Palo Alto Research Center

Implementing Long Lived Transactions Using Log Record Forwarding

Robert B. Hagmann and Hector Garcia-Molina

XEROX

Implementing Long Lived Transactions Using Log Record Forwarding

Robert B. Hagmann
Xerox Palo Alto Research Center

Hector Garcia – Molina
Princeton University

CSL – 91 – 2 February 1991 [P91 – 00032]

Abstract: Many database systems use a disk log for fast crash recovery. Over time, the log fills up. Old long lived uncommitted transactions must be aborted since some of their records in the log are about to be overwritten. This paper proposes a way to extend the lifetime of transactions by forwarding a copy of the to – be – overwritten records to the end of the log. Crash recovery and transaction abort processing is adapted to process log records different than the order in which they were created. The normal operation and crash recovery performance for this system are also described. Overhead for forwarding during normal operation is shown to be negligible, except for degenerate cases. Crash recovery is shown to run faster than having a very long log.

CR Categories and Subject Descriptors: H.2.2 [Database Management]: Physical Design — Recovery and restart; H.2.7 [Database Management]: Database Administration — Logging and recovery;

Additional Keywords and Phrases: Long Lived Transactions

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

1. Introduction

Most systems that provide transaction support, whether they be database, file or operating systems, typically use a disk based log for recording the actions that are performed by transactions. One common way to organize the disk log is as a circular file. A compromise is reached, implicitly, in the disk log size. Where the log is short, recovery is faster but moderately long lived transactions will have to be aborted (see [Lind79]). Only part of the file has valid data. The START position refers to start of valid data, and the TAIL position refers to end of valid data. As new log data is written, the TAIL position is advanced. If the end of the file is reached, the pointer wraps around to the beginning of the file (i.e., modular arithmetic is used for the pointers).

As the log fills up, the TAIL eventually catches up to the START. At this point, START must be advanced, reclaiming old log records that are no longer needed. In most cases, the area to be reclaimed will contain no records of active (uncommitted) transactions, so START can be safely advanced. However, in some cases there may still be active records (of long running transactions) in the area. Something must be done in this case, as the system cannot run without available log space. There are two possible solutions. The first is to abort the transactions that hold the records that are in the way, thus clearing the space.

A second solution, proposed in this paper, is to use record forwarding. The basic idea is quite simple: a copy of active records in the log area to be reclaimed ahead of the START position, are appended (forwarded) to the end of the log. The challenge here is to perform adequately fast crash recovery and not degrade normal operations.

In this paper we will argue that the overhead of forwarding (if implemented correctly) is acceptable and always less than the overhead of aborting transactions. Furthermore, in a distributed system there are cases where abortion may be unacceptable. For instance, say a transaction T runs at nodes A and B. During the commit protocol, nodes A and B become disconnected, and T is blocked at A [Skee82]. This means that A does not know the fate of T and cannot commit or abort it. If the failure lasts long enough, the log TAIL will eventually wrap around to where the T records are found. Since T cannot be aborted, the only alternative in this case is to use forwarding.

It could be argued that both abortion and forwarding are unnecessary if ``enough'' space is allocated to the log. For example, if we expect all transactions to commit in at most 3600 seconds (one hour), and we expect the system to generate 10,000 bytes of log data per second, then we can use a log file of more than 36

megabytes. With such a file, the situation where START cannot advance will ``never'' occur. We offer two counter arguments to this idea. The first is that a well designed system should avoid making assumptions about how long transactions will run and how much they will log. (We could mention a number of systems designed under the assumption ``no one will ever need more than X of this'' that eventually and painfully broke down!) The load of the database, restructuring major relations, massive updates, an unusually long transaction, or a failure like the one illustrated above can easily violate our assumptions.

Our second counter argument is that in some applications ``large enough'' would lead to a very high storage cost. For example, some high performance systems write their logs to so called RAM disks (devices with disk interfaces that store data in non – volatile electronic memory). Allocating tens or hundreds of megabytes to the log may be unacceptable, especially when there is an inexpensive solution like log forwarding that can dramatically reduce the log size.

A second example is a file system or an objected oriented database system where transactions may be very long lived and may generate large volumes of log data. A transaction to reorganize a large file or to process a collection of images from a satellite may take days and may generate megabytes of log. The log would have to hold not only the megabytes generated by that transaction, but also the logs of all other transactions that run concurrently. Again, we believe that forwarding is a very attractive alternative.

Although the basic idea of log record forwarding is relatively simple, there are at least two critical challenges that exist. One is to design an efficient strategy. Forwarding causes non – sequential log activity: the disk head must be moved from the log tail to where the records to forward are. This can affect logging performance during normal operation. Thus, the forwarding overhead must be kept at a minimum. Also, forwarding causes records to appear in a different order in the log. A recovery procedure that sorts records into their original order could be very expensive and should be avoided.

A second challenge is to design a correct strategy. The logging mechanism must concurrently log new records as old ones are being moved. At recovery time, it must process log records out of order. If the mechanism is properly designed, the rules for this processing can be quite simple. However, showing that the rules are correct is non – trivial.

This paper will outline how recovery can be performed for a particular

implementation of log record forwarding. Our description must be detailed enough so that some of the subtle issues related to forwarding can be explained, yet it should contain a minimal number of irrelevant details. To achieve this balance, we have selected a relatively simple yet efficient form of logging based on physical logging with complete before and after images. We also assume pessimistic locking.

There is a large variety of logging mechanisms (some even proprietary), and this paper cannot show how to use the log record forwarding technique for all of these systems. However, we believe that record forwarding will help nearly all disk log recovery systems, after it has been adapted to their environment. For example, instead of a circular log, one can use a file with multiple extents, where old extents are archived to tape. Log forwarding makes it possible to archive extents even before all their records are inactive. As another example, forwarding can be used in conjunction with log compaction [Hagm86, Kaun84]. In Section 4 we discuss some of the other major types of logging and the impact of record forwarding.

2. Design Overview

In this section we describe a particular implementation of log record forwarding. Our description of record forwarding is accompanied by an Appendix that gives pseudo – code for the principal logging and recovery functions.

2.1 Normal system operation

We start by describing how logging would work without forwarding. As described in the introduction, the log is implemented as a circular file. Each log record must have some marking to make it possible to identify the TAIL after a crash. For example, each record could be written with an increasing sequence number. The TAIL would then be the record with the largest sequence number. (There many other ways to mark the records, though.)

As updates are made, the logger is told the before and after images for the updated object. The first time an object is updated by a transaction, both the before and after images are logged. On subsequent updates to the same object by the same transaction, only the after image is logged. Database objects can either be pages, records, or byte ranges (see Section 4). We assume that pessimistic object level locking is used. This implies that a before image in the log is the latest committed version of the object.

To make recovery faster, systems periodically force some buffer pages and key

state information to disk. This operation is called a checkpoint. First, a log entry is made recording all the active transactions together with a (back) pointer to their latest log record. Committed transactions need not be recorded; any of their writes to disk that have not been performed will be done by the checkpoint and hence these transactions will be fully completed. Next, all dirty database pages in the buffer pool are flushed. Finally, a special block CHKPBLOCK is written at a fixed known position. This block contains a pointer to the latest checkpoint record, as well as the current START record on disk. The checkpoint completes when CHKPBLOCK is written safely. Before the CHKPBLOCK is written, crash recovery should start with the previous checkpoint. Notice that the state of the database is not consistent at the time of a checkpoint.

After a crash, CHKPBLOCK is read and the latest checkpoint record (C) is found. The set of active transaction is read from C. Next, the log TAIL is found, for example, by performing a binary search for the record with the largest sequence number. From TAIL, the records are read in reverse order until C. If a record is found for a transaction not in active list, then that transaction is added to the list. (Such a transaction started after the checkpoint.) During the scan, we keep track of the state of each transaction. Initially it is assumed that all transactions are to be aborted. When a commit record is found, the state of that transaction is changed to committed. (Note that in the backward scan, the commit record for a transaction will be found before any action records for that transaction.) For each log record found that records an action, the action is undone or redone, according to the Application Rule given below.

Once record C is reached, all committed transactions have been redone (any writes made before the checkpoint were propagated to the database by the checkpoint at C). However, it is still necessary to undo the actions performed by aborted transactions before the checkpoint. Thus, for each aborted transaction, the system follows its chain of log records (each record contains a pointer to the previous one), using the Application Rule, until the corresponding begin – transaction record is found. All the chains are followed together, thereby processing the log in reverse order.

Application Rule. For each database object o we keep a flag done(o). When recovery starts, done(o) is set to false for all objects. When the first undo or redo is applied to o, done(o) is set to true. (Note that done can be implemented as a hash table. If object o is not in the table, then done(o) is false. Initialization is trivial and lookup is fast.) The following rule is applied to each action record for transaction T_i and object o seen in the backward pass:

if T_i is committed and done(o) is false and there is a redo image in the record
then

begin apply the redo on o; set done(o) to true end

else if T_i is aborted and done(o) is false and there is an undo image in the record
then

begin apply the undo on o; set done(o) to true end

else skip this action

Before discussing record forwarding, it is important to understand the Application Rule. Consider for instance a particular database object o that has been modified by some transactions just before a crash. At recovery time the log might look as shown in the following figure.

Figure 1

We only show the log records pertaining to object o, plus the latest checkpoint C. Say we begin the backward scan and find the first record R_1. Say the "?" record was a commit for T_1, the transaction that wrote R_1. Since T_1 will be in committed state at R_1, the last image produced by T_1 will be placed in the database. After this step, no further actions on o need be performed. Now suppose that the "?" is not a commit for T_1. Then we must place in the database the image of o that existed when T_1 started. This will be found in one of the T_1 records, but not necessarily in R_1 (it may only have an after image). So we scan further back until we do find the before image in one of the records, say R_3. (Note that none of the records between R_1 and R_3, referring to o, can be for a transaction other that T_1 because T_1 had locked o exclusively.) This R_3 image is installed. It represents the latest committed value for o, so again, no further actions on o are necessary.

2.2 Log record forwarding

The basic idea of record forwarding is to copy records from active transactions near the START position to the log tail. Once all useful information has been cleared, the START position can be advanced. The START position changes by some DELTA number of records, thus creating some free space for the log.

When the START position is advanced, the vast bulk of records skipped are usually for short committed or aborted transactions. These transactions have already had their effects done or undone to the disk resident part of the database. They may have committed or aborted, or have had their effects superseded by other transactions. Dropping these records has no adverse side effects. Typically, only a small portion of the records at the end of the log belong to long lived transactions.

Although DELTA can be any number (that does not get us too close to the tail), we choose to make it such that START + DELTA is the next valid checkpoint in the log. This makes it much more efficient to discover active records to forward. Let us illustrate this through the following example (details are given in the Appendix.)

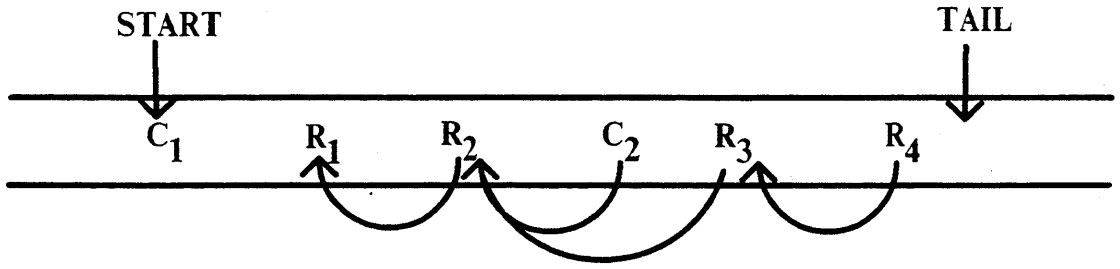

Figure 2

In Figure 2, C_2 is the oldest checkpoint record. (C_1 is the next oldest, but now is an invalid checkpoint.) Records R_1, R_2, R_3, and R_4 were made by a long lived transaction T_i that is still active. The T_i records have backward pointers. In memory, a pointer is kept to the most recent log record of T_i, in this case R_4.

Say a new checkpoint C_7 is to be made. After the checkpoint record is written and the buffer pool is flushed, it is noticed that more free log space is desirable. As part of the checkpoint, the oldest checkpoint C_2 is read (its position was stored in memory). Any transaction like T_i with active records in the area to be reclaimed was recorded in C_2. The log record chain for T_i is read, starting by the record identified in C_2, i.e., R_2. (As described above, every checkpoint contains the list of active transactions at the time, plus a pointer to the most recent log record of each.) All the chains are followed together, thereby processing the log in reverse order. This way records R_2 and R_1 are found and copied to free positions at the tail. The copied records are treated as standard records, being linked in the normal fashion. They are added to the log without force. When forwarding is done, START is advanced to C_2, the log is forced, and the position of the latest checkpoint C_7 recorded by writing a new CHKPBLOCK

block. The situation at this point is illustrated by the next figure.

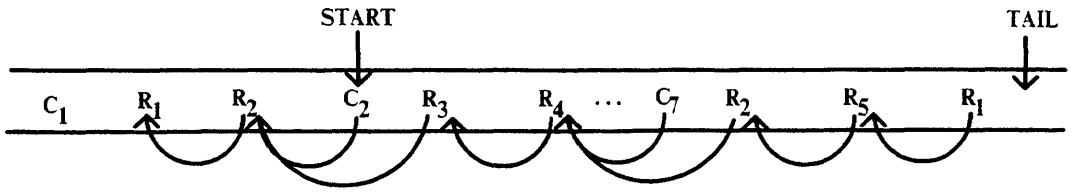

Figure 3

There are several important things to notice. One is that pointers like the one from R_3 to the old copy of R_2 are now invalid. Thus, a chain of records actually ends with a nil or an invalid pointer. Invalid pointers can be detected because they cross the START position. In particular, suppose that the log is implemented by a contiguous vector of N disk pages, and that R_3 and R_2 are in pages p_3 and p_2 respectively. Then the pointer from R_3 to R_2 is invalid if $(p_3 - p_2) \bmod N > (p_3 - \text{START}) \bmod N$. (This assumes checkpoints and thus START records are always at the beginning of a page.)

A second thing to observe is that transaction processing is not suspended during forwarding. In the example, T_i produced a new log record, R_5, after R_2 was copied but before R_1 was copied. A third observation is that the log records for a transaction can be out of order due to forwarding. For example, record R_1 will now be the first record found in a backward scan, instead of being the last one. In the next section we show that crash recovery is still efficient, in spite of this lack of order.

A fourth observation is that not all active records need to be forwarded. In particular, there are three situations where records are not forwarded:

(1) Begin – transaction records can be ignored. The end of the record chain for a transaction is indicated by an invalid pointer, so there is no need to end the chain explicitly.

(2) Redo portions of records can always be dropped: the update in the record must have been propagated to disk by the current checkpoint when it flushed the buffer pool.

(3) Records of transactions that commit during forwarding do not have to be forwarded at all. In our example, suppose that T_i was uncommitted when the checkpoint starts (and record C_7 is made). However, it commits soon after, and say record R_5 is actually its commit record. Record R_1 can then be ignored. The reason is as follows. Since T_i has committed, the undo information in R_1 is useless. As stated above in (2), redo portions of records can always be dropped. Thus, the entire record is useless.

It is interesting to notice that optimization (3) not only reduces the amount of data to be copied, but also makes recovery very simple. To illustrate, suppose that R_5 is a commit record but that R_1 was forwarded anyway, resulting in the state shown in Figure 3. At recovery time, we would then get to R_1 first in the backward scan. At this point, we would not know whether T_i was to be undone or redone, so the recovery action could not be applied immediately. On the other hand, if records like R_1 are not forwarded, then for committed transactions it is certain that the commit record will be the first one found in the backward scan. If an action record is seen without first having seen a commit record, then that transaction can be aborted. If one did not take advantage of optimization (2), recovery would also be more complex.

A final observation is that only a few disk reads may be needed to acquire the forwarded records. For long lived transactions, the system can group together their log records and write them to the log in bursts. When these bursts are forwarded they will also stay together. Thus, we can expect the forwarded records to be found in a "few" cylinders. We will return to this issue in Section 3.

2.3 Crash recovery and abort processing

As we have seen, forwarding may cause log records to be out of order. Surprisingly, however, crash recovery proceeds in exactly the same way as without forwarding. That is, a single backwards scan is done, from TAIL to the latest checkpoint, using the Application Rule on each undo or redo record found. Then the remaining aborted transactions are aborted, again using the Application Rule.

To see why this restores the database to a correct state, let us consider once again the fate of a particular object o. Figure 1 showed the actions in the log that involved o, when the log was ordered. Now that we have forwarding, some action, call it R_f, may appear ahead of its original position.

When the log was ordered (no forwarding), there were basically three outcomes of the crash recovery phase: an after image was stored into o, a before image was stored, or no action on o was performed. Now we will show that forwarding does not affect the three outcomes.

(I) Suppose that if the log were ordered, an after image would be stored into o at recovery. This occurs when a commit record is found for some transaction T_k and then a redo record is found before the latest checkpoint. In Figure 1, say R_1 is the undo record (and "?" the commit record). There are now two subcases to consider: a forwarded record R_f appears ahead of, or behind R_1.

(i) Any record R_f that was forwarded after R_1 was written must also belong to T_k. (Transaction T_k holds an exclusive lock on o at least between R_1 and the commit record. Since only active transactions forward, the transaction that owns R_f must hold the same lock from the latest checkpoint through the point where the record is forwarded.) Clearly, R_f cannot appear after the commit record for T_k, so in this subcase, it must be that R_f is between R_1 and the commit record. In this region of the backward scan, the state of T_k will be committed. Since R_f only has undo information (see the fourth observation, part 2 in Section 2.2), it will be ignored (see Application Rule).

(ii) The forwarded record R_f is behind R_1. When the redo image R_1 is installed, done(o) is set to true. Thus, all records behind R_1 will be ignored.

So, for case (I) we see that forwarded records have no effect. The state of o is restored to its last committed value, just as if the log has been in order.

(II) Suppose that with the ordered log, an undo image was stored into o at recovery. Say that R_k was the record that contained the before image applied to o and that T_k was the transaction that wrote it. Note that T_k must have been the last transaction that modified o before the crash occurred. Now let us look at the unordered log. Say b is the position in the unordered log where R_k appears (it may or may not be a forwarded record). At time b (i.e., when TAIL was at b), object o must have been locked by T_k.

We now proceed by contradiction. Say that when the unordered log is used for recovery a different undo record for o, R_x, is applied. Record R_x must have been made by a different transaction T_x because T_k only wrote a single undo record referring to o. Record R_x must appear at a position c ahead of b (closer to TAIL), else R_k would still be applied. Record R_x must be a forwarded record, else T_k would not have been the last transaction to update o. For a similar reason, the original place where R_x was logged, call it a, must be behind b. (Going from START to TAIL, a occurs first, then b, then c.) At time a, T_x held a lock on o. The lock must have been released by time b when T_k holds it. Therefore, T_x is no longer active at b. But this means that R_x could not have been forwarded at time c. This is a contradiction, so R_k must be the undo record that is applied when recovery is done from the unordered log.

(III) The last case is when recovery with the ordered log yields no actions on object o. This occurs when none of the active transactions at checkpoint time access o, and when there are no o records ahead of the checkpoint record. Forwarding cannot introduce o records ahead of the checkpoint record, because this would imply that

some transaction that modified o was active at the checkpoint. Thus, with forwarding the outcome is the same: no recovery actions are performed on o.

2.4 Safety and fairness

Fairness and safety provisions must also be incorporated in the log record forwarding. The log can fill up with the records of very long lived transactions. A single transaction can use a large fraction of the log. As the start pointer is advanced, an equal amount of data is logged (the forwarded log records.) The system is just wildly copying the log. This should be rare, but the system should protect itself against this situation.

A final observation regards a full log. The system should attempt to ensure that a full log never happens by keeping far enough ahead. During normal logging, if a transaction finds a full log, then it can wait in hopes that a checkpoint is in progress and about to reclaim space. However, if the checkpointer runs into a full log when forwarding records, then it cannot wait. The checkpointer must abort any active transactions that hold active records in the area to be reclaimed. Only when the area is free, can the checkpoint complete and transaction logging resume.

3. Performance considerations

There is clearly a performance cost associated with record forwarding. To forward a record, it must be read and copied onto the tail of the log. In addition to the cost in terms of CPU time and disk bandwidth, there is possibly a penalty for causing non-sequential activity on the log disk. That is, to read a record that will be forwarded, the disk head must move away from the log tail.

Many systems can piggyback reading and writing of the log on existing operations. The forwarding occurs without force, so that it (usually) does not incur an IO. Recall that the log manager was careful to write the records for long lived transactions in clusters. If a tape copy of the log is written, then forwarding can piggyback the reading of the log at this time (see Section 4.1).

3.1 Comparison with abort

To be fair, a performance comparison of record forwarding must be made against some alternative. In this case, one alternative (in addition to just not running long lived transactions) is to abort the long transactions when log space runs out. It is not hard to see that record forwarding is a clearly superior alternative. Suppose that j

records belonging to a LLT lie in the log area that is about to be reclaimed. At that time, the same LLT has an additional k records in the rest of the log. If forwarding takes place, j records will be read and a subset of them copied. If the LLT is aborted, all j + k records have to be read to obtain the undo information. After the abort is complete, the LLT must be re – run, requiring j + k new log writes just to get us back to the same position we would be in with forwarding.

3.2 IO cost model

We have argued that the proposed log record forwarding technique is superior to aborting long lived transactions. So, given that log forwarding is to be implemented, we now try to evaluate its cost. The major cost is the extra IO's that must be performed to forward records.

In order to study this cost, we present a simple model. It is a "strawman" model whose goal is to illuminate the major issues, not to predict the performance of an actual system. The model parameters are as follows:

d: the duration (seconds) of a long lived transaction (LLT).

t_u: the rate (records/second) at which a LLT writes undo records into the log

t_r: the rate (records/second) at which a LLT writes redo records into the log

l: the number of concurrent long lived transactions

n: the number of records in the circular log. Assume that all records are of equal size.

m: the number of checkpoint records on disk

r: the total rate (records/second) at which records (of any type) are written on the log.

i: the maximum number of contiguous log records that we want to read in a single IO

Note that the LLT log writes and any forwarding is included in the total rate r. This rate is assumed to be constant. During normal processing, the log tail moves along the log. The time for each rotation (time until reuse of a log page) is n/r seconds. The time between checkpoints is n/(rm).

Suppose that a checkpoint is to be taken and that a LLT is active at the time. The LLT records in the log area to be reclaimed must be read. The number of redo records that will be read (but not forwarded) is $(t_r n)/(rm)$. If there are l concurrent LLT's, then the number of total redo records will be roughly that amount times l. Since undo records are forwarded, in computing the number of undo records we must take into

account all the undo records that were written by the LLT during its lifetime. Assuming that on the average each LLT is half done, we would expect to see $(dt_u)/2$ of its undo records at a given time. Since there are l concurrent LLT, the expected number of undo records is $(ldt_u)/2$. In the area to be reclaimed we would expect to see 1/m of these records. Adding the undo and the redo records, we expect to find a total of

(Eq. 1) $T = (ldt_u)/(2m) + (lt_r n)/(rm)$

records in the reclaimed area. Assuming that we can pack all these LLT records into a continuous run (of size i pages), then the number of IO's per checkpoint due to reading records off the reclaimed area is T/i. (In reality the number might be slightly larger if there is not one but two runs.)

We can now argue that under "normal" loads, this quantity will be relatively small, and hence the IO impact of forwarding is negligible. For example, say we expect each LLT to write a total of 10,000 undo records (dt_u), and that there are 100 checkpoints (m) and 2 concurrent LLT's. Then we expect to find 100 undo records in the area to reclaim. Similarly, suppose that each LLT writes 10 redo records between a pair of checkpoints (this is $(t_r n)/(rm)$). Then two LLT's will write 20 redo records. If we are willing to use runs of 100 records, the we would have 2 IO's to perform per checkpoint. It is not hard to see from Equation 1 that most realistic sets of parameters yield relatively small numbers of IO's. And if for some reason the IO load does become larger than what can be tolerated, then the number of concurrent LLT's, l, can be reduced. Or as an alternative, the log can be made larger, increasing the number of checkpoints m. Yet another alternative is to only have undo records in the linked list kept for each transaction (redo records could be kept on a separate linked list, or not chained at all). This makes it unnecessary to read redo records, eliminating the second term from Equation 1.

3.3 Estimation of the number of forwarded records

In our IO cost model we did not consider the cost of processing the forwarded records in memory and then writing them at the log tail. Since these costs involve no disk arm movement (forwarded records are not forced), they will usually be less significant than the IO cost. Yet, it is still important to understand the CPU and disk bandwidth costs and to make sure they do indeed stay at reasonable levels. Both the CPU overhead and the disk bandwidth consumed will be proportional to the number of records that are forwarded, F. We now estimate F.

As mentioned earlier, the time for one rotation is n/r seconds. During each

rotation, the LLT will produce E new undo records, where

(Eq. 2) $E = (n/r)\, t_u$.

During the lifetime of the LLT, the tail will make

(Eq. 3) $R = d / (n/r)$

rotations. For simplicity, let us assume that R is an integer. (If it is not, our following equations will be approximately correct but not exact.) During the first rotation of the LLT, E new undo records are made but none are forwarded. During the second rotation, the first E records have to be forwarded, and E new ones are made. During the third rotation, the 2E records of the first two rotations must be forwarded; in the fourth rotation 3E must be forwarded, and so on. The total number of forwarded records is then

(Eq. 4) $$F = \sum_{i=0}^{R} E(i-1) = \tfrac{1}{2} \sum R(R-1)$$

This equation can be rewritten as

(Eq. 5) $F = (dt_u / 2)\,[\, dr/n - 1\,]$

There are two important observations that can be made from this equation. The first one is that the forwarding overhead is proportional to the square of the LLT duration. An LLT that takes twice as long as another one will have to forward four times as many log records. (Of course, if the shorter LLT makes only one revolution, then the ratio is E to zero forwarded records.) Clearly, if d becomes too large we can have a serious problem.

The key to avoiding this problem is to choose a sufficiently large log that reduces the number of revolutions of an LLT. A "rule of thumb" that may be useful for this can be derived from Equation 5 (this constitutes our second observation).

During the lifetime of a LLT, a total of dr log records are written by all transactions combined. Suppose that we choose the log size so that it can hold 1/k of these records (i.e., the log will undergo k rotations during the lifetime of the LLT). Thus, n can be expressed as

(Eq. 6) $n = d\,r / k$

where k is some small integer we choose. Substituting into Equation 5 we see that the number of forwarded records is (k − 1)/2 times the number of original undo records made by the LLT (i.e., $d\,t_u$). For example, if the log can hold a third of the total records, then on the average each undo record written by the LLT will be forwarded one time. If the log holds a fifth, then we expect each record will be forwarded twice. One can

also use the formula in reverse fashion. Say that for each record logged by the LLT, we are only willing to forward it two times. Then the log must at least hold a fifth of the log records that the system will produce during the lifetime of the LLT.

Another option for coping with high bandwidth requirements for the log to to use disk striping, i.e., writing in parallel to multiple disk drives. This technique is discussed in [Kim86, Sale86].

3.4 Long transaction abort

An LLT can be aborted during normal operation voluntarily or because of deadlocks. Such an abort should take about the same time as an abort of a same sized transaction in a system without forwarding. In either case the same number of records must be randomly fetched from the log. Random I/O is the dominating cost in this process because back chaining is used to find the next record and buffering is not of much help. Since the same number of records must be fetched, roughly the same number of I/O's will occur.

However, since transactions now can live longer, they can also get bigger. Hence, an individual transaction abort can take longer using log record forwarding. However, this is simply because the transaction is larger and is doing more work.

3.5 Performance during recovery

This section discusses the impact of record forwarding on recovery time. To illustrate, the system is assumed to have the following characteristics:

Five percent of the log is used for long lived transactions.

The system can do disk scheduling. The disks have 19 tracks, seek to adjacent cylinders in 5 milliseconds, and can read a track in 16 milliseconds.

The buffer pool is cleaned via a checkpoint whenever the log enters a new cylinder.

During recovery, the only difference from normal recovery is the forwarded records. For the records after the last checkpoint, there is negligible difference in how recovery works. Before the last checkpoint, on the average there is one cluster per cylinder of records, either forwarded or not, that must be read for LLT's. The records are clustered together and are about a track (about 5% of the cylinder). Since the records are clustered, they can usually be read using a single read. This takes an adjacent cylinder seek, a latency, and a transfer (5 + 8 + 14 = 27 milliseconds). Normal log reading for a cylinder takes a adjacent cylinder seek and 19 revolutions (5 + 19 * 16

= 309 milliseconds). Hence the forwarded log can be read about eleven times faster than a scan of a normal log.

4. Other Logging Strategies

To describe record forwarding we have assumed that full before and after images are logged, and that pessimistic locking is used. In this section we discuss some other types of logging and the impact on forwarding.

4.1 Tape copy of the log

If a tape copy of the log is kept, then the copying of the log to tape is often delayed to allow for compaction [Kaun84]. The forwarded records can be extracted during the copy operation at little cost and no additional head motion. Thus, reading records for forwarding is basically free.

4.2 Different flush policies

Our simple checkpoint algorithm flushes all dirty pages at checkpoint. Hence, we only have to scan backwards during recovery to the first checkpoint. If a different flush policy is used, then more backwards scan for redo will be needed. All that we can say here is that the backwards scan for redo must be far enough to see all the redos needed. Exactly how far is dependent on the checkpoint buffer pool invariant. Forwarding is not significantly affected.

4.3 Redo only logging

Redo only logging for LLT's writes into the log the new value of pages modified by a transaction. The buffer pool of modified pages is not written to the database until commit. Modified pages that are not in the buffer pool are read from the log.

Forwarding now forwards redo object images. It does not forward redo images that have been remodified. There may be many redo images for an object in the log for a transaction, but the last one is guaranteed to be the committed value for a transaction, if it commits.

4.4 Record or Page Logging

In our algorithm we have not specified what an "object" actually is. The solution works regardless of whether an object is a page, a record, or a byte range within a page. Of course, if an object is smaller than a page, an undo or redo cannot be applied without first reading into memory the image of the entire page. However, as long as full before and after images are logged, forwarding proceeds as we have described.

4.5 When Record Order is Important

Our mechanism is not affected by the order of forwarded records in the log.

However, under some circumstances order is important. This includes logical operation logging and locking with non-pessimistic locking. Because of space limitations we cannot explain how recovery and forwarding operate in these cases. All we can do is sketch out a common solution that can be used.

The basic idea is to include a log sequence number (LSN) in each log record. Each time an action is logged, a counter in memory is incremented and its value stored as the LSN of the log record. When a record is forwarded, its LSN is preserved unchanged.

At recovery time, the necessary log records can be read into memory and the correct ordering can be reconstructed from the LSN's. In practice, it is not necessary to read in the entire log, sort it, and then apply it. Various optimizations can be used, depending on the particulars of the logging used, to start applying recovery actions early.

The main observation to make is that even with more complex logging strategies, the overhead of forwarding during normal operation is still minimal. Recovery may be more complex, but the overhead still very low.

5. Conclusions

This paper proposes a method to do log compression in the normal database log. The proposal is a way to extend the lifetime of transactions by forwarding a copy of the to-be-overwritten records to the end of the log. The compressed log is interspersed with current log data.

Crash recovery and transaction abort processing has been adapted to process log records different than the order in which they were created. Overhead for normal operation was shown to be negligible, except for degenerate cases. Crash recovery performance for this system was shown to be superior to recovery with a very long log.

References

[Gray79] J. Gray, ``Notes on Data Base Operating Systems,'' in Operating Systems, An Advanced Course. Edited by R. Bayer, R. M Graham and G. Seegmuller, Springer – Verlag 1979.

[Hagm86] R. Hagmann. ``A Crash Recovery Scheme for a Memory Resident Database System,'' IEEE Transactions on Computer Systems, Vol. 11, No. 1, pp. 839 – 843, March, 1986.

[Kaun84] J. Kaunitz and L. Van Ekert. ``Audit Trail Compaction for Database Recovery,'' Communications of the ACM, Vol. 27, No. 7, pp. 678 – 683, July, 1984.

[Kim86] M. Kim. ``Synchronized Disk Interleaving,'' IEEE Transactions on Computers, Vol. 35, No. 11, pp. 978 – 988, Nov., 1986.

[Lind79] B. G. Lindsay, et al. Notes on Distributed Databases, IBM Research Report RJ2571, 1979.

[Sale86] K. Salem and H. Garcia – Molina, Disk Striping. Princeton Department of EE & CS Technical Report 332, Dec. 1984.

[Skee82] D. Skeen, Crash Recovery in a Distributed Database Management System. University of California at Berkeley Technical Report UCB/ERL M82/45, 1982.

Appendix: Pseudo Code for Logging with Record Forwarding

In this Appendix we describe in more detail the proposed log forwarding mechanism. We use informal pseudo code to describe the various steps, relying on English to describe some of the minor operations. Our goal is to, in a very limited space, give enough details so that an experienced programmer could convert this description to actual code.

For simplicity we assume that all records are written on a separate disk page. This avoids having to describe here how records are packaged into pages. Records are identified by an integer between 0 and N – 1, where N is the number of the page that holds it. Extending the algorithm to multiple records within a page and to variable size records is straight forward.

Data Structures in Memory

N: the number of records in the circular log on disk.

AST: The set of active transactions. For each T in AST we have

T.tid = the transaction id

T.lle = the position in the log of the most recent log record for this transaction.

T.state = committed, aborted (only used at recovery time)

CHKPLIST: a list of checkpoint records. CHKPLIST.old points to the oldest valid checkpoint log record. CHKPLIST.new points to the newest element of the list. Each record R in the list contains:

R.position: the position of the corresponding checkpoint record in the log.

R.next = a pointer to the record in memory that represents the next checkpoint that occurred.

START: identifies start of valid log. START + 1 is first valid record.

TAIL: the latest valid record in the log. Note that START = TAIL when the system is started.

The system uses two semaphores:

logsem = ensures that only one process at a time logs

forsem = ensures that at most one process is checkpointing

Format of Log Records

Every log record L has the following two fields:

L.mark = a mark used to identify the TAIL quickly. We do not discuss this further here.

L.type = the type of record; can be checkpoint, tbegin, taction, or tcommit.

A log record of type checkpoint in addition contains a set L.ast of active transaction at the time of the checkpoint. Note that all of these transactions are uncommitted (at checkpoint time all buffers are flushed, so any committed transaction is all done and can be forgotten). For each T in L.ast we have the following fields:

T.tid = the transaction id

T.lle = the position in the log of the most recent log record for this transaction.

A log record L of type tbegin, taction, or tcommit in addition contains the following fields:

L.tid = the transaction id

L.lle = the location in the log of the previous log record for this transaction. If this is a tbegin record, then L.lle is null (e.g., – 1).

A log record of type taction in addition contains the undo/redo information. We assume that full before/after images are used for each object. The three additional fields are:

L.object = the id of the object

L.undo = the before image

L.redo = the after image

Format of Checkpoint Block

The checkpoint block CHKPBLOCK is a disk block stored at a known fixed position. It is used at recovery time to discover the log position of the most recent log checkpoint record. The block has the following fields:

CHKPBLOCK.start = the start record of the log

CHKPBLOCK.tail = the position of the most recent checkpoint record.

Logging Actions

The following describes what a transaction does when it logs an action. Let T be the element in AST for the transaction performing this action.

```
P(logsem)
wait until { (TAIL + 1) mod N not= START} (wait is outside logsem critical section)
TAIL __ (TAIL + 1) mod N
Allocate a new buffer for constructing log record L
L.type __ taction; L.tid __ T.tid; L.lle __ T.lle; T.lle __TAIL;
```

```
L.redo __ after image; L.object __ object id
if this is first update to object by T then L.undo __ before image
write buffer to position TAIL on disk (not forced)
V(logsem)
```

Logging a transaction begin or commit is similar and is not described here. Note that when a transaction aborts, no "abort" record is written. After a transaction aborts or commits, it is removed from AST.

Checkpointing

The checkpointer is called every $DELTA_T$ seconds, or whenever (START − TAIL − 1) mod N < $DELTA_S$. $DELTA_T$ and $DELTA_S$ are administrator defined constants.

```
P(forsem)
P(logsem)
if (TAIL + 1) mod N = START then abort – some – transaction (log is full)
TAIL __ (TAIL + 1) mod N
Allocate a new buffer for constructing log record L
L.type __ checkpoint
For each T in AST do begin
    if T.state = aborted or committed then delete T from AST else
     add T (with corresponding T.tid and T.lle) to L.ast
write buffer to position TAIL on disk (not forced)
Save __ TAIL;
V(logsem)
Allocate new memory cell for checkpoint cell. Call it R.
 R.position __ Save;  R.next __ null
if CHKPLIST.new = null then CHKPLIST.new __ CHKPLIST.old __ R
else begin CHKPLIST.new.next __ R;  CHKPLIST.new __ R end
Flush all data buffer pages to disk
If (START − TAIL − 1) mod N < DELTAS then begin {start forwarding}
    Let C be the checkpoint record at CHKPLIST.old.position
    CHKPLIST.old __ CHKPLIST.old.next
    for each T in C.ast, if T.lle is invalid then T.lle __ null  (T.lle is invalid if (C − T.lle)
              mod N > (C − START) )
    While {there is a T in C.ast with T.lle not null} do begin
```

```
        select T from C.ast such that T.lle is not null and (C - T.lle) mod N is the
                smallest
        Let E __ T.lle, the record to be forwarded
        if E.type = taction and E.undo exists then begin {not necessary to forward
                tbegin or redo entries}
          P(logsem)
          if { T is in AST with state not = committed } then begin
            if (TAIL + 1) mod N = START, then abort - some - transaction.
            TAIL __ (TAIL + 1) mod N
            Allocate a new buffer for constructing log record L
            L.type __ taction; L.tid __ E.tid; L.lle __ AST.T.lle; AST.T.lle __TAIL;
            L.redo __ null; L.object __ E.object
            L.undo __ E.undo;
            if E.lle is invalid, set T.lle (in C.ast) __ null, else set T.lle __ E.lle (E.lle is
                    invalid if (E - E.lle) mod N > (E - START) mod N)
            write buffer to position TAIL on disk (not forced)
            end
          V(logsem)
          end {not necessary to forward}
        end {While T}
      START < - C
      end {end of forwarding}
    Flush log, ensuring pages through TAIL + 1 are actually written
    Update checkpointing block on disk (atomically) so that new checkpoint is in effect:
      CHKPBLOCK.start __ START
      CHKPBLOCK.tail __ Save
    V(forsem)
```

Crash Recovery

During crash recovery, we keep a hash table called done(o) that indicates whether a recovery action has been performed on object o.

This is the procedure to be executed after a crash:

```
START __ CHKPBLOCK.start; TAIL __ CHKPBLOCK.tail
Let C be the log record identified by CHKPBLOCK.tail
```

```
Copy C.ast into AST
For each T in AST set T.state __ abort
Let TAIL be the last valid log record (found using L.mark in log records)
Scan the log backwards from TAIL to C, in order. For each record Y do begin
   let T be the transaction in AST with id Y.tid. (If it does not exist, make a record
         with T.tid __ Y.tid, T.lle __ null, and T.state __ abort).
   if Y.type = tcommit then T.state __ commit {next, use Application Rule, Sec 2.1}
   if T.state = commit and done(Y.object) is false
     and Y.redo is not null then
      begin apply the redo on Y.object; done(Y.object) __ true end
   else if T.state = aborted and done(Y.object) is false
        and Y.undo is not null then
        begin apply the undo on Y.object; done(Y.object) __ true end
   end {of back scan to last checkpoint C}
Delete all AST transactions with committed state
for each T in AST, if T.lle is invalid then T.lle __ null (T.lle is invalid if (C - T.lle) mod
        N > (C - START) )
While {there is a T in AST with T.lle not null} do begin {abort remaining
        transactions}
   select T from AST such that T.lle is not null and (C - T.lle) mod N is the smallest
   Let E __ T.lle, the record to be undone
   if done(Y.object) is false and Y.undo is not null then
      begin apply the undo on Y.object; done(Y.object) __ true end
   if E.lle is invalid, set T.lle __ null, else set T.lle __ E.lle (E.lle is invalid if (E - E.lle)
         mod N > (E - START) mod N)
   end {abort pending transactions}
Delete all transactions from AST;
START __ TAIL ;
CHKPLIST.new __ null; initialize log buffer in memory
Do a checkpoint
Resume transaction processing
```