Performance Evaluation of Ephemeral Logging *

John S. Keen
johnk@ai.mit.edu
(617) 253-7706

William J. Dally
billd@ai.mit.edu
(617) 253-6043

Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Ephemeral logging (EL) is a new technique for managing a log of database activity on disk. It does not require periodic checkpoints and does not abort lengthy transactions as frequently as traditional firewall logging for the same amount of disk space. Therefore, it is well suited for highly concurrent databases and applications which have a wide distribution of transaction lifetimes.

This paper briefly explains EL and then analyzes its performance. Simulation studies indicate that it can offer significant savings in disk space, at the expense of slightly higher bandwidth for logging and more main memory. The reduced size of the log implies much faster recovery after a crash as well as cost savings.

EL is the method of choice in some but not all situations. We assess the limitations of our current knowledge about EL and suggest promising directions for further research.

1 Introduction

Recent technological developments and new application requirements have changed the nature of the logging and recovery problem. In particular, two recent developments have highlighted shortcomings of existing solutions.

First, technology has advanced. The advent of highly concurrent systems consisting of hundreds or thousands of processors has offered much greater processing power, but has made synchronization much more difficult. Traditionally, checkpointing has been a part of all database management system (DBMS) designs. Although there are numerous variations on checkpointing, they all rely on some form of synchronization of activity in the entire system. Naive solutions that demand system quiescence are wasteful of processing resources if many processors are idle for unnecessarily long periods of time. More efficient solutions are complicated and impose overhead. Large main memories are another important technological development. Extrapolation of current trends suggests that many computer systems will have 64 MBytes or more per processor within the next several years.

Second, transactions of widely varying lifetimes may exist simultaneously in a system. Until a transaction commits, some record of its activity must be kept in the log. Traditionally, the log of database activity must hold all records which have been written (by all transactions) since the oldest active transaction[1] began; this space in the log cannot be freed up until the oldest active transaction finishes. There may be many log records which are no longer needed for recovery purposes. However, their space is unavailable for re-use as long as the oldest record must be retained. This "firewall" (FW) approach, originally proposed for System R [4], poses disk management problems. If a transaction lives too long, the log may run out of disk space to hold new records. System R's solution is to simply kill off excessively lengthy transactions.

*The research described in this paper was supported in part by a Natural Sciences and Engineering Research Council of Canada (NSERC) 1967 Scholarship and in part by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738 and N00014-91-J-1698 and in part by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation, IBM Corporation and AT&T Corporation.

SIGMOD /5/93/Washington, DC,USA
© 1993 ACM 0-89791-592-5/93/0005/0187...$1.50

[1] An *active* transaction is one which is still in progress (it has not requested termination nor been aborted).

This solution is clearly unpalatable in an environment where applications require transactions of widely varying lifetimes.

The *logging manager (LM)* is the component of a DBMS which is responsible for managing a log of database activity. *Ephemeral logging (EL)*, a new disk management technique that avoids the two disadvantages described above, is intended as part of a LM design. EL manages the log as a chain of queues to which new records are continuously added. It performs continuous garbage collection and log compression. Only records which must be retained in the log are forwarded from the head of one queue to the tail of the next. In tandem with this activity, EL continuously updates the disk version of the database with the new values of objects[2] that have been updated by committed transactions so that the log records for these modifications are no longer needed. EL does not require checkpointing in the traditional sense and it can accommodate log records from transactions of widely varying lifetimes. It relies on the ephemeral nature of log records which must be retained and exploits differences between the lifetimes of transactions.

EL relies on large quantities of main memory. It attempts to minimize disk space, processing and communication requirements at the expense of increased main memory requirements. It requires more disk bandwidth for log information, but we expect this increase to be small in a well designed system.

Throughout our discussion, we conveniently assume that transactions never write out uncommitted updates to the disk version of the database; a change to an object is propagated to disk only after the associated transaction commits. Hence, log records for modified objects (physical state logging on the access path level [5]) contain only the updated values (REDO logging [4, 5, 1]). This simplifies our discussion. Nevertheless, the techniques proposed in this paper can be extended to the more general situation of UNDO/REDO logging with little difficulty.

The rest of the paper presents an explanation of EL (Section 2), a description of our simulation environment (Section 3), the results of experiments (Section 4) and a review of related work (Section 5). We conclude with a summary of our progress to date and suggestions for further promising research directions.

[2]We use the term *object* broadly to denote any distinct item of data in a database. It may be a record in a hierarchical or network database, a tuple in a relational database or an object in an object-oriented database.

2 Data Structures and Algorithms

2.1 Conceptual Design

Ephemeral logging (EL) manages the log as a chain of fixed-size queues. Each queue is called a *generation*. If there are N generations, then generation 0 is the *youngest* generation and generation $N-1$ is the *oldest* generation. New log records are added to the tail of generation 0. Log records at the head of generation i, for $i < N-1$, are forwarded to the tail of generation $i+1$ if they must be retained in the log; otherwise, their information is *flushed* (transferred) to a permanent version of the database elsewhere on disk or simply discarded. In the special case of generation $N-1$, log records at its head which must be retained are recirculated in it by adding them to its tail. The disk space within each queue is managed as a circular array [2]; the head and tail pointers rotate through the positions of the array so that records conceptually move from tail to head but physically they remain in the same place on disk.

An example of EL with N=3 generations is shown in Figure 1. A stable version of the database resides elsewhere on disk. It does not necessarily incorporate the most recent changes to the database, but the log contains sufficient information to restore it to the most recent consistent state if a crash were to occur. The garbage pail does not actually exist, but is a convenient concept for explanatory purposes.

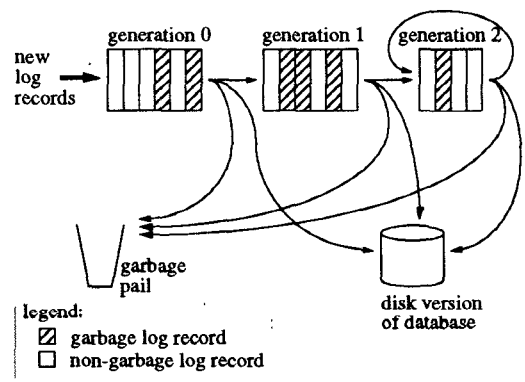

Figure 1: Ephemeral Logging with Three Generations

A *non-garbage* log record is one which may be needed for recovery, and hence must be kept in the log. All remaining log records are *garbage* records. Every log record is initially a non-garbage record. After becoming a garbage record, a log record cannot switch back to become non-garbage again.

The arrows at the head of each generation in Figure 1 portray the three possible fates for a log record at the head. If the record is garbage, it is ignored (conceptually thrown away in the garbage pail). If it is non-garbage and must be retained in the log because its transaction is still active, then it is either forwarded to the tail of the next generation or recirculated in the last generation. For now, suppose that when a non-garbage log record for an update to an object by a committed transaction arrives at the head of a generation, the LM flushes the updated object's new value to the disk version of the database; after this has been done, the record is garbage and is thrown away.

This segmentation of the log is particularly effective if a large proportion of transactions finish execution before their log records reach the head of generation 0; none of their records are forwarded to generation 1 and their disk space can quickly be reclaimed for more incoming log records. Only a small proportion of log records, from transactions with longer lives, are forwarded to subsequent generations.

Recirculation in the last generation means that the physical order of its records no longer necessarily corresponds to the temporal order in which they were originally generated. We assume that all log records are timestamped, so that the recovery manager can establish the temporal order of the records. A recovery method for EL is described in [9].

A *cell* exists for every non-garbage record in any generation of the log. Each cell resides in main memory and points to the record's location on disk. The cells corresponding to each generation are joined in a doubly linked list. The linked list "wraps around" in a circular manner; the cells at the head and tail have right and left pointers to each other, respectively. For generation i, pointer h_i points to the cell for the non-garbage record nearest the head. There is no tail pointer for the generation, but the cell for the non-garbage record nearest to the tail can be found quickly by following the right pointer of the cell pointed to by h_i.

There are two types of log records. *Data log records* chronicle changes to the contents of the database (creation, modification or deletion of data objects). *Transaction (tx) log records* mark important milestones (e.g., begin, commit or abort) during the lives of transactions. The *logged object table (LOT)* has an entry for every data object which has at least one non-garbage data log record somewhere in the log. Likewise, the *logged transaction table (LTT)* has an entry for every transaction with a non-garbage tx log record. Cells for an object's non-garbage data log records are accessible via its LOT entry. Similarly, cells for a transaction's tx log records are associated with its LTT entry. Although cells belong to these two different tables, they may nonetheless simultaneously belong to the same doubly linked list.

Figure 2 illustrates the most important aspects of the data structures for EL. The LOT and LTT, with their constituent cells, reside in RAM. Other internal details of the LOT and LTT have been omitted; the circular doubly linked lists of cells are the important aspect of the LOT and LTT in this figure.

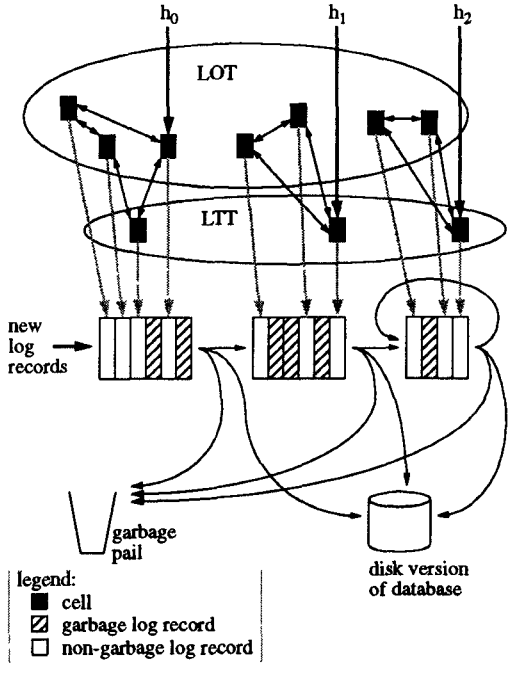

Figure 2: Data Structures for Ephemeral Logging

The log is managed as write-only disk storage. At any given time, the LM can determine whether the record at the head of generation i is non-garbage by checking if h_i points to its head; if not, the LM can safely conclude that the record is garbage. When a record must be forwarded to the tail of generation $i+1$, the LM writes its contents to disk at the tail of generation $i+1$. Its cell, c, is updated to point to its new position in the log and is transferred from the circular linked list for generation i to the circular linked list for generation $i+1$. Pointer h_i is updated to point to the cell previously to the left of c, if such a cell exists for generation i; otherwise, h_i is set to *NULL*. If h_{i+1} was *NULL* immediately before the record was forwarded, then it is updated to point to c (and c's left and right pointers point to itself). Recirculation in the last generation is handled similarly, although it

may occasionally be necessary to kill a transaction if one of its log records cannot be recirculated because of an absence of space in the last generation.

After addition of new records to the tail of a generation, the LM advances the head (by forwarding, flushing or discarding records) so that there is always some gap between the head and tail in the circular array of disk space. This unused disk space is available to hold incoming records at the tail.

2.2 Pragmatic Details

Two characteristics of current disk technology exert an important influence on the implementation of EL. Information is written to disk in fixed sized blocks (with each block typically some multiple of 1024 bytes). Sequential disk I/O is faster than random disk I/O. The technique introduced above must accommodate the constraint of fixed sized disk blocks, and ought to take advantage of the performance benefits of sequential I/O.

Suppose that each disk block is of size B bytes. The LM has a pool of buffers, each of size B bytes. At any given time, there is a current buffer for generation 0. New log records are added to this buffer until it is full, at which time it is written to disk and a different buffer becomes the current buffer[3]. The collection of disk block locations for a particular generation are continually written in the same cyclic order. The head and tail pointers for a generation indicate only block locations; they do not point to more precise locations within a block. Similarly, a cell indicates merely the block to which its record belongs. This coarse resolution for pointers suffices for implementation of EL.

The movement of head and tail pointers in block sized quanta has implications. When the head of a generation is advanced to a new block, the LM must deal with all log records in this block. Some are forwarded, some flushed, and others discarded. Suppose that several log records are forwarded from generation i to generation $i+1$. These records are added to an empty buffer and are generally insufficient to fill the buffer but the LM must ensure that the forwarded records are immediately written to disk. Therefore, it attempts to fill the buffer as full as possible before writing it. After forwarding records from the block at the head of generation i, the LM works backward from the head to gather enough other non-garbage log records to fill the buffer that is destined for the tail of generation $i+1$. In summary, the requirements of generation i dictate that records be removed from its head in quanta of size at least a block. The requirements associated with forwarding records to the tail of generation $i+1$ imply that records are forwarded as a group from the first several blocks at the head of generation i.

Recirculation is not as complicated. The LM can remove records from only the block at the head of the last generation and place them in a buffer without immediately writing it to disk. The existing copies of these records will not be overwritten until after the tail has advanced, but the recirculated records will belong to the disk block written at the tail.

The introductory explanation of EL in section 2.1 suggested that committed updates are flushed to the stable version of the database kept elsewhere on disk when their log records next reach the head of a generation. In general, there is negligible locality of access between the updates of independent transactions. Flushing updates in the order that they are written to the log would lead to random disk I/O. Instead, the LM attempts to schedule flushes so that it can take advantage of locality in the disk version of the database and thus improve I/O performance. At any given time, there should be a significantly large number of committed updates from which the LM can choose the next object to be flushed; too small a "pool" of updates leads to random I/O. The LM can flush a data log record's update to disk any time after its transaction has committed. Flushing can proceed continuously at as high a rate as possible; it is no longer triggered by the arrival of a new block at the head of a generation. After the LM flushes an update from a data log record, the record becomes garbage. Ideally, every committed update is flushed before it arrives at the head of its generation so that records at the head of any generation are either forwarded (or recirculated) or thrown away. In practice, a few may reach the head of a generation and require flushing; there will be a small amount of random I/O, but much less than for the naive approach originally described. Alternatively, we can keep an unflushed update's record in the log by forwarding or recirculating it until the update is eventually flushed.

As before, the LM continues to ensure that there is always enough of a gap between the head and the tail of every generation. Now, this gap is measured in terms of available disk blocks.

The circular linked list of cells for generation i is still managed according to the description in section 2.1.

[3] Several buffers are necessary because a disk write generally requires a significant amount of time, such as 10 ms, during which many other log records may arrive. While one buffer is being written to disk, new records can be added to a different buffer without risk of interference.

The h_i pointer keeps track of the non-garbage log record nearest the head of generation i. The h_i pointer points to a cell; it does not (directly) point to the block nearest the head which contains at least one non-garbage record.

2.3 Management of the LOT and LTT

The LOT and LTT keep track of all non-garbage log records. The LM updates them on a continual basis as records enter the log and progress through it.

When the DBMS initiates a new transaction, it sends a `BEGIN` record to the log. In response, the LM adds the `BEGIN` record to a buffer which will be written to the tail of generation 0, creates a cell to point to the record and creates a new entry in the LTT. Even though the LM has not yet written the buffer to disk, it knows the position of the disk block to which it will eventually be written. The cell's pointer to the record's position in the log consists of the address of this disk block. The LTT entry for the transaction points to the cell. Additionally, each LTT holds a set of object identifiers (*oids*) to keep track of which objects were updated by the transaction; this set is initially empty and grows as the transaction progresses and performs work. Entries in the LTT are associatively accessed using transaction identifiers (*tids*) as keys. A hash table implementation is therefore appropriate. The dynamic nature of the LTT strongly suggests that *chaining* [2] (rather than *open addressing*) is the most suitable technique for collision resolution.

We assume that only the most recent tx log record is ever required for any transaction; all earlier tx log records are garbage. Throughout the time that a transaction has an LTT entry, there is one cell which points to its most recent tx log record. Whenever a transaction writes another tx log record (such as `COMMIT`), the LM adds it to the log (at the tail of generation 0) and updates the cell for the transaction's previous tx log record to point to the disk block of this newest record[4].

The LOT is accessed associatively by object identifiers (*oids*). Like the LTT, it is implemented as a hash table with chaining. An object's LOT entry has one or more cells, each of which points to the disk block of a non-garbage data log record for the object. An object has a cell for the most recently committed update (if any) if this update has not yet been flushed; it may have several cells for uncommitted updates.

Whenever a transaction modifies an object in the database, it sends a data log record to the log. If an entry does not already exist for the object in the LOT, the LM creates one. The LM creates a cell to point to the data log record's block in the log, adds it to the set of cells maintained in the object's LOT entry, inserts the cell in the doubly linked list for generation 0 and adds the object's oid to the set of oids in the LTT entry of the transaction which performed the update.

Every transaction eventually commits or aborts. An abort is easy to handle. All data and tx log records from an aborted transaction immediately become garbage; the cells which pointed to these records are disposed. The transaction's LTT entry is deleted.

When a transaction commits, the LM updates its tx log record cell to point to the `COMMIT` record in the log and processes the set of oids in its LTT entry. For each oid, the LM retrieves the object's LOT entry. If a data log record for an earlier committed update existed, it is now garbage; its cell is removed from the object's LOT entry and disposed. The current transaction's most recent update now becomes the most recently committed update for the object.

After the LM flushes an update to the disk version of the database, the associated data log record is garbage. The LM removes the cell for this record from the object's LOT entry and disposes it. If the set of remaining cells is empty (i.e., the object no longer has any non-garbage data log records in the log), the LM deletes the object's entry from the LOT.

Whenever a data log record becomes garbage and its cell is disposed, the oid is removed from the LTT of the transaction which originally wrote the record. When the set of oids in a committed transaction's LTT entry is empty, the LM disposes the cell for the transaction's most recent tx log record (because the record is garbage) and removes the transaction's entry from the LTT.

To summarize, every object with non-garbage data log records in the log has an entry in the LOT. An object's LOT entry keeps track of the positions within the log of its non-garbage data log records. There is an LTT entry for every transaction currently in progress and every committed transaction which still has non-garbage data log records. A transaction's LTT entry keeps track of all objects which it updated and the position within the log of its most recent tx log record. The LM continually updates the LOT and LTT to reflect the current state of the system as transactions and log records come and go. At any given time, the cells associated with the LOT and LTT entries point to all non-garbage records in the log.

[4] The cell is also transferred from its previous doubly linked list of cells to the tail of the doubly linked list for generation 0.

3 Simulation Environment

We have implemented an event-driven simulator to study EL. The simulator is written in C and runs on SPARCstations. The user can specify the following input parameters:

pdf:	statistical mix of transactions
arrival rate:	rate of transaction initiation
flush rate:	for flushing committed updates
generations:	the number and size of generations
recirculation:	flag to turn circulation on or off
runtime:	duration of simulated time span

The user specifies an arbitrary number of different transaction types and their probability distribution function (pdf). For each type of transaction, the user states the probability of occurrence, the duration of execution, the number of data log records written and the size of each data log record. This transaction model is graphically represented in Figure 3 for a transaction that generates two data log records.

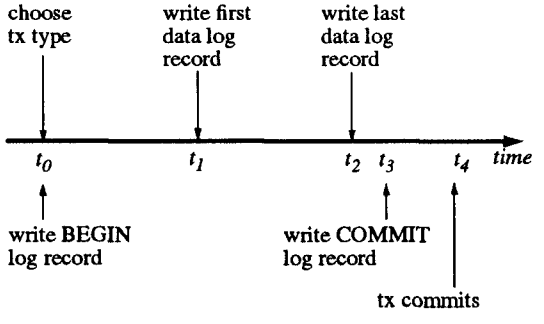

Figure 3: Simulation Transaction Model

Whenever a new transaction must be initiated, the simulator randomly (according to the pdf) selects its type. After choosing its type, the simulator schedules when its log records will be written. The `BEGIN` tx log record is written immediately after it is initiated (at time t_0). The data log records are written at equally spaced intervals, with the last being written only some short time ϵ (equal to $t_3 - t_2$) prior to completion. Suppose that the transaction's lifetime (specified as part of its type) is T. It will finish execution and write a `COMMIT` tx log record (at time t_3) T seconds after it started. Its last data log record is written (at time t_2) $T - \epsilon$ before it finishes, and each data log record is written $(T - \epsilon)/N$ after the preceding one, where N is the number of data log records written by a transaction of this type. After writing the `COMMIT` record, the transaction waits for acknowledgement (at time t_4) from the LM before it actually commits; this delay occurs because the LM waits until a buffer is almost full before writing it to disk at the tail of generation 0, and then there is some delay (typically 10 to 20 ms) for transferring the contents to disk.

Transactions are initiated at regular intervals, according to the specified arrival rate (transactions per second). We believe that this simple, deterministic arrival pattern is sufficient for a first order evaluation of EL. More complicated probabilistic models (such as Markov arrivals) may be investigated in future work.

We do not model feedback in the transaction scheduling. In reality, the performance of the database system may affect the rate at which new transactions are submitted for execution, the times at which they write records to the log and their duration. The details of this *arrival mechanism* are beyond the scope of this paper.

Whenever a transaction writes a data log record, we randomly pick some integer for the oid, subject to the constraint that the number has not already been chosen for an update by a transaction which is still active. The set of integers from which an oid can be chosen consists of all integers from 0 up to NUM_OBJECTS-1, where NUM_OBJECTS is the total number of objects (a fixed value).

To control the rate at which updates are flushed, the user specifies some number of disk drives and the time required to write a block to any of these drives. We assume that there can be at most one request at a time for any particular drive. The user can increase the maximum rate at which updates are flushed by increasing the number of drives or decreasing the time to write a block to any drive. The objects are range partitioned [3] evenly over these drives. That is, for NUM_OBJECTS objects and D drives, the first NUM_OBJECTS/D objects[5] reside on drive 0, and so on. We assume that each updated object requires a separate disk write (i.e., there is negligible locality of updates within a disk block). Each disk drive attempts to service pending flush requests in a manner that minimizes access time. In our simulator, we assume that the difference between two objects' oids corresponds to their locality on disk. When calculating the difference between two oids, we assume that the range of integers assigned to their disk drive wraps around.

The user specifies the number and size (number of disk blocks) of each generation. The size of each disk block is fixed in the simulator.

In some experiments, we examine the behavior of EL without recirculation in the last generation, just so

[5] For simplicity, we ignore the case where NUM_OBJECTS is not a multiple of D.

that we can see the effect of simply segmenting the log. There is an input flag to specify whether recirculation in the last generation is turned on or off. If recirculation is disabled and a transaction's non-garbage log record reaches the head of the last generation while it is still executing, the LM kills the transaction.

Several parameters are fixed. The delay ϵ between the writes for the last data log record and the **COMMIT** tx log record for a transaction is fixed at 1 ms. The capacity of each disk block is 2000 bytes[6]. At any given time, at least k blocks must be available to hold new log records. This threshold distance is currently fixed at k=2 blocks. Four disk block buffers (2048 bytes each) are provided for each generation. The **BEGIN** and **COMMIT** tx log records are assumed to both require 8 bytes. We conservatively assume a fixed delay of τ_{Disk_Write}=15 ms to transfer a buffer's contents to disk when writing out records to the tail of the log. The simulator uses the *group commit* technique [1]; a log record is not written to disk until its buffer is as full as possible. Therefore, the delay between the time a record is added to a buffer and the time it is written to disk is generally longer than τ_{Disk_Write}. The number of objects in the database is fixed at NUM_OBJECTS=10^7.

4 Experimental Results

We ran several experiments to observe the behavior of EL and understand the effects of varying different parameters. We also simulated the traditional FW technique by using a single log with no recirculation and compared the results of EL to those of FW. We did not implement a checkpoint facility for the FW technique; the firewall was always the oldest non-garbage log record from the oldest active transaction. This omission favors FW because it ignores the overhead (in terms of disk space and bandwidth) associated with checkpointing.

We consider several evaluation criteria. Disk space, disk bandwidth (in terms of block writes per second) and RAM requirements (for the LOT and LTT) are the main criteria we examine. We also try to estimate the degree to which disk I/O is random or sequential. Another important consideration is how fast recovery time is after a crash. We do not simulate recovery so we cannot cite any quantitative results. However, it is generally true that recovery time is proportional to the amount of log information and so less disk space means faster recovery.

For all experiments, we considered two transaction types. The first is of 1 s duration and writes 2 data log records, each of size 100 bytes. The second lasts 10 s, in which time it writes 4 data log records of size 100 bytes each. These are characteristic of transactions for many common interactive applications. Most transactions are relatively short and modify only a small amount of the database. Other transactions are more complex and so they take longer to execute; they may also update more objects in the database. We vary the relative frequencies with which these two types of transactions occur.

The arrival rate was 100 TPS (transactions per second) and the simulation time was 500 s for all experiments. All tests of EL used two generations.

As the fraction of 10 s transactions increases from 5% to 40%, the average number of updates per second rises from 210 to 280. To provide sufficient bandwidth for flushing updates, we specified 10 disk drives with a transfer time of 25 ms (net bandwidth is 400 flushes per second). The conservative 25 ms time allows for some read operations to be interspersed with writes.

We estimate that the FW method requires 22 bytes for each transaction (including a pointer to the position within the log of its oldest log record) in the system. The EL method requires 40 bytes for each transaction and 40 bytes for each updated (but unflushed) object.

Figure 4 plots the disk space requirements (number of blocks) versus the transaction mix for both FW and EL. Recirculation in the last generation is disabled for EL, so that we can assess the effect of simply segmenting the log. For both FW and EL, we continued to run simulations and reduce the disk space until we observed transactions being killed. Hence, these results reflect the minimum disk space requirements to support 500 s of logging activity in which no transaction is killed. The corresponding graphs of disk bandwidth (to only the log) and main memory requirements for these tests are shown in Figures 5 and 6, respectively.

The advantages of EL are most apparent for the 5% mix. It reduces disk space by a factor of 3.6 with only an 11% increase in bandwidth; memory requirements are modest. As the proportion of 10 s transactions increases, EL's relative advantage over FW diminishes. The reduction in disk space is not as large, but the increase in bandwidth is greater.

We enabled recirculation in EL's last generation and progressively decreased its size until we observed transactions being killed. For all tests, we used the

[6] A block size of 2048 is typical, but we assume 48 bytes are reserved for bookkeeping purposes and so only the remaining 2000 bytes are available to hold log records.

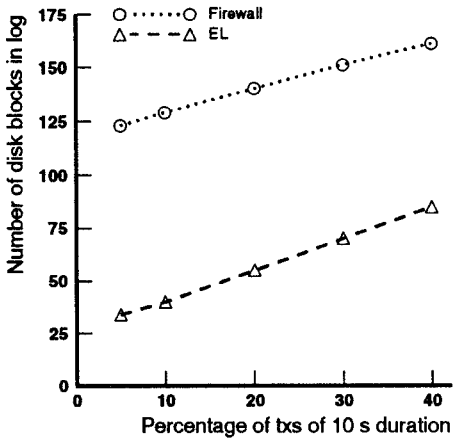

Figure 4: Disk Space Requirements vs. Tx Mix

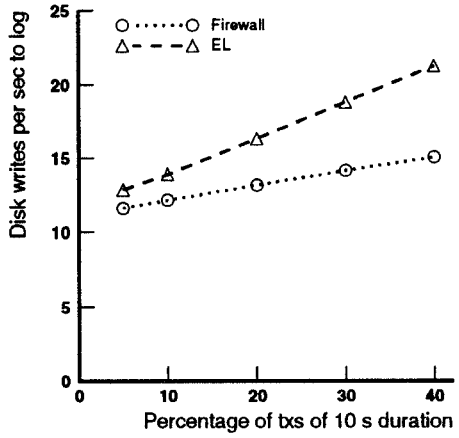

Figure 5: Disk Bandwidth vs. Tx Mix

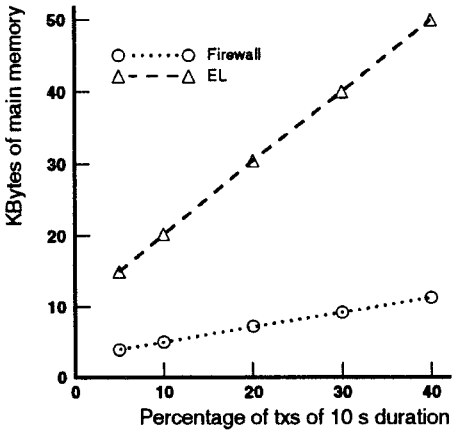

Figure 6: Memory Requirements vs. Tx Mix

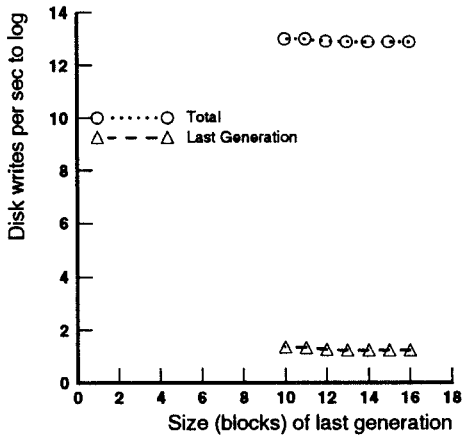

Figure 7: EL Disk Bandwidth vs. Space

same mix of 5% 10 s transactions. The size of the first generation remained fixed at 18 blocks (for which the minimum space was obtained in the case of no recirculation). Figure 7 plots the bandwidth to the last generation[7] and the total overall logging bandwidth (both generations) as the size of the last generation is varied. The amount of space required for EL decreases from 34 to 28 blocks, while bandwidth increases from 12.87 to 12.99 writes/sec (main memory consumption is unchanged since the number of non-garbage log records is the same). Compared to the FW case of 123 blocks and 11.63 writes/sec, these results for EL constitute a factor of 4.4 reduction in disk space and a 12% increase in bandwidth.

We point out that 28 blocks of 2 KBytes each can all fit in the main memory of many workstations with lots of room to spare. The traditional two pass (undo, redo) recovery method [1] that was appropriate for databases with large logs and small main memories is no longer appropriate. Now, we can read the entire log into memory and perform recovery with a single pass [9]. Recovery in less than a second may be feasible.

To see how EL performs when bandwidth for flushing is scarce, we increased the flush transfer time to 45 ms so that the 10 disk drives together provide a maximum bandwidth of 222 writes per sec. For a transaction mix with 5% 10 s transactions, the database is updated at an average rate of 210 objects per second. Familiarity with queueing theory suggests that a backlog of unserviced flush requests will accumulate as the flushing service rate approaches the rate of new arrivals. Under these circumstances, EL with recirculation requires 31 disk blocks (20 and 11

[7] The first generation has the same bandwidth as before. Only the second generation experiences increased bandwidth, because of the recirculated records.

in generations 0 and 1, respectively) and a bandwidth of 13.96 writes per sec. Unflushed committed updates are recirculated in generation 1 until they are eventually flushed, but the extra disk space and bandwidth are not prohibitive. The average distance between oids of successively flushed objects is now 109,000. Compared to the average of 235,000 which we observed for previous tests when the transfer time was 25 ms, this indicates a significant increase in locality. As a backlog accumulates, disk I/O for flushing becomes less random and more sequential. This negative feedback provides some stability. We conclude that EL is suitable when flushing bandwidth is scarce.

5 Related Work

EL essentially performs log compression as it passes records from one generation to the next. It filters out records which are no longer needed and keeps only those records that would be required if a crash were to suddenly happen. The continuous nature of EL distinguishes it from previous log compression methods [8, 6]. These earlier methods are intended for "batch mode" log compression; records cannot be added to the existing log while it is being compressed. EL employs very different techniques to perform log compression.

The idea of segmenting a fixed amount of storage space into several temporal generations for the purposes of garbage collection is not new. Lieberman and Hewitt [10] adopt such a strategy when there are more general reference patterns amongst a dynamically varying collection of objects held in main memory. However, a review of the literature reveals no prior pursuit of a similar strategy for the management of disk space for a database system's log.

Rosenblum and Ousterhout [11] use some similar ideas in the log-structured file system (LFS). The LFS adds all changes to data, directories and metadata to the end of an append-only log so that it can take advantage of sequential disk I/O. The log consists of several large segments. A segment is written and garbage collected ("cleaned") all at once. To reclaim disk space, the LFS merges non-garbage pieces from several segments into a new segment; it must read the contents of a segment from disk to decide what is garbage and what isn't. There is a separate checkpoint area and the LFS performs periodic checkpoints. The LFS takes advantage of the known hierarchical reference patterns amongst the blocks of the file system during logging and recovery.

Hagmann and Garcia-Molina [7] propose recirculation of log records from long lived transactions within an unsegmented (i.e., single generation) log. Records to be recirculated must be read from disk. Their method still relies on checkpoints.

6 Concluding Remarks

This paper has presented a quantitative evaluation of ephemeral logging and demonstrated that it has clear advantages over the established firewall technique for some applications. EL can significantly reduce the amount of disk space required to retain log information when a small proportion of transactions have relatively long lifetimes. The smaller this proportion and the longer the lifetimes, relative to other transactions, the greater is the reduction in disk space. The benefits of reduced disk space for the log are twofold. First, it reduces the cost for a system. Second, recovery time after a crash is expected to be much shorter because there is less log information to process. The cost of EL is higher disk bandwidth for managing log information and greater main memory requirements. The amount of extra bandwidth required by EL decreases as the fraction of long-lived transactions decreases.

We originally formulated EL for a database which retains a version number timestamp with each object. For the more general case of no timestamps in the database, a broader definition of non-garbage records is required to ensure correct recovery; some log records may need to wait longer before becoming garbage.

EL challenges the conventional wisdom that checkpoints are an inherent necessity for any logging algorithm. By offering continuous logging service, EL is particularly well suited for highly concurrent database systems, in which checkpointing is expected to be much more awkward than for sequential systems.

There are some situations in which the FW technique is better than EL. When all transactions are approximately the same duration and checkpoints do not impose significant overhead, the FW technique requires no more disk space than EL, but it consumes less bandwidth and main memory. Therefore, EL and FW should be regarded as two alternative design techniques available to database system designers.

EL provides fault tolerance to *system failures* [1], in which the contents of main memory are lost. We assume that main memory is large enough to buffer the original and updated values for all objects which an active transaction has modified, and so a *transaction*

failure can easily be handled without needing to read anything from disk. Previously known techniques [4, 5, 1] for *archiving* continue to provide fault tolerance to *media failures* in which some information on disk is irretrievably lost.

Implementation of EL requires significant control over the management of information on disk. The DBMS must have control over a set of disk blocks which are dedicated for the log. The DBMS chooses when new contents are written to any of these blocks. Not all operating systems provide such a high degree of control over "low level" I/O. Stonebraker [12] wrote an excellent critique of the shortcomings of existing operating systems and file systems, from the viewpoint of a DBMS designer. We add our support to Stonebraker's criticisms, and we encourage operating system designers to heed the needs of DBMS designers.

The optimal number of generations and their sizes depends on the application. We cannot offer any provably correct analytical methods as tools to a database administrator (DBA) who must specify these parameters when a system is configured. Ideally, we would like an adaptable version of EL that dynamically chooses the number and sizes of generations itself.

Another promising avenue of investigation is a possible EL-FW hybrid scheme. Like EL, the log is segmented into a chain of FIFO queues. Like FW, a firewall is maintained for each queue; The oldest non-garbage record in a queue is its firewall. Now, the LM retains a pointer to only the oldest log record from each transaction. This can drastically reduce main memory consumption if each transaction updates many objects, but at a price of higher bandwidth. When a transaction's oldest non-garbage log record reaches the head of one queue, *all* of its log records must be regenerated and added to the tail of the next queue because the LM does not have pointers to know their whereabouts in the current queue.

Suppose the transaction manager can estimate the expected lifetime of a transaction when it begins (for example, the type of the transaction may indicate its expected duration). The LM might exploit this information to reduce bandwidth. Rather than letting the transaction's records progress through successively older generations, it directly adds the transaction's log records to the tail of a generation in which the records are unlikely to reach the head before the transaction finishes. This technique would be particularly beneficial in conjunction with the hybrid EL-FW approach described in the previous paragraph.

Acknowledgements

We gratefully thank all members of the MIT CVA (Concurrent VLSI Architecture) Group for their valuable criticisms, comments and suggestions on the material presented in this paper. Prakash Sundaresan brought [7] to our attention and we thank him for it. We appreciate the many good criticisms and suggestions of the SIGMOD referees.

References

[1] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Massachusetts, 1987.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 1990.

[3] David DeWitt and Jim Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Comm. of the ACM*, 35(6):85–98, June 1992.

[4] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13(2):223–242, June 1981.

[5] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[6] Robert B. Hagmann. A Crash Recovery Scheme for a Memory-Resident Database System. *IEEE Trans on Computers*, C-35(9):839–843, September 1986.

[7] Robert B. Hagmann and Hector Garcia-Molina. Implementing Long Lived Transactions Using Log Record Forwarding. Technical Report CSL-91-2, Xerox Palo Alto Research Center, February 1991.

[8] John Kaunitz and Louis Van Ekert. Audit Trail Compaction for Database Recovery. *Comm. of the ACM*, 27(7):678–683, July 1984.

[9] John S. Keen. Logging and Recovery in a Highly Concurrent Stable Object Store. Technical Report CVA Memo #37, MIT, May 1991. *Revised November, 1991.*

[10] Henry Lieberman and Carl Hewitt. A Real-Time Garbage Collector Based on the Lifetime of Objects. *Comm. of the ACM*, 26(6):419–429, June 1983.

[11] Mendel Rosenblum and John K. Ousterhout. The LFS Storage Manager. In *Proc Summer '90 USENIX Technical Conf*, Anaheim, California, June 1990.

[12] Michael Stonebraker. Operating System Support for Database Management. *Comm. of the ACM*, 24(7):412–418, July 1981.