Ra [illegible]

Multics Technical Bulletin

MTB-370

To: Distribution

From: Michael Asherman

Subject: Transaction Processing Extensions

Date: 1 May 1978

Transaction Processing Extensions

contents:

Introduction

Considerable interest exists in having a Multics transaction processing capability. The term has been used so loosely that, strangely enough, it isn't exactly clear what this means. Does Multics already support transaction processing? There is no point in debating over terminology; what are the people who want this feature really getting at? It has something to do with data base sharing and recovery, but these are very broad issues, and there is no agreement on precisely how the concept of a transaction relates to them.

Purpose

This MTB offers a precise definition to the term "transaction" that is in the spirit of its common usage. Having done this, I propose that the featu re be implemented in Multics by making the specific changes described below.

Summary

A transaction is defined as an atomic operation on a data base comprised of vfile_ files. Only indexed files will be supported in the initial implementation. Arbitrarily complex procedures can be given the appearance of taking place indivisibly if they are invoked as transactions, without requiring any explicit user programming to handle synchronization and avoid inconsistencies arising from interrupted operations. The implementation involves using a permanent transaction control file (tcf) in conjunction with a collection of data base files; temporary files are used as reference lists while performing transactions. Vfile_ automatically takes care of synchronizing access and adjusting inconsistencies due to interruption at the level of individual records and index pages, using a before/after image representation of these items. When one considers the expense and inconvenience typically associated with solving this problem, it may come as a surprise that a highly efficient implementation can be built on Multics with minimal reprogramming of applications.

Recovery

Recovery from interrupted transactions is handled automatically by vfile_, as it encounters individual items (records and index pages) left locked by dead processes. Thus no explicit user intervention is required, although one has the option of cleaning up such garbage at any time, so long as the reference lists are still available.

Usage

The most basic usage of transactions requires only a knowledge of vfile_ and the new transact command. One must define a transaction control file, and associate an attachment to this file with the individual data base files via the new -transaction attach option. Having done this, transactions can be executed as command lines given to the transact command. Even if the data base is not shared, this feature may be desirable as a means of avoiding inconsistencies from the interruption of complex data base transformations. When the data base is shared, automatic synchronization provides an additional incentive for using transactions.

transaction size

If a large transaction can be broken up into several smaller transactions without jeopardizing the data base's consistency between parts, this should be done for efficiency. In other words, one should not use unnecessarily long atomic operations, although there is no actual limit imposed by the implementation. Thus, for example, one might break up a background job that modifies a substantial portion of the entire data base into a series of smaller transactions. In typical usage, individual transactions should involve a relatively small part of the entire data base, but still be sufficiently complex to make the fixed overhead per transaction small compared to the cost of the entire transaction.

explicit synchronization

Users are never required to lock explicitly, since synchronization is entirely automatic under transactions. However, there is a provision for explicit user synchronization as well, to facilitate enforcing protocols for minimizing contention and eliminating the possibility of deadlock between transactions. Unless all transactions observe a conventional order of modifications to individual data base items, the possibility of deadlock exists. If arbitrary sequences of changes may occur in transactions, then deadlock can at least be detected and prevented, although the situation that calls for special action and the interference have not been eliminated. Such a capability for detecting and preventing deadlock does not presently exist, and I do not propose to deal with the matter any further at this time; this is not to belittle the issue, but to avoid delaying the basic TP extensions. When time permits, a number of improvements should be made to the system set_lock_ procedure, including deadlock prevention as well as a complete redesign for better performance.

Implementation

Transactions are atomic because their state of completion is reduced to the setting of a flag on a tcf entry for the given transaction number. Several internal vfile_ changes are proposed to support transactions. Each record and index node (page) will have some additional header words to support having both a before and an after image. When an item is modified on behalf of a transaction, only the after image is affected; the item is left locked until a checkpoint or rollback occurs, indicating the normal or abnormal completion of this transaction. Every data base item altered by a given transaction acquires an after image marked with the transaction's identifying number. Passive references never need wait or lock in order to examine an item. If a reference is made while another transaction is modifying the same object, the passive reference may have to examine the tcf in order to determine whether the before or the after image applies. In general, however, the tcf need not be examined, because the before image always applies except while a checkpoint of a transaction that modified the object is in progress. A temporary list of references is implicitly maintained by vfile_ to keep track of all items either modified or referenced passively in the course of each transaction. When a subsequent reference is made to an item for which a previous passive reference occurred in the same transaction, the fact that this item has undergone an asynchronous modification in another transaction is automatically detected and treated as an error. The list of passive references is always examined at checkpoint time as well. Thus the passive reference list serves as a basis for verifying the atomicity of transactions before they are permitted to complete successfully.

Reference list entries for modified objects are used at checkpoint or rollback time in order to unlock any modified records and index pages after either resetting their before images or discarding their after images.

Cost

storage

Use of transactions entails some additional cost over the basic expense of the underlying vfile_ operations The added storage requirement is several words per record and index node allocation; this may be insignificant if the records are long, but not if the records are very short (i.e. on the order of 10 words or less). With short records, the storage overhead can be kept small by aggregating logical records into groups stored in vfile_ records of a more reasonable size (e.g. pages); in fact, the implementors of the Multics Data Base Management facility have proposed such usage. In addition to the fixed overhead per record, one has the expense of keeping a transaction control file, and there is temporary storage required for the reference list files and storage for any after images in addition to the before images while transactions are in progress. However, this expense would tend to be small compared to the overall data base size, and proportional to the rate of file activity.

processing

An additional amount of processing takes place on each vfile_ operation in order to add entries into the reference list for the current transaction. This expense can be reduced to a small fixed cost per operation, since the reference list would tend to be very small in comparison to the data base being referenced. In general, the processing associated with manipulating the tcf and reference list should be less than or comparable to that implied by typical data base operations, a small fraction of the total processing required per transaction.

I/O

No extra I/O is required to guarantee the atomicity of transactions if they are sufficiently small to keep the data base pages they touch in core until checkpoint. In the limit of very large transactions, the extra I/O would not exceed the minimum I/O (i.e. total I/O can no more than double); however, the benefit of aggregating half of each modification at checkpoint time may reduce this additional cost. To be protected from losing any transactions, one also needs to journalize transactions before acknowledging their receipt. This requires 1 extra I/O for every N transactions, where N is the number of transactions accepted in the input queue before acknowledgement. The implementation has no inherent added I/O requirement for atomicity, because before images are not copied, but temporarily retained in their initial allocations.

interference

Any number of concurrent transaction may be operating on a common data base without necessarily interfering with each other. They may actually benefit from concurrency, to the degree that commonly referenced pages tend to stay in core because of the virtual paging behavior. Since passive operations don't wait or lock, they never interfere with other transactions; only modifications can lead to interference. Each low level item modified (record or index node) remains locked until the end of the transaction, but the files as a whole are never implicitly left locked. Therefore, the most significant component of interference arises from multiple transactions attempting to modify the same low level object at the same time. The larger the transaction, the longer it will interfere with others attempting to modify common items. However, at the level of the file as a whole, transactions do not increase the amount of contention over the file lock, because they don't leave the file locked any longer than would be the case without transactions.

recovery

If a transaction is interrupted and not resumed, no immediate adjustment is needed in order to continue processing other transactions on the data base, so it is not necessary to wait in order to recover from pure interruptions. As vfile_ encounters individual items left locked by a transaction in a dead process, the items are adjusted before proceeding with a new modification. The cost of adjusting any interrupted vfile_ operation is comparable to the basic cost of the operation itself. Thus the total cost of cleaning up an interrupted transaction is comparable to the cost of completing the transaction successfully; this expense may be postponed indefinitely, being incurred piecemeal through automatic garbage collection. After a system failure that causes a loss of data, as well as pure interruption, the recovery procedure is more involved, unless one is willing to lose some number of transactions. In such cases, an extra delay will be needed to restore a complete data base snapshot and/or reapply some number of journalized transactions. For a more thorough discussion of recovery in general, see MTB-369.

Example

Consider the following application in a simple banking system. The data base is a file with singly keyed records, where the keys give the account number, and the records contain a number which is the balance in that account. Suppose that there is also a single record in which the bank's total assets are maintained; this record corresponds to account number 000000.

Withdrawals might be made through this exec_com:

withdraw.ec

```
&command_line off io control sw seek_head &1 &if [nless [io read
sw 15] &2] &then &goto insufficient_funds io position sw 0 -1 io
rewrite sw [minus [io read sw 15] &2] io control sw seek_head
000000 io rewrite sw [minus [io read sw 15] &2] &quit &label
insufficient_funds io position sw 0 -1 ioa_ "Withdrawal refused.
Balance is only $^a" [io read sw 15]
```

This routine takes two arguments, the account number followed by the amount to be withdrawn. If the account has sufficient funds, the given amount is subtracted from both this account, and from the bank's total assets. Otherwise, the withdrawal is refused, and a message is printed with the account's present balance. The data base file is assumed to have been attached and opened for keyed_sequential_update on an I/O switch named "sw".

problems without transactions

As it stands, there are several potential difficulties in using withdraw.ec, all of which can be avoided by performing withdrawals as transactions. For example, if the process that is performing this operation dies before adjusting the bank's total assets, but after subtracting from the individual account, then the data base will be left in an inconsistent state where the total bank assets figure does not reflect the sum of the individual accounts. Another potential cause of inconsistency arises if, for example, we suppose that one may have joint accounts. In this case, it may happen that several concurrent withdrawals are attempted on the same account. The concurrent operations might interfere with each other in such a way as to yield inconsistent results; e.g. the first withdrawal might have obtained the current balance just before the second withdrawal began, and the second withrawal might complete before the first. If this happens, the result of the second withdrawal might get overritten on finishing the first, and the net result would be to lose any record of the second withdrawal.

Besides these cases, one also has the problem of producing a meaningful report of a cross section of the data base while concurrent withdrawals are permitted. Unless some further action is taken, the report may not be a valid snapshot; the subtotal of individual accounts displayed might not actually correspond to any instantaneous subtotal that ever existed.

solution

The aforementioned difficulties are eliminated by using the proposed transaction processing features. This entails associating a tcf with the data base before opening it, and invoking withdrawals as individual transactions via the transact command. The setup procedure might consist of the following sequence of command lines:

```
io attach tcf_sw vfile_ tcf_path -share io open tcf_sw
keyed_sequential_update io attach sw vfile_ data_base_path -share
-stationary -transaction tcf_sw io open sw
keyed_sequential_update
```

A withdrawal is done by typing the following command lines:

```
assign_transaction tcf_sw transact tcf_sw "ec withdraw
<acct_number> <amount>"
```

The purpose of using assign_transaction is to cause a transaction number to be printed out before initiating the transaction, so that in the event of an interruption, there is a basis for determining whether the most recently issued transaction has been performed, even though the message acknowledging this fact may not yet have been produced at the user's terminal. Thus, after returning from such an interruption, one would examine the tcf entry for any transaction whose completion is in question.

MPM Documentation

Draft MPM documentation for the proposed TP related changes follows.

command: assign_transaction, atc

Function

reserves a unique transaction number for the current transaction, and optionally prints the new transaction code. The tcf switch must be opened for modification, so that a new entry can be created.

Usage

```
atc tcf_sw {-brief , -bf} {t_code}
```

tcf_sw

names an I/O switch attached to the transaction control file (tcf).

-brief

optionally suppresses the standard printout of this command.

t_code

optionally specifies the new transaction number to be assigned to the current transaction and inserted into the tcf. If omitted, the next available unique transaction code will be assigned.

Output

is of the form: transaction N
where N is the new transaction number.

Reference

See the writeup of the transact command in the MPM.

command: checkpoint, chp

Function

attempts to complete the current transaction on a data base associated with the given transaction control switch. The current transaction number becomes undefined after a successful checkpoint.

Usage

```
chp tcf_sw {-brief , -bf}
```

tcf_sw

names an I/O switch attached to the transaction control file (tcf).

-brief

optionally suppresses the standard printout of this command.

Output

is of the form: checkpoint: N
where N is the number of the transaction just completed.

Reference

See the writeup of the transact command in the MPM.

command: rollback, rlb

Function

undoes all modifications made on behalf of the current transaction in the specified data base. The transaction number for this tcf switch is then reset to zero; i.e., the current transaction becomes undefined.

Usage

```
rlb tcf_sw {-brief , -bf}
```

tcf_sw

names an I/O switch attached to the transaction control file (tcf).

-brief

optionally suppresses the standard printout of this command.

Output

is of the form: rollback: N
where N is the number of the transaction just aborted.

Reference

See the writeup of the transact command in the MPM.

command: transact, trn

Function

executes a given command line as an atomic transaction on a specified data base.

Usage

```
trn tcf_sw {-brief , -bf} command_line
```

tcf_sw

names an I/O switch attached to the transaction control file (tcf).

command_line

is a Multics command line which need not be enclosed in quotes unless it contains special characters.

-brief

optionally suppresses the standard printout of this command.

Output

is of the form: Done transaction N.
where N is the number of the transaction just completed.

References

See the writeup of the transact subroutine in the MPM. Also see writeups of the following related commands: assign_transaction checkpoint rollback transaction_code transaction_status

command: transaction_code, trc

Function

prints, and optionally resets the current transaction number for a given tcf switch. The control file itself is not referenced or altered by this operation, permitting purely passive transactions to have only read access to the tcf.

Usage

```
trc tcf_sw {-brief , -bf} {t_code}
```

tcf_sw

names an I/O switch attached to the transaction control file (tcf).

-brief

optionally suppresses the standard printout of this command.

t_code

specifies the number to be taken as the current one. If omitted, the current transaction number is unchanged.

Output

is of the form: transaction code: N
where N is the current transaction number (obtained before changing).

Reference

See the writeup of the transact command in the MPM.

command: transaction_status, trs

Function

prints items of information about a transaction for a specified tcf switch. This includes the transaction number, its completion status, and optionally counts of passive and non-passive references.

Usage

```
trs tcf_sw {-brief , -bf , -verify , -vf} {t_code}
```

tcf_sw

names an I/O switch attached to the transaction control file (tcf).

-brief, -bf

optionally suppresses the examination of reference list entries.

-verify, -vf

causes a check of all passive references made in the transaction for possible asynchronous changes. If a previously referenced item has been changed, an error message is printed, indicating that this transaction will be unsuccessful.

t_code

is the transaction number whose status is to be printed. If omitted, the current transaction is assumed.

Output

is of the form: transaction N passive refs: p, non-passive refs: n
where N is the transaction number,
p is the number of data base items referenced without alteration,
n is the number of items modified so far in this transaction.

Reference

See the writeup of the transact command in the MPM.

entry: transact_$assign_code

Function

reserves a unique transaction number for the current transaction and returns the new transaction code. The tcf switch must be opened for modification, so that a new entry can be created.

Usage

call transact_$assign_code (tcfp , cur_tcode , code);

declaration

dcl transact_$assign_code entry (ptr, fixed (35), fixed (35));

arguments

tcfp

points to an iocb for the transaction control file (Input).

cur_tcode

is set to the new transaction number (Output).

code

is a standard system error code (Output).

Notes

A transaction number can also be assigned via the transaction_code entry. The user is not required to preassign a transaction number at all, in which case one will automatically be assigned upon making the first reference to a data base item for the new transaction.

entry: transact_$checkpoint

Function

attempts to complete the current transaction on a data base associated with a given transaction control switch. The current transaction number becomes undefined if the checkpoint is successful.

Usage

call transact_$checkpoint (tcfp , cur_tcode , code);

declaration

dcl transact_$checkpoint entry (ptr, fixed (35), fixed (35));

arguments

tcfp

points to an iocb for the transaction control file (Input).

cur_tcode

is set to transaction number just completed (Output).

code

is a standard system error code (Output).

entry: transact_$rollback

Function

undoes all modifications that have been made on behalf of the current transaction in a specified data base.

Usage

```
call transact_$rollback (tcfp , cur_tcode , code);
```

declaration

```
dcl transact_$rollback entry (ptr, fixed (35), fixed (35));
```

arguments

tcfp

points to an iocb for the transaction control file (Input).

cur_tcode

is set to the transaction number just aborted (Output).

code

is a standard system error code (Output).

Notes

The effect of a rollback is logically invisible outside the current transaction, except possibly in its immediate cleaning up of accumulated garbage (after images). The transaction code for a rolled back transaction is not reused. After issuing a rollback, the caller's transaction number for the given tcf switch becomes undefined, and the data base is restored to its state following the last checkpoint.

entry: transact_$status

Function

returns various items of information about a transaction for a specified tcf switch. This includes the transaction number, its completion status, and optionally counts of passive and non-passive references.

Usage

```
     call transact_$status (tcfp , cur_tcode ,  ts_status_word  ,
ts_infop , code);
```

declaration

```
     dcl  transact_$status  entry  (ptr,  fixed  (35),  bit  (36)
aligned,
  ptr,fixed (35));
```

ts_info

```
     dcl 1 ts_info based ( ts_infop ) ,
     2 flags,
          3 verify bit ( 1 ) unal, /* causes data base  items  to
be checked */
          3 version fixed , /* set to current version by user  --
Input */
     2 passive_refs fixed ( 34 ) , /* Output */
     2 non_passive_refs fixed ( 34 ) , /* Output */
     2  pad  fixed  ;  /*  reserved  for  future  use  */  dcl
ts_info_version_0 static internal fixed options ( constant ) init
( 0 ) ;
```

ts_status_flags

```
     dcl 1 ts_status_flags based ( addr ( ts_status_word ) ) ,
     2 defined bit ( 1 ) unal , /* set if transaction code  found
in tcf */
     2 status fixed ( 34 ) unal; /* 0 = incomplete , 1 = done , 2
= aborted */
```

arguments

tcfp

points to an iocb for the transaction control file (Input).

cur_tcode

is the transaction number for which status information is desired, or set to 0 to specify the current transaction. If this is zero, then the returned value will be the current transaction number (Input/Output).

ts_status_word

contains a code defining the status of this transaction as one of the following (Output): undefined - no tcf entry exists incomplete - in progress, but not yet checkpointed done - successfully checkpointed (can't rollback) aborted - rolled back (can't checkpoint)

ts_infop

points to a structure, ts_info, in which the counts of references made by the transaction are to be returned. If null, this information is not obtained (Input).

ts_info.verify

if set, causes the list of passively referenced items for this transaction to be checked for possible asynchronous changes. If a change is detected, the returned code is set to error_table_$asynch_change, indicating that this transaction will be unsuccessful (Input).

ts_info.version

is the version number for this info structure, which should be set to ts_info_version_0 (Input).

ts_info.passive_refs

is the number of distinct items referenced passively (not modified) so far in this transaction (Output).

ts_info.non_passive_refs

is the number of distinct data base items modified so far in this transaction (Output).

code

is a standard system error code (Output).

entry: transact_$transaction_code

Function

returns and optionally resets the current transaction number for a given tcf switch. The control file itself is not referenced or altered by this operation, permitting purely passive transactions to have only read access to the tcf.

Usage

call transact_$transaction_code (tcfp , cur_tcode , next_tcode , code);

declaration

```
dcl transact_$transaction_code entry (ptr, fixed (35), fixed
(35),
  fixed (35);
```

arguments

tcfp

points to an iocb for the transaction control file (Input).

cur_tcode

is the current transaction number (before changing) (Output).

next_tcode

is the new transaction number or zero, if no change is desired (Input).

code

is a standard system error code (Output).

Notes

When a transaction is known to involve no data base alterations, this entry may be used to initialize the transaction number to a unique value, thereby avoiding the necessity of modifying the tcf in order to reserve new code. Unless the transaction number has been initialized, a tcf entry will automatically be assigned on the first reference to a data base item in the current transaction; the default behavior requires that the tcf be opened for modification.

entry: transact_

Function

executes a given command line as an atomic transaction on a specified data base. Handlers are established the the cleanup and program_interrupt conditions. The cleanup handler causes the transaction to be rolled back if, for example, the user quits and releases. The program_interrupt handler permits one to rollback and reexecute the command line by typing pi from command level.

Usage

```
call transact_ (tcfp , cur_tcode , command_line , code);
```

declaration

```
dcl transact_entry (ptr, fixed (35), char (*), fixed (35));
```

arguments

tcfp

points to an iocb for the transaction control file (Input).

cur_tcode

is set to transaction number just completed (Output).

command_line

is a Multics command line which need not be enclosed in quotes unless it contains special characters.

code

is a standard system error code (Output).

Transactions

definition

A transaction is a unit of processing which has the appearance of taking place as an indivisible, atomic operation. Arbitrary procedures involving any collection of vfile_ indexed files may be invoked as as transactions via this subroutine.

appearance

A partially completed transaction terminates either by a successful checkpoint operation, or by a rollback. That is to say, until a checkpoint occurs, the data base appears unchanged, except within the current transaction. Any data base modifications which a transaction makes appear simultaneously, outside the transaction which makes them, when the checkpoint takes place.

purpose

There are two major reasons for encapsulating a procedure as a transaction. The first is to simplify the user's task of handling inconsistencies that can arise from interrupted operations which are not resumed (e.g. because of a system crash or an application program error). Second, in the event that a data base is shared among independent processes, the entire burden of synchronizing file access is removed from the user and automatically managed by the system transaction processing facility.

tcf switch

Each independent transaction server (task or process which performs transactions) requires an I/O switch that associates the transactions with a particular data base. This switch is attached by the user to a permanent transaction control file (tcf) that is used in conjunction with the collection of files comprising a single logical data base.

transaction codes

A transaction has a unique identifying code associated with its tcf switch. Initially and after a checkpoint or rollback, this number is zero, indicating that no current transaction is defined for the given tcf switch. A transaction number will be assigned automatically when a data base file attached via -transaction to the tcf switch is referenced, unless a non-zero code already has been set explicitly.

reference lists

A temporary reference list is automatically maintained with each tcf switch. This structure, which is implemented as an indexed file without records, contains the necessary information for keeping track of passive references made during the course of each transaction, so that asynchronous changes that might invalidate the transaction can be detected. The reference list also identifies all items modified during each transaction, in order to clean up the data base at checkpoint or rollback time.

Files

data base

Any collection of vfile_ indexed files may be defined as a data base upon which to apply transactions. All that is required is that a common tcf always be used in connection with references to any file in the given data base, and that the individual data base files be attached with the -transaction option specifying a tcf switch attached to the tcf for the data base.

transaction control file

The tcf is a permanent indexed file containing only index entries (i.e. no records). The user is responsible for its creation, but the tcf is implicitly manipulated by vfile_ and the various transact_ routines, so that no explicit user operations on this file are required. If concurrent transactions are performed on a common data base, the -share option must be given in the tcf attchment, as well as in the attachments to the data base files that are shared.

tcf entries

Keys are added to the tcf when a transaction code is assigned for a new transaction. Each key's descriptor is a flag indicating the state of logical completion of a single transaction. Thus the atomicity of a transaction is reduced to changing the flag on its tcf entry.

Usage

opening constraints

In order to use transactions, the user must first attach and open the tcf for the data base. The user is also responsible for attaching and opening all data base files to be referenced before issuing any transactions, and none of these files should be closed within a related transaction.

abnormal termination

When a checkpoint is attempted, or upon referencing a data base item previously read in the same transaction, it is possible that an error resulting from an asynchronous change in another transaction will be detected. This situation makes it impossible to correctly complete the current transaction, and the transaction must be aborted. To determine whether an unexpected error was caused by an asynchronous data base change, one may use the transact_$status entry with the verify option.

References

See the writeup of the vfile_ I/O module in the MPM Subroutines. For a description of the command level interfaces corresponding to the transact_ entries, see the writeup of the transact command.

I/O Module: vfile_

Attach Description

control_args

-transaction , -trans tcf_sw

indicates that all operations on this switch are performed within transactions associated with a control file attached to the I/O switch named tcf_sw. The file must be indexed with stationary type records. Refer to the sections on the transact command and transact_ subroutine described elsewhere in the MPM.

Control Operation

record_status

CHANGE:insert the following after line: 2 block_ptr ptr unal,

```
          2 last_image_modifier fixed ( 35 ) ,
```

2. lock_sw {Input}

CHANGE:replace the last two lines with the following

error_table_$higher_inconsistency The code no_room_for_lock is returned if the allocated record block is too small to contain a lock (see "Record Locks" below) . The code higher_inconsistency is returned if the lock was set by a transaction which cannot be adjusted, either because it is another transaction in the caller's process, or because the lock was set by a dead process and no tcf entry can be found for the record modifier.

If the first modification of a record in a transaction is to lock (and not unlock) via record_status, then vfile_ automatically initializes an after image for the record with a copy of its before image. The record_ptr returned in this case points to the after image, so that based manipulations of the record via its pointer do not affect the before image; this guarantees that modifications made in this manner can be rolled back. After image initialization is suppressed by setting rs_info.unlock_sw.

3. unlock_sw {Input}

CHANGE:add the following paragraph

When the -transaction attach option applies, records can not be unlocked explicitly, as they must be left locked until the transaction completes; then unlocking is done automatically. The only permissible use of setting rs_info.unlock_sw under -trans is in the case where rs_info.lock_sw is also set, in which case the effect is to suppress setting the record's after image and return a pointer to the before image allocation, leaving the record locked. This usage permits explicit synchronization for avoiding interference and deadlocks without incurring the added expense of preparing an after image when one has no immediate intention to rewrite. Based modifications of the record contents should not be made via the record_ptr returned by record_status in this case, but passive based references are allowed. The only valid way to perform based alterations of a record in a transaction is by obtaining a pointer to its after image.

15. modifier {Input/Output}

CHANGE:replace paragraph with the following

if nonzero, this is the identifying number of a transaction on whose behalf the record was locked. When rs_info.lock_sw is set, the user should set this value to 0 before calling record_status.

17. last_image_modifier {Output}

is the transaction number for the most recent modification of this record. If zero, then the most recent modification was not made under the -transaction option.

Logically Absent Records
CHANGE:insert after paragraph beginning 'Garbage collection of keys ...'

If the -transaction attach option was used, garbage collection of the last key and record's stationary header is suppressed. This is done to insure that any passive reference to the record prior to its deletion can find the record header afterwards to detect the asynchronous change. Thus, to completely recover the storage occupied by records deleted in transactions, one must periodically collect garbage by opening the file without the -transaction attachment. Only those items which can't have been referenced by any transactions currently in progress may be collected.

Record Locks
CHANGE:insert after paragraph beginning 'Attempting a rewrite_record ...'

If a record has been locked by a transaction, the above error codes are suppressed, except for the case of record_busy on an attempt to alter a record locked by a live process. If the record's modifier can not be found in the transaction control file, or if the caller has not used the -transaction attach option, then the code error_table_$higher_inconsistency is returned.

Multiple Openings

4. Openings with the -share control argument.
CHANGE:add at end of this section

The code error_table_$asynch_change is returned on a subsequent reference to an item previously referenced in the same transaction, if an asynchronous change is detected; when this is the case, it is impossible to successfully complete the transaction by checkpoint, and the current transaction must be aborted.