RJ2623(33801)8/15/79
Computer Science

Research Report

THE RECOVERY MANAGER OF A DATA MANAGEMENT SYSTEM

Jim Gray
Paul McJones
Mike Blasgen
Raymond Lorie
Tom Price
Franco Putzolu
Irving Traiger

IBM Research Laboratory
San Jose, California 95193

IBM Research Division
Yorktown Heights, New York · San Jose, California · Zurich, Switzerland

Copies may be requested from:
IBM Thomas J. Watson Research Center
Distribution Services
Post Office Box 218
Yorktown Heights, New York 10598

RJ2623(33801)8/15/79
Computer Science

THE RECOVERY MANAGER OF A DATA MANAGEMENT SYSTEM

Jim Gray
Paul McJones*
Mike Blasgen
Raymond Lorie
Tom Price
Franco Putzolu
Irving Traiger

IBM Research Laboratory
San Jose, California 95193

ABSTRACT: This paper describes and evaulates the recovery subsystem of System R, an experimental data management system. The features of the recovery system include: The transaction concept which allows application programs to commit, abort or partially undo their effects. The DO-UNDO-REDO protocol allows new recoverable types and operations to be added to the recovery system. Application programs can record data in the transaction log to facilitate application-specific recovery. The checkpoint mechanism is based on differential files (shadows). The recovery log is based on disk rather than tape.

---

* Present address: Xerox Corporation, 3333 Coyote Hill Road, Palo Alto, California, 94304

The Recovery Manager of a Data Management System

Jim Gray, Paul McJones*, Mike Blasgen, Raymond Lorie
Tom Price, Franco Putzolu, Irving Traiger
IBM San Jose Research Laboratory

ABSTRACT: This paper describes and evaluates the recovery subsystem of System R, an experimental data management system. The features of the recovery system include: The transaction concept which allows application programs to commit, abort or partially undo their effects. The DO-UNDO-REDO protocol allows new recoverable types and operations to be added to the recovery system. Application programs can record data in the transaction log to facilitate application-specific recovery. The checkpoint mechanism is based on differential files (shadows). The recovery log is based on disk rather than tape.

CONTENTS:

* Present address: Xerox Corporation, 3333 Coyote Hill Road, Palo Alto, California, 94304

SYSTEM R OVERVIEW

System R [1] is broken into two major layers: An external layer called the *Research Data System (RDS)*, and a completely internal layer called the *Research Storage System (RSS)*.

The external layer provides a relational data model, and operators on this model. It also provides catalog management, data dictionary, authorization, and alternate views of data. The RDS is manipulated using the language SQL [2]. The SQL compiler maps SQL statements into sequences of RSS calls.

The internal layer is a non-symbolic 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 via key) and *parent-child set* (access path supporting the operations PARENT, FIRST_CHILD, (NEXT | PREVIOUS) SIBLING with direct pointers), and *cursor* which navigates over access paths to locate records. (Unfortunately, these objects have the non-standard names segment, relation, tuple, field, image, link and scan in the System R documentation. The more standard names are used here.)

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

Another major aspect of the RSS is its support of the notion of *transaction:* 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. Transactions are units of recovery. The RSS assumes all responsibility for running concurrent transactions and for assuring that each transaction sees a consistent view of the data base. The RSS is also responsible for recovering the data to its most recent consistent state in the event of user, transaction, action, system or media failure.

A final component of System R is the operating system. System R was originally designed to run under the VM/370 operating system. VM/370 provides processes (tasks) in the form of virtual IBM/370's. The VM/370 inter-process communication facility (VMCF) is used to send signals (interrupts) between virtual machines. VM/370 was extended to allow machines to share page tables (shared memory) [3]. System R has also been adapted to run on the MVS operating system. This adaptation substitutes an MVS address space (and several MVS tasks) for each VM virtual machine. The principal difficulties encountered in this conversion were in the areas of alternate device support and differing task structure. In particular, the recovery component is independent of the operating system.

DESCRIPTION OF SYSTEM R RECOVERY MANAGER

TRANSACTION DEFINITION

The RSS provides **actions** on the objects it implements. These actions include operations to create and destroy RSS objects such as files, record types, record instances, indices, sets, and cursors as well as actions which manipulate, retrieve and modify such objects. Each such RSS action is *atomic:* (a) it either happens or has no effect and (b) if any two actions relate to some object they appear to execute in some serial order. This is accomplished by (a) undoing the partial effects of any actions which fail and (b) by locking necessary RSS resources for the

duration of the action. Each RSS action is a ' mini-transaction '. In the vernacular of [4] each RSS action is a two-phase well-formed transaction on the RSS structure. Hence each user sees an RSS consistent state.

RSS actions are rather primitive. In general several actions are required to ' hire an employee ' or ' make a deposit in an account. ' 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 a transaction is a funds transfer which debits one account, credits another account, writes an activity record, and does some terminal input and output. The author of such a transaction wants it to be an all or nothing affair: he doesn't want only some of the actions of the transaction to have occurred. In a multi-user environment, this atomicity takes on the additional attribute that any two transactions concurrently operating on common objects should appear to run serially. This later problem is handled by the lock subsystem [4], [5], [6].

The application declares a sequence of actions to be a transaction by beginning the sequence with a BEGIN action and ending the sequence with a COMMIT action. All intervening actions by that application (process or collection of processes) are considered to be parts of a single unit. If the application gets into trouble, it may issue the ABORT action which undoes all actions by the transaction. Further, the system may unilaterally abort in-progress transactions in certain cases (e.g. authorization violation, resource limits, deadlock, system shutdown or crash). Figure 1 demonstrates these three cases.

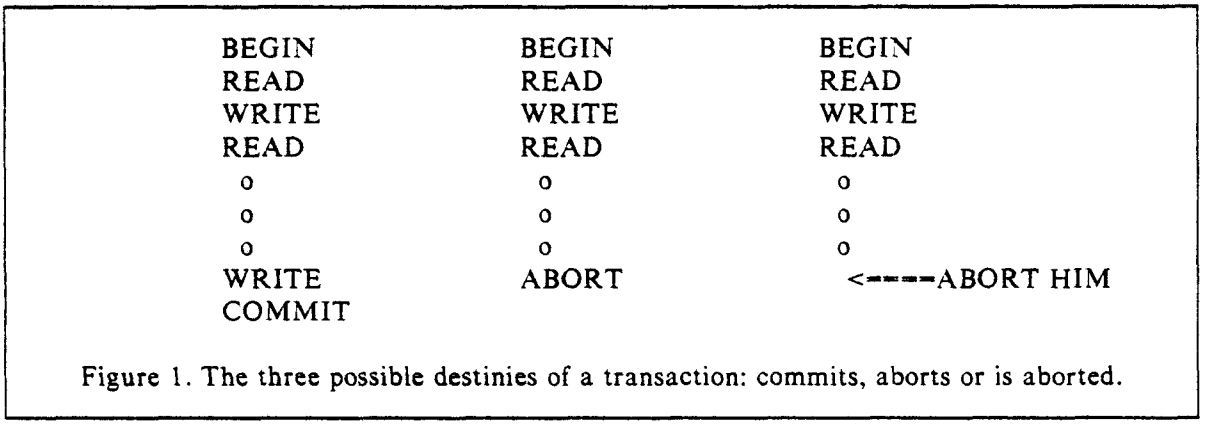

Figure 1. The three possible destinies of a transaction: commits, aborts or is aborted.

If a transaction aborts or is aborted then the system must undo all its actions on recoverable objects. However the effects of committed transactions must be *durable:* once the transaction commits, its updates and messages to the external world must persist. 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.

TRANSACTION SAVE POINTS

The RSS defines the additional notion of **transaction save point.** A save point is a fire-wall which allows transaction undo to stop. If a transaction gets into trouble (e.g. deadlock or authority violation) it may be sufficient to back up to such an intermediate save point rather than undoing all the work of the transaction. Each save point is numbered, the beginning of a transaction is save point 1 and successive save points are numbered 2, 3, ... By issuing a save action specifying a save point record to be saved in the log, the application program (RDS or

user program) declares a save point. This save point record may be retrieved if (when) the transaction returns to that save point.

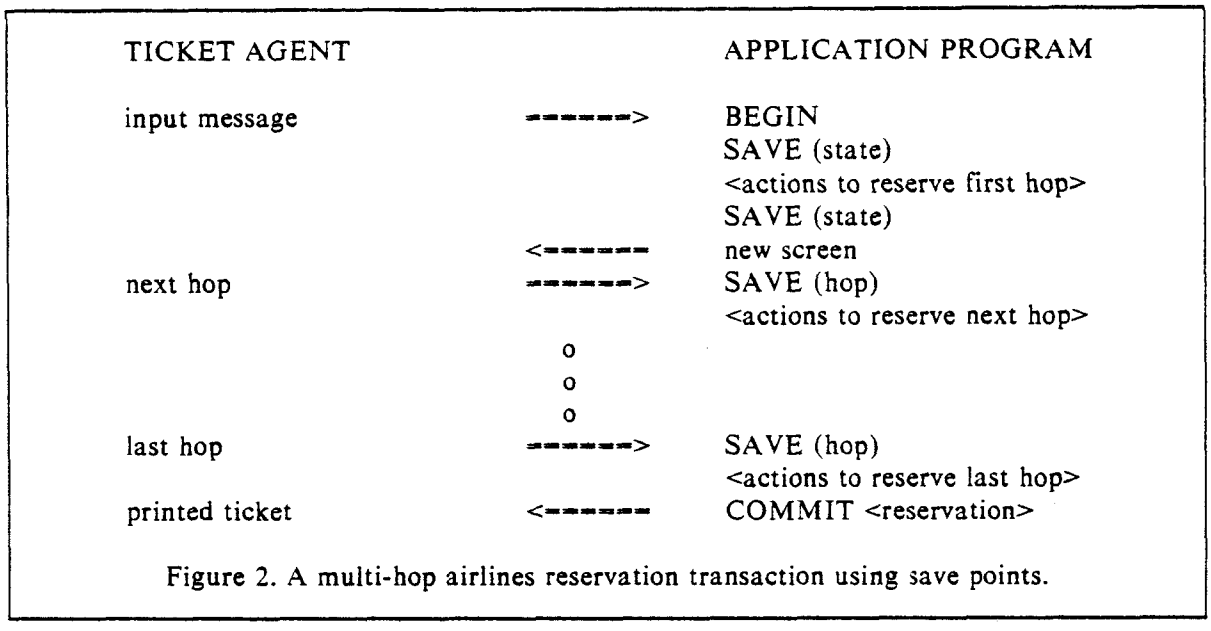

Figure 2. A multi-hop airlines reservation transaction using save points.

Figure 2 is a graphic example of the use of save points. It describes a conversational transaction making a multi-hop airline reservation involving several ticket agent interactions: one per hop. The application program establishes a save point before and after each interaction (the save point data includes the current state of the program variables and the ticket agent screen). If the ticket agent makes an error or if one flight is unavailable, the agent or program can back up to the most recent mutually acceptable save point thereby minimizing re-typing by the agent. Once the entire ticket is composed, the transaction commits the database changes and prints the ticket. Of course all the save points may be washed away by a system crash or serious deadlock, but most error situations will 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 manually backs out previous steps of the transaction when something goes wrong or the transaction defers all updates until the last step. The deferred update strategy (which is the most common) is complicated to program and has the problem that the flight status may change during the transaction (locks are not held between steps).

Save points can also be used by the RDS to implement complex operations. Suppose a single RDS operation requires many RSS operations and suppose that the RDS makes the guarantee that each SQL statement is atomic: it either ' happens ' or has no effect and raises an error condition. The RDS could support this 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.

TRANSACTIONS AND SYSTEM RESTART

As described above, a transaction has one of two destinies, it commits or aborts. Successful application programs issue commit actions. The abort may be generated by the transaction itself (or its user) or the abort may be generated by an external stimulus like deadlock, system crash or shutdown (see Figure 1).

After a system crash, all uncommitted transactions in progress at the time of the crash are *undone* and all committed transactions are preserved. If a committed action did not survive the restart (update was not written to disk or message was not delivered) then the action must be redone at restart. Undo and redo of transactions is based on a *log* of the actions performed by transactions. Each RSS action which changes the RSS state records enough information in the log to set the new state to the old state (undo) and to set the old state to the new state (redo). Transactions are undone or redone by applying the 'old' or 'new' values (respectively) from the log to the disk version of the database and the network state. If the disk version of the database does not survive, an archive version of the state is used. The log is optionally duplexed so that any single failure will be tolerated. Figure 3 illustrates what survives a system crash.

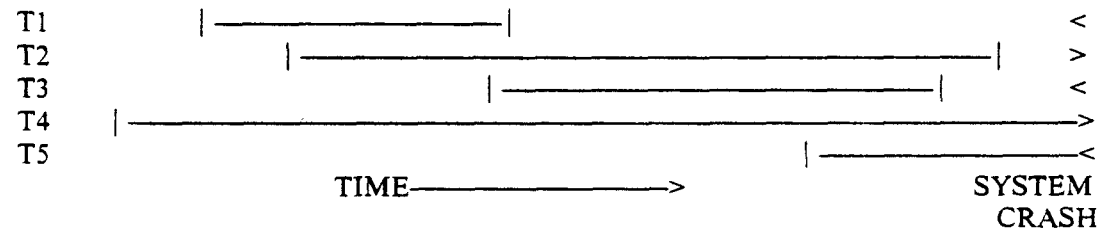

Figure 3. Five transactions. T1, T2 and T3 will survive a system crash. But T4 and T5 will not survive because they were in-progress at the time of the crash.

SUMMARY

To summarize, the RSS recovery manager provides the actions:

BEGIN: designates the beginning of a transaction.

SAVE: designates a fire-wall 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.

BACKUP: undoes the effects of a transaction to a earlier save point.

ABORT: undoes all effects of a transaction.

COMMIT: signals successful completion of transaction and causes outputs to be committed.

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

This model of recovery was formulated by Davies and Bjork [7], [8]. Unlike their model, RSS transactions have no parallelism within a transaction (e.g. if multiple nodes of a network execute a single transaction, only one node at a time is executing it). Further, the RSS 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.

The transaction model is an ideal which can never be attained in reality. At best, careful use of redundancy minimizes the probability of unrecoverable failures and consequent loss of commit-

ted updates. Redundant copies are designed to have independent failure modes so it is unlikely that all records are lost at the same time. However, Murphy's law: ' Anything that can go wrong, will go wrong! ' as amended by Randell: ' And at the worst possible time in the worst possible way ', implies that all recovery techniques will sometimes fail to recover. As will be seen below, System R can tolerate any single failure and often tolerates multiple failures.

IMPLEMENTATION OF SYSTEM R RECOVERY

FILES, VERSIONS AND SHADOWS

All persistent System R data is stored in files. A file is a paged linear space of up to sixty eight billion bytes. A user may define as many files as he likes. Files are 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 the System R users. The buffer manager uses an LRU algorithm to regulate occupancy of the pool. The buffer pool is volatile and is presumed not to survive system restart.

Each file carries a particular recovery protocol and corresponding overhead of recovery. Files are dichotomized as **shadowed** and **non-shadowed.**

Non-shadowed files have no automatic recovery. The user is responsible for making redundant copies of these files and storing such copies in a safe place. The RSS simply updates non-shadowed files (in the buffer pool). Changes to non-shadowed files migrate to 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 on-line versions of shadowed files: a **shadow version** and a **current version.** RSS actions affect only the current version of a file, the shadow version is never altered (except by file save and restore commands). The current version of a file can be SAVEd as the shadow version thereby committing the recent updates to the file, or the current version can be RESTOREd to the shadow version thereby ' undoing ' all recent updates to the file (see Figure 4).

If data is 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.

The current version of a file does not survive restart because recent updates to the file may still be in the buffer pool. However, the shadow version of a file does survive restart. Hence at RSS restart (i.e. after a crash or shutdown) all non-shadowed 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 will be discussed below, the transaction recovery then uses the log to remove the effects of aborted transactions and to restore the effects of committed transactions (see System restart section.)

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 file directory, and freeing superseded shadow pages. Restoring a file is achieved by discarding pages of that file in the buffer pool, freeing all the *new* disk pages of that file and returning to the shadow file directory. The paper by Lorie [9] describes the implementation in greater detail.

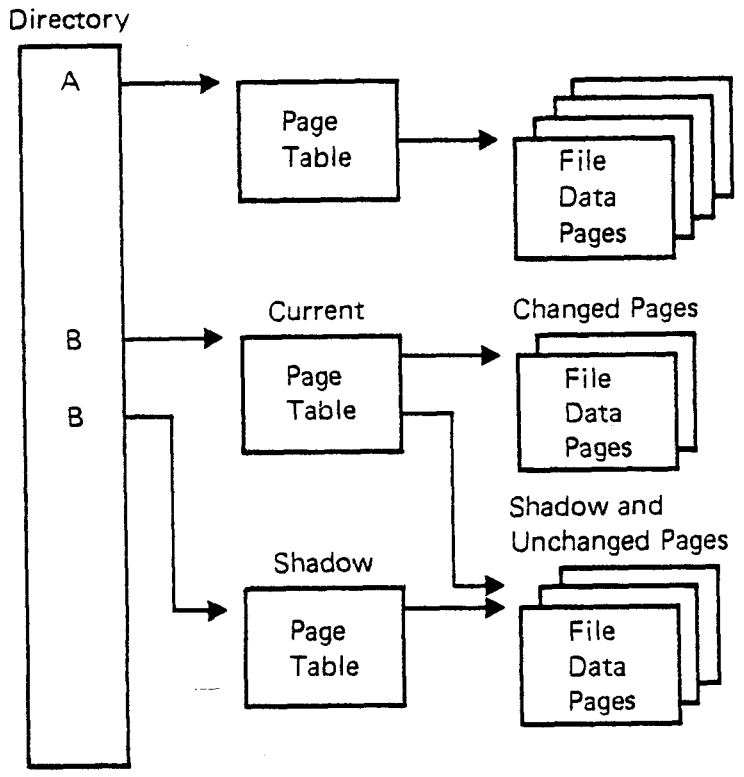

Figure 4. The directory structure for non-shadowed and shadowed files. File A is not shadowed. File B is shadowed and has two directory entries, a current version and a shadow version.

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: If a run fails, one goes back to the old master and tries again. If the run succeeds, the new master becomes the old master. Unhappily, this technique does not seem to generalize to concurrent transactions. If several transactions concurrently alter a file, then 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 BACKUP actions as well as to handle such problems as deadlock, system overload and unexpected user disconnect. Further, as shown in Figure 3, selective transaction back up and commit are required for system restart.

We were unable to architect a transaction mechanism which supported multiple users or save points and which was based solely on shadows. Rather, 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 action writes a log record giving the old and new value of the updated object. As Figure 5 shows, these records are aggregated by transaction and collected in a common system *log file* (which optionally is duplexed).

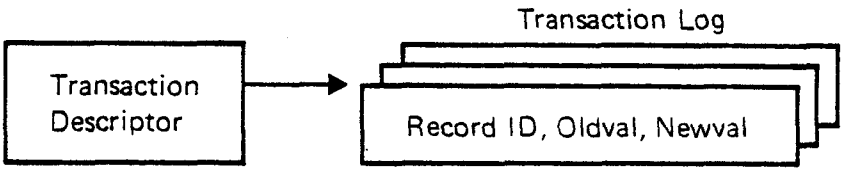

Figure 5. A transaction log is the sequence of changes made by this transaction.

When a shadowed file is defined, 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 non-logged shadowed files. Non-shadowed files have none of the virtues or corresponding overhead of recovery (see Figure 6).

	NO SHADOW	SHADOW
NO LOG	NON-SHADOW: contents unpredictable after crash	SHADOWED: contents = shadow after crash
LOG	not supported	LOGGED: All updates logged. Committed updates survive crash.

Figure 6. Recovery attributes of files.

Each time a transaction modifies a logged file, a new record is written in the log. Read actions need generate no log records, but update actions on logged files must write enough information in the log so that, given the log record at a later time, the action can be completely undone or redone. As will be seen below, most log writes do not require I/O, rather they can be buffered in central memory.

Thus every RSS action on a logged file must have two ' extra ' actions (see figure 7):

DO: does the action and also writes a log record sufficient to undo and to redo the action.

UNDO: undoes the action given the log record written by the DO action.

REDO: redoes the action given the log record written by the DO action.

DISPLAY: translates the log into a human-readable format (optional).

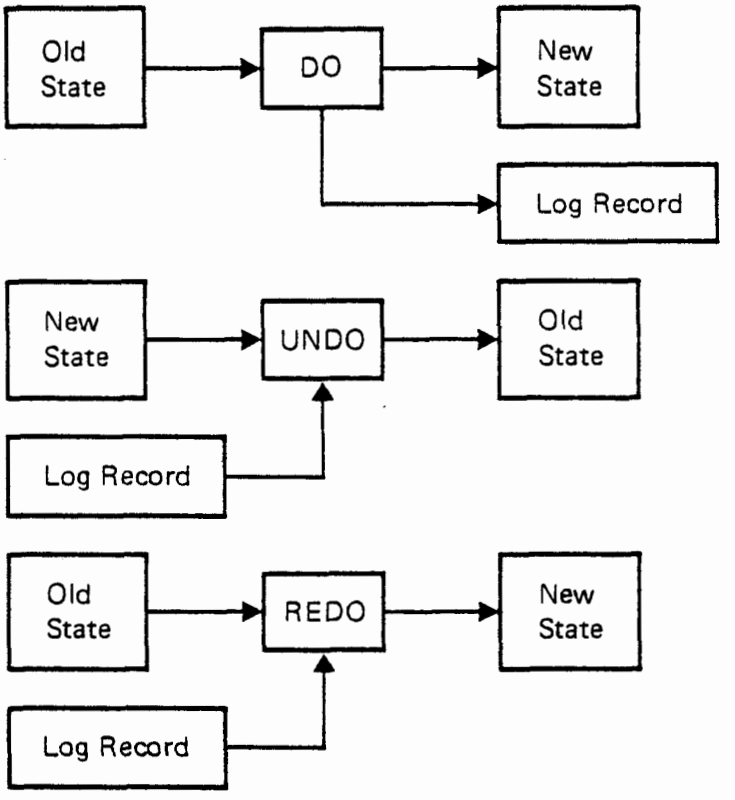

Figure 7. Three aspects of an action: Action DO generates a new state and a log record. Action UNDO generates an old state from a new state and a log record. Action REDO generates a new state from an old state and a log record.

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

file name
record identifier
old record value
new value.

The log subsystem augments this with the additional fields:

transaction identifier
action identifier
time stamp
length of log record
pointer to previous log record of this transaction.

The undo operation restores the record to its old value appropriately updating associated structures such as indicies (B-trees) 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 will traverse the log

of a particular transaction in either direction (forward or backward in time).

COMMIT PROCESSING

The essential property of a transaction is that each transaction ultimately either commits or aborts and that:

- Once it COMMITs, its updates persist.
- Once it ABORTs or is aborted, its updates are erased.

Achieving this property is non-trivial. In System R, a transaction commits when it issues the commit action and it aborts when it issues the abort action or is unilaterally aborted by the system. In order to insure that a transaction's effects will survive system and media failures the system must be able to redo committed transactions. System R does two things to insure that uncommitted transactions can be undone and that committed transactions can be redone:

1. The transaction log is written to disk before the shadow database is discarded.
2. The transaction commit action writes a commit log record and then forces all the transaction's log records to disk.

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 above), 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 using the log records which were forced to disk by the commit. In the vernacular of Gray [5], item 1 above is the Write Ahead Log protocol (WAL) and item 2 is the two phase commit protocol.

TRANSACTION UNDO

The logic of action UNDO is very simple. It reads a log record, looks at the name of the action that wrote the record and invokes the undo entry point of that action passing it the log record. Thus recovery is 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 *backwards* undoing each action in turn. Given the log of a transaction T:

```
UNDO(T):
   DO WHILE (LOG(T) ¬= NULL);
   LOG_RECORD = LAST_RECORD(LOG(T));
   UNDOER = WHO_WROTE(LOG_RECORD);
   CALL UNDOER(LOG_RECORD);
   LOG(T)=LOG(T)'EVIOUS;
   END;
END UNDO;
```

Clearly, this process can be stopped half-way, returning the transaction to an intermediate transaction save point. Transaction save points allow the transaction to backtrack in case of some failure and yet salvage all successful work.

From this discussion it follows that a transaction's log is a push down stack and that writing a new log record pushes it onto the stack while undoing a record pops it off the stack by invalidating it (see Figure 5). 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 threaded together and anchored off of the transaction descriptor. Notice that UNDO requires that the log be directly addressable while the transaction is uncommitted. This is the reason that 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.

TRANSACTION SAVE POINTS

A transaction save point records enough information to restore the transaction's view of the RSS as of the save point. Also, the user may record up to 64k bytes of 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 point. For performance reasons, changes to cursors and locks are not incrementally recorded in the log. Otherwise, every read action would have to write a log entry. Rather, the state of locks and cursors is only recorded at intermediate save points. Further, the status of the locks of the transaction must be altered so that all locks held at the save point are held while the save point is valid. To see why this is necessary, suppose a read lock on a record were released by a cursor (when the cursor moved to a new record). If this record is subsequently deleted by another transaction one cannot restore the transaction to an earlier save point since the cursor cannot be reset to address the deleted record. (This problem is only an issue for programs which opt for less than the Degree 3 consistency lock protocol [10].) To avoid this problem, each save point marks all locks as locks which are held to the end of transaction. This allows transaction back up to 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 name of the top of the list. At back up to a save point, all locks subsequent to this lock are released.

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 on and off dual logging) 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 ideas like SYSGEN and cold-start.

A *monitor* process (task or virtual machine) is responsible for system startup, shutdown, 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 user's process, and then leave the system. In theory, 254 users may be joined to a system at one time.

SYSTEM CHECKPOINT

System checkpoints limit the amount of work (undo and redo) necessary at restart. A checkpoint records information on disk which helps to locate the end of the log at restart, and also a checkpoint forces all changes to disk 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. This optimum depends critically on the cost of a checkpoint. Hence 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 at which point no transactions are in progress and hence a logically consistent snapshot can be recorded. However, quiescing the system causes long interruptions in system availability (while the system is being quiesced) and hence argues for infrequent snapshots. This in turn increases the amount of work that is 'lost' in a catastrophe and hence the amount of work that must be redone. So system quiesce 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 *checkpoint* which is a snapshot of the system at a time when no RSS actions are in progress (an **action consistent state**).

This approach has the virtue that it does not require a system quiesce (which could easily take minutes) but only an RSS quiesce. 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' to allow checkpoints and user attentions to interrupt them).

Checkpoints are taken after a specified amount of log activity or at system operator request. Whenever the system is checkpointed, a checkpoint record is written in the log and then all log pages in the buffer are written to the log disk file. This implements the Write Ahead Log Protocol [5]. 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). This involves flushing the data base 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 [9]. The directory root is ping-ponged and duplexed on disk so that this process can tolerate failures during disk write. At restart the system will be able to locate the corresponding checkpoint record (see Figure 8).

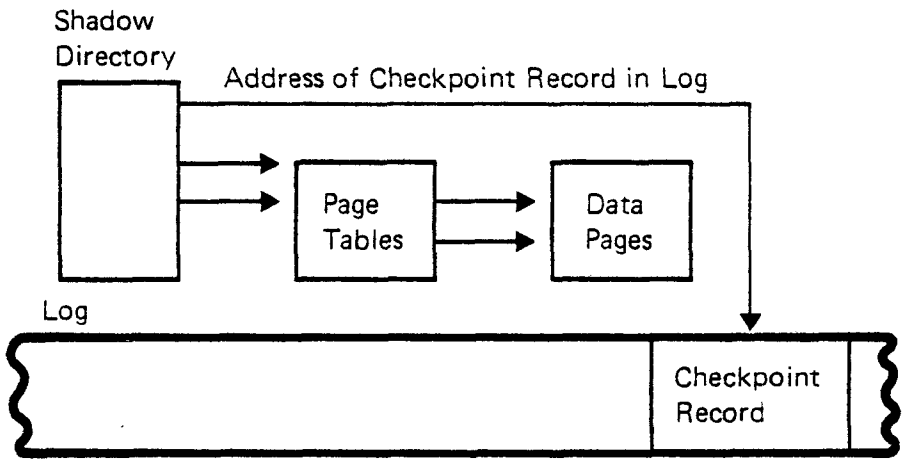

Figure 8. Map of directory root pointing at most recent checkpoint.

The whole checkpoint process is bracketed by an RSS quiesce. Originally the quiesce of the RSS was done using locking; each RSS call acquired the 'RSS' lock in shared mode and quiesce acquired the 'RSS' lock in exclusive mode. Performance considerations dictated that we special case this lock protocol with bit flags [11].

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 logically consistent version of the state can be constructed by undoing all updates (prior to time T) of transactions which were uncommitted at time T+E and then redoing the updates (after time T and before 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. So 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 8 shows is pointed at by the current directory). If there was no work in progress at 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. In this case, restart initializes System R and opens up the system for general use.

On the other hand if there was work in progress at the checkpoint, or if there are log records after the checkpoint record then this is a restart from a crash. Figure 9 illustrates the five possible types of transactions at this point: T1 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, and T5 began after the checkpoint and apparently never ended. To honor the commit of T1, T2 and T3 transactions requires that their updates appear in the system state (done). But T4 and T5 have not committed and so their updates must not appear in the state (undone).

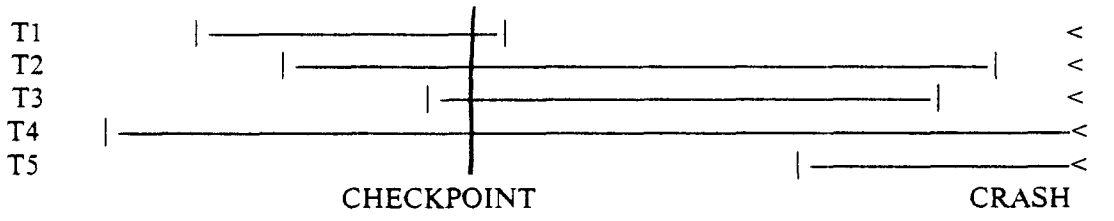

Figure 9. Five transaction types with respect to the most recent checkpoint and the crash point.

At restart, the shadowed files are as they were at the most recent checkpoint (call it time T). 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 T1 are in the state so it need not be redone. Hence only T2, T3, and T4 remain. T2 and T3 must be redone from the checkpoint forward, the updates of the first half of T2 are already reflected in the checkpoint. On the other hand, T4 must be undone from the checkpoint backwards. (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.)

After a crash, the system wakes up with amnesia remembering only the shadow versions of the shadowed files and the address of the most recent checkpoint record in the log. The end of the log is found by looking for the log page beyond the checkpoint with the highest log page sequence number. This along with the shadows of the logged files gives an action consistent system state at time T and a log that is current to time T+E. Restart uses the log as follows. The restart process never sees transactions of type T1. It reads the most recent checkpoint record and assumes that all the transactions active at the time of the checkpoint are of type T4. It then reads the log in the forward direction starting from the checkpoint record. If it encounters a BEGIN record it remarks 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 then reads the log backwards from the checkpoint undoing all log records of losers. Then restart reads the log forwards from the checkpoint redoing any winners. (This undo then redo order is critical.) Once this is done a new checkpoint is written so that the restart work will not be lost. The following is the basic logic of restart:

```
RESTART:
   DEDUCE WINNERS AND LOSERS FROM LOG;
   FOR EACH LOSER P ACTIVE AT TIME T;
      UNDO(P) FROM  TIME T;
   END;
   FOR EACH WINNER'S REDO RECORD AFTER TIME T ;
      REDO(action); /* i.e. put in ' new ' value*/
   END;
   TAKE CHECKPOINT;
 END RESTART;
```

Restart must be prepared to tolerate failures during the restart process. This problem is very subtle in most systems, but the System R shadow mechanism makes it trivial: System R restart does not update the log or the shadow version of the database until restart is complete. Writing a checkpoint record signals the end of a successful restart. Any failure prior to the writing of the checkpoint record will return the restart process to the original shadow state. Any failure after the checkpoint record will return the database to the new (restarted) state. Further, writing a checkpoint record is atomic and so there are only the two cases. (The checkpoint record is validated by a single disk write which also updates the shadow. The directory is duplexed and designed to withstand failures in the writing process.)

MEDIA FAILURE

In the event of system failure which causes a loss of disk storage integrity, it must be possible to continue with a minimum of lost work. It is important that the archive mechanism have failure modes independent from the failure modes of the on-line storage system. Using doubly redundant disk storage protects against a disk head crash, but does not protect against errors in disk programs or fire in the machine room.

The archive mechanism chosen for System R periodically dumps a transaction consistent copy of the database to magnetic tape.

In the event of a single media failure,

- If one of the duplexed on-line logs fails then allocate a new log and copy the good version of the duplexed logs onto the new log.

- If any other file fails, locate the most recent surviving dump tape, loading it back into the data base, and then redo all updates from that point forward using the log.

Although performing a checkpoint causes relatively few disk writes and takes only a few seconds, dumping the entire data base is a lengthy operation. Maintaining store quiesce for the duration of the dump operation is undesirable or even impossible, depending on the real time constraints imposed by system users. Lorie [9] describes a scheme based on shadows for taking a dump while the system is operating. Gray [5] 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.

MANAGING THE LOG

The log is a very long sequence of pages. Each page has a unique sequence number. In-progress transaction undo and system redo need to have quick access to the log but most of the log can be kept off-line or discarded. The on-line log file is used to hold the 'useful' parts of the log. It is managed as a ring buffer. LOG_END points just beyond the last useful byte of the log. CHECK_POINT points at the most recent system checkpoint. Clearly system restart must keep all records since the system checkpoint on-line (in support of transaction redo). Further, transaction undo needs to keep all records of incomplete transactions on-line (LOG_BEGIN(I) for active transaction I). Lastly, one cannot free the space occupied by a log record in the ring until the record is archived (this point is addressed by LOG_ARCHIVED). 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.

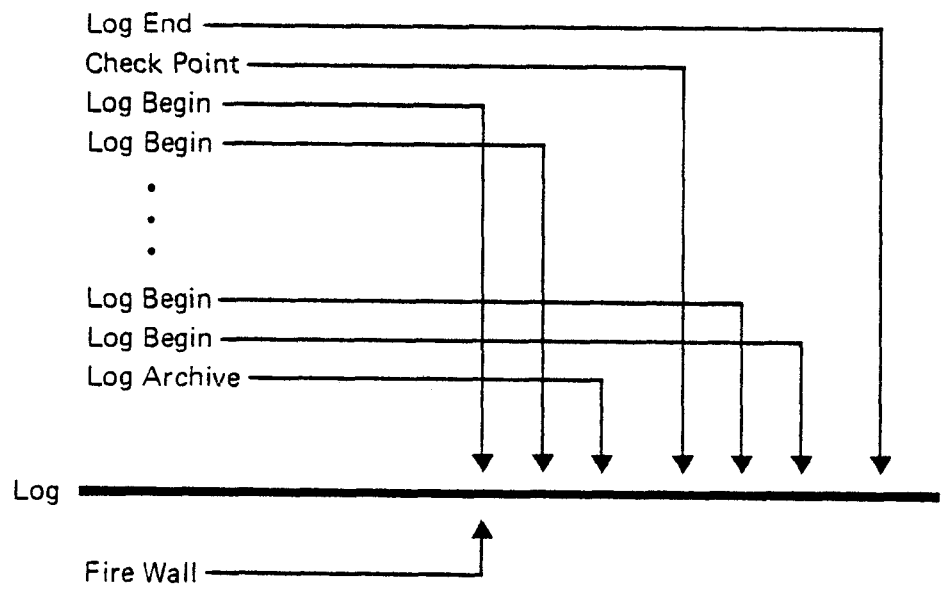

Figure 10. Bytes between FIRE_WALL and LOG_END are needed for system restart and hence are kept in the log ring buffer.

If the on-line ring buffer fills it is because archiving of the log is required or a checkpoint is required or 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 latter problem is solved by aborting very old transactions. For many applications, an on-line log of one megabyte is adequate.

Duplexing of the log is a system option which may be changed at each restart. Duplexing is transparent above the log interface; reads 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).

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 use *at least* a Degree 1 consistent lock protocol: lock all updates exclusive and hold all such locks until the update is committed or undone [10]. In fact, System R also supports Degree 2 and Degree 3 consistency and almost all transactions opt for Degree 3 consistency which gives complete isolation from concurrency anomalies [4].

A second issue is that transaction undo cannot tolerate deadlock (we don't want to have to undo undo's). Further, transaction undo need not rerequest locks it already holds. Undo may take

advantage of the fact that it only accesses records the transaction locked in the do step. (This is a consequence of Degree 1 consistency: all exclusive locks are held to 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 call (e.g. read locks on the root of a B-tree index). To deal with 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 doesn't produce unbreakable deadlock cycles (i.e. those containing only golden transactions), the additional rule is adopted that only one golden transaction can perform an RSS undo action at a time (and hence that no cycle involves two golden transactions). A special lock: RSS_BACKUP is requested in exclusive mode by each golden transaction before it begins the next 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.

EVALUATION

We were apprehensive about several things 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. Secondly we were apprehensive about the performance and complexity of such programs. Third, we were concerned that the added complexity might create more crashes than it cured.

IMPLEMENTATION COST

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

It is the case that writing recoverable actions (ones which can undo and redo themselves) is quite hard. *Subjectively*, writing a recoverable action is 30% harder and requires about 20% more code than a non-recoverable action. In addition, the recovery system itself (log management, system restart, checkpoint, commit, abort, ...) contributes about 15% to 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. So the marginal cost of implementing recovery is about 10%.

EXECUTION COST

Another component of cost is the instructions added to execution of a transaction (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% increased path length. There is also a fixed cost associated with recovery: the cost of periodic checkpoints and of shutdown and restart. Restart is quite fast, typically running at ten times real time. Hence if the checkpoint interval is five minutes, the system averages 15 seconds to restart. As described in the next section, checkpoint is I/O bound.

I/O COST

Yet a third component of cost is I/O added by the recovery system. Each transaction commit adds two I/Os to the cost of the transaction when the 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 all transactions are a few (50) RSS calls it constitutes a 20% 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 amortizing log I/O across several transactions 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 [9] are correct: a checkpoint requires several seconds of I/O after five minutes of work on a 100 megabyte database. This work increases with larger databases and with higher transaction rates. It becomes significant at ten transactions per second or for billion byte files.

SUCCESS RATE

Perhaps the most significant aspect of the System R recovery manager is the confidence it inspires in users. We routinely crash the system knowing that the recovery system will be able to pick up the pieces: all committed transactions will persist and all others will be aborted. 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.

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.

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 log. The rationale for the use of disk is that disks cost about as much as tapes (typically 30,000 dollars per unit if one includes controllers), but disks are more capacious than tapes (500 megabytes rather than 50 megabytes per unit) and disks can be allocated in smaller units (a disk based log can use half of the disk cylinders but 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.

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.

SAVE POINTS

Although we continue to believe that transaction save points are an elegant idea which is cheap to implement and easy to use, transaction save points are not used by or exposed by the SQL language. Unfortunately, the RDS implementors let PL/I do most of the environmental control (e.g. controlled storage). So the RDS processor does not know how to save its state and PL/I does not offer a facility which allows the RDS to reset its state even if the RDS could remember the state. Hence the RSS transaction save point facility is not used.

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.

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. The fact that restart always starts with an RSS-action consistent state is quite a simplification and probably contributes to its success.

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, one must be prepared to deal with the other two cases in some way.

One alternative is the Write Ahead Log (WAL) protocol used by IMS [12]. 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 idempotent: attempting to redo a done page will have no effect and attempting to undo an undone page will have no effect (this allows restart to fail and then retry as though it was a first attempt). WAL is extensively discussed by Gray [5].

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 re-write of the RSS is required to reverse the decision. Fortunately, the performance of shadows is not unacceptable. In fact for small data bases (less than 10 meagabytes) shadows have excellent performance.

Our adoption of shadows is largely historical. Lorie implemented a single-user relational system called XRM 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, notably QBE [13], the DataComputer [14], and the Lampson and Sturgis file system [15], 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*. Allocation units are typically a group of disk cylinders, called *extents*. If the file grows, new extents are added to the list. Such a file might be represented by ten extents and the corresponding descriptor might be 200 bytes. Accessing a page consists of indexing the extent table and then the extent to find the page.

By contrast, a shadow mechanism is much more complex and expensive. Each block of the file must have an individual descriptor in the page table. Such descriptors need to be at least four 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 house keeping items. Hence the directories needed for a file are about 0.2% of the file size (actually 0.2% of the maximum file size).

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 4K byte 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 [9] suggests a shadow scheme which maintains physical clustering within a cylinder). So it appears that shadows are bad for sequential and for direct (random) processing.

Shadows consume an inconsequential amount of disk space for directories (less than 1%). On the other hand, in order to use the shadow mechanism, one must reserve a large amount (20%) 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% shadow overhead or the operation must be able to tolerate several checkpoints (i.e. discard shadow) while it is in progress. This problem complicates archival recovery for the RSS.

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. (The data structure in [9] requires five such I/O but the RSS uses a more efficient data structure.) 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/O's.

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 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 6. If we had done this, the current and shadow directories for such files 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 for such files. 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 updates to indexes be explicitly written rather than being deduced from record updates). But the performance of WAL is better for *large shared databases* (bigger than 100 megabytes). Shadows seem to be excellent for medium to small files which do not support multiple concurrent updators, save points or incremental media recovery (i.e. do not require a log) and we would expect to continue to use shadowed files in such applications.

MESSAGE RECOVERY, AN OVERSIGHT

As pointed out by the example in Figure 2, a transaction's database actions and output messages must either all be committed or all 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 (e.g. IMS and CICS).

NEW FEATURES

We recently added two new facilities to the recovery component and to the SQL interface. First the COMMIT verb was extended to allow an application to combine a COMMIT with a BEGIN and preserve its 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. The locks are down-graded from exclusive to share 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 then processes the 'B's', then commits, then processes the 'C's', ... 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 [5]. A PHASE_ONE action was added to the RSS and to SQL to allow transactions to prepare to commit. This caused 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 [16].

At present, our major interest is in a distributed version of System R. As a prerequisite, this requires integrating System R with some network system or designing a network system from scratch. We are considering both alternatives. We expect that either approach will have implications for the design and function of the recovery system.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Astrahan, M.M., M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, V. Watson, 'System R: a Relational Approach to Database Management', ACM Transactions on Database Systems, Vol. 1, No. 2, June 1976.

[2] Chamberlin, D.D., M.M. Astrahan, K.P. Eswaran, P.P. Griffiths, R.A. Lorie, J.W. Mehl, R. Reisner, B.W. Wade, 'SEQUEL 2: A Unified Approach to Data Definition, Manipulation and Control', IBM J. Res. Develop., Vol 20, No. 6, Nov. 1976, pp. 560-576.

[3] Gray, J.N., V. Watson, 'A Shared Segment and Interprocess Communication Facility for VM/370', IBM San Jose Research Laboratory Report: RJ 1579, May 1975.

[4] Eswaran, K.E., J.N. Gray, R.A. Lorie, I.L. Traiger, 'On the Notions of Consistency and Predicate Locks in a Relational Database System,' CACM, Vol. 19, No. 11, Nov. 1976, pp. 624-634.

[5] Gray, J.N., 'Notes on Data Base Operating Systems', *Operating Systems - An Advanced Course*, R. Bayer, R.M. Graham, G. Seegmuller editors, Springer Verlag, 1978 pp. 393-481. Also IBM Research Report: RJ 2188, Feb. 1978.

[6] Nauman, J.S., 'Observations on Sharing in Data Base Systems', IBM Santa Teresa Laboratory Technical Report: TR 03.047, May 1978.

[7] Bjork L., 'Recovery Scenario for a DB/DC System,' Proceedings ACM National Conference, 1973, pp. 142-146.

[8] Davies, C.T., 'Recovery Semantics for a DB/DC System,' Proceedings ACM National Conference, 1973, pp. 136-141.

[9] Lorie, R.A., Physical Integrity in a Large Segmented Database. ACM Transactions on Database Systems, Vol. 2, No. 1, March 1977, pp. 91-104

[10] Gray, J.N., R.A. Lorie, G.F. Putzolu, I.L. Traiger, 'Granularity of Locks and Degrees of Consistency in a Shared Data Base', *Modeling in Data Base Management Systems.* G.M. Nijssen editor, North Holland, 1976, pp. 365-394. Also IBM Research Report: RJ 1606.

[11] Blasgen, M.W., J.N. Gray, M. Mitoma, T. Price, 'The Convoy Phenomenon', ACM Operating Systems Review, Vol. 13, No. 2, April 1979, pp. 20-25.

[12] 'Information Managagement System/Virtual Systems (IMS/VS), Programming Reference Manual' IBM Form No. SH20-9027-2, sect. 5.

[13] 'Query By Example Program Description/Operators Manual', IBM Form No. SH20-2077.

[14] Marill, T., D.H. Stern. 'The Datacomputer: A Network Utility', Proc. AFIPS NCC, Vol. 44, 1975.

[15] Lampson B.W., H. E. Sturgis. 'Crash Recovery in a Distributed Data Storage System', To appear in CACM.

[16] The prepare action was implemented by Tom Szczygelski.