SIMULATION OF A SELECTIVE ROLLBACK AND RECOVERY METHODOLOGY

by

Kirk Allen Norsworthy

B. S., University of Kansas, Lawrence, Kansas, 1975

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements of the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1978

Approved by:

[illegible]
Major Professor

Document
LD
2668
.R4
1978
N66
c. 2

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

Special thanks to Dr. Fred Maryanski, Dr. Beth Unger, and Dr. Dave Gustafson for all their help and guidance with this project. Also, I wish to thank Dave Schmidt, Robert Young, and Jim Ratliff for answers to numerous queries. This work was sponsored in part by the U.S. Army Research Office Grant No. DAAG 29-78-0018.

1.0 INTRODUCTION

In the past few years, information management systems have rapidly established themselves as the best form of computer-based information storage for both business and government. The advantages are numerous, but principally they allow for multiple applications to share data without the need for redundancy. Information management systems offer the best possible utilization of system resources when understood and used correctly. Accompanying the growth of information management systems has been a parallel growth in computer networks. Together, information management systems and computer networks, have brought about a possibility of sharing resources among different computing installations. Eventually, the integration of computing resources may extend to the interconnecting of dissimilar computer architectures into distributed data sharing.

Before distributed data base processing can be fully achieved, several areas of technological intergration must be solved. These areas include data and language translation facilities, multiple data base synchronization, deadlock, and concurrent data base access. It is these areas and more specifically data base deadlock, which provide the motivation for this research. This paper addresses the deadlock problem by simulating a proposed rollback and recovery algorithm that could be coupled with a deadlock detection scheme. From this model, statistics on

CPU overhead, data base accesses, and the number of data base operations rolled back during the recovery, were recorded. The statistics form a basis for the projection of the efficiency of the rollback and recovery algorithm.

This paper begins by surveying the existing literature pertaining to the areas of rollback and recovery. Special mention is made of those studies which pertain to recovery procedures in a distributed data processing environment. Also, examples of the current commonly used deadlock detection and prevention algorithms are examined. A high level description of the Selective Rollback and Recovery algorithm and its operation during rollback is the basis for the second chapter. A description of the selective rollback and recovery simulation and its parameters is followed next with a discussion of the results that were recorded for the different job environments. Lastly, a summary of the experiment and its impact on the area of distributed processing is evaluated.

2.0 BACKGROUND LITERATURE

Over the years, a great amount of research has been done in the area of integrity of information managemnt systems. Most work in this area has been devoted to the study of deadlock. A deadlock situation occurs when all of the following conditions are satisfied:

(A) A resource is granted exclusively to one process.

(B) Resources are not released in case of an incremental request.

(C) Resources can not be pre-empted temporarily from a process and granted to another process.

(D) Processes are permitted to wait for the release of resources. [1]

The resolution of deadlock situations has been the concern of software operating systems writers for some time. Their approaches for handling deadlock have focused on three basic strategies; eliminating or allowing deadlock, preventing deadlock, and detecting deadlock.

2.1 Eliminating or Allowing System Deadlock

It is not always necessary to employ an explicit mechanism or procedure for avoiding deadlock in a multi-user environment. A system can be specified as a multi-user on-line read only environment. Any requests for updates to the system are written to a tape from which a batch update is made at a convenient time. By not allowing users the option of updating on-line, the problem of deadlock is eliminated through system design.

In some data processing environments, the information management systems have a low volume of shared data usage. Most processes in these systems access their own set of files repeatedly, thus reducing the potential for deadlock dramatically. In this type of environment, the problem of deadlock is handled by simply ignoring the problem. Deadlock discovery in this system is made by a user who waits an extraodinarily long time for a system response or an operator who may notice two or more tasks waiting to be executed after the system has been cleared of waiting jobs. Allowing or designing away deadlock has the advantage of the lowest possible overhead and simplest implementation. This approach does have the disadvantage of being increasingly less effective as the trend toward shared data and need for accurate on-line information grows.

2.2 Deadlock Prevention

Deadlock prevention is another way of handling the problem of deadlock. Prevention of deadlock is usually handled thru a method of prior resource declaration or a group set method. The prior declaration of required resources is considered the simplest of the two methods.

Prior declaration of resources in a distributed data base network requires that all requested files for each new process be declared and sent to a monitor node. The monitor node uses a fixed path of examination for determining if a deadlock situation exists after granting the requesting process the files it needs. The fixed path of examination is determined by assigning each node in the computer network a unique number. The list of file(s) requested by the new process are passed sequentially around the examination path to each node. Each node in its turn considers the request independently of the other requested files and reports to the monitor node whether the request can be accepted without resulting in a deadlock.

Each node's examination is further improved by determining the routing of the fixed path of examination based upon the current file utilization (the usage rate and the length of each file transaction for the process at each node). The higher the node's utilization and file storage capability, the lower its fixed path of examination number.

In a centralized data base system, the requests for files is checked for deadlock prior to allowing the process access to any files. A process is delayed until all of its requested file(s) can be granted. A distributed data base system first examines the requested files to determine if all the files are local files (located at the network node where the request is initiated). If the requesting process asks for all local files, then the system checks for deadlock in the same way as a centralized data base. If some of the files are located at other network nodes, then the request is routed around the network and the process waits until an affirmative response is sent from all nodes.

The preallocation method of deadlock prevention is attractive for its ease of implementation and low overhead. Inter-computer communication is usually the most expensive portion of any deadlock prevention scheme for a distributed network. The preallocation method is able to keep the inter-computer communication at a minimum because it only occurs when remote files are requested. The major drawback hampering the effectiveness of this method is that the declaring of files prior to task initiation can result in an extensive delay. If a task accessing a remote file is blocked, system performance could be seriously hurt by forcing the task to have access to all the files it will require. [2]

A second form of deadlock prevention is the process set method. This method grants a process´s request immediatley unless doing so will result in deadlock. The system collects the file(s) requested and initiates the process. The process is allowed to proceed until it requests a file (exclusive access). The access request results in a system check for a potential deadlock situation with other processes. If no potential deadlock situation exists the process continues .

Deadlock checks are made through the maintenance of process sets. A process set is a list for each active process indicating the active processes sharing resources with it. A new process set is formed whenever a new process is initiated. All the current process sets that have file(s) requested by the new process are combined with the new process to form a new process set. If the new process requests files that are not included in any of the active process sets, then the new process forms a process set by itself.

The advantage of using process sets is that after allocation of a process to a set, the progress of the process is independent of all processes in the other process sets. The process set results in better system utilization through increased multiprocessing. Also, system performance is enhanced because a process does not require all the files it will access prior to execution and because the computation required to determine deadlock is reduced with process sets.

In a distributive environment a process set may include processes and files from several network nodes and be located at remote network nodes. The system transmits a request for file(s) to the network node containing the process set pertaining to the request. File access is granted in the same fashion for a distributed network as a centralized system. [3]

2.3 Deadlock Detection

Deadlock detection allows user freedom in requesting files. Deadlock detection is accomplished via a similar method to that used in deadlock prevention. The system maintains two lists, a list of active processes and a list of files requested by the active process. Each list is dynamic and has a pointer associated with each element of the list. When a process requests a file, the system sets a pointer from the file to the requesting process´s pointer address, if the request can be granted. When a request cannot be granted, the system sets a pointer associated with the requesting process to the requested file´s pointer address.

Deadlock is detected by traversing the file and the process lists until either an open process pointer is found or return to the requesting process is made. When return to the requesting process occurs, a deadlock situation exists. At detection of a deadlock, the system returns a process to its initial state and restores the files controlled by the terminated process.

The deadlock detection mechanism uses a monitor node for implementation in a distributed network. The monitor node updates the process lists, file lists, and accepts information on each process and file. In some cases, further efficiency is realized by separate monitor nodes for the process list and the file list. [4]

Advantages of using a deadlock detection mechanism include its ease of implementation and no requirement of prior declaration of required files. Also, the overhead is approximately the same as deadlock prevention when the need for a rollback and recovery method is considered. A deadlock detection mechanism works well for computer environments where deadlock is rare. The detecting of deadlock encourages system thruput by allowing processes to proceed without waiting. The overriding disadvantage of a deadlock detection system is the uncertainty of a system's rollback algorithm to insure integrity of the data. Rollback systems used today are also costly in system down-time, which in a multi-user on-line environment can result in user idleness and dissatisfaction. Deadlock detection systems may become more popular if the confidence in rollback can be improved.

2.4 System Rollback and Recovery Methods

If data integrity is to be fully protected, then a system restart and recovery methodology must be implemented. Restart and recovery in almost all systems begins with a system dump, copying all of the data base on to secondary storage backup (usually magnetic tape or disk). System dumps are usually made at the beginning of each processing day, but may occur less often if the size of the data base makes a dump prohibitive. System dumps are sometimes made during the daily processing on a predetermined basis. These dumps are called system checkpoints and are usually done on a fixed interval schedule or when the data base reaches a point of system inactivity.

Another system mechanism used in most rollback and recovery methods is a journal file (sometimes called an audit trail or log file). The journal file is a chronological list of data base activity automatically written to secondary storage (usually blocked and written to tape). When modifications of the data base are made, a corresponding journal file entry is made that comprises a user's identification, the nature of his data base modification, description for identifying the file update and the file image. The file images can be beforeimages (a copy of the modified files prior to a data base alteration), afterimages (a copy of the modified files after a data base alteration) or both.

Rollback of the data base modifications must be possible for both a deadlock prevention and deadlock detection scheme. The need for rollback and its associated recovery is necessary for several hardware and software problems:

(A) When an operator initiates a rollback of the system because of a system timer interrupt or a probable software loop error.

(B) When the hardware fails for some reason.

(C) Or when the processes reach a state of deadlock and the data base needs to be returned to a point where processing can begin.

The general strategy for recovery from data contamination (ie. tainted or invalid information) is dependent upon the type of system failure. System failures have been classified in a number of ways. The most commonly used terms are _hard_ and _soft crashes._ [5]

Hard crashes are damages to the data base that result in physical conditions that leave it unreadable. Crashes on a disk storage and hardware failures are forms of hard crash. Software too, can be responsible for hard crashes. The general rule to follow in handling hard crashes is to restore the data base with the most recent system dump and then follow it with one of the following two methods:

(A) Ask system users to reenter all data and commands from the point of the system dump.

(B) Reconstruct the data base by copying after image journal file entries to execute the process again. Repetitive output messages are suppressed during the recovery.

Soft crashes in the system result from semantic and structural problems. A structural problem is one that results in inaccurate structure of the data. A possible structural problem is the incomplete linking of lists. A semantic problem is a sequence of data base commands that do not finish their intended operation successfully. Semantic failures leave a semantic inconsistency in the data base. A semantic problem could be a discrepancy between the value within the NO_RECORDS field in a file and the actual number of existing records.

Soft crashes are usually correctable through automatic system recovery techniques. The two most often used methods of recovering from soft crashes are:

(A) A copy of the last system checkpoint is used in conjunction with the beforeimages stored on the journal file. The beforeimages are copied into the data base until the last completed process is finished. At this point, users resume execution from this point by making any neccessary inputs.

(B) The data base is backed up to a point prior to the insertion of the system error by inserting the beforeimages into the data base in reverse chronological order. After the error condition is corrected the processes are allowed to continue their execution while suppressing any duplicate messages.

2.5 System Granule Locking

Computer systems are usually considered to be static in regards to their actions on data in the data base. However, this is not the case if we consider system actions as interrelated transactions. Consistency is assumed to be constant throughout the processing when it actually contains a period of inconsistency. The use of electronic funds transfer in the banking industry has this problem when updating accounts. An inconsistency exists because as an account is debited, there is a period where the second account has not as yet been credited. The actions of processing data of the data base from a consistent state into a new consistent state are called transactions. Transactions that are run in a concurrent environment face the simultaneous execution and interleaving of data base requests from several transactions.

To preserve a consistency in the data base during concurrent operation, the data base uses a granule locking protocol. A granule, is a physical data base size that is normally system dependent. The problem of temporary inconsistency in the data base is inherent in distributed systems. However, the problem does not neccessarily mean that the inconsistency should cause a serious problem.

One method for avoiding inconsistency in the system is to run transactions serially and avoid any problems from transactions running in parallel. The loss in system performance from blocking concurrent operations makes this solution unacceptable. Another method is to divide the data base into logical subsets (partitions) and allow the transactions to proceed only if it accesses a subset that is not currently being used by a transaction. Transactions that use the same subsets are scheduled serially to avoid problems. The problem with this method is that it is difficult to examine a transaction and decide which data base subsets it will use.

The best method of insuring data consistency is to lock all granules that a transaction accesses. This locking works well for a centralized data base system, but suffers numerous problems in a distributed data base environment. Many times in a data base it is desirable to lock all entities with a given value (ie. "key addressing", lock all the employees with a DEPT_NUM EQUAL 10). If we consider the relational data model, we would be locking all the tuples (records) that contain a match in one of their fields with the value of the key. Tuple locking can be very costly in terms of system overhead and results in large portions of the data base being locked. An alternative method, is to lock a tuples relation (row) or domain (column) whenever a match is found. This method results in many data base locks and can limit system concurrency.

A proposal was made by Gray, Lorie, and Putlolu, of IBM, [6] to find an efficient method for the locking of granules. They suggest that the locking take place at several levels of granularity (different logical sizes of locking) in a hierarchical fashion. A hierarchy would resemble a tree structure where the node representing the "root" has branches in ever smaller detail descending from it. A locking protocol for the data base will include a provision for the creation and destruction of sets, files, and other portions of the data base. Implementation of this proposed system is called an _index_ _interval_ _lock_. Instead of locking an entire record, the locking is done if possible on key values. By locking on key values, the system is in a sense locking at the field level of a record since the index of the key can point to all records that contain the key value as a field.

To implement this system, the data base is broken into several areas. Each area contains a set of relations, their indicies and tuples, along with a catalog of the area. Each tuple has a unique tuple identifier (key) which is used for direct addressing of the tuple. Each tuple identifier maps to a set of field values. In this way, the tuple identifiers can directly access the tuples and the indicies can give fast associative access to field values and their corresponding tuples.

Colliat and Bachman [7] of Honeywell describe a system for distributed data base management that insures data integrity. A process's request is divided into execution at the node of the request (local node) and distant nodes. After completion of access at the local node, a local journal file entry is made with a mark called "commit-pending". At this point, the distant node is notified of the successful local node completion. The distant node follows the same procedure and marks its local journal file in the same fashion. When the local node recieves confirmation that the distant node has completed it records a journal entry mark, "completely committed". The distant node responds similarly by recording in its local journal the same entry. In this method for two nodes, rollback takes place if one of the nodes sends a message requesting rollback and that node has not recorded a "completely committed" mark.

For the Colliat and Bachman method to work for a network greater than two nodes, a hierarchical precedence relationship between the nodes must be established. A node only records a "commit-pending" mark after recieving acknowledgement from its descendent nodes. It must then request "completely committed" acknowledgement from its parent node if there exists one. After a node recieves acknowledgement from its parent node, the node completes its work and records a "completely-committed" mark followed by acknowledgement to its descendents. A process is rolled back only when a request is made by a parent node and never after recording a "completely-committed" mark.

The above method is based upon a considerable amount of message protocol. This protocol is costly and must guarantee that a message path represents a valid hierarchical relationship. Similarly, each node must never have more than one parent node and there must never be any hieararchical loops established. A hierarchical loop would be where a root node is dependent upon a descendent node.

The authors of this method acknowledge that they do not know of any existing algorithm that can guarantee against a loop in the hierarchy relationship.

A proposal for quicker recovery is the idea of a <u>differential</u> <u>file</u>. A differential file system as proposed by Severance and Lohman would the maintence of large data bases. Similar to an errata list that is maintained for a book, the idea is to collect changes to the data base over a period of time and then apply them at one time to reduce the overhead involved. It is less expensive to keep a differential file as an errata list than to directly access and update the data base after each modification transaction. Other savings are realized from reduced system complexity, maintence overhead, and storage costs. The differential file collects all pending changes to the data base and is searched as a first step by any data base request. Use of the differential list, means an increased cost in access time, but this may be offset by reduced update time. The differential file is incorporated into the data base when it reaches a sufficiently large size to make the update cost effective.

Severance and Lohman [8] report a number of benefits to be derived from a differential file structure:

(A) Reduced Data Base Dumping Costs; In almost any data base, a recovery from a hard crash requires that a dump be copied into the data base and journal file afterimages be rerun. In a very large data base, this "quick recovery" could easily take hours. However, a differential file and its associated entries could be rewritten to the data base in just a few minutes.

(B) Facilitates Incremental Dumping; Dumping very large data bases at one time can be prohibitive. An alternative is to dump physical sections of the data base at different intervals. This is easily done by simply searching the differential file for the data within the section and copying it onto the backup storage.

(C) Realtime Dumping and Reorganization with Concurrency; By dumping only a differential file, concurrent users can be blocked for a very short time or by building a seperate differential file during update, concurrent use can continue unhampered. Reorganization of files can be done with a differential file that is copied after reorganization is complete.

(D) Quicker Recovery From Hard and Soft Crashes; A differential file can be used as an on-line journal file for rapid recovery and rollback.

(E) Reduces Serious Data Loss; Because dumps and changes are made in a small area of secondary storage, several possible advantages arise:

(1) Critical area data can be allocated to a more reliable system device than the main file.

(2) Exposure to the critical area is minimized.

(3) Critical areas may be duplicated with little system overhead.

(F) Memo Concept; Used in banking, the concept of a memo uses a scratch copy of the data base throughout the day. Updates are applied to the scratch copy and kept in a differential file. At the end of the day, the differential file updates that are correct are copied into the data base.

(G) Software Development; In software development where it is infeasible to make a duplicate data base copy, differential files can be particularly useful and effective.

(H) Simpler Software; Since the main data base is only read, data base accesses can be considerably easier with concurrent update and deadlock problems avoided. The data base is stored more efficiently because no free space for growth must be allocated. Predictions are that the cost of read/write memory will continue to escalate but, the cost of read only memory will continue to fall. This indicates that systems based on a read-only main memory will become increasingly attractive.

Several data base simulation studies have been made to examine problems of deadlock, rollback, and recovery.

J. E. Shemer and A. J. Collmeyer [9], simulated the deadlock problem based on a granule locking protocol at the record level. They ran the simulation for 10,000 transactions and measured the frequency of system deadlocks. The simulation used a varied number of users and a data base size of 100 and 200 records. The simulated transactions were divided into readers (no modification to the data base) and writers (modification of the data base) and were applied to the data base in a random fashion in different percentages. Users would perform a series of consecutive transactions on the data base for each access.

Simulation results showed that for twenty users in a 100 record data base with seventy percent write transactions, the system overhead is still acceptable. As the data base size increases the proportion of deadlocks in the system decreases.

A data base concurrency simulation was conducted by Munz and Krenz [10], to answer the following questions:

(A) How frequent are deadlocks in certain environments?

(B) How does the concurrency relate to the locking of granules?

(C) What types of deadlock criteria are useful in rollback?

(D) What is the optimum number of system checkpoints?

When a task siezes a data set exclusively it must release it before it can sieze another. When a data base is divided into an increasing number of data sets, the waiting time of the processes goes down in an exponential fashion. The simulation showed that the usual method of handling a deadlock (rollback the process causing the deadlock) was not the best. The following were the best methods for rolling back a computer system experiencing occasional deadlock:

(1) Rollback the task which has made the smallest number of exclusive requests.

(2) Rollback the least expensive task.

(3) Rollback the task which has the fewest number of granules.

In cases where the data base system experiences frequent deadlocks, rolling back the task which will lose the least amount of CPU is the best method. The simulation shows that frequent checkpointing was little better than rolling the errant task back with no system checkpoints at all.

A paper by Ries and Stonebraker [11] looked into what size of granule locks should be used on the data base. The tradeoff in locking a data base is that locking large granules inhibits concurrency, but minimizes locking overhead. Locking many small data base granules encourages concurrency but, at an added system overhead. Findings by Ries and Stonebraker indicate that a small number of granules are sufficient to allow enough concurrency for efficient system thruput. The cost overhead in locking a large number of granules far outweighs the savings from additional concurrency. Ries and Stonebraker go on to mention that the model is designed to favor small numbers of granules. For almost all cases simulated, ten granules on the data base are apparently sufficient.

Predicate locking, the locking of a logical subset of the data base, is more costly than granule locking. However, results from the simulation indicate that with only a few granules in the data base the added system cost can be economically absorbed by using predicate locks, the system needs to only maintain a few locks.

A paper by Maryanski and Fisher [12] makes use of a rollback and recovery algorithm that guarantees total data integrity. The algorithm is designed to rollback all possibly contaminated data through a series of dynamically maintained lists. The rollback and recovery algorithm is explained in the next chapter and a simulation using this algorithm is reported in a later chapter.

3.0 Selective Rollback and Recovery Algorithm

The approach to solving distributed data base problems are being judged by their cost competetiveness to centralized data processing. Before any real acceptance can be achieved, the performance problems of reliable rollback and recovery must be overcome. If this and other problems can be solved many of the benefits associated with distributed processing can be realized. An algorithm outlined by Maryanski and Fisher [13] addresses the problem of rollback and recovery in a distributed environment.

A performance advantage of this algorithm over other proposals is that commands used in recovery are the inverse of the data manipulation commands which made the modification. The inverse of any data manipulation command used in a data base operation, is applicable for any data base system. Inverse data base operations are usually in the data base language, providing efficient storage of the rollback algorithm and the associated information.

The algorithm is based on a group of lists that are maintained by the system. The interaction of a task with any other tasks is reflected in a dynamic list for each task called a potential shared data list. The potential shared data list is generated when the sub-schema for a task is constructed. Each task intersects its sub-schema with the sub-schema for all currently active sub-schemas. The result is a global list containing the names of the tasks and their granules that had intersections with the task's subschema. The potential shared data lists are maintained by sending a message to all active computer tasks whenever a task enters or leaves the computer system.

A system journal file is kept of the data base accesses and the information neccessary to effect selective rollback and recovery. The form of a journal file entry is displayed in Table 3.1. The journal file records the beforeimages of the data base modifications prior to the modification. The journal file requires the following fields for each entry:

TABLE 3.1

<u>Journal File Entry Format</u>

JOURNAL TYPE

*

TIME STAMP

*

TASK NAME

*

TASK GRANULE

*

SET MEMBERSHIP

*

DATABASE COMMAND

*

POINTER LINK

(A) A flag field indicating the nature of the journal entry (was the entry a rollback entry or a data base command).

(B) A time stamp field indicating the relative system clock time or the chronological order in which the data base transaction took place.

(C) A name field of the task involved in the data base operation.

(D) A field indicating the granule of the task that is involved in the data base operation.

(E) A field indicating the set membership modification introduced by the data base operation.

(F) The data base manipulation command.

(G) A field with a pointer to the area of memory where the before image of the data base is stored.

(H) Pointers that link the journal entries together into a list.

A list is maintained for each CODASYL type of set in the system. The set lists contain the task and record entries for the records connected into the set. When a data manipulation command connects, orders, or disconnects a record from a set, the corresponding set list is interrogated and updated.

3.1 Rollback

The secondary rollback list, rollback file, task list and update lists are all part of the rollback procedure. When an application task in the system terminates abnormally, its effect if any must be removed from the data base to insure data integrity. In an on-line computer system it is possible that data written by the terminating application task may have been read and used by other tasks sharing the data. If the tasks unknowingly use the bad data to make modifications, it is possible that a series of bad reads and writes can occur in the data base. To offset this result a rollback of data base modifications is necessary to restore the data base to a state of data integrity. To preserve the data the selective rollback and recovery system rolls back only tasks which could have had interaction with the terminated task. This method is an effort at confining the rollback to only those tasks which have used erroneous data. Unlike the method of rolling back the entire data base, the data base is not blocked to users who are not involved in the rollback.

Selective rollback is initiated when an application has terminated abnormally or a deadlock situation is found to exist. The host processor or front-end processor sends an inter-computer communication message to all backend processors notifying them that a task has terminated improperly. The host processor provides the backend processors the errant task's name, its potential shared data list (usually kept dynamically in the system library), a list of parameters open for update, and the time of the task initiation. The rollback procedure follows the following steps:

1. The initiation time of a task is derived by reading the journal file backwards to the location of the last RESTART entry for the terminated task. A RESTART entry identifies a stable point at which a task can be restarted (such as a system checkpoint). If no RESTART entries appear for a task after initiation, the task will be rolled back to initiation. All backend processors refuse any data manipulation requests that access areas of the data base not listed in the update parameters. The next operation is the initializing of the task list to null. The task list is used to record the names of all tasks rolled back so that multiple rollback of the same task is avoided.
2. From the point of the last terminated task's RESTART entry, the journal file is read forward until the point of system termination. The following operations are performed:
   - (A) The secondary rollback flag is initialized and the update list is initialized.
   - (B) If the journal entry is an update for the task being rolled back (the terminated task the first time), called the primary rollback task, an entry is made

in the update list consisting of the record whose contents or set occurrence has been modified. The secondary rollback flag is set and the journal entry is copied into the rollback file.

(C) If a journal entry for the primary rollback task alters a granule occurrence previously inserted into the update list, the journal entry is ignored. This action insures that each updated granule is rolled back to its earliest correct value in the rollback procedure.

(D) If a journal entry for a task other than the primary rollback task occurs and that granule exists in the potential shared data list of the primary rollback task, and the task references a record or set occurence for which an update list entry exists, then the task must be rolled back. To insure against the task operating with incorrect data, the task's name and the time of the entry are written in the secondary rollback list.

(E) All entries encountered in the journal file that already have entries in the secondary rollback list are skipped.

(F) All commands of the primary rollback task that do not update the data base are skipped.

(G) All commands of a task other than the primary rollback task that do not have entries in the potential shared data list of the primary rollback task are skipped.

(H) Any task already occurring in the task list is skipped .

3. When the journal file has been processed to the point of termination, the rollback file entries are processed. Effects of the rollback file entries is nullified by using inverse commands. As an example, a STORE command is nullified by a ERASE command.

4. When all rollback file entries have been processed, a task on the secondary rollback list is declared the primary rollback task. The task is removed from the secondary rollback list and the journal file searched for the first RESTART command prior to the time the incorrect operation was logged in the secondary rollback list. A message is sent from the host processor to the backend processors indicating the task now being rolled back and its parameters.

5. Rollback of the tasks proceeds until no more tasks exist in the secondary rollback list. Rollback in a distributed data base can proceed in a concurrent fashion, but the algorithm does not allow for duplicate data base copies.

A note should be made about the data base administrator´s role in this recovery method. Because of the potential for a great deal of rollback to occur in a highly integrated data base, operating on the contents of all data items in a record can be costly.

It is the role of the data base administrator to identify only those data items that are critical to the data base. These critical data items should be the entries that are intersected between tasks and included in the potential shared data lists.

Two methods suggested for identifying critical data items are:

(A) The maintenence of a data dictionary of non-critical data items or critical data items for each task.

(B) A mark field on each data item with a flag set indicating if the data item is critical.

4.0 Selective Rollback and Recovery Simulation

The selective rollback and recovery algorithm was simulated through the implementation of a CODASYL type data base. The data base system was written in PL/1 and designed to accept CODASYL type input commands. The data base management commands take the form:

```
COMMAND TASK_ID RECORD_ID.
or
COMMAND TASK_ID RECORD_ID = [SET OR VALUE].
```

The input command used is dependent upon the CODASYL operation.

The simulation parses the input command and calls the appropriate command subroutine. This subroutine calls a subroutine which logs the input command in the journal file and then performs the required data base operation. As the model completes a command , statistics are recorded on the cumulative number of accesses to the data base files and a system time stamp is recorded. The system time stamp is a recording of the computer system's 24 hour internal clock. The statistics recorded during the simulation are summarized in Table 4.1.

4.1

Simulation Statistics Recorded

PSDB	Potential shared data list access during data base commands.
PSRB	Potential shared data list access during data base rollback.
LDB	Journal file access during data base commands.
LRB	Journal file access during data base rollback.
SDB	Set list access during data base commands.
SRB	Set list access during data base rollback.
RDB	The number of data base records accessed during data base commands.
RRB	The number of data base commands rolled back during rollback.
STDB	The number of commands that result in the storing of information by the journal file.
STRB	The number of accesses to information stored by the journal file during the rollback.
SRRB	Secondary rollback list accesses.
RFRB	The number of retrievals of the primary rollback file.
URB	The number of retrievals of the update list.
TL	The number of accesses to the task list.

*** The retrievals of a list involves bringing a list into memory and operating upon it.

The first operation in the simulation is the initialization of the data base. The data base is built around a PL/1 implementation of based variables. Each based variable structure represents a single task and includes the following:

(A) The task name.

(B) The records associated with the task.

(C) The task's potential shared data list (task and record).

(D) A count of the number of potential shared data list entries for the task.

After the data base is initialized, the potential shared data list for each task is established by intersecting the subschemas of all the tasks and recording the matches. At the same time, all lists used in rollback and data base operations are established. When a rollback command is encountered, the previously described selective rollback and recovery algorithm is initiated.

The simulation processed 100 CODASYL type commands in percentages of 30, 50, and 70 percent writers. For consecutive executions of the simulation, the job stream was modified by randomizing the order of the six consecutive transactions executed by a task. The job stream was further randomized by changing the order in which the tasks executed. Data sharing in the data base was simulated for three seperate cases:

(A) No Shared Data - The tasks have no data in common (each task's potential shared data list is empty).

(B) Bi-task Data - A task shares fifty percent of its data with one other task and shares no data with all other tasks.

(C) Common Shared Data - All tasks share one half of their data in common with all other tasks (each task's potential shared data list contains one half of the records of all the other tasks).

For each of the three cases of data sharing, a data base of 200 total records was simulated with the following combination of parameters:

Users ... Each case was run with 8 tasks / 25 records, 12 tasks / 17 records, and 16 tasks / 13 records.

Writers ... Data base modification commands were 30, 50, and 70 percent of the input command stream.

Rollbacks ... Statistics were recorded for simulations of two, one, and no rollbacks.

System time stamp recordings were taken for the potential shared data list creation, data base command stream execution, and total rollback execution time. The amount of time required to process the data base commmand stream was used as a comparison against the total rollback execution time. The simulations for each job configuration (a minimum of three for each) compared the rollback time against the job command time to calculate a percentage rollback time to execution time. These percentages were averaged and multiplied by a deadlock factor derived from a simulation by Shemer and Collmeyer. The deadlock factors of the study are in Table 5.1. The simulation figures measure the predicted frequency of deadlocks encountered for a shared data base of 200 records under different job stream mixes.

5.0 Results and Analysis

The simulation was run on the ITEL AS 5 system under the various parameter settings. The CPU figures for the program execution varied from a low of 5.62 seconds for 8 tasks in a thirty percent writers environment with no rollback, to that of 18.44 seconds for a simulation of 16 tasks in a seventy percent writers environment with two rollbacks. The average rollback time as a percentage of the command stream execution time for all similar simulations,

AVG_RBT = (([JSE(1) / RET(1)] + .. + [JSE(N) / RET(N)]) / N)

AVG_RBT	The average time the rollback took with respect to the job stream execution time for the simulation.
JSE	The execution time required to process the data base commands.
RET	The rollback execution time for the simulation.
N	The number of simulations.

was averaged and weighted by a deadlock factor from the Shemer and Collmeyer simulation shown in table 5.1.

TABLE 5.1

Shemer and Collmeyer Deadlock Simulation Results

% WRITE \ TASKS	16	12	8
70	0.4	1.37	1.90
50	0.12	0.31	0.70
30	0.02	0.08	0.10

*** NUMBER OF DEADLOCKS PER 100 ACCESES ***

Each individual rollback to job stream percentage was weighted by a deadlock factor and was used to obtain a set of confidence intervals for the mean statistics. The confidence intervals were calculated from the following formula:

HI_PARM = MEAN + [1.96 * STDEV] / [SQRT [NUMBER OF VALUES]]

LO_PARM = MEAN - [1.96 * STDEV] / [SQRT [NUMBER OF VALUES]]

*** STDEV is the standard deviation calculation.

1.96 is the constant for calculating a 95% confidence interval, based upon the student T distribution.

The results of the simulation are displayed in Table 5.2 and in a graphical representation in Figures 5.1 - 5.6.

TABLE 5.2

Simulation Results

% CPU Overhead for Rollback Due to Deadlock

% WRITE	TASKS	MEAN	LOW	HIGH
		Common Shared Data		
70	8	0.5412	0.5000	0.5824
70	12	2.4888	1.9388	3.0388
70	16	3.7504	2.5598	4.9410
50	8	0.1483	0.1409	0.1556
50	12	0.3975	0.3527	0.4423
50	16	0.4025	0.3735	0.4315
30	8	0.0099	0.0094	0.0104
30	12	0.0284	0.0246	0.0322
30	16	0.0358	0.0173	0.0543
		Bi-task Data		
70	8	0.1304	0.1186	0.1422
70	12	0.3662	0.3339	0.3983
70	16	0.4219	0.2724	0.5714
50	8	0.0253	0.0227	0.0278
50	12	0.4930	0.0374	0.0623
50	16	0.0539	0.0306	0.0772
30	8	0.0018	0.0016	0.0020
30	12	0.0099	0.0072	0.0126
30	16	0.0059	0.0048	0.0070
		No Shared Data		
70	8	0.0582	0.0173	0.0543
70	12	0.2195	0.2036	0.2216
70	16	0.2846	0.2447	0.3246
50	8	0.0172	0.0121	0.0223
50	12	0.0274	0.0262	0.0286
50	16	0.0342	0.0209	0.0475
30	8	0.0010	0.0006	0.0014
30	12	0.0056	0.0038	0.0074
30	16	0.0032	0.0029	0.0035

The simulation for common shared data has the highest system overhead for the simulation run. The seventy percent update as indicated in Figure 5.1 still projects a reasonable figure of less than four percent system overhead. The comparison of Figures 5.1 through 5.3 indicates that the amount of CPU increases at an approximate factor of ten from thirty percent update to fifty percent and another factor of ten from fifty percent update to seventy percent.

The results of Figures 5.4 through 5.6 indicate a trend of similar overhead statistics for all three percentage updates. The overhead for the Bi-task and No Shared data is very similar with the curves following each other closely. Another intresting trend is that the overhead CPU figures are about the same for the cases of 12 and 16 users. The discrepancy between the Common Shared data and the other two types in each percent update indicates strongly that the greater the task integration the higher the CPU overhead. The calculated confidence intervals for all the Figures (5.1 - 5.6) are small enough at the 95% level to indicate a strong relationship among the figures.

FIGURE 5.1

Simulation Results for Common Shared Data

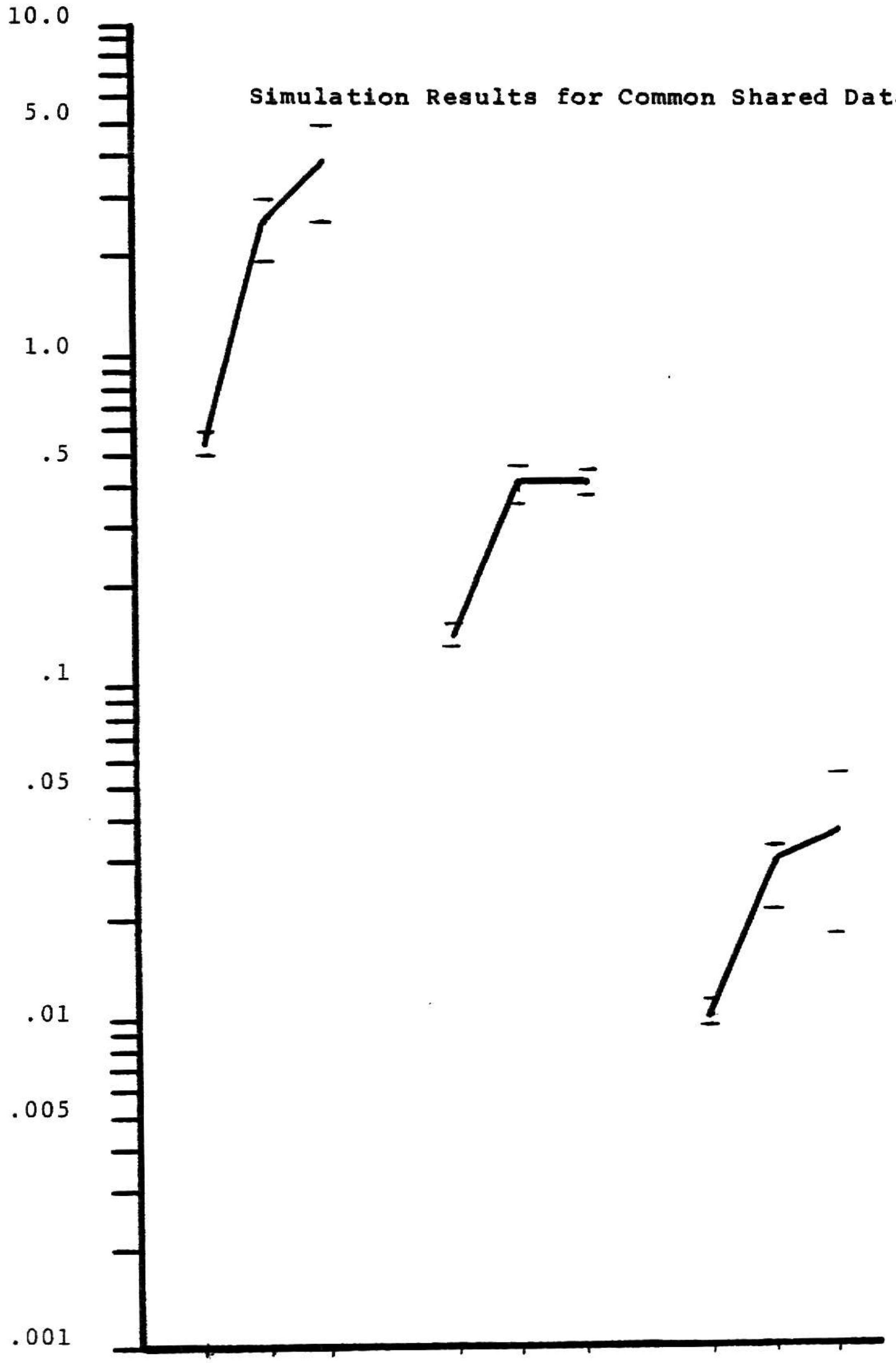


FIGURE 5.2

Simulation Results for Bi-task Data

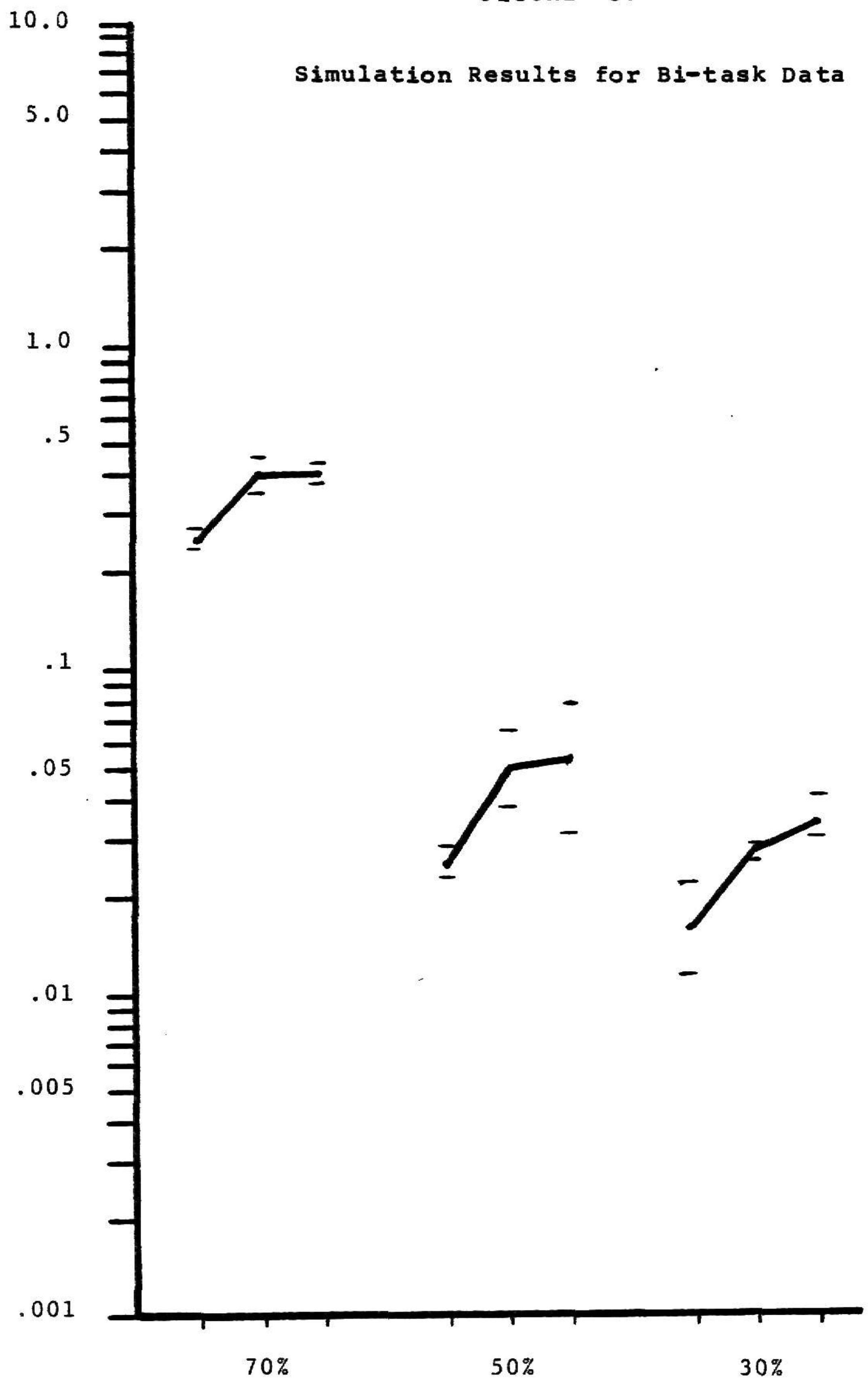


FIGURE 5.3

Simulation Results for No Shared Data

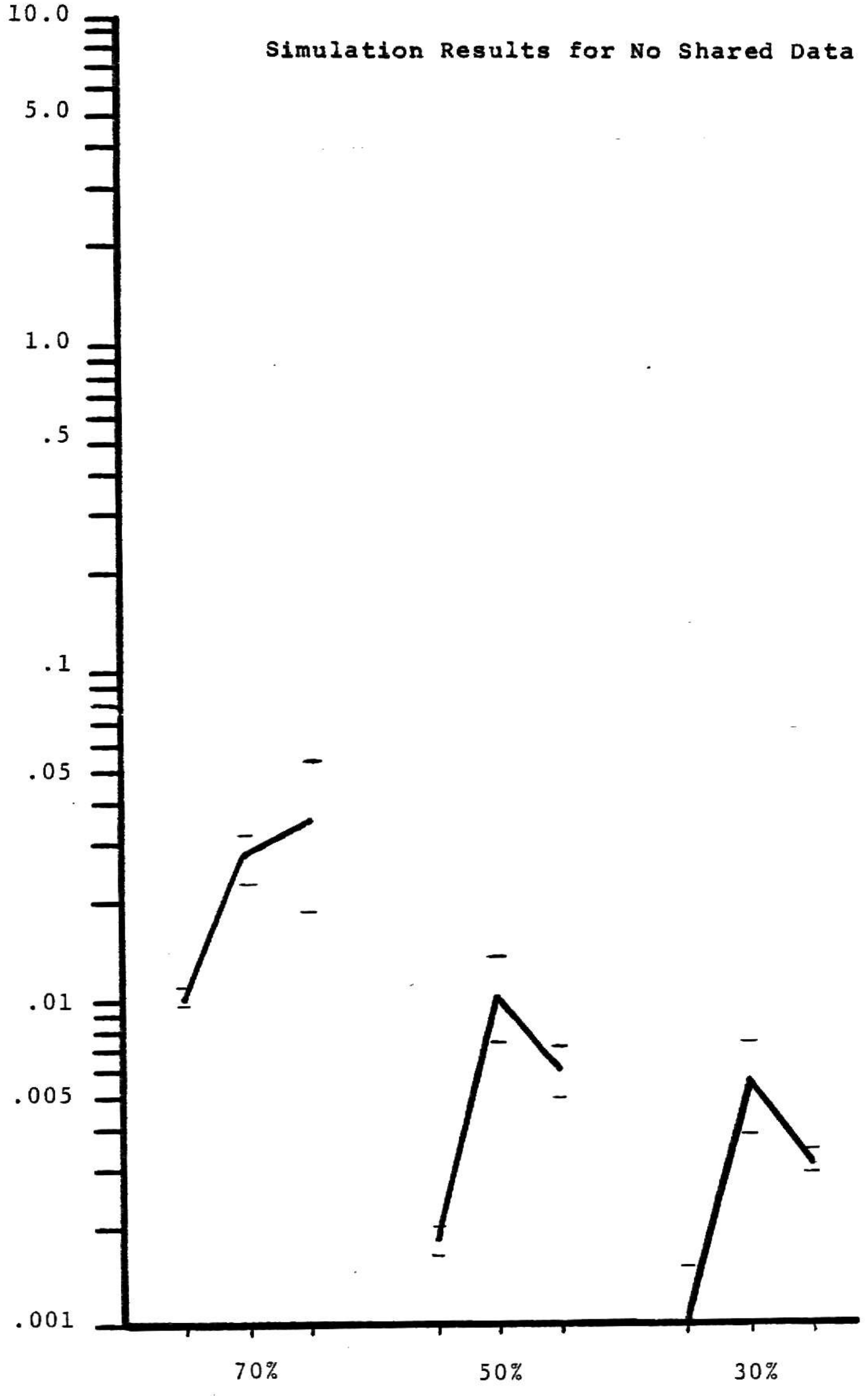


FIGURE 5.4

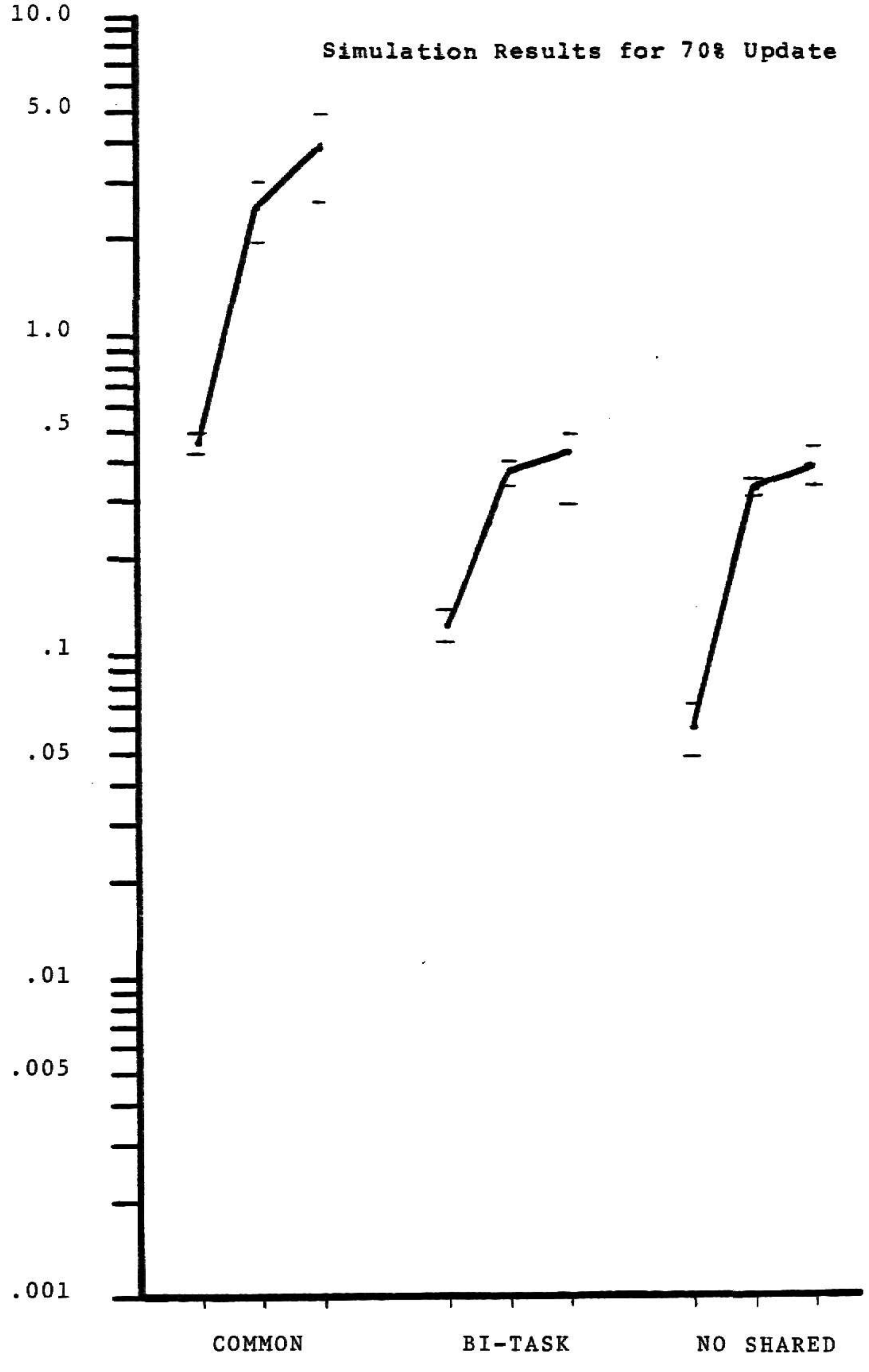

10.0
5.0
1.0
.5
.1
.05
.01
.005
.001
Simulation Results for 70% Update
COMMON
BI-TASK
NO SHARED

FIGURE 5.5

Simulation Results for 50% Update

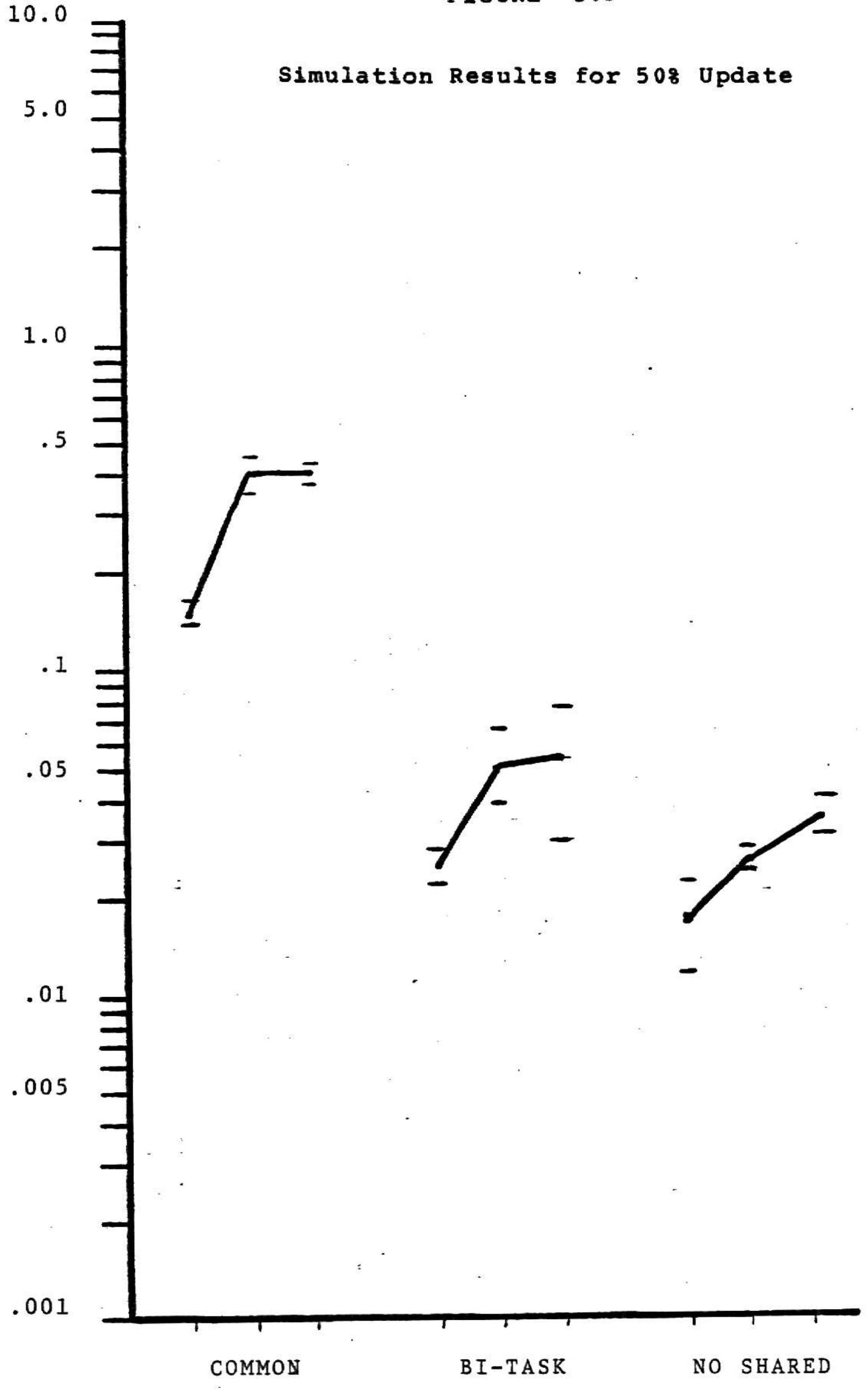
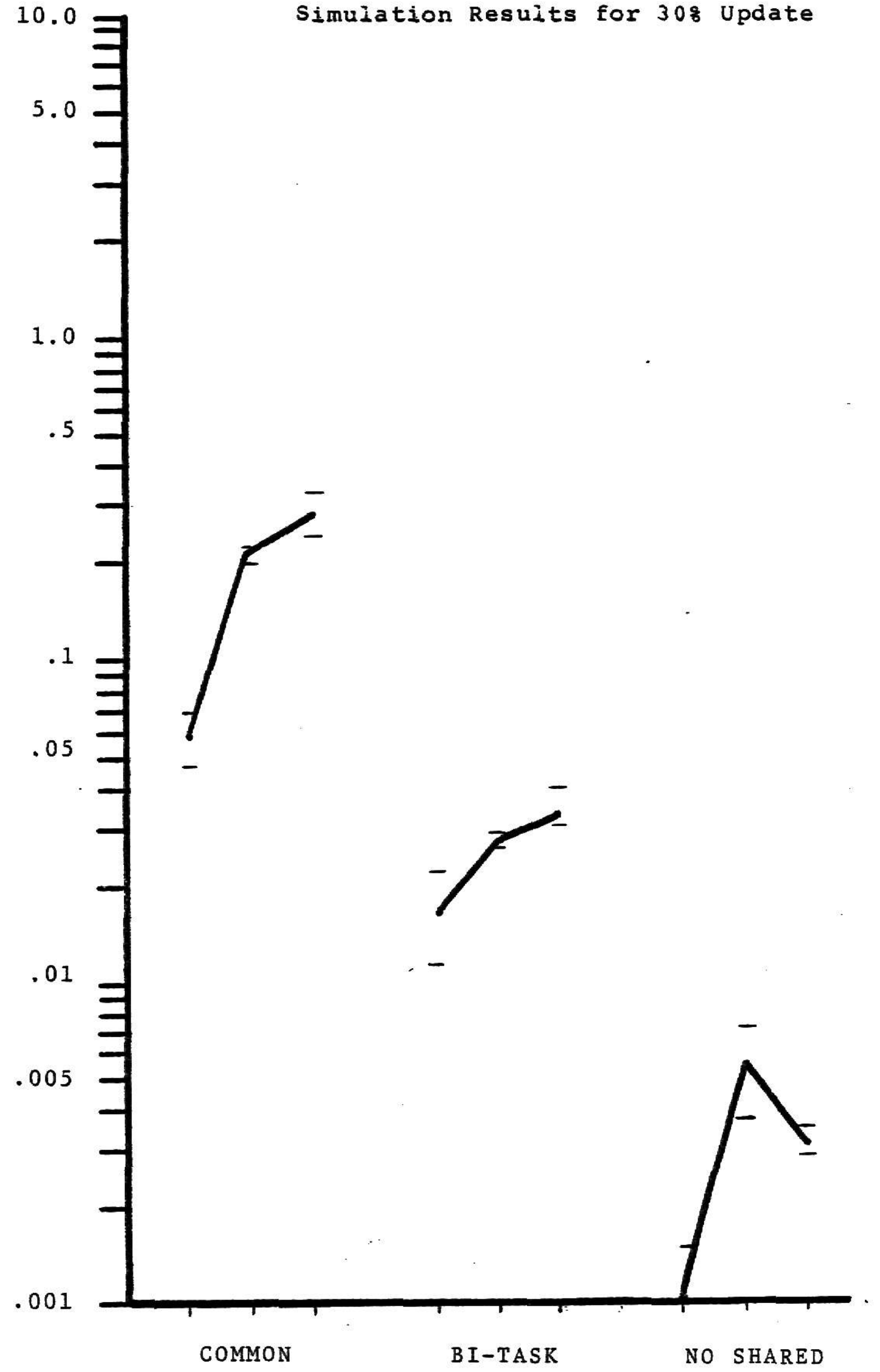


FIGURE 5.6

Simulation Results for 30% Update



The simulation results indicate that as the amount of data sharing between tasks increases, the percentage of system CPU overhead to support the selective rollback and recovery algorithm increases. The greatest amount of data sharing, the common shared data case, results in the highest percentage of system CPU overhead but, does not exceed four percent of the CPU overhead. System overhead is higher at higher levels of data sharing because the number of deadlock occurrences increases with data sharing. Also, the increased data sharing results in more system locks on granules during the execution of the data base commands.

The figures show a trend toward higher system overhead as the percentage of the command job stream writers (modifications) increases. These figures follow those of Shemer and Collmeyer and seem reasonable since a rollback task having modified the data base percipitates the future rollback of other tasks. The number of actual task operations rolled back are summarized in Table 5.3. The number of task operations rolled back is small until the data sharing becomes highly integrated (common shared data).

TABLE 5.3

Number of Data Commands Rolled Back

TASKS / % WRITE	8	12	16
	Common Shared Data		
30	11.3	11.3	17.0
50	11.7	27.7	30.7
70*1	22.3	27.7	34.7
70*2	49.7	53.7	67.7
	Bi-task Shared Data		
30	1.7	4.7	2.7
50	2.3	4.7	9.7
70*1	3.3	8.7	12.7
70*2	7.7	10.7	16.7
	No Shared Data		
30	1.3	1.3	1.3
50	1.3	2.7	3.3
70*1	2.0	5.7	5.7
70*2	5.0	7.0	11.0

*N Indicates the Number of Rollbacks.

NOTE: Statistics are not weighted for expected rollback.

As mentioned, from Figures 5.4 - 5.6 there is a leveling of overhead factors, the 12 user configurations and the 16 user configurations require about the same amount of overhead. This leveling is a result of 16 tasks having fewer records but, more tasks potentially involved in rollback than the 12 user configurations.

Table 5.4 summarizes the number of accesses to the journal file and the potential shared data list during rollback. From these figures, the number of accesses increases rapidly with the increase in data sharing, suggesting consideration should be given to the form of storage used with these. One possible solution is to keep the journal file, potential shared data list, or both in main memory at all times. Another possibility is to keep the journal file and potential shared data list on external storage until rollback is initiated and then keep them in main storage until rollback is completed. The journal file is not normally kept in main storage because it is not required except in the case of a hard system crash. Updates to the journal file are kept in a buffer until the data base is inactive and a checkpoint can be established. For most recovery methods the update buffers are applied against the data base to

TABLE 5.4

Accesses to Journal File
and
Potential Shared Data List
During Rollback

TASKS / % WRITE	16	12	8
		No Shared Data	
30	7.0 - 32.7	13.7 - 27.3	9.7 - 24.3
50	8.3 - 21.7	13.7 - 58.3	24.3 - 87.3
70*1	10.3 - 43.0	22.3 - 99.7	29.3 - 116.0
70*2	136.7 - 428.3	150.0 - 572.7	166.0 - 701.3
		Bi-task Data	
30	5.0 - 17.0	9.0 - 15.0	5.0 - 16.3
50	15.0 - 51.0	9.0 - 42.0	5.0 - 17.0
70*1	15.0 - 50.0	13.0 - 49.3	7.0 - 19.3
70*2	26.0 - 101.7	20.0 - 86.3	16.0 - 79.0
		Common Shared Data	
30	38.3 - 82.7	37.0 - 63.7	49.0 - 118.7
50	35.7 - 64.7	75.7 - 308.3	77.0 - 336.3
70*1	60.7 - 129.7	79.0 - 339.7	83.0 - 350.7
70*2	136.7 - 428.3	150.0 - 572.7	166.0 - 701.3

*n Indicates Number of Rollbacks.

Another reason for keeping only a limited amount of the journal file in memory is that in case of system crash the journal file could be damaged. Also, the journal file, as is seen from this simulation, requires a considerable amount of information and would be better kept in a mass storage device.

Evaluation and Conclusions

The simulation study of the performance of the selective Rollback and Recovery Algorithm indicates the feasibility of that methodology for the recovery of distributed data bases. The security offered by the selective rollback and recovery algorithm contributes to the likelihood that distributed data base management systems could with future additions be a commercially acceptable product. The selective rollback and recovery algorithm overall CPU system cost is especially exciting in light of several factors:

--- The need for distributed data base management systems exists now.

--- A data base management system must maintain some form of rollback and recovery mechanism no matter what type of deadlock approach is used.

--- The selective rollback and recovery algorithm requires less than four percent system CPU overhead in the worst case measured.

--- For a typical data processing environment where data sharing is limited, the system CPU overhead is less than one percent.

--- The selective rollback and recovery algorithm uses existing system commands meaning that it will require little data base modification and space to run.

With the demand for increased sharing of data among users and the advent of on-line computing services, the neccessity of a working distributed data base system is a must.

6.1 Future Investigation

The research investigated has indicated the potential for a efficient mechanism for ensuring data integrity. One possibility is to couple the selective rollback and recovery algorithm with a differential file system. This combination should show a considerable performance advantage in rollback and recovery over the usual methods of checkpoints and before or afterimages. The marrying of a differential file structure with a selective rollback and recovery system offers the best of both and should mean an improved method of system recovery applicable to various computing environments. Parsing the differential file structure during rollback, is especially well suited to the design of the selective rollback algorithm.

A second suggestion for further work would be the extension of the work by Shemer and Collmeyer. The simulation indicates statistics for up to 20 system users, further investigation would validate their statistics and possibly test the effect of even a greater number of users. Shemer and Collmeyer investigated a locking for three or six transactions at every task request, investigation of this parameter seems realistic.

Lastly, investigation of this work should continue. Replication of this work is a good idea for firm statistical assumption. Also, it seems natural that the actual porting of this methodology into a real data base management system would provide many useful performance statistics. Finally, the testing of this methodology in a distributed environment would be a major accomplishment.

REFERENCES

1. Munz, R., and Krenz, G., "Concurrency in Database Systems - A Simulation Study", ACM Sigmod , (1977): 111.

2. Chu, W. W., and Ohlmacher, G., "Avoiding Deadlock in Disrtibuted Data Bases", ACM National Symposium, Vol.1 (November 1974): 156-157.

3. Chu, pp. 157-159.

4. Chu, pp. 159.

5. Deppe, Mark E., "Recovery and Restart in a Distributed Environment", Hewlett Packard, Santa Clara, CA, (June 1978) 1-10.

6. Gray, J. N., Lorie, R. A., and Putola, G. R., "Granularity of Locks in a Shared Data Base", Proc. First Conference on Very Large Data Bases, (September 1975) 428,430-435.

7. Colliat, G., and Bachman, C., "Commitment in a Distributed Database", Proc. Fourth Conference on Very Large Data Bases, (September 1978) 3-9,12-16.

8. Severance, Dennis G., and Lohman, Guy M., "Differential Files: Their Application to the Maintenence of Large Database Systems", ACM Transactions on Databse Systems, Vol. 1, No. 3 (September 1976): 263,266.

9. Shemer, J. E., and Collmeyer, A. J., "Database Sharing: A Study of Interference, Roadblock and Deadlock", ACM SIGFIDET, (1972): 147-156.

10. Munz, pp. 111-112.

11. Ries, Daniel R., and Stonebraker, Michael, "Effects of Locking Granularity in a Database Management System", ACM Transactions on Database Systems, Vol. 2, No. 3, (September 1976) 233-240.

12. Maryanski, Fred J., and Fisher, Paul S., "Rollback and Recovery in Distributed Data Base Management Systems", Proc. ACM Annual Conference on Very Large Data Bases, (October 1977) pp. 33-38.

13. Maryanski, pp. 35-38.

BIBLIOGRAPHY

Chu, W. W., and Ohlmacher, G. "Avoiding Deadlock in Distributed Data Bases". ACM National Symposium , Vol. 1 (November 1974): 158-160.

Colliat, G., and Bachman, C. "Commitment in a Distributed Database". Proc. Fourth Conference on Very Large Databases, (September 1978): 2-19.

Deppe, Mark E., "Recovery and Restart in a Distributed Environment", Hewlett Packard, Santa Clara, CA, (June 1978).

Gray, J. N., Lorie, R. A., and Putzola, G. R. "Granularity of Locks in a Shared Data Base". Proc. First Conference on Very Large Data Bases, (September 1975): 428-451.

Maryanski, Fred J., and Fisher, Paul S. "Rollback and Recovery in Distributed Data Base Management Systems". Proc. ACM Annual Conference, (October 1977): 33-38.

Munz, R., and Krenz, G. "Concurrency in Database Systems - A Simulation Study". ACM Sigmod , (1977): 111-120.

Ries, Daniel R., and Stonebraker, Michael "Effects of Locking Granularity in a Database Management System". ACM Transactions on Database Systems, Vol. 2, No. 3, (September 1977): 233-246.

Severance, Dennis G.,and Lohman, Guy M. "Differential Files: Their Application to the Maintenance of Large Databases". ACM Transactions on Database Systems , Vol. 1, No. 3 (September 1976): 256-267.

Shemer, J. E., and Collmeyer, A. J. "Database Sharing: A Study of Interference, Roadblock and Deadlock". ACM SIGFIDET (1972): 147-163.

APPENDIX

Simulation Program Listing

```
PROG5:   PROCEDURE OPTIONS(MAIN);

/*************************************************************/
      DATA BASE TASK .. NAME, RECORDS, AND P.S.D.L.
/*************************************************************/

DCL      1 PROCESS BASED(@PR),
              2 NAME CHAR(1),
              2 REC(13),
                   3 DATA BIN FIXED(15),
              2 SDLT(120) CHAR(1),
              2 SDLR(120) BIN FIXED(15),
              2 NSDL BIN FIXED(15),

         @PNTR(16) STATIC EXTERNAL POINTER,

/**********************************************************/
                        SYSTEM JOURNAL FILE
/**********************************************************/

         1 LOG BASED(@LOG),
              2 LFLAG BIN FIXED(15),
              2 LTIME BIN FIXED (15),
              2 LTASK CHARACTER(1),
              2 LSET FIXED DEC(1),
              2 LREC BIN FIXED (15),
              2 @ROLL POINTER,
              2 @BACK POINTER,
              2 @FWD POINTER,
              2 LDML CHARACTER(10),

       (@HEAD,@TAIL)POINTER STATIC EXTERNAL,

/**********************************************************/
      STORAGE FOR BEFOREIMAGES USED IN ROLLBACK
/**********************************************************/

         1 STAK BASED(@STAK),
              2 DATA BIN FIXED(15),
              2 S1FIDT CHAR(1),
              2 S1FIDR BIN FIXED(15),
              2 S1BIDT CHAR(1),
              2 S1BIDR BIN FIXED(15),
              2 S2FIDT CHAR(1),
              2 S2FIDR BIN FIXED(15),
              2 S2BIDT CHAR(1),
              2 S2BIDR BIN FIXED(15),

      (@PRED,@SUCC,@CURR)STATIC EXTERNAL POINTER,

/*********************************************************/
                 ROLLBACK FILE DECLARATION
/*********************************************************/

         1 RFILE BASED(@RFILE),
              2 RFTIME BIN FIXED(15),
              2 RFDML CHARACTER(10),
              2 RFREC BIN FIXED(15),
              2 @RFWD POINTER,
              2 @RFBK POINTER,

         (@TRF,@HRF)POINTER STATIC EXTERNAL,

/*********************************************************/
               SECONDARY ROLLBACK LIST DECLARATION
/*********************************************************/

         1 SRL BASED(@SRL),
              2 TASK CHARACTER(1),
              2 STIME BIN FIXED(15),
              2 @SRLBK POINTER,
              2 REC BIN FIXED(15),
              2 @SRLFWD POINTER,
         (@SRLH,@SRLT)POINTER STATIC EXTERNAL,

/*********************************************************/
      SET LISTS .. USED FOR CONNECTS AND DISCONNECTS
/*********************************************************/

          1 SET1 BASED(@SET1),
              2 S1TASK CHAR(1),
              2 S1REC BIN FIXED(15),
              2 S1FWD POINTER,
              2 S1BK POINTER,

           1 SET2 BASED(@SET2),
              2 S2TASK CHAR(1),
              2 S2REC BIN FIXED(15),
              2 S2FWD POINTER,
              2 S2BK POINTER,

      (@S1T,@S1H,@S2T,@S2H)POINTER STATIC EXTERNAL,
 /**************************************************************/
         DECLARATIONS FOR OTHER VARIABLES USED
 /**************************************************************/

 DCL (TLIST(16),CHNG,TSK) CHARACTER(1);
 DCL ERROR_CODE CHARACTER(30)VAR;
 DCL (X,Y,Z,I,J,K,L,M,N,QT)BIN FIXED(15)INIT(1);
 DCL
(II,SRFLAG,TRTIME,FLAG,UPDATE(13),TL,EFLAG,NNSD,CNG)BIN
FIXED(15)INIT(0);
 DCL (CNT,TTIME,RECR,OO,PSDB,RDB,LDB,SDB,LRB,SRRB,PSRB)BIN
FIXED(15)INIT(0);
 DCL (RFRB,URB,STDB,RRB,STRB,SRB)BIN FIXED(15)INIT(0);
 DCL SET FIXED DEC(1);
 DCL (COMMAND,CARD) CHARACTER(80)VAR;
 DCL DML CHARACTER(10)VAR;
 DCL (TASK,R) CHARACTER(5)VAR;
 DCL (@PTR,@PTS,@ADJ)STATIC EXTERNAL POINTER;
 DCL (@TRVS,@R,@SRCH,@SRLL,@SCN)POINTER STATIC EXTERNAL;

 /**************************************************************/
       INITIALIZATION OF VARIABLES AND POINTERS
 /**************************************************************/

 @S1H = NULL;
 @S1T = NULL;
 @S2H = NULL;
 @S2T = NULL;
 @HEAD = NULL;
 @TAIL = NULL;
 @SRLH = NULL;
 @SRLT = NULL;

 /**************************************************************/
               INITIALIZATION OF THE DATA BASE
 /**************************************************************/

 NNSD = 120;
 NREC = 13;
 NTASK = 16;
 DO I = 1 TO NTASK;
 ALLOCATE PROCESS;
 @PNTR(I) = @PR;
 END;

 PUT SKIP LIST(TIME);

 DO I = 1 TO NTASK;
 @PTR = @PNTR(I);
      IF I = 1 THEN @PTR -> NAME = 'A';
 ELSE IF I = 2 THEN @PTR -> NAME = 'B';
 ELSE IF I = 3 THEN @PTR -> NAME = 'C';
 ELSE IF I = 4 THEN @PTR -> NAME = 'D';
ELSE IF I = 5 THEN @PTR -> NAME = 'E';
ELSE IF I = 6 THEN @PTR -> NAME = 'F';
ELSE IF I = 7 THEN @PTR -> NAME = 'G';
ELSE IF I = 8 THEN @PTR -> NAME = 'H';
ELSE IF I = 9 THEN @PTR -> NAME = 'I';
ELSE IF I = 10 THEN @PTR -> NAME = 'J';
ELSE IF I = 11 THEN @PTR -> NAME = 'K';
ELSE IF I = 12 THEN @PTR -> NAME = 'L';
ELSE IF I = 13 THEN @PTR -> NAME = 'M';
ELSE IF I = 14 THEN @PTR -> NAME = 'N';
ELSE IF I = 15 THEN @PTR -> NAME = 'O';
ELSE IF I = 16 THEN @PTR -> NAME = 'P';
DO J = 1 TO NNSD;
@PTR -> SDLT(J) = ' ';
@PTR -> SDLR(J) = 0;
END;
@PTR -> NSDL = 0;
END;

DO I = 1 TO NTASK;
@PTR = @PNTR(I);
DO J = 1 TO 7;
@PTR -> REC(J).DATA = J;
END;              END;

DO I = 1 TO NTASK;
@PTR = @PNTR(I);
DO J = 8 TO NREC;
IF I = 1 THEN
@PTR -> REC(J).DATA = (10+J);
ELSE IF I = 2 THEN
@PTR -> REC(J).DATA = (40+J);
ELSE IF I = 3 THEN
@PTR -> REC(J).DATA = (70+J);
ELSE IF I = 4 THEN
@PTR -> REC(J).DATA = (100+J);
ELSE IF I = 5 THEN
@PTR -> REC(J).DATA = (130+J);
ELSE IF I = 6 THEN
@PTR -> REC(J).DATA = (160+J);
ELSE IF I = 7 THEN
@PTR -> REC(J).DATA = (190+J);
ELSE IF I = 8 THEN
@PTR -> REC(J).DATA = (220+J);
ELSE IF I = 9 THEN
@PTR -> REC(J).DATA = (250+J);
ELSE IF I = 10 THEN
@PTR -> REC(J).DATA = (280+J);
ELSE IF I = 11 THEN
@PTR -> REC(J).DATA = (310+J);
ELSE IF I = 12 THEN
@PTR -> REC(J).DATA = (340+J);
ELSE IF I = 13 THEN
```

```
@PTR -> REC(J).DATA = (370+J);
ELSE IF I = 14 THEN
@PTR -> REC(J).DATA = (400+J);
ELSE IF I = 15 THEN
@PTR -> REC(J).DATA = (430+J);
ELSE IF I = 16 THEN
@PTR -> REC(J).DATA = (460+J);
END;
END;

/*************************************************************/
           LISTING OF THE DATA BASE COMMANDS
/*************************************************************/

DO I = 1 TO NTASK;
@PTR = @PNTR(I);
DO J = 1 TO NREC;
PUT SKIP LIST(@PTR -> NAME,J,@PTR -> REC(J).DATA);
END;
END;

/**************************************************************/
      CALCULATION OF THE POTENTIAL SHARED DATA LIST
/**************************************************************/

DO U = 1 TO NTASK-1;
@PTR = @PNTR(U);
DO V = 1 TO NREC;
DO W = U+1 TO NTASK;
@PTS = @PNTR(W);
DO X = 1 TO NREC;
IF (@PTR -> REC(V).DATA = @PTS -> REC(X).DATA)
   THEN CALL PDLIN;
END;         END;          END;             END;
PUT SKIP LIST(TIME);

/************************************************************/
      ECHO THE CONTENTS OF THE P.S.D.L FOR EACH TASK
/************************************************************/

DO I = 1 TO NTASK;
@PTR = @PNTR(I);
PUT SKIP(2)EDIT(' PSDL FOR ',@PTR ->
NAME)(X(10),A,X(2),A);
DO J = 1 TO @PTR -> NSDL;
PUT SKIP LIST(@PTR -> SDLT(J), @PTR -> SDLR(J));
END;            END;

PUT SKIP LIST(TIME);
```

```
 /************************************************************/
        MAIN PROGRAM LOOP .. INPUT A DATA BASE COMMAND,
        DECODE COMMAND, CALL APPROPRIATE SUBROUTINE
 /************************************************************/

 PUT SKIP
LIST('PSDB',PSDB,'RDB',RDB,'LDB',LDB,'SDB',SDB,'LRB',LRB,
 'SRRB',SRRB,'PSRB',PSRB,'RFRB',RFRB,'URB',URB,'STDB',STDB,
 'RRB',RRB,'STRB',STRB,'SRB',SRB);
 ON ENDFILE(SYSIN) QT = 2;
 GET LIST(COMMAND);
 INP: DO WHILE (QT=1);
 FLAG = 0;
 TTIME = TTIME + 1;
 PUT SKIP(2) EDIT('THE PROGRAM INPUT IS :       ',COMMAND)
    (COL(20),A,A);
 V = INDEX(COMMAND,' ');
 DML = SUBSTR(COMMAND,1,V-1);
 CARD = SUBSTR(COMMAND,V+1);
 VV = INDEX(CARD,' ');
 TASK = SUBSTR(CARD,1,VV-1);
    IF DML ^= 'ORDER' THEN CALL RREC;
 IF DML = 'GET' THEN CALL PGET;
    ELSE IF DML = 'MODIFY' THEN CALL PMOD;
    ELSE IF DML = 'RBACK' THEN CALL PROL;
    ELSE IF DML = 'ERASE' THEN CALL PERA;
    ELSE IF DML = 'STORE' THEN CALL PSTO;
    ELSE IF DML = 'RESTART' THEN CALL PRES;
    ELSE IF DML = 'CONNECT' THEN CALL PCON;
    ELSE IF DML = 'DISCONNECT' THEN CALL PDIS;
    ELSE DO;
      ERROR_CODE = 'INVALID COMMAND';
      CALL ERROR;
    END;
 PUT SKIP LIST(TIME);
 PUT SKIP
LIST('PSDB',PSDB,'RDB',RDB,'LDB',LDB,'SDB',SDB,'LRB',LRB,
 'SRRB',SRRB,'PSRB',PSRB,'RFRB',RFRB,'URB',URB,'STDB',STDB,
 'RRB',RRB,'STRB',STRB,'SRB',SRB,'TL',TL);
 GET LIST(COMMAND);
 END INP;


 /************************************************************/
                      ***  RREC SUBROUTINE  ***
        DETERMINES THE RECORD SPECIFIED IN THE INPUT
                      COMMAND TO THE DATA BASE
 /************************************************************/

 RREC: PROCEDURE;
 EQ = INDEX(CARD,'=');
 IF EQ > 0 THEN R = SUBSTR(CARD,VV+1,(EQ-VV-2));
 ELSE DO;
```

```
DOT = INDEX(CARD,'.');
R = SUBSTR(CARD,VV+1,(DOT-VV-1));
END;
RECR = BINARY(R);
END RREC;


/**************************************************************/
            ***  PDLIN  SUBROUTINE  ***
      INSERTION OF A POTENTIAL SHARED DATA LIST PAIR
         IN THE TASKS POTENTIAL SHARED DATA LISTS
/**************************************************************/

PDLIN: PROCEDURE;
@PTR -> NSDL = 1 + @PTR -> NSDL;
Y = @PTR -> NSDL;
@PTR  -> SDLT(Y) = @PTS -> NAME;
@PTR -> SDLR(Y) = X;
@PTS -> NSDL = 1 + @PTS -> NSDL;
Y = @PTS -> NSDL;
@PTS -> SDLT(Y) = @PTR -> NAME;
@PTS -> SDLR(Y) = V;
END PDLIN;


/**************************************************************/
                ***  PDIS SUBROUTINE  ***
      DISCONNECTS A TASK'S RECORD FROM A SET
/**************************************************************/

PDIS: PROCEDURE;
PUT SKIP LIST(' ENTER PDIS**');
IF FLAG = 0 THEN CALL FST;
IF FLAG = 0 THEN LDB = LDB + 1;
ELSE LRB = LRB + 1;
DO I = 1 TO NTASK;
@PTR = @PNTR(I);
DO J = 1 TO NREC;
IF (TASK = @PTR -> NAME & RECR = J)
   THEN GO TO FND;
END;          END;
ERROR_CODE = 'TASK FOR DIS NF IN PROCESS';
CALL ERROR;
FND: @ADJ = @S1H;
IF SET = 1 THEN DO;
DO WHILE(1 = 1);
IF(@ADJ -> SET1.S1TASK = @PTR -> NAME &
  @ADJ -> SET1.S1REC = J) THEN GO TO F1;
ELSE IF @ADJ = @S1T THEN DO;
IF EFLAG = 1 THEN DO;
CALL PSTK1;
GO TO SS1;
END;
```

```
ELSE DO;
IF FLAG = 1 THEN RETURN;
ERROR_CODE = 'PDIS 1 ERROR';
CALL ERROR;
END;
END;
ELSE  @ADJ = @ADJ -> S1FWD;
END;        END;
ELSE IF SET = 2 THEN DO;
PUT SKIP LIST('ENTER PD2');
@ADJ = @S2H;
DO WHILE(1 = 1);
IF(@ADJ -> SET2.S2TASK = @PTR  -> NAME &
  @ADJ -> SET2.S2REC = J) THEN GO TO F1;
ELSE IF @ADJ = @S2T THEN DO;
IF EFLAG = 1 THEN DO;
CALL PSTK2;
GO TO SS2;
END;
ELSE DO;
IF FLAG = 1 THEN RETURN;
ERROR_CODE = 'PDIS 2 ERROR';
CALL ERROR;
END;
END;
ELSE  @ADJ = @ADJ -> S2FWD;
END;        END;
F1: @CURR = @ADJ;
IF EFLAG = 0 THEN ALLOCATE STAK;
IF SET = 1 THEN DO;
IF FLAG = 0 THEN DO;
STDB = STDB + 1;
@ADJ = @CURR -> SET1.S1FWD;
IF @ADJ ^= NULL THEN DO;
STAK.S1FIDT = @ADJ -> SET1.S1TASK;
STAK.S1FIDR = @ADJ -> SET1.S1REC;
END;
ELSE DO;;
STAK.S1FIDT = ' ';
STAK.S1FIDR = 0;
END;
@ADJ = @CURR -> SET1.S1BK;
IF @ADJ ^= NULL THEN DO;
STAK.S1BIDT = @ADJ -> SET1.S1TASK;
STAK.S1BIDR = @ADJ -> SET1.S1REC;
END;
ELSE DO;;
STAK.S1BIDT = ' ';
STAK.S1BIDR = 0;
END;
END;
IF @CURR -> SET1.S1BK ^= NULL THEN @PRED = @CURR ->
SET1.S1BK;
```

```
 ELSE @PRED = NULL;
 IF @CURR -> SET1.S1FWD ^= NULL THEN @SUCC = @CURR ->
SET1.S1FWD;
 ELSE @SUCC = NULL;
 IF (@PRED ^= NULL & @SUCC ^= NULL) THEN DO;
 @PRED -> SET1.S1FWD = @SUCC;
 @SUCC -> SET1.S1BK = @PRED;
 END;
 ELSE IF (@PRED = NULL & @SUCC ^= NULL )THEN DO;
 @SUCC -> SET1.S1BK = NULL;
 @S1H = @SUCC;
 END;
 ELSE IF (@SUCC = NULL & @PRED ^= NULL )THEN DO;
 @PRED -> SET1.S1FWD = NULL;
 @S1T = @PRED;
 END;
 ELSE DO;
 @S1H = NULL;
 @S1T = NULL;
 END;        END;
 ELSE IF SET = 2 THEN DO;
 IF FLAG = 0 THEN DO;
 STDB = STDB + 1;
 @ADJ = @CURR -> SET2.S2FWD;
 IF @ADJ ^= NULL THEN DO;
 STAK.S2FIDT = @ADJ -> SET2.S2TASK;
 STAK.S2FIDR = @ADJ -> SET2.S2REC;
 END;
 ELSE DO;;
 STAK.S2FIDT = ' ';
 STAK.S2FIDR = 0;
 END;
 @ADJ = @CURR -> SET2.S2BK;
 IF @ADJ ^= NULL THEN DO;
 STAK.S2BIDT = @ADJ -> SET2.S2TASK;
 STAK.S2BIDR = @ADJ -> SET2.S2REC;
 END;
 ELSE DO;;
 STAK.S2BIDT = ' ';
 STAK.S2BIDR = 0;
 END;
 END;
 IF @CURR -> SET2.S2BK ^= NULL THEN @PRED = @CURR ->
SET2.S2BK;
 ELSE @PRED = NULL;
 IF @CURR -> SET2.S2FWD ^= NULL THEN @SUCC = @CURR ->
SET2.S2FWD;
 ELSE @SUCC = NULL;
 IF (@PRED ^= NULL & @SUCC ^= NULL) THEN DO;
 @PRED -> SET2.S2FWD = @SUCC;
 @SUCC -> SET2.S2BK = @PRED;
 END;
 ELSE IF (@PRED = NULL & @SUCC ^= NULL )THEN DO;
```

```
@SUCC -> SET2.S2BK = NULL;
@S2H = @SUCC;
END;
ELSE IF (@SUCC = NULL & @PRED ^= NULL )THEN DO;
@PRED -> SET2.S2FWD = NULL;
@S2T = @PRED;
END;
ELSE DO;
@S2H = NULL;
@S2T = NULL;
END;         END;
SS1: CALL STT1;
SS2: CALL STT2;
IF FLAG = 0 THEN DO;
LDB = LDB + 1;
RDB = RDB + 1;
I = 1;
@PTR = @PNTR(I);
DO WHILE(@PTR -> NAME ^= TASK);
I = I +1;
@PTR = @PNTR(I);
END;
@STAK -> STAK.DATA = @PTR -> REC(RECR).DATA;
IF EFLAG = 0 THEN DO;
@LOG -> @ROLL = ADDR(STAK);
END;
END;
END PDIS;


/******************************************************************/
                *** FST SUBROUTINE ***
    DETERMINES THE SET NUMBER SPECIFIED IN THE INPUT
/******************************************************************/

FST: PROCEDURE;
IF EFLAG = 0 THEN DO;
CALL LOGG;
DOT = INDEX(CARD,'.');
CHNG = SUBSTR(CARD,EQ+2,DOT-1);
SET = BINARY(CHNG);
@LOG -> LSET = SET;
PUT SKIP(2)LIST('        THE SET IS  ',SET);
END;
END FST;
```

```
/*************************************************************/
                    ***  PCON SUBROUTINE  ***
       THIS PROCEDURE CONNECTS A RECORD TO A SET
/*************************************************************/

 PCON: PROCEDURE;
 PUT SKIP LIST(' ENTER PCON**');
 IF FLAG = 0 THEN SDB = SDB + 1;
 ELSE SRB = SRB + 1;
 IF FLAG = 0 THEN CALL FST;
 IF SET = 1 THEN DO;
 CALL STT1;
 ALLOCATE SET1;
 IF @S1H = NULL THEN @S1H = @SET1;
 ELSE @S1T ->  S1FWD = @SET1;
  S1BK = @S1T;
 @S1T = @SET1;
  S1FWD = NULL;
 @SET1 ->  S1REC = RECR;
 @SET1 ->  S1TASK = TASK;
 PUT SKIP LIST('CON-S-1    ',@SET1 -> S1REC, @SET1 ->
S1TASK);
 END;
 ELSE IF SET = 2 THEN DO;
 CALL STT2;
 ALLOCATE SET2;
 IF @S2H = NULL THEN @S2H = @SET2;
 ELSE @S2T ->  S2FWD = @SET2;
  S2BK = @S2T;
 @S2T = @SET2;
  S2FWD = NULL;
 @SET2 ->  S2TASK = TASK;
 @SET2 ->  S2REC = RECR;
 PUT SKIP LIST('CON-S-2    ',@SET2 -> S2REC, @SET2 ->
S2TASK);
 END;
 IF FLAG = 0 THEN DO;
 STDB = STDB + 1;
 ALLOCATE STAK;
 IF SET = 1 THEN DO;
 @ADJ = @SET1;
 IF @ADJ ->   S1FWD = NULL THEN DO;
 @STAK -> S1FIDT = ' ';
 @STAK -> S1FIDR = 0;
 END;
 ELSE DO;
 @ADJ = @ADJ->  S1FWD;
 @STAK -> S1FIDT = @ADJ -> S1TASK;
 @STAK -> S1FIDR = @ADJ -> S1REC;
 END;
 @ADJ = @SET1;
 IF @ADJ->  S1BK = NULL THEN DO;
 @STAK -> S1BIDT = ' ';
```

```
@STAK -> S1BIDR = 0;
END;
ELSE DO;
@ADJ = @ADJ-> S1BK;
@STAK -> S1BIDT = @ADJ -> S1TASK;
@STAK -> S1BIDR = @ADJ -> S1REC;
END;
CALL STT1;
END;

ELSE IF SET = 2 THEN DO;
@ADJ = @SET2;
IF @ADJ ->  S2FWD = NULL THEN DO;
@STAK -> S2FIDT = ' ';
@STAK -> S2FIDR = 0;
END;
ELSE DO;
@ADJ = @ADJ-> S2FWD;
@STAK -> S2FIDT = @ADJ -> S2TASK;
@STAK -> S2FIDR = @ADJ -> S2REC;
END;
@ADJ = @SET2;
IF @ADJ-> S2BK = NULL THEN DO;
@STAK -> S2BIDT = ' ';
@STAK -> S2BIDR = 0;
END;
ELSE DO;
@ADJ = @ADJ-> S2BK;
@STAK -> S2BIDT = @ADJ -> S2TASK;
@STAK -> S2BIDR = @ADJ -> S2REC;
END;
END;
I = 1;
@PTR = @PNTR(I);
DO WHILE(@PTR -> NAME ^= TASK);
I = I +1;
@PTR = @PNTR(I);
END;
@STAK -> STAK.DATA = @PTR -> REC(RECR).DATA;
@LOG -> @ROLL = ADDR(STAK);
RDB = RDB + 1;
LDB = LDB + 1;
CALL STT2;
END;
END PCON;
```

```
/*************************************************************/
                    ***  ERROR SUBROUTINE  ***
     ROUTINE TO PRINT ERROR MESSAGES AND STOP PROGRAM
/*************************************************************/

ERROR: PROCEDURE;
PUT SKIP LIST(ERROR_CODE);
STOP;
END ERROR;


/*************************************************************/
                 ***  LOGG SUBROUTINE  ***
       ROUTINE TO LOG ALL INCOMING DATA BASE COMMANDS
              AND DATA BASE ROLLBACK ACTIONS
/*************************************************************/

LOGG: PROCEDURE;
IF FLAG = 0 THEN LDB = LDB + 1;
ELSE LRB = LRB + 1;
IF FLAG = 1 THEN PUT SKIP LIST('   ROLLBACK  LOG ENTRY');
ELSE PUT SKIP EDIT(' LOG ENTRY')(COL(25),A);
PUT SKIP EDIT(TASK,RECR)(COL(35),A,F(5));
PUT SKIP EDIT(DML)(COL(35),A);
ALLOCATE LOG;
IF @HEAD = NULL THEN @HEAD = @LOG;
ELSE @TAIL -> @FWD = @LOG;
@LOG -> @BACK = @TAIL;
@TAIL = @LOG;
@LOG -> @FWD = NULL;
@LOG -> LTIME = TTIME;
@LOG -> LTASK = TASK;
@LOG -> LREC = RECR;
@LOG -> LDML = DML;
@LOG -> LFLAG = FLAG;
PUT SKIP LIST(@LOG -> LTIME);
PUT SKIP LIST(@LOG -> LTASK);
PUT SKIP LIST(@LOG -> LREC);
PUT SKIP LIST(@LOG -> LDML);
END LOGG;


/*************************************************************/
                    ***  PSTK1 SUBROUTINE  ***
                    ERASE A MEMBERSHIP IN SET 1
/*************************************************************/

PSTK1: PROCEDURE;
STAK.S1FIDT = ' ';
STAK.S1FIDR = 0;
STAK.S1BIDT = ' ';
STAK.S1BIDR = 0;
END PSTK1;
```

```
/**************************************************************/
                    ***  PSTK2 SUBROUTINE  ***
                    ERASE A MEMBERSHIP IN SET 2
/**************************************************************/

PSTK2: PROCEDURE;
STAK.S2FIDT = ' ';
STAK.S2FIDR = 0;
STAK.S2BIDT = ' ';
STAK.S2BIDR = 0;
END PSTK2;


/**************************************************************/
                  ***  PSRCH SUBROUTINE  ***
       SEARCH ROUTINE USED IN ROLLBACK TO LOCATE THE
       ENTRY OF THE RECORD TO BE MODIFIED IN THE JOUNAL
       FILE .. A BEFORE IMAGE IS OFTEN NEEDED AND THE
                 LINKING TO IT IS DONE HERE
/**************************************************************/

PSRCH: PROCEDURE;
PUT SKIP LIST(' ENTER PSRCH');
LRB = LRB + 1;
DO WHILE(@SRCH -> LOG.LTIME ^= @TRVS -> RFTIME);
IF (@SRCH -> LOG.LTIME = TRTIME) THEN DO;
ERROR_CODE = 'INCORRECT RFILE TIME';
CALL ERROR;
END;
@SRCH = @SRCH -> LOG.@BACK;
END;
@R = @SRCH -> LOG.@ROLL;
PUT SKIP LIST(' LEAVE PSRCH');
END PSRCH;


/**************************************************************/
                     ***  RMOD SUBROUTINE  ***
       ROLLBACK ROUTINE TO CHANGE A MODIFIED RECORD
                   BACK TO ITS BEFOREIMAGE
/**************************************************************/

RMOD: PROCEDURE;
PUT SKIP LIST(' ENTER RMOD');
SRB = SRB + 1;
CALL PSRCH;
RECR = @SRCH -> LOG.LREC;
TASK = @SRCH -> LOG.LTASK;
I = 1;
@PTR = @PNTR(I);
DO WHILE(@PTR -> NAME ^= TASK);
```

```
I = I +1;
@PTR = @PNTR(I);
END;
CNG = @R -> STAK.DATA;
CALL PMOD;
PUT SKIP LIST(' LEAVE RMOD');
END RMOD;


/*********************************************************/
                  ***  RCON SUBROUTINE  ***
       ROLLBACK ROUTINE TO RECONNECT A RECORD TO A SET
                  IT WAS DISCONNECTED FROM
/*********************************************************/

RCON: PROCEDURE;
PUT SKIP LIST(' ENTER RCON');
RRB = RRB + 1;
CALL PSRCH;
SET = @SRCH -> LOG.LSET;
CALL PCON;
I = 1;
@PTR = @PNTR(I);
DO WHILE(@PTR -> NAME ^= TASK);
I = I +1;
@PTR = @PNTR(I);
END;
@R -> STAK.DATA = @PTR -> REC(RECR).DATA;
PUT SKIP LIST(' LEAVE RCON');
END RCON;


/*********************************************************/
                  ***  RDIS SUBROUTINE  ***
       ROLLBACK ROUTINE TO DISCONNECT A RECORD FROM A
                       SET CONNECTION
/*********************************************************/

RDIS: PROCEDURE;
PUT SKIP LIST(' ENTER RDIS');
RRB = RRB + 1;
CALL PSRCH;
SET = @SRCH -> LOG.LSET;
CALL PDIS;
I = 1;
@PTR = @PNTR(I);
DO WHILE(@PTR -> NAME ^= TASK);
I = I +1;
@PTR = @PNTR(I);
END;
@R -> STAK.DATA = @PTR -> REC(RECR).DATA;
PUT SKIP LIST(' LEAVE RDIS');
END RDIS;
```

```
/**********************************************************/
                  ***  PERA SUBROUTINE  ***
       ROUTINE TO ERASE A RECORD FROM THE DATA BASE
/**********************************************************/

PERA: PROCEDURE;
PUT SKIP LIST('  ENTER PERA***');
EFLAG = 1;
IF FLAG = 0 THEN DO;
STDB = STDB + 1;
RDB = RDB + 1;
 CALL LOGG;
@LOG-> LSET = 0;
ALLOCATE STAK;
I = 1;
@PTR = @PNTR(I);
DO WHILE(@PTR -> NAME ^= TASK);
I = I +1;
@PTR = @PNTR(I);
END;
@STAK -> STAK.DATA = @PTR -> REC(RECR).DATA;
@LOG -> @ROLL = ADDR(STAK);
END;
IF (RECR > NREC ) THEN DO;
ERROR_CODE = 'NO RECORD FOR ERASE';
CALL ERROR;
END;
SET = 1;
CALL PDIS;
SET = 2;
CALL PDIS;
I = 1;
@PTR = @PNTR(I);
DO WHILE(@PTR -> NAME ^= TASK);
I = I +1;
@PTR = @PNTR(I);
END;
@PTR -> REC(RECR).DATA = 0;
EFLAG = 0;
PUT SKIP LIST('  LEAVE PERA***');
END PERA;


/**********************************************************/
                  ***  RERA SUBROUTINE  ***
       ROLLBACK ROUTINE TO ERASE A STORED RECORD
/**********************************************************/

RERA: PROCEDURE;
PUT SKIP LIST('  ENTER RERA***');
CALL PSRCH;
```

```
RECR = @R -> STAK.DATA;
STRB = STRB + 1;
CALL PERA;
PUT SKIP LIST('  LEAVE RERA***');
END RERA;


/***********************************************************/
                        ***  PGET SUBROUTINE  ***
       ROUTINE TO READ THE CONTENTS OF A RECORD ..
                             NO MODIFICATION
/***********************************************************/

PGET: PROCEDURE;
CALL LOGG;
@LOG -> @ROLL = NULL;
@LOG -> LSET = 0;
RDB = RDB + 1;
I = 1;
@PTR = @PNTR(I);
DO WHILE(@PTR -> NAME ^= TASK);
I = I +1;
@PTR = @PNTR(I);
END;
PUT SKIP LIST(@PTR -> REC(RECR).DATA);
END PGET;


/***********************************************************/
                       ***  PRES SUBROUTINE  ***
       RESTART ROUTINE .. RECORDS A RESTART POINT
/***********************************************************/

PRES: PROCEDURE;
CALL LOGG;
@LOG -> @ROLL = NULL;
@LOG -> LSET = 0;
END PRES;


/***********************************************************/
                       ***  PSTO SUBROUTINE  ***
       ROUTINE TO STORE A RECORD IN THE DATA BASE
/***********************************************************/

PSTO: PROCEDURE;
II = 1;
IF FLAG = 0 THEN RDB = RDB + 1;
ELSE RRB = RRB + 1;
I = 1;
@PTR = @PNTR(I);
DO WHILE(@PTR -> NAME ^= TASK);
I = I +1;
```

```
@PTR = @PNTR(I);
END;
DO WHILE(@PTR -> REC(II).DATA ^= 0 & II <= NREC);
 II = II + 1;
END;
IF II > NREC THEN DO;
ERROR_CODE = 'NO RECORD FOR STORE';
CALL ERROR;
END;
IF FLAG = 0 THEN DO;
RECR = II;
CALL LOGG;
SDB = SDB + 1;
ALLOCATE STAK;
@STAK -> STAK.DATA = II;
@LOG -> @ROLL = ADDR(STAK);
@LOG -> LSET = 0;
DOT = INDEX(CARD,'.');
CHNG = SUBSTR(CARD,EQ+2,DOT-1);
CNG = BINARY(CHNG);
END;
RDB = RDB + 1;
@PTR -> REC(II).DATA = CNG;
RECR = II;
PUT SKIP EDIT('RECORD STORED AT
',TASK,II)(COL(35),A,A,F(5));
END PSTO;


/***********************************************************/
                  ***  PREV SUBROUTINE  ***
       ROLLBACK COMMAND TO FIND THE LAST CHECKPOINT OR
               START OF THE PRIMARY ROLLBACK TASK
/***********************************************************/

PREV: PROCEDURE;
LRB = LRB + 1;
@CURR = @TAIL;
II = 1;
DO WHILE(II = 1);
IF (@CURR -> LOG.LDML = 'RESTART   ') THEN DO;
IF (@CURR -> LOG.LTASK = TASK ) THEN II = 2;      /*
TASK*/
ELSE @CURR = @CURR -> LOG.@BACK;
END;
ELSE IF @CURR -> LOG.@BACK = NULL THEN DO;
ERROR_CODE = ' NO LOG ENTRY FOR P-RBACK TASK';
CALL ERROR;
END;
ELSE @CURR = @CURR -> LOG.@BACK;
END;
PUT SKIP LIST('RESTARTED TASK IS :    ',TASK,'  AT THE
TIME  ',
     @CURR -> LOG.LTIME);
TRTIME = @CURR -> LOG.LTIME;
END PREV;


/**********************************************************/
                 ***  PROL SUBROUTINE  ***
     MAIN ROLLBACK PROCEDURE CALLS AND SECONDARY CALLS
/**********************************************************/

PROL: PROCEDURE;
CALL LOGG;
CALL PREV;
CALL PLIN;
CALL PFOR;
CALL PRBK;
CALL PSCN;
@SRLH = NULL;
@SRLT = NULL;
END PROL;


/**********************************************************/
                   ***  PLIN SUBROUTINE  ***
       INITALIZE THE TASK LIST AND SET COUNTER 'K'
/**********************************************************/

PLIN: PROCEDURE;
TL = TL + 1;
DO II = 1 TO NTASK;
TLIST(II) = ' ';
END;
K = 1;
END PLIN;


/**********************************************************/
                    ***  PLIS SUBROUTINE  ***
       UPDATE THE TASK LIST WITH THE NAME OF CORRECTED
                     (ROLLED BACK) TASK
/**********************************************************/

PLIS: PROCEDURE;
TL = TL + 1;
K = K + 1;
TLIST(K) = TASK;
END PLIS;
```

```
/*************************************************************/
                    ***  PFOR SUBROUTINE  ***
       THE FORWARD SEARCH OF THE JOURNAL FILE FOR ENTRYS
       IN THE ROLLBACK FILE AND SECONDARY ROLLBACK LIST
/*************************************************************/

 PFOR: PROCEDURE;
 PUT SKIP LIST(' ENTER PFOR ');
 SRFLAG = 1;
 @TRF = NULL;
 @HRF = NULL;
 LRB = LRB + 1;
 @CURR = @CURR -> LOG.@FWD;
 DO WHILE(@CURR -> LOG.LTIME ^= TTIME);
 IF ((@CURR -> LOG.LDML ='MODIFY    ' ] @CURR -> LOG.LDML
= 'STORE     '
  ] @CURR -> LOG.LDML = 'CONNECT   ' ] @CURR -> LOG.LDML =
'DISCONNECT'
  ] @CURR -> LOG.LDML = 'ERASE     ' ) & @CURR ->
LOG.LFLAG = 0) THEN
    DO;
 IF (TASK = @CURR -> LOG.LTASK ) THEN DO;       /*  TASK*/
 II = 1;
 URB = URB + 1;
 DO WHILE(UPDATE(II) ^= 0 & II <= NREC);
 IF (UPDATE(II) = @CURR -> LOG.LREC) THEN GO TO ZIP;
 II = II + 1;
 PUT SKIP LIST('   LEAVE   PFOR *2*');
 END;
 UPDATE(II) = @CURR -> LOG.LREC;
 SRFLAG = 0;
 PUT SKIP LIST(UPDATE(II));
 ALLOCATE RFILE;
 RFRB = RFRB + 1;
 IF @HRF = NULL THEN @HRF = @RFILE;
 ELSE @TRF -> @RFWD = @RFILE;
 @RFBK = @TRF;
 @TRF = @RFILE;
 @RFWD = NULL;
 PUT SKIP LIST ('ADDITION TO ROLL BACK FILE');
 @RFILE -> RFDML = @CURR -> LDML;
 @RFILE -> RFREC = @CURR -> LREC;
 @RFILE -> RFTIME = @CURR -> LTIME;
 PUT SKIP LIST(@RFILE -> RFDML, @RFILE -> RFREC, @RFILE ->
RFTIME);
 PUT SKIP LIST('   LEAVE   PFOR *1*');
 END;
 PUT SKIP LIST('   LEAVE   PFOR ***');
 IF (SRFLAG = 0 ) THEN DO;
 I = 1;
 @PTR = @PNTR(I);
 DO WHILE(@PTR -> NAME ^= TASK);
 I = I +1;
```

```
@PTR = @PNTR(I);
IF I > NTASK  THEN DO;
ERROR_CODE = 'ERROR PFOR';
CALL ERROR;
END;
END;
CALL PASK;
END;
END;
ELSE;
ZIP: @CURR = @CURR -> LOG.@FWD;
END;
PUT SKIP LIST('   LEAVE   PFOR *3*');
END PFOR;


/*************************************************************/
                    ***  PSCN SUBROUTINE  ***
       THIS ROUTINE TAKES THE TASKS PLACED ON THE
       SECONDARY ROLLBACK LIST AND MAKES THEM A PRIMARY
       ROLLBACK TASK TO COMPLETE THE ROLLBACK PROCEDURE
/*************************************************************/

PSCN: PROCEDURE;
PUT SKIP LIST('ENTER PSCN');
SRRB = SRRB + 1;
@SCN = @SRLH;
DO WHILE(CNT ^= 0);
RECR = @SCN -> SRL.REC;         /*   RECORD  */
TASK = @SCN -> SRL.TASK;
CALL PREV;
CALL PFOR;
CALL PRBK;
@SCN = @SCN -> @SRLFWD;
SRRB = SRRB + 1;
CNT = CNT - 1;
END;
END PSCN;


/*************************************************************/
                   ***  PASK SUBROUTINE  ***
       ROUTINE TO CHECK A JOURNAL ENTRY OTHER THAN THE
       PRIMARY ROLLBACK TASK FOR DETERMINING IF IT
       SHOULD BE ADDED TO THE SECONDARY ROLLBACK LIST
/*************************************************************/

PASK: PROCEDURE;
PSRB = PSRB + 1;
PUT SKIP LIST('  ENTER PASK ***');
J = 1;
DO WHILE( @PTR -> NSDL >= J);
IF (@CURR -> LOG.LTASK = @PTR -> SDLT(J) &
   @CURR -> LOG.LREC = @PTR -> SDLR(J)) THEN DO;
PUT SKIP LIST(' ENTER PASK *1*');
SRRB = SRRB + 1;
IF @SRLH = NULL THEN CALL PSRL;
ELSE DO;
@SRLL = @SRLH;
DO WHILE(@SRLL -> SRL.@SRLFWD ^= NULL);
PUT SKIP LIST('   LEAVE   PASK *1*');
IF (@CURR -> LOG.LTASK = @SRLL -> SRL.TASK ) THEN RETURN;
@SRLL = @SRLL -> SRL.@SRLFWD;
END;
IF (@CURR -> LOG.LTASK = @SRLL -> SRL.TASK ) THEN RETURN;
CALL PSRL;
END;
END;
J = J + 1;
END;
PUT SKIP LIST('   LEAVE   PASK ***');
END PASK;


/***********************************************************/
                    ***  PSRL SUBROUTINE  ***
       ROUTINE TO ADD JOURNAL ENTRY TO THE SECONDARY
                        ROLLBACK LIST
/***********************************************************/

PSRL: PROCEDURE;
PUT SKIP LIST(' ENTER PSRL ***');
TL = TL + 1;
DO L = 1 TO K;
IF TLIST(L) = @CURR -> LOG.LTASK THEN RETURN;
END;
CALL PLIS;
ALLOCATE SRL;
IF @SRLH = NULL THEN @SRLH = @SRL;
ELSE @SRLT -> @SRLFWD = @SRL;
@SRLBK = @SRLT;
@SRLT = @SRL;
@SRLFWD = NULL;
@SRL -> SRL.TASK = @CURR -> LOG.LTASK;
@SRL -> SRL.REC = @CURR -> LOG.LREC;
@SRL -> SRL.STIME = @CURR -> LOG.LTIME;
PUT SKIP LIST('NEW SEC. ROLL BACK ALLOCATION ');
PUT SKIP LIST(@SRL -> SRL.TASK, @SRL -> SRL.STIME);
CNT = CNT + 1;
PUT SKIP LIST('   LEAVE   PSRL ***');
END PSRL;
```

```
/***************************************************************/
                    ***  PRBK SUBROUTINE  ***
       BEGINS THE ROLLBACK OF ENTRIES IN THE PRIMARY
                         ROLLBACK LIST
/***************************************************************/

PRBK: PROCEDURE;
PUT SKIP LIST(' ENTER PRBK ***');
RFRB = RFRB + 1;
@TRVS = @HRF;
DO WHILE(@TRVS -> @RFWD ^= NULL);
RECR = @TRVS -> RFREC;
DML = @TRVS -> RFDML;
CALL RCOM;
@TRVS = @TRVS -> @RFWD;
END;
DML = @TRVS -> RFDML;
RECR = @TRVS -> RFREC;
CALL RCOM;
@TRF = NULL;
@HRF = NULL;
PUT SKIP LIST('   LEAVE   PRBK ***');
END PRBK;


/***************************************************************/
                    ***  RCOM SUBROUTINE  ***
        SENDS THE ROLLBACK COMMANDS TO CORRECT ROUTINE
/***************************************************************/

RCOM: PROCEDURE;
PUT SKIP LIST(' ENTER RCOM ***');
CALL STT1;
CALL STT2;
@SRCH = @TAIL;
IF (DML = 'ERASE      ') THEN CALL RSTO;
ELSE IF (DML = 'STORE      ') THEN CALL RERA;
ELSE IF (DML = 'MODIFY     ') THEN CALL RMOD;
ELSE IF (DML = 'CONNECT    ') THEN CALL RDIS;
ELSE IF (DML = 'DISCONNECT') THEN CALL RCON;
CALL STT1;
CALL STT2;
PUT SKIP LIST('   LEAVE   RCOM *1*');
END RCOM;


/***************************************************************/
                    ***  RSTO SUBROUTINE  ***
        ROLLBACK ROUTINE TO RESTORE ERASED RECORD
/***************************************************************/

RSTO: PROCEDURE;
CALL PSRCH;
```

```
PUT SKIP LIST(' ENTER RSTO ***');
CNG = @R -> STAK.DATA;
CALL PSTO;
IF((@R -> STAK.S1BIDT ^= ' ')|(@R -> STAK.S1FIDT ^= '
'))THEN DO;
SET = 1;
CALL PCON;
END;
IF((@R -> STAK.S2BIDT ^= ' ')|(@R -> STAK.S2FIDT ^= '
'))THEN DO;
SET = 2;
CALL PCON;
END;
STRB = STRB + 1;
PUT SKIP LIST('   LEAVE   RCTO *1*');
END RSTO;


/************************************************************/
                    ***  PMOD SUBROUTINE  ***
       ROUTINE TO MODIFY THE CONTENTS OF THE DATA BASE
/************************************************************/

PMOD: PROCEDURE;
CALL LOGG;
@LOG -> LSET = 0;
IF FLAG = 0 THEN DO;
DOT = INDEX(CARD,'.');
CHNG = SUBSTR(CARD,EQ+2,DOT-1);
CNG = BINARY(CHNG);
STDB = STDB + 1;
ALLOCATE STAK;
I = 1;
@PTR = @PNTR(I);
DO WHILE(@PTR -> NAME ^= TASK);
I = I +1;
@PTR = @PNTR(I);
END;
@STAK -> STAK.DATA = @PTR -> REC(RECR).DATA;
LDB = LDB + 1;
@LOG -> @ROLL = ADDR(STAK);
END;
@PTR -> REC(RECR).DATA = CNG;
IF FLAG = 0 THEN RDB = RDB + 1;
ELSE RRB = RRB + 1;
END PMOD;
```

```
/*********************************************************/
                   ***  LGR SUBROUTINE  ***
         ROUTINE TO LIST THE JOURNAL FILE CONTENTS
/*********************************************************/

LGR: PROCEDURE;
PUT SKIP LIST('     THE LOG RECORD LIST');

@ADJ = @HEAD;
DO WHILE(@ADJ -> LOG.@FWD ^= NULL);
PUT SKIP LIST(@ADJ -> LOG.LTIME, @ADJ -> LOG.LTASK,
  @ADJ ->  LOG.LREC, @ADJ -> LOG.LDML);
@ADJ = @ADJ -> LOG.@FWD;
END;
PUT SKIP LIST(@ADJ -> LOG.LTIME, @ADJ -> LOG.LTASK,
  @ADJ ->  LOG.LREC, @ADJ -> LOG.LDML);
END LGR;


/*********************************************************/
                ***  STT1 SUBROUTINE  ***
           LIST THE CONTENTS OF RECORDS IN SET 1
/*********************************************************/

STT1: PROCEDURE;
PUT SKIP LIST ('SET 1 ... TASK  & RECORD');
IF @S1H ^= NULL THEN DO;
@ADJ = @S1H;
DO WHILE(@ADJ -> SET1.S1FWD ^= NULL);
PUT SKIP(2)LIST(@ADJ -> SET1.S1TASK, @ADJ -> SET1.S1REC);
@ADJ = @ADJ -> SET1.S1FWD;
END;
PUT SKIP(2)LIST(@ADJ -> SET1.S1TASK, @ADJ -> SET1.S1REC);
END;
PUT SKIP LIST('  END OF SET 1 LIST');
END STT1;


/*********************************************************/
                 ***  STT2 SUBROUTINE  ***
           LIST THE CONTENTS OF RECORDS IN SET 2
/*********************************************************/

STT2: PROCEDURE;
PUT SKIP LIST ('SET 2 ... TASK  & RECORD');
IF @S2H ^= NULL THEN DO;
@ADJ = @S2H;
DO WHILE(@ADJ -> SET2.S2FWD ^= NULL);
PUT SKIP(2)LIST(@ADJ -> SET2.S2TASK, @ADJ -> SET2.S2REC);
@ADJ = @ADJ -> SET2.S2FWD;
END;
PUT SKIP(2)LIST(@ADJ -> SET2.S2TASK, @ADJ -> SET2.S2REC);
END;
```

```
 PUT SKIP LIST('  END OF SET 2 LIST');
 END STT2;

 /************************************************************/
                      END OF PROGRAM
 /************************************************************/

 PUT SKIP LIST('NORMAL TERM');
 END PROG5;


//GO.SYSIN DD *
'RESTART A 2.    '
'RESTART B 2.    '
'RESTART C 2.    '
'GET J 5.'
'MODIFY J 6 = 4.'
'CONNECT A 2 = 2.'
'GET A 5.'
'GET A 1.'
'GET A 13.'
'DISCONNECT A 2 = 2.'
'MODIFY A 12 = 9.'
'GET C 5.'
'GET C 1.  '
'GET C 13.'
'GET C 11.'
'GET C 11.'
'GET C 5.'
'MODIFY E 8 = 1.'
'GET E 5.'
'CONNECT E 7 = 2.'
'DISCONNECT E 7 = 2.'
'GET E 7.'
'GET E 13.'
'MODIFY F 8 = 5.'
'MODIFY F 11 = 6.'
'MODIFY F 8 = 1.'
'GET F 5.'
'GET F 11.'
'GET F 13.'
'GET K 11.'
'GET K 5.'
'GET K 7.'
'GET K 4.'
'CONNECT K 7 = 2.'
'DISCONNECT K 7 = 2.'
'RBACK G 0.'
```

SIMULATION OF A SELECTIVE ROLLBACK AND RECOVERY METHODOLGY

by

Kirk A. Norsworthy

B. S., University of Kansas, Lawrence, Kansas, 1975

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements of the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1978

In order to check the effectiveness of a selective rollback and recovery algorithm, a simulation of the algorithm was run under a Codasyl type data base. This simulation was weighted against the performance figures from a simulation on deadlocks in data base management systems and calculations of the predicted CPU overhead were made.

This simulation showed that the selective rollback and recovery algorithm worked well in even the highest cases of data integration. This simulation modeled the deadlock problem in a multi-user environment to simulate the performance of the algorithm coupled with a deadlock detection scheme. The motivation for this research is the eventual integration of computing resources into distributed data sharing.