANSACTION will undo all actions since the issuance of the BEGIN_TRANSACTION command (see Figure 1).

The System R recovery manager supports these commands and guarantees an atomic, durable execution.

System R generally runs several transactions concurrently. The concurrency control mechanism of System R hides such concurrency from the programmer by a locking technique [ESWA76, GRAY78, NAUM78] and gives the appearance of a consistent system.

Structure of System R

System R consists of an external layer called the *Research Data System (RDS),* and a completely internal layer called the *Research Storage System (RSS)* (see Figure 2).

The external layer provides a relational data model, and operators thereon. It also provides catalog management, a data dictionary, authorization, and alternate views of data. The RDS is manipulated using the language SQL [CHAM76]. The SQL compiler maps SQL statements into sequences of RSS calls.

The RSS is a nonsymbolic record-at-a-time access method. It supports the notions of *file, record type, record instance, field within record, index* (B-tree associative and sequential access path), *parent-child set* (an access path supporting the operations PARENT, FIRST_CHILD, NEXT_SIBLING, PREVIOUS_SIBLING with direct pointers), and *cursor* (which navigates over access paths to locate

```
FUNDS_TRANSFER. PROCEDURE,
  $BEGIN_TRANSACTION;
  ON ERROR DO;                                          /* in case of error */
              $RESTORE_TRANSACTION,                     /* undo all work */
              GET INPUT MESSAGE;                        /* reacquire input */
              PUT MESSAGE ('TRANSFER FAILED');          /* report failure */
              GO TO COMMIT;
              END;
  GET INPUT MESSAGE;                                    /* get and parse input */
  EXTRACT ACCOUNT_DEBIT, ACCOUNT_CREDIT,
    AMOUNT FROM MESSAGE,
  $UPDATE ACCOUNTS                                      /* do debit */
                      SET BALANCE = BALANCE - AMOUNT
        WHERE ACCOUNTS. NUMBER = ACCOUNT_DEBIT;
  $UPDATE ACCOUNTS                                      /* do credit */
                      SET BALANCE = BALANCE + AMOUNT
        WHERE ACCOUNTS. NUMBER = ACCOUNT_CREDIT;
  $INSERT INTO HISTORY                                  /* keep audit trail */
      <DATE, MESSAGE>;
  PUT MESSAGE ('TRANSFER DONE');                        /* report success */
 COMMIT:                                                /* commit updates */
  $COMMIT_TRANSACTION
  END;                                                  /* end of program */
```

Figure 1. A simple PL/I-SQL program which transfers funds from one account to another.

Application Programs in PL/I or COBOL, plus SQL

Research Data System (RDS)
- Supports the relational data model
- Supports the relational language SQL
- Does naming and authorization
- Compiles SQL statements into RSS call sequences

Research Storage System (RSS)
- Provides nonsymbolic record-at-a-time database access
- Maps records onto operating system files
- Provides transaction concept (recovery and locking)

Operating System
- Provides file system to manage disks
- Provides I/O system to manage terminals
- Provides process structure (multiprogramming)

Hardware

Figure 2. System R consists of two layers above the operating system. The RSS provides the transaction concept, recovery notions, and a record-at-a-time data access method. The RDS accepts application PL/I or COBOL programs containing SQL statements. It translates them into COBOL or PL/I programs plus subroutines which represent the compilation of the SQL statements into RSS calls.

records). Unfortunately, these objects have the nonstandard names "segment," "relation," "tuple," "field," "image," "link," and "scan" in the System R documentation. The former, more standard, names are used here. RSS provides actions to create instances of these objects and to retrieve, modify, and delete them.

The RSS support of data is substantially more sophisticated than that normally found in an access method; it supports variable-length fields, indices on multiple fields, multiple record types per file, interfile and intrafile sets, physical clustering of records by attribute, and a catalog describing the data, which is kept as a file which may be manipulated like any other data.

Another major contribution of the RSS is its support of the notion of *transaction,* a unit of recovery consisting of an application-specified sequence of RSS actions. An application declares the start of a transaction by issuing a BEGIN action. Thereafter all RSS actions by that application are within the scope of that transaction until the application issues a COMMIT or an ABORT action. The RSS assumes all responsibility for running concurrent transactions and for assuring that each transaction sees a consistent view of the database. The RSS is also responsible for recovering the data to their most recent consistent state in the event of transaction, action, system, or media failure or a user request to cancel the transaction.

A final component of System R is the operating system. System R runs under the VM/370 [GRAY75] and the MVS operating system on IBM S/370 processors. The System R recovery manager is also part of the SQL/DS product running on DOS/CICS. The operating system provides processes, a simple file system, and terminal management.

System R allocates an operating system process for each user to run both the user's application program and the System R database manager. Application programs are written in a conventional programming language (e.g., COBOL or PL/I) augmented with the SQL language. A SQL preprocessor maps the SQL statements to sequences of RSS calls. Typically, a single application program or group of programs (main plus subroutines) constitute a transaction. In this paper we ignore the RDS and assume that application programs, like those produced by the SQL compiler, consist of conventional programs which invoke sequences of RSS operations.

Model of Failures

The recovery manager eases the task of writing *fault-tolerant* programs. It does so by the careful use of redundancy. Choosing appropriate redundancy requires a quantitative model of system failures.

In our experience about 97 percent of all transactions execute successfully. Of the remainder, almost all fail because of incorrect user input or because of user cancellation. Occasionally (much less than 1 percent) transactions are aborted by the system as a result of some overload such as deadlock. In a typical system running one transaction per second, transaction undo occurs about twice a minute. Because of its frequency, transaction undo must run about as fast as forward processing of transactions.

Every few days the system *restarts* (following a crash). Almost all crashes are due to hardware or operating system failures, although System R also initiates crash and restart whenever it detects damage to its data structures. The state of primary memory is lost after a crash. We assume that the state of the disks (secondary and tertiary storage) is preserved across crashes, so at restart the most recently committed state is reconstructed from the surviving disk state by referencing a log of recent activity to restore the work of committed and aborted transactions. This process completes within a matter of seconds or minutes.

Table 1. Frequency and Recovery Time of Failures

Recovery manager trade-offs		
Fault	Frequency	Recovery time
Transaction abort	Several per minute	Milliseconds
System restart	Several per month	Seconds
Media failure	Several per year	Minutes

Occasionally, the integrity of the disk state will be lost at restart. This may be caused by hardware failure (disk head crash or disk dropped on the floor) or by software failure (bad data written on a disk page by System R or other program). Such events are called *media failures* and initiate a reconstruction of the current state from an archive version (old and undamaged version of the system state) plus a log of activity since that time. This procedure is invoked once or twice a year and is expected to complete within an hour.

If all these recovery procedures fail, the user will have lost data owing to an *unrecoverable failure*. We have very limited statistics on unrecoverable failures. The current release of System R has experienced about 25 years of service in a variety of installations, and to our knowledge almost all unrecoverable failures have resulted from operations errors (e.g., failure to make archive dumps) or from bugs in the operating system utility for dumping and restoring disks. The fact that the archive mechanism is only a minor source of unrecoverable failure probably indicates that it is appropriately designed. Table 1 summarizes this discussion.

If the archive mechanism fails once every hundred years of operation, and if there are 10,000 installations of System R, then it will fail someone once a month. From this perspective, it might be underdesigned.

We assume that System R, the operating system, the microcode, and the hardware all have bugs in them. However, each of these systems does quite a bit of checking of its data structures (defensive programming). We postulate that these errors are detected and that the system crashes before the data are seriously corrupted. If this assumption is incorrect, then the situation is treated as a media failure. This attitude assumes that the archive and log mechanism are *very* reliable and have failure modes independent of the other parts of the system.

Some commercial systems are much more demanding. They run hundreds of transactions per second, and because they have hundreds of disks, they see disk failures hundreds of times as frequently as typical users of System R (once a week rather than once a year). They also cannot tolerate downtimes exceeding a few minutes. Although the concepts presented in this paper are applicable to such systems, much more redundancy is needed to meet such demands (e.g., duplexed processors and disks, and utilities which can recover small parts of the database without having to recover it all every time). The recovery manager presented here is a textbook one, whose basic facilities are only a subset of those provided by more sophisticated systems.

The transaction model is an unrealizable ideal. At best, careful use of redundancy minimizes the probability of unrecoverable failures and consequent loss of committed updates. Redundant copies are designed to have independent failure modes, making it unlikely that all records will be lost at once. However, Murphy's law ensures that all recovery techniques will sometimes fail. As seen below, however, System R can tolerate any single failure and can often tolerate multiple failures.

1. DESCRIPTION OF SYSTEM R RECOVERY MANAGER

1.1 What is a Transaction?

The RSS provides *actions* on the objects it implements. These actions include operations to create, destroy, manipulate, retrieve, and modify RSS objects (files, record types, record instances, indices, sets, and cursors). Each RSS action is *atomic*—it either happens or has no effect—and *consistent*—if any two actions relate to the same object, they appear to execute in some serial order. These two qualities are ensured by (1) undoing the partial effects of any actions which fail and (2) locking necessary RSS resources for the duration of the action.

RSS actions are rather primitive. In general, functions like "hire an employee" or "make a deposit in an account" require several actions. The user, in mapping abstractions like "employee" or "account" into such a system, must combine several actions into an *atomic transaction.* The classic example of an atomic transaction is a funds transfer which debits one account, credits another, writes an activity record, and does some terminal input or output. The user of such a transaction wants it to be an all-or-nothing affair, in that he does not want only some of the actions to have occurred. If the transaction is correctly implemented, it looks and acts atomic.

In a multiuser environment, transactions take on the additional attribute that any two transactions concurrently operating on common objects appear to run serially (i.e., as though there were no concurrency). This property is called *consistency* and is handled by the RSS lock subsystem [ESWA76, GRAY76, GRAY78, NAUM78].

The application declares a sequence of actions to be a transaction by beginning the sequence with a BEGIN action and ending it with a COMMIT action. All intervening actions by that application (be it one or several processes) are considered to be parts of a single recovery unit. If the application gets into trouble, it may issue the ABORT action which undoes all actions in the transaction. Further, the system may unilaterally abort in-progress transactions in case of an authorization violation, resource limit, deadlock, system shutdown, or crash. Figure 3 shows the three possible outcomes—commit, abort, or system abortion—of a transaction, and Figure 4 shows the outcomes of five sample transactions in the event of a system crash.

If a transaction either aborts or is aborted, the system must undo all actions of that transaction. Once a transaction commits, however, its updates and messages to

```
BEGIN     BEGIN    BEGIN
READ      READ     READ
WRITE     WRITE    WRITE
READ      READ     READ
  .         .        .
  .         .        .
WRITE     ABORT    ⇐SYSTEM ABORTS
COMMIT               TRANSACTION
```

Figure 3. The three possible destimes of a transaction. commits, aborts, or is aborted.

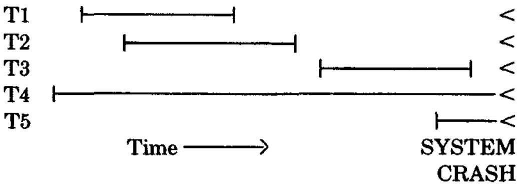

Figure 4. Five transactions. The effects of actions by transactions T1, T2, and T3 will survive a system crash because they have committed. This is called durability. But the effects of transactions T4 and T5 will be undone because they were in progress at the time of the crash (had not yet committed).

the external world must persist—its effects must be *durable*. The system will "remember" the results of the transaction despite any subsequent malfunction. Once the system commits to "open the cash drawer" or "retract the reactor rods," it will honor that commitment. The only way to undo the effect of a committed transaction is to run a new transaction which compensates for these effects.

1.2 Transaction Save Points

The RSS defines the additional notion of *transaction save point*. A save point is a firewall which allows transaction undo to stop short of undoing the entire transaction. Should a transaction get into trouble (e.g., deadlock or authority violation), it may be sufficient to back up only as far as an intermediate save point. Each save point is numbered, with the beginning of a transaction being save point 1. The application program declares a save point by issuing a SAVE action specifying a save point record to be entered in the log. This record may be retrieved if and when the transaction returns to the corresponding save point.

```
TICKET AGENT          APPLICATION PROGRAM
   input message  ⇒  BEGIN
                     SAVE (state)
                     ⟨actions to reserve first hop⟩
                     SAVE (state)
                  ⇐  new screen
        next hop  ⇒  SAVE (hop)
                     ⟨actions to reserve next hop⟩
                   .
                   .
        last hop  ⇒  SAVE (hop)
                     ⟨actions to reserve last hop⟩
  prmted ticket   ⇐  COMMIT ⟨reservation⟩
```

Figure 5. A multihop airlines reservation transaction using save points. If the application program or ticket agent detects an error, the transaction can undo to a previous save point and continue forward from there

Figure 5 illustrates the use of save points. It describes a conversational transaction making a multihop airline reservation involving several ticket agent interactions—one per hop. The application program establishes a save point before and after each interaction, with the save point data including the current state of the program variables and the ticket agent screen. If the ticket agent makes an error or if a flight is unavailable, the agent or program can back up to the most recent mutually acceptable save point, thereby minimizing the agent's retyping. Once the entire ticket is composed, the transaction commits the database changes and prints the ticket. The program can request a return to save point N by issuing the UNDO N action. Of course all the save points may be washed away by a system restart or serious deadlock, but most error situations can be resolved with a minimal loss of the ticket agent's work.

The System R save point facility is in contrast to most systems in which each message causes the updates of the transaction to be committed. In such systems either the agent must manually delete previous steps of the transaction when something goes wrong, or the application program must implement recovery by keeping an undo log or deferring all updates until the last step. Such application program recovery schemes complicate the program and may work incorrectly because locks are not held between steps (e.g., the lost update problem described in GRAY76).

Save points are used by the RDS to implement certain complex operations. Some RDS operations require many RSS

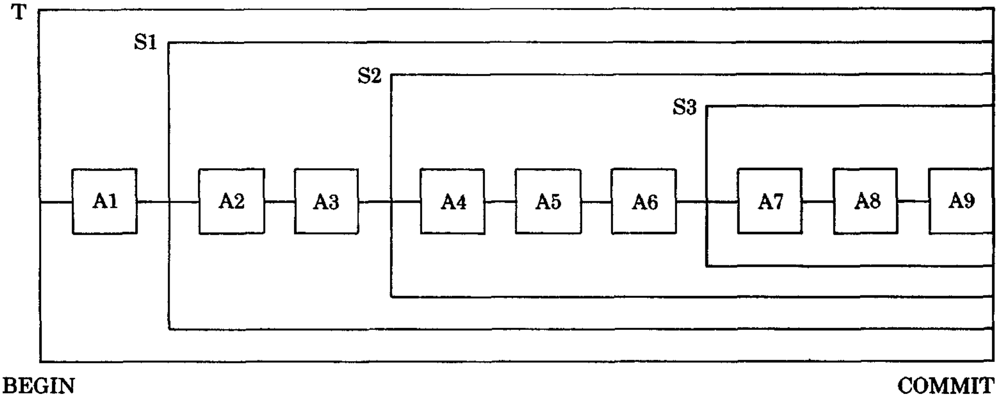

Figure 6. An example of nested transactions using save points. The transaction T consists of RSS actions A1, ..., A9. Actions A2, A4, and A7 defined save points S1, S2, and S3 which become nested subtransactions of T. These are probably RDS actions which consist of several RSS actions. Each RSS action is a minitransaction on the RSS state. If the RSS action gets into trouble, it backs up to the beginning of the action and retries. If that fails, undo propagates to the nearest save point that can resolve the issue. This is the spheres of control notation of DAVI73.

operations, and the RDS guarantees that each SQL statement is atomic. The RDS supports atomic SQL statements by beginning each such complex operation with a save point and backing up to this save point if the RSS or RDS fails at some point during the operation. Figure 6 illustrates the use of save points.

1.3 Summary

To summarize, the RSS recovery manager provides the following actions:

- BEGIN designates the beginning of a transaction.
- SAVE designates a firewall within the transaction. If an incomplete transaction is backed up, undo may stop at such a point rather than undoing the entire transaction.
- READ_SAVE returns the data saved in the log by the application at a designated save point.
- UNDO undoes the effects of a transaction to an earlier save point.
- ABORT undoes all effects of a transaction (equivalent to UNDO 0).
- COMMIT signals successful completion of transaction and causes updates to be committed.

Using these primitives, the RDS and application programs using the RDS can construct groups of actions which are atomic and durable.

This model of recovery is a subset of the recovery model formulated by Davies and Bjork [BJOR73, DAVI73]. Unlike their model, System R transactions have no parallelism within a transaction (i.e., if multiple nodes of a network are needed to execute a single transaction, only one node executes at a time). Further, System R allows only a limited form of transaction nesting via the use of save points (each save point may be viewed as the start of an internal transaction). These limitations stem from our inability to find an acceptable implementation for the more general model.

2. IMPLEMENTATION OF SYSTEM R RECOVERY

2.1 Files, Versions, and Shadows

All persistent System R data are stored in files. A user may define any number of files. A file, for our purposes, is a paged linear space of up to 68 billion (2^{36}) bytes, which has been dynamically allocated on disk in units of 4096-byte pages. A buffer manager maps all the files into a virtual memory buffer pool shared by all System R users. The buffer manager uses a "Least Recently Used" (LRU) algorithm to regulate occupancy of pages in the pool. The buffer pool

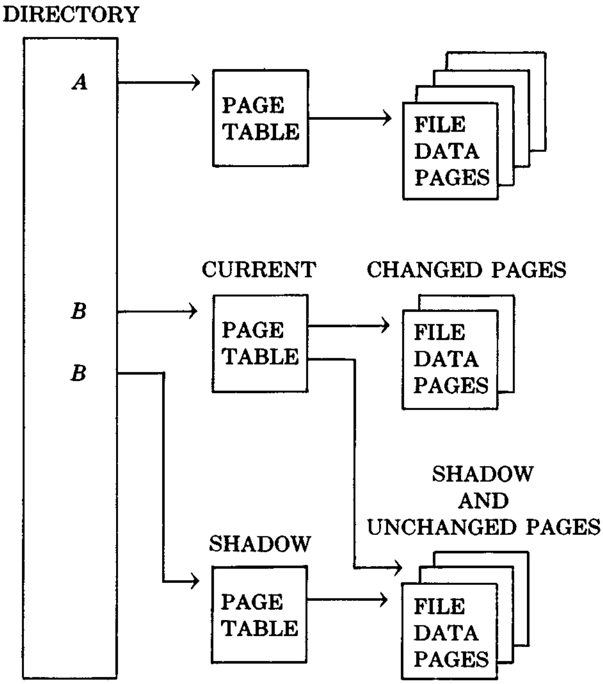

Figure 7. The directory stucture for nonshadowed and shadowed files. File *A* is not shadowed. File *B* is shadowed and has two directory entries, a current version and a shadow version.

is volatile and presumed not to survive system restart.

Each file carries a particular recovery protocol and corresponding overhead of recovery. Files are dichotomized as *shadowed* and *nonshadowed.* Nonshadowed files have no automatic recovery. The user is responsible for making and storing redundant copies of these files. System R simply updates nonshadowed file pages in the buffer pool. Changes to nonshadowed files are recorded on disk when the pages are removed from the buffer pool (by the LRU algorithm) and when the file is saved or closed.

By contrast, the RSS maintains two online versions of shadowed files, a *shadow version* and a *current version.* RSS actions affect only the current version of a file and never alter the shadow version (except for file save and restore commands). The current version of a file can be SAVEd as the shadow version, thereby making the recent updates to the file permanent; the current version can also be RESTOREd to the shadow version, thereby "undoing" all recent updates to the file (see Figure 7). If data are spread across several files, it is desirable to save or restore all the files "at once." Therefore file save or restore can apply to sets of shadowed files.

Although the current version of a file does not survive restart, because recent updates to the file may still reside in the buffer pool, the shadow version of a file does. Hence at RSS restart (i.e., after a crash or shutdown) all nonshadowed files have their values as of the system crash (modulo updates to central memory which were not written to disk) and all shadowed files are reset to their shadow versions. As discussed below, starting from this shadow state, the log is used to remove the effects of aborted transactions and to restore the effects of committed transactions.

The current and shadow versions of a file are implemented in a particularly efficient manner. When a shadow page is updated in the buffer pool for the first time, a new disk page frame is assigned to it. Thereafter, when that page is written from the buffer pool or read into the buffer pool, the new frame is used (the shadow is never updated). Saving a file consists of writing to disk all altered pages of the file currently in the buffer pool and then writing to disk the new page table, and freeing superseded shadow pages. Restoring a file is achieved

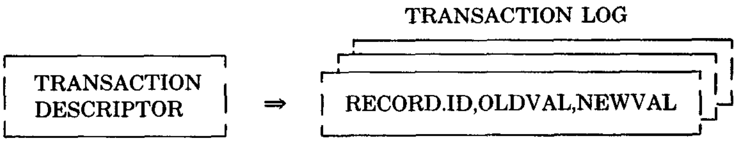

Figure 8. A transaction log is the sequence of changes made by this transaction

by discarding pages of that file in the buffer pool, freeing all the new disk pages of that file, and returning to the (old) shadow page table. The paper by Lorie [LORI77] describes the implementation in greater detail.

2.2 Logs and the DO, UNDO, REDO Protocol

The shadow-version/current-version dichotomy has strong ties to the old-master/new-master dichotomy common to most batch EDP systems in which, if a run fails, the old master is used for a successive attempt; if the run succeeds, the new master becomes the old master. Unhappily, this technique does not seem to generalize to concurrent transactions on a shared file. If several transactions concurrently alter a file, file save or restore is inappropriate because it commits or aborts the updates of *all* transactions to the file. It is desirable to be able to commit or undo updates on a per-transaction basis. Such a facility is required to support the COMMIT, ABORT, and UNDO actions, as well as to handle such problems as deadlock, system overload, and unexpected user-disconnect. Further, as shown in Figure 4, selective transaction back up and commit are required for system restart.

We were unable to architect a transaction mechanism based solely on shadows which supported multiple users and save points. Instead, the shadow mechanism is combined with an incremental *log* of all the actions a transaction performs. This log is used for transaction UNDO and REDO on shared files and is used in combination with shadows for system checkpoint and restart. Each RSS update action writes a log record giving the old and new value of the updated object. As Figure 8 shows, these records are aggregated by transaction and collected in a common system *log file* (which is optionally duplexed).

Table 2. Recovery Attributes of Files

	No shadow	Shadow
No log	Contents unpredictable after crash	Contents equal shadow after crash
Log	Not supported	All updates logged Transaction consistent after a crash

When a shadowed file is defined by a user, it is designated as *logged* or *not logged.* The RSS controls the saving and restoration of logged files and maintains a log of all updates to logged files. Users control the saving and restoration of nonlogged shadowed files. Nonshadowed files have none of the virtues or corresponding overhead of recovery (see Table 2).

In retrospect, we regret not supporting the LOG and NO SHADOW option. As explained in Section 3.8, the log makes shadows redundant, and the shadow mechanism is quite expensive for large files.

Each time a transaction modifies a logged file, a new record is appended to the log. Read actions need generate no log records, but 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. As seen below, most log writes do not require I/O and can be buffered in central memory.

Every RSS operation on a logged file must be implemented as a set of operations (see Figure 9):

- a DO operation, which does the action and also writes a log record sufficient to undo and redo the action;
- an UNDO operation, which undoes the action given the log record written by the DO action;
- a REDO operation, which redoes the action given the log record written by the DO action;

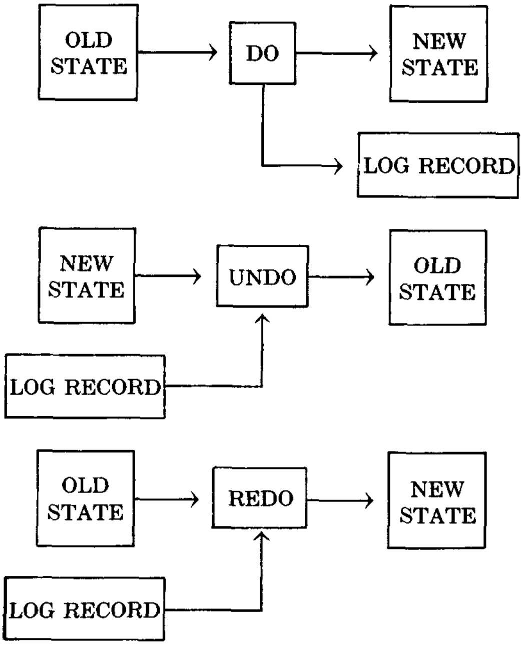

Figure 9. Three aspects of an actıon Actıon DO generates a new state and a log record. Action UNDO generates an old state from a new state and a log record Actıon REDO generates a new state from an old state and a log record.

- optionally, a DISPLAY operation which translates the log record into a human-readable format.

To give an example of an action and the log record it must write, consider the record update action. This action must record the following in the log:

- file name,
- record identifier,
- old record value,
- new record value.

The log subsystem augments this with the additional fields:

- transaction identifier,
- action identifier,
- timestamp,
- length of log record,
- pointer to previous log record of this transaction.

The UNDO operation restores the record to its old value and appropriately updates associated structures such as indices and storage management information. The REDO operation restores the record (and its associated structures) to its new value. The display operation returns a text string giving a symbolic display of the log record.

Once a log record is recorded, it cannot be updated. However the log manager provides a facility to open read cursors on the log which will traverse the system log or traverse the log of a particular transaction in either direction.

2.3 Commit Processing

The essential property of a transaction is that it ultimately commits, aborts, or is aborted and that once it commits, its updates persist, and once it aborts or is aborted, its updates are suppressed. Achieving this property is nontrivial. In order to ensure that a transaction's effects will survive restarts and media failures, the system must be able to redo committed transactions. System R ensures that uncommitted transactions can be undone and that committed transactions can be redone as follows:

(1) The transaction log is written to disk before the shadow database is replaced by the current database state.
(2) The transaction commit action writes a commit log record in the log buffer and then forces all the transaction's log records to disk (with the commit record being the last such record).

A transaction commits at the instant its commit record appears on disk. If the system crashes prior to that instant, the transaction will be aborted. Because the log is written before the database (item 1), System R can always undo any uncommitted updates which have migrated to disk. On the other hand, if the system crashes subsequent to the writing of the commit record to disk, then the transaction will be redone, from the shadow state, using the log records which were forced to disk by the commit. In the terminology of Gray [GRAY78], item 1 is the "write ahead log protocol."

2.4 Transaction UNDO

The logic of action UNDO is very simple. It reads a log record, looks at the name of the action identifier in the log record, and invokes the undo operation of that action, passing it the log record. Recovery is thereby table driven. This table-driven design has allowed the addition of new recoverable objects and actions to the RSS without any impact on recovery management.

The effect of any uncommitted transaction can be undone by reading the log of that transaction *backward*, undoing each action in turn. Clearly, this process can be stopped halfway, returning the transaction to an intermediate transaction save point. Transaction save points allow the transaction to backtrack.

From this discussion it follows that a transaction's log is a push-down stack; writing a new log record pushes it onto the stack, and undoing a record pops it off the stack (see Figure 8). To minimize log buffer space and log I/O, all transaction logs are merged into one system log, which is then mapped into a log file. But the log records of a particular transaction are chained together as a linked list anchored off of the transaction descriptor. Notice that UNDO requires that the log be directly addressable while the transaction is uncommitted. For this reason at least one version of the log must be on disk (or some other direct-access device). A tape-based log would be inconvenient for in-progress transaction undo.

2.5 Transaction Save Points

A transaction save point records enough information to restore the transaction's view of the RSS as of the save point. The user may record up to 64 kilobytes of application data in the log at each save point.

One can easily restore a transaction to its beginning by undoing all its updates and then releasing all its locks and dropping all its cursors (since no cursors or locks are held at the beginning of the transaction and since a list of cursors and locks is maintained on a per-transaction basis). To restore to a save point, the recovery manager must know the name and state of each active cursor and the name of each lock held at the save point. For performance reasons, changes to cursors and locks are not recorded in the log. Otherwise, every read action would have to write a log record. Instead, the state of locks and cursors is only recorded at save points. Assuming that all locks are held to end of the transaction, transaction backup can reset cursors without having to reacquire any locks. Further, because these locks are kept in a list and are not released, one can remember them all by remembering the lock at the top of the list. At backup to a save point, all subsequent locks are released.

2.6 System Configuration, Startup, and Shutdown

A System R database is created by installing a "starter system" and then using System R commands to define and load new files and to define transactions which manipulate these files. Certain operations (e.g., turning dual logging on and off) require a system shutdown and restart, but most operations can be performed while the system is operational. In particular we worked quite hard to avoid functions like SYSGEN and cold-start.

A *monitor* process (a task or virtual machine) is responsible for system startup, shutdown, and checkpoints and for servicing system operator commands. If several instances of System R (several different databases) are running on the same machine, each instance will have a monitor. System R users join a particular instance of System R, run transactions in the users' process, and then leave the system. In theory, 254 users may be joined to a system at one time.

2.7 System Checkpoint

System checkpoints limit the amount of work (undo and redo) necessary at restart. A checkpoint records information on disk which helps locate the end of the log at restart and correlates the database state with the log state. A checkpoint saves all logged shadow files so that no work prior to the checkpoint will have to be redone at restart. If checkpoints are taken frequently, then restart is fast but the checkpoint overhead is high. Balancing the cost of checkpoints against the cost of restart gives an optimum checkpoint interval. Because this optimum depends critically on the cost of

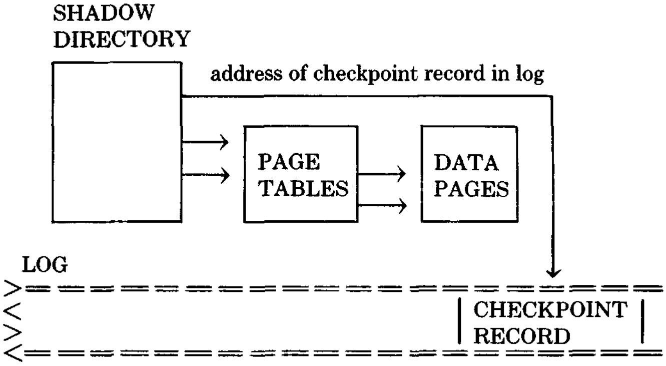

Figure 10. The directory root points at the most recent checkpoint record in the log.

a checkpoint, one wants a cheap checkpoint facility.

The simplest form of checkpoint is to record a *transaction-consistent state* by quiescing the system, deferring all new transactions until all in-progress transactions complete, and then recording a logically consistent snapshot. However, quiescing the system causes long interruptions in system availability and hence argues for infrequent checkpoints. This, in turn, increases the amount of work that is lost at restart and must be redone. System quiesce, consequently, is not a cheap way to obtain a transaction-consistent state.

The RSS uses a lower level of consistency, augmented by a transaction log, to produce a transaction-consistent state. The RSS implements *checkpoints* which are snapshots of the system at a time when no RSS actions are in progress (an *action-consistent state*). Since RSS actions are short (less than 10,000 instructions) system availability is not adversely affected by frequent checkpoints. (Long RSS actions, e.g., sort or search, occasionally "come up for air" in an action-consistent state to allow checkpoints to occur.)

Checkpoints are taken after a specified amount of log activity or at system operator request. At checkpoint, a checkpoint record is written in the log. The checkpoint record contains a list of all transactions in progress and pointers to their most recent log records. After the log records are on disk, all logged files are saved (current state replaces shadows), which involves flushing the database buffer pool and the shadow-file directories to secondary storage. As a last step, the log address of the checkpoint record is written as part of the directory record in the shadow version of the state [LORI77]. The directory root is duplexed on disk, enabling the checkpoint process to tolerate failures while writing the directory root. At restart the system will be able to locate the corresponding checkpoint record by examining the most recent directory root (see Figure 10).

2.8 System Restart

Given a checkpoint of the state at time T along with a log of all changes to the state made by transactions prior to time $T + E$, a transaction-consistent version of the state can be constructed by undoing all updates, logged prior to time T, of transactions which were uncommitted or aborted by time $T + E$, and then redoing the updates, logged between time T and time $T + E$, of committed transactions.

At system restart the system R code is loaded and the file manager restores any shadowed files to their shadow versions. If a shadowed file was not saved at shutdown, the then-current version will be replaced by its shadow. In particular, all logged files will be reset to their state as of the most recent system checkpoint.

Recovery manager is then given control and it examines the most recent checkpoint record (which, as Figure 10 shows, is pointed at by the current directory). If there was no work in progress at the time of the checkpoint and if the checkpoint is the last record in the log, then the system is restarting from a shutdown in a quiesced state. No transactions need be undone or redone, and restart initializes System R and opens up the system for general use.

On the other hand, if work was in progress at the checkpoint, or if there are log records after the checkpoint record, then this is a restart from a crash. Figure 11 illustrates the five possible states of transactions at this point:

- Tl began and ended before the checkpoint.
- T2 began before the checkpoint and ended before the crash.
- T3 began after the checkpoint and ended before the crash.
- T4 began before the checkpoint but no commit record appears in the log.
- T5 began after the checkpoint and apparently never ended.

To honor the commit of Tl, T2, and T3 transactions requires their updates to appear in the system state (done). But T4 and T5 have not committed and so their updates must not appear in the state (undone).

At restart the shadowed files are as they were at the most recent checkpoint. Notice that none of the updates of T5 are reflected in this state, so T5 is already undone. Notice also that all of the updates of Tl are in the shadow state, so it need not be redone. T2 and T3 must be redone from the checkpoint forward. (The updates of the first half of T2 are already reflected in the shadow state.) On the other hand, T4 must be undone from the checkpoint backward. (Here we are skipping over the following anomaly: If, after a checkpoint, T2 backs up to a save point prior to the checkpoint, then some undo work is required for T2.)

Restart uses the log as follows. It reads the most recent checkpoint record and assumes that all the transactions active at the time of the checkpoint are of type T4 (active at checkpoint, not committed). It then reads the log in the forward direction starting from the checkpoint record. If it encounters a BEGIN record, it notes that this is a transaction of type T5. If it encounters the COMMIT record of a T4 transaction, it reclassifies the transaction as type T2. Similarly, T5 transactions are reclassified as T3 transactions if a COMMIT record is found for that transaction. When it reaches the end of the log, the restart manager knows all the T2, T3, T4, and T5 transactions. T4- and T5-type transactions are called "losers" and T2- and T3-type transactions are called "winners." Restart reads the log backward from the checkpoint, undoing all actions of losers, and then reads the log forward from the checkpoint, redoing all actions of winners. Once this is done, a new checkpoint is written so that the restart work will not be lost.

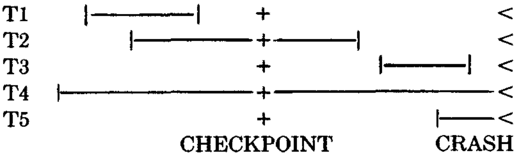

Figure 11. Five transaction types with respect to the most recent checkpoint and the crash point.

Restart must be prepared to tolerate failures during the restart process. This problem is subtle in most systems, but the System R shadow mechanism makes it fairly straightforward. System R restart does not update the log or the shadow version of the database until restart is complete. Taking a system checkpoint signals the end of a successful restart. System checkpoint is atomic, so there are only two cases to consider. Any failure prior to completing the checkpoint will return the restart process to the original shadow state. Any failure after the checkpoint is complete will return the database to the new (restarted) state.

2.9 Media Failure

In the event of a system failure which causes a loss of disk storage integrity, it must be possible to continue with a minimum of lost work. Such situations are handled by periodically making a copy of the database state and keeping it in an archive. This copy, plus a log of subsequent activity, can be used to reconstruct the current state. The archive mechanism used by System R periodically dumps a transaction-consistent copy of the database to magnetic tape.

It is important that the archive mechanism have failure modes independent of the failure modes of the on-line storage system. Using duplexed disks protects against a disk head crash, but does not protect against

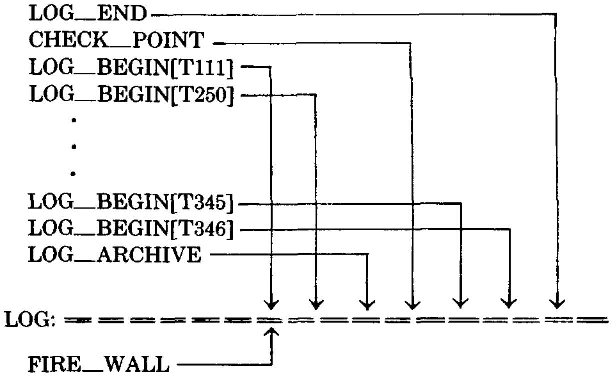

Figure 12. Bytes between FIRE_WALL and LOG_END are needed for system restart and hence are kept ın the log rıng buffer.

errors in disk programs or fire in the machine room.

In the event of a single media failure, the following occurs:

- If one of the duplexed on-line logs fails, then a new log is allocated and the good version of the duplexed log is copied to the new one.
- If any other file fails, the most recent surviving dump tape is loaded back into the database, and then all committed update actions subsequent to the checkpoint record of the archive state are redone using the log.

Recovery from an archive state appears to the recovery manager as a restart from a very old checkpoint. No special code has been written for archive recovery.

The most vulnerable part of this system is the possible failure of both instances of the on-line log. Assuming the log software has no bugs, we estimate the mean time between such coincident failures at about 1000 years. As mentioned in the summary to the Introduction, operational errors are much more frequent, and so make log failure a minor source of unrecoverable errors.

Although performing a checkpoint causes relatively few disk writes and takes only a short time, dumping the entire database is a lengthy operation (10 minutes per 100 megabytes). Maintaining transaction quiesce for the duration of the dump operation is undesirable, or even impossible, depending on the real-time constraints imposed by system users. Lorie [LORI77] describes a scheme based on shadows for making an archive dump while the system is operating. Gray [GRAY78] describes a "fuzzy" dump mechanism which allows the database to be dumped while it is in operation (the log is later used to focus the dump on some specified time). IMS and other commercial systems provide facilities for dumping and restoring fragments of files rather than whole databases. We did not implement one of these fancier designs because the simple approach was adequate for our needs.

2.10 Managing the Log

The log is a very long sequence of pages, each with a unique sequence number. Transactions undo and redo need quick access to the log but most of it can be kept off-line or discarded. Figure 12 illustrates the bookkeeping of the on-line log. The on-line log file is used to hold the "useful" parts of the log. It is managed as a ring buffer, with LOG_END pointing just beyond the last useful byte of the log and CHECK_POINT pointing at the most recent checkpoint. Clearly, all records since the checkpoint must be kept on-line in support of transaction redo. Further, transaction undo needs all records of incomplete transactions on-line (LOG_BEGIN(I) for active transaction I). Last, one cannot free the space occupied by a log record in the ring until the database is archived (this point is addressed by LOG_ARCHIVE). So at restart the system may need records back to the FIRE_WALL where FIRE_WALL is the minimum of LOG_ARCHIVE, CHECK_POINT, and LOG_BEGIN(I) for each active transaction I. Bytes prior to FIRE_WALL need not reside in the on-line ring buffer.

If the on-line ring buffer fills, it is because (1) archiving of the log is required, (2) a checkpoint is required, or (3) a transaction has been running for a very long time (and hence has a very low LOG_BEGIN). The first two problems are solved by periodic archiving and checkpoints. The third problem is solved by aborting very old transactions. For many applications, dual on-line logs of 1 megabyte are adequate, although 10 to 100 megabytes are more typical.

Duplexing of the log is a system option which may be changed at each restart. Duplexing is transparent above the log interface; reads (usually) go to one instance of the log, and writes go to both instances. If one instance fails, the other is used at restart (on a page-by-page basis).

2.11 Recovery and Locking

Recovery has implications for and places requirements on the lock subsystem.

In order to blindly undo the actions of one transaction without erasing subsequent updates by other transactions, it is essential that all transactions lock all updates in exclusive mode and hold all such locks until the transaction is committed or undone. In fact System R automatically acquires both share and exclusive locks and holds them all until the end of the transaction. See GRAY76 for a detailed discussion of the various locking options supported by System R.

A second issue is that transaction undo cannot tolerate deadlock (we do not want to have to undo undo's). Undo may take advantage of the fact that it only accesses records the transaction locked in the do step; hence it need not re-request these locks. (This is a consequence of holding all exclusive locks until the end of the transaction.) But transaction undo may have to set some locks because other RSS actions are in progress and because RSS actions release some locks at the end of each RSS action (e.g., physical page locks when logical record locking is the actual granularity). To solve this problem, transactions which are performing transaction undo are marked as "golden." Golden transactions are never chosen as deadlock victims; whenever they get into a deadlock with some other transactions, the other transactions are preempted. To assure that this rule does not produce unbreakable deadlock cycles (ones containing only golden transactions), the additional rule is adopted that only one golden transaction can execute at a time (and hence that no deadlock cycle involves more than one golden transaction). A special lock (RSSBACKUP) is requested in exclusive mode by each golden transaction before it begins each undo step. This lock is released at the end of each such undo step.

During restart, locking is turned off. Essentially, the entire database is locked during the restart process which sequentially executes actions in the order they appear in the log (making an undo pass followed by a redo pass).

3. EVALUATION

We were apprehensive on several counts when we first designed the System R recovery system. First, we were skeptical of our ability to write RSS actions which could always undo and redo themselves. Second, we were apprehensive about the performance and complexity of such programs. And third, we were concerned that the added complexity might create more crashes than it cured. In retrospect, the recovery system was comparatively easy to write and certainly has contributed to the system's reliability.

3.1 Implementation Cost

The RSS was designed in 1974 and 1975 and became operational in 1976. Since then we have had a lot of experience with it.

Writing recoverable actions (ones which can undo and redo themselves) is quite hard. Subjectively, writing a recoverable action is 30 percent harder and, objectively, it requires about 20 percent more code than a nonrecoverable action. In addition, the recovery system itself (log management, system restart, checkpoint, commit, abort, etc.) contributes about 15 percent of the code of the RSS. However, the RSS is less than half of System R, so these numbers may be divided in half when assessing the overall system implementation cost, leaving the marginal cost of implementing recovery at about 10 percent.

3.2 Execution Cost

Another component of cost is the instructions added to execution of a transaction to support recovery (frequently called "path length"). Fortunately, most transactions commit and hence make no use of the undo or redo code. In the normal case the only cost imposed by recovery is the writing of log records. Further, only update operations need write log records. Typically, the cost of keeping a log is less than 5 percent increased path length. In addition, the execution cost, associated with recovery, of periodic checkpoints is minimal. Restart is quite fast, typically running ten times faster than the original processing (primarily because updates are infrequent). Hence if the checkpoint interval is 5 minutes, the system averages 15 seconds to restart. As described in the next section, checkpoint is I/O bound.

3.3 I/O Cost

A third component of recovery cost is I/O added by the recovery system. Each transaction commit adds two I/Os to the cost of the transaction when the duplexed log is forced to disk (as part of the commit action). This cost is reduced to one extra I/O if dual logging is not selected. Log force is suppressed for read-only transactions. Depending on the application, log force may be a significant overhead. In an application in which each transaction is a few (50) RSS actions, it constitutes a 20 percent I/O overhead. The transaction of Figure 1 would have about a 25 percent log I/O overhead. In another application in which the database is all resident in central memory, the log accounts for *all* of the disk I/O. IMS Fast Path solves this problem by logging several transactions in one I/O so that one gets *less than* one log I/O per transaction. The shadow mechanism when used with large databases often implies extra I/O, both during normal operation and at checkpoint. Lorie's estimates in LORI77 are correct: a checkpoint requires several seconds of I/O after 5 minutes of work on a 100-megabyte database. This work increases with larger databases and with high transaction rates. It becomes significant at 10 transactions per second or for billion-byte files.

3.4 Success Rate

Perhaps the most significant aspect of the System R recovery manager is the confidence it inspires in users. During development of the system, we routinely crashed the system knowing that the recovery system will be able to pick up the pieces. Recovery from the archive is not unheard of, but it is very uncommon. This has created the problem that some users do not take precautions to protect themselves from media failures.

3.5 Complexity

It seems to be the case that the recovery system cures many more failures than it introduces. Among other things this means that everybody who coded the RSS understood the do–undo–redo protocol reasonably well and that they were able to write recoverable actions. As the system has evolved, new people have added new recoverable objects and actions to the system. Their ability to understand the interface and to fit into its table-driven structure is a major success of the basic design.

The decision to put all responsibility for recovery into the RSS made the RDS much simpler. There is essentially no code in the RDS to handle transaction management beyond the code to externalize the begin, commit, and abort actions and the code to report transaction abort to the application program.

3.6 Disk-Based Log

A major departure of the RSS from other data managers is the use of a disk-based log and its merging of the undo and redo logs. The rationale for the use of disk is that disks cost about as much as tapes (typically $30,000 per unit if controllers are included), but disks are more capacious than tapes (500 megabytes rather than 50 megabytes per unit) and can be allocated in smaller units (a disk-based log can use half of the disk cylinders; it is not easy to use half of a tape drive). Further, a disk-based log is consistent with the evolution of tape archives such as the IBM 3850 and the AMPEX Terabit Store. Last, but most important, a disk-based log eliminates operator intervention at system restart. This is essential if restart is to occur automatically and within seconds. Since restart is infrequent, operators are likely to make errors during the restart process (even if they have regular "fire drills"). A disk-based log reduces opportunities for operator error.

Several systems observe that the undo log is not needed once a transaction commits. Hence they separate the undo and redo log and discard the undo log at transaction commit. Merging the two logs causes the log to grow roughly twice as fast but leads to a simpler design. Since transactions typically write only 200 to 500 bytes of log data, we do not consider splitting the undo and redo logs to be worth the effort.

3.7 Save Points

Transaction save points are an elegant idea which the RSS can implement cheaply. Transaction save points are used by the RDS to undo complex RDS operations (i.e., make them atomic). However, transaction save points are not available to application programs using the SQL language. Unfortunately, the RDS implementors let PL/I do most of the storage control, so the RDS processor does not know how to save its state and PL/I does not offer it a facility to reset its state even if the RDS could remember the state. The RDS, therefore, does not show the RSS save point facility to users.

Supporting save points is an unsolved language-design issue for SQL. If SQL were imbedded in a language which supported backtrack programming, save points might be implemented rather naturally. INTERLISP is a natural candidate for this [TEIT72], since it already supports the notion of undo as an integral part of the language.

We had originally intended to have system restart reset in-progress transactions to their most recent save point and then to invoke the application at an exception entry point (rather than abort all uncommitted transactions at restart). (CICS does something like this.) However, the absence of save point support in the RDS and certain operating system problems precluded this feature.

3.8 Shadows

The file shadow mechanism of System R is a key part of the recovery design. It is used to create and discard user scratch files, to store user work files, and to support logged files. A major virtue of shadows is that they ensure that system restart always begins with an RSS action-consistent state. This is quite a simplification and probably contributes to the success of system restart.

To understand the problem that shadows solve at restart, imagine that System R did not use shadows but rather updated pages in place on the disk. Imagine two pages P1 and P2 of some file F, and suppose that P1 and P2 are related to one another in some way. To be specific, suppose that P1 contains a reference R1 to a record R2 on P2. Suppose that a transaction deletes R2 and invalidates R1 thereby altering P1 and P2. If the system crashes there are four possibilities:

(1) Neither P1 nor P2 is updated on disk.
(2) P1 but not P2 is updated on disk.
(3) P2 but not P1 is updated on disk.
(4) Both P1 and P2 are updated on disk.

In states 2 and 3, P1 and P2 are not RSS-action consistent: either the reference, R1, or referenced object, R2, is missing. System restart must be prepared to redo and undo in any of these four cases. The shadow mechanism eliminates cases 2 and 3 by checkpointing the state only when it is RSS-action consistent (hence restart sees the shadow version recorded at checkpoint rather than the version current at the time of the crash). Without the shadow mechanism, the other two cases must be dealt with in some way.

One alternative is the "write ahead log" (WAL) protocol used by IMS [IBMa]. IMS log records apply to page updates (rather than to actions). WAL requires that log records be written to secondary storage *ahead of* (i.e., before) the corresponding updates. Further, it requires that undo and redo be restartable: attempting to redo a done page will have no effect and attempting to undo an undone page will have no effect, allowing restart to fail and retry as though it were a first attempt. WAL is extensively discussed by Gray [GRAY78].

There is general consensus that heavy reliance on shadows for large shared files was a mistake. We recognized this fact rather late (shadows have several seductive properties), so late in fact that a major rewrite of the RSS is required to reverse the decision. Fortunately, the performance of shadows is not unacceptable. In fact for small databases (fewer than 10 megabytes) shadows have excellent performance.

Our adoption of shadows is largely historical. Lorie implemented a single-user relational system called XRAM which used the shadow mechanism to give recovery for private files. When files are shared, one needs a transaction log of all changes made to the files by individual users so that the changes corresponding to one user may be undone independently of the other users' changes. This transaction log makes the shadow mechanism redundant. Of course the shadow mechanism is still a good recovery technique for private files. A good system should support both shadows for private files and log-based recovery for shared files. Several other systems, notable QBE [IBMb], the DataComputer [MARI75], and the Lampson and Sturgis file system [LAMP81] have a similar use of shadows. It therefore seems appropriate to present our assessment of the shadow mechanism.

The conventional way of describing a large file (e.g., over a billion bytes) is as a sequence of *allocation units.* An allocation unit is typically a group of disk cylinders, called an *extent.* If the file grows, new extents are added to the list. A file might be represented by 10 extents and the corresponding descriptor might be 200 bytes. Accessing a page consists of accessing the extent table to find the location of the extent containing the page and then accessing the page.

By contrast, a shadow mechanism is much more complex and expensive. Each page of the file has an individual descriptor in the page table. Such descriptors need to be at least 4 bytes long and there need to be two of them (current and shadow). Further, there are various free-space bit maps (a bit per page) and other housekeeping items. Hence the directories needed for a file are about 0.2 percent of the file size (actually 0.2 percent of the maximum file size). For a billion-byte file this is 2 megabytes of directories rather than the 200 bytes cited for the extent oriented descriptors.

For large files this means that the directories cannot reside in primary storage; they must be paged from secondary storage. The RSS maintains two buffer pools: a pool of 4-kilobyte data pages and another pool of 512-byte directory pages. Management and use of this second pool added complexity inside the RSS. More significantly, direct processing (hashing or indexing single records by key) may suffer a directory I/O for each data I/O.

Another consequence of shadows is that "next" in a file is not "next" on the disk (logical sequential does not mean physical sequential). When a page is updated for the first time, it moves. Unless one is careful, and the RSS is not careful about this, getting the next page frequently involves a disk seek. (Lorie in LORI77 suggests a shadow scheme which maintains physical clustering within a cylinder.) So it appears that shadows are bad for direct (random) processing and for sequential processing.

Shadows consume an inconsequential amount of disk space for directories (less than 1 percent). On the other hand, in order to use the shadow mechanism, one must reserve a large amount (20 percent) of disk space to hold the shadow pages. In fact some batch operations and the system restart facility may completely rewrite the database. This requires either a 100 percent shadow overhead or the operation must be able to tolerate several checkpoints (i.e., reclaim shadow) while it is in progress. This problem complicates system restart (its solution was too complex to describe in the system restart section).

The RSS recovery system does not use shadowed files for the log; rather, it uses disk extents (one per log file). However the recovery system does use the shadow mechanism at checkpoint and restart. At checkpoint all the current versions of all recoverable files are made the shadow versions. This stops the system and triggers a flurry of I/O activity. The altered pages in the database buffer pool are written to disk, and much directory I/O is done to free obsolete pages and to mark the current pages as allocated. The major work in this operation is that three I/Os must be done for every directory page that has changed since the last checkpoint. If updates to the database are randomly scattered, this could mean three I/Os at checkpoint for each update in the interval between checkpoints. In practice updates are not scattered randomly and so things are not that bad, but checkpoint can involve many I/Os.

We have devised several schemes to make the shadow I/O asynchronous to the checkpoint operation and to reduce the quantity of the I/O. It appears, however, that much of the I/O is inherent in the shadow mechanism and cannot be eliminated entirely. This means that the RSS (System R) must stop transaction processing for seconds. That in turn means that user response times will occasionally be quite long.

These observations cause us to believe that we should have adopted the IMS-like approach of using the WAL protocol for large shared files. That is, we should have supported the log and no-shadow option in Figure 9. If we had done this, the current and shadow directories would be replaced by a much smaller set of file descriptors (perhaps a few thousand bytes). This would eliminate the directory buffer pool and its attendant page I/O. Further, checkpoint would consist of a log quiesce followed by writing a checkpoint record and a pointer to the checkpoint record to disk (two or three I/Os rather than hundreds). WAL would not be simpler to program (for example, WAL requires more detailed logging). But the performance of WAL is better for large shared databases (bigger than 100 megabytes).

3.9 Message Recovery, an Oversight

As pointed out by the examples in Figures 1 and 4, a transaction's database actions and output messages must either all be committed or all be undone. We did not appreciate this in the initial design of System R and hence did not integrate a message system with the database system. Rather, application programs use the standard terminal I/O interfaces provided by the operating system, and messages have no recovery associated with them. This was a major design error. The output messages of a transaction must be logged and their delivery must be coordinated by the commit processor. Commercial data management systems do this correctly.

3.10 New Features

We recently added two new facilities to the recovery component and to the SQL interface. First the COMMIT command was extended to allow an application to combine a COMMIT with a BEGIN and preserve the transaction's locks and cursors while exposing (committing) its updates. COMMIT now accepts a list of cursors and locks which are to be kept for the next transaction. These locks are downgraded from exclusive to shared locks, and all other cursors and locks are released. A typical use of this is an application which scans a large file. After processing the *A*'s it commits and processes the *B*'s, then commits and then processes the *C*'s, and so on. In order to maintain cursor positioning across each step, the application uses the special form of commit which commits one transaction and begins the next.

A second extension involved support for the two-phase commit protocol required for distributed systems [GRAY78]. A PHASE_ONE action was added to the RSS and to SQL to allow transactions to prepare to commit. This causes the RSS to log the transaction's locks and to force the log. Further, at restart there are now three kinds of transactions: winners, losers, and in-doubt. In-doubt transactions are redone and their locks are reacquired at restart. Each such transaction continues to be in-doubt until the transaction coordinator commits or aborts it (or the system operator forces it). During the debugging of this code, several transactions were in-doubt for two weeks and for tens of system restarts.

At present our major interest is in a distributed version of System R. We are extending the System R prototype to support transparent distribution of data among multiple database sites.

ACKNOWLEDGMENTS

We have had many stimulatıng discussions with Dar Busa, Earl Jenner, Homer Leonard, Dieter Gawlick, John Nauman, and Ron Obermarck. They helped us better understand alternate approaches to recovery. John Howard and Mike Mitoma did several experıments which stress tested the recovery system. Jim Mehl and Bob Yost adapted the recovery manager to the MVS environment. Tom Szczygielski implemented the two-phase commit protocol. We also owe a great deal to the recovery model formulated by Charlie Davies and Larry Bjork.

REFERENCES

(Note [BLAS79] is not cited in text.)

ASTR76 ASTRAHAN, M M., BLASGEN, M. W., CHAMBERLIN, D. D., ESWARAN, K. P., GRAY, J. N , GRIFFITHS, P. P., KING, W F , LORIE, R. A , MCJONES, P R., MEHL, J. W., PUTZOLU, G. R., TRAIGER, I. L , WADE, B W., AND WATSON, V "System R A relational approach to database management," *ACM Trans Database Syst* **1,** 2 (June 1976), 97-137.

BJOR73 BJORK, L. "Recovery scenario for a DB/DC system," in *Proc ACM Nat Conf.,* 1973, pp. 142-146.

BLAS79 BLASGEN, M. W , GRAY, J. N , MITOMA, M., AND PRICE, T. "The convoy phenomenon," *ACM Oper. Syst Rev.* **14,** 2 (April 1979), 20-25.

CHAM76 CHAMBERLIN, D D., ASTRAHAN, M. M., ESWARAN, K. P., GRIFFITHS, P P., LORIE, R. A., MEHL, J. W , REISNER, R., AND WADE, B. W. "SEQUEL 2: A unified approach to data definition, manipulation and control," *IBM J Res Dev.* **20,** 6 (Nov. 1976), 560-576

DAVI73 DAVIES, C. T. "Recovery semantics for a DB/DC system," in *Proc. ACM Nat. Conf.,* 1973, pp. 136-141.

ESWA76 ESWARAN, K. E , GRAY, J N., LORIE, R. A., AND TRAIGER, I. L. "On the notions of consistency and predicate locks in a relational database system," *Commun. ACM* **19,** 11 (Nov. 1976), 624-634.

GRAY75 GRAY, J. N., AND WATSON V. "A shared segment and interprocess communication facility for VM/370," IBM San Jose Research Lab. Rep. RJ 1679, May 1975.

GRAY76 GRAY, J. N , LORIE, R A., PUTZOLU, G. F , AND TRAIGER, I. L. "Granularity of locks and degrees of consistency in a shared data base," *Modeling in data base management systems,* G. M. Nijssen, Ed., North-Holland, Amsterdam, 1976, pp. 365-394; also IBM Research Rep. RJ 1706.

GRAY78 GRAY, J. N. "Notes on data base operating systems," in *Operating systems—an advanced course,* R. Bayer, R M. Graham, and G. Seegmuller, Eds., Springer Verlag, New York, 1978, pp. 393-481; also IBM Research Rep. RJ 2188, Feb 1978.

IBMa IBM "Information management system/virtual systems (IMS/VS), programming reference manual," IBM Form No. SH20-9027-2, sect. 5.

IBMb IBM "Query by example program description/operators manual," IBM Form No SH20-2077.

LAMP81 LAMPSON, B. W., AND STURGIS, H E. "Crash recovery in a distributed data storage system," *Commun ACM,* to appear.

LORI77 LORIE, R. A. Physical integrity in a large segmented database," *ACM Trans Database Syst* **2,** 1 (March 1977), 91-104.

MARI75 MARILL, T., AND STERN, D. H "The Datacomputer· A network utility," in *Proc AFIPS Nat. Computer Conf*, vol. 44, AFIPS Press, Arlington, Va., 1975

NAUM78 NAUMAN, J. S. "Observations on sharing in data base systems," IBM Santa Teresa Lab. Tech. Rep. TR 03.047, May 1978.

TEIT72 TEITLEMAN, W. "Automated programming—The programmer's assistant," in *Proc. Fall Jt. Computer Conf*, Dec. 1972

Received February 1980, final revision accepted November 1980