Log Partitioning Techniques for Transaction Processing Systems in a Shared-nothing, Multiprocessor Environment

Michael J. Skarpelos
Susanne Englert
Robbert van der Linden
Jim Carley

February 20, 1997

Log Partitioning Techniques for Transaction Processing Systems in a Shared-nothing, Multiprocessor Environment

0. Abstract

The Log Manager component of a transaction management system often becomes a bottleneck during very heavy workloads. In this paper, we present two complimentary log partitioning techniques for increasing Log Manager throughput. Both techniques rely on an underlying shared-nothing, multiprocessor hardware platform. We also present an analytical model for determining when to use the techniques and experimental results which serve to illustrate the model.

1. Introduction

At a conceptual level, one can regard a *log* (also called an audit trail or journal) in a transaction processing system as a history of changes to a database. A typical log record describes a particular database change: it consists of a *before* and *after image* of a modified database record. With before images, a transaction processing system can undo modifications made by transactions which abort or fail to complete due to system failures. With after images, a transaction processing system can recover from media failures by restoring old (possibly inconsistent) copies of database files and redoing the earlier modifications. Logs also contain *commit* records that indicate which transactions have successfully completed. (For a lucid and thorough account of logs and transaction management see [GRAY78] chapter 5, and [BERN87].)

In high performance, transaction management systems, bottlenecks often occur in the system component which writes log records to disk (hereafter called the *Log Manager*). In this paper, we present two complimentary log partitioning techniques for increasing Log Manager throughput. Both techniques rely on an underlying shared-nothing, multiprocessor hardware platform. (See [GRAY93] page 59ff. and [DEWI92].)

The ideas presented here arose out of recent software development work at Tandem Computers which in turn built upon earlier transaction management work at Tandem. (See [BORR81] and [BORR84].)

In section 2, we describe a general log management architecture. In sections 3 and 4 we present the two log partitioning techniques: vertical and horizontal log partitioning, respectively. In section 5 we present an analytical model for determining when to use the techniques, and in section 6 we illustrate the model with the results of actual experiments. We conclude with a summary (section 7), future research (section 8) and acknowledgments (section 9).

2. Log Management Architecture

In this section we present an log management architecture for a typical transaction management system running on a shared-nothing, multiprocessor platform. (For a more detailed view of a Log Manager, see [GRAY93] chapter 9.)

2.1. The Log

A log consists of an ordered sequence of log files. The most recent log files typically reside on a duplexed disk to decrease the likelihood of losing log information if a media failure occurs. The Log Manager writes log records to a particular file (known as the *current* log file) until the file becomes full. At that point, the Log Manager begins writing to a newly created file while an archiving process (known as the *Archiver*) backs up the old log file to tape or other archiving media. We call this transition from old to new log file a *log file rollover*.

After the Archiver has backed up the old log file, the Log Manager can reclaim disk space by deleting the file once all system components no longer need it. (For example, the Log Manager should not delete any file containing records associated with a currently active transaction since it will need this file if the transaction aborts.)

Figure 1 below depicts a Log Manager, labeled **L**, writing log data to a duplexed disk. The Archiver, labeled **A**, backs up log files to archival media so the Log Manager can later reclaim disk space for new log files.

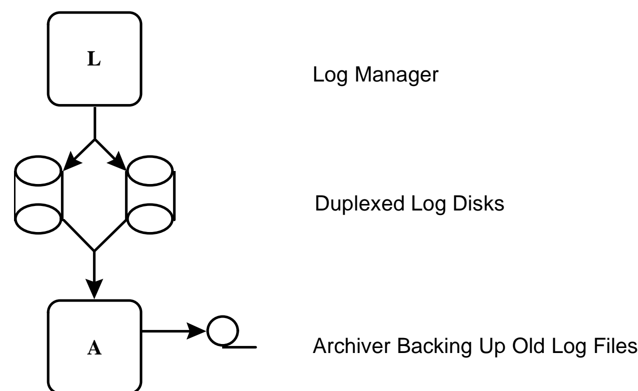

Figure 1

2.2. Log Record Generation

Log record generation occurs whenever the database system updates, deletes or inserts database records. Given a shared-nothing, multiprocessor hardware platform, we assume a physical partitioning of the database (see [DEWI92]). Database files can span partitions, but at any single point in time, a database record resides in exactly one database partition (i.e. no data replication). Each database partition has an associated process which accepts requests from application programs and database users to modify or retrieve data records in that particular partition (and only in that partition) of the database. As part of modifying their database partition, the process generates log records and sends those log records to the Log Manager. Hence we call the process a *Log Generator*.

A given transaction can access any of the database partitions, so (potentially) many Log Generators can produce log records on the transaction's behalf. In addition to the Log Generators which access and modify database partitions, a special Log Generator called the *Commit Coordinator* sends commit records to the Log Manager. Since the Log Manager must immediately force commit records to disk (see the “Write Ahead Log Protocol” described in [GRAY78] section 5.8.3.2 and [BERN87] pages 177ff.), a single Commit Coordinator can take advantage of "boxcarring" techniques to improve transaction throughput and response time (see [HELL87]).[1]

Readers familiar with the X/Open XA specification [XOPE91] can think of Log Generators for database partitions as Resource Managers and the Commit Coordinator as a Transaction Manager, all of which share a common logging facility.

Figure 2 below illustrates the generation of log records. The database partitions labeled $\mathbf{P_1}$ through $\mathbf{P_n}$ each have an associated Log Generator $\mathbf{G_1}$ through $\mathbf{G_n}$. The Log Generator labeled **CC** is the special Commit Coordinator. The labels **L** and **A** refer to the Log Manager and Archiver respectively (as in Figure 1). Arrows from Log Generators (including the Commit Coordinator) to the Log Manager indicate the flow of log records.

[1] For readers familiar with Tandem parlance, Log Generators for database partitions correspond to Data Disk Processes, the Commit Coordinator corresponds to the Transaction Monitor Process, and the Log Manager corresponds to the Audit Trail Disk Process.

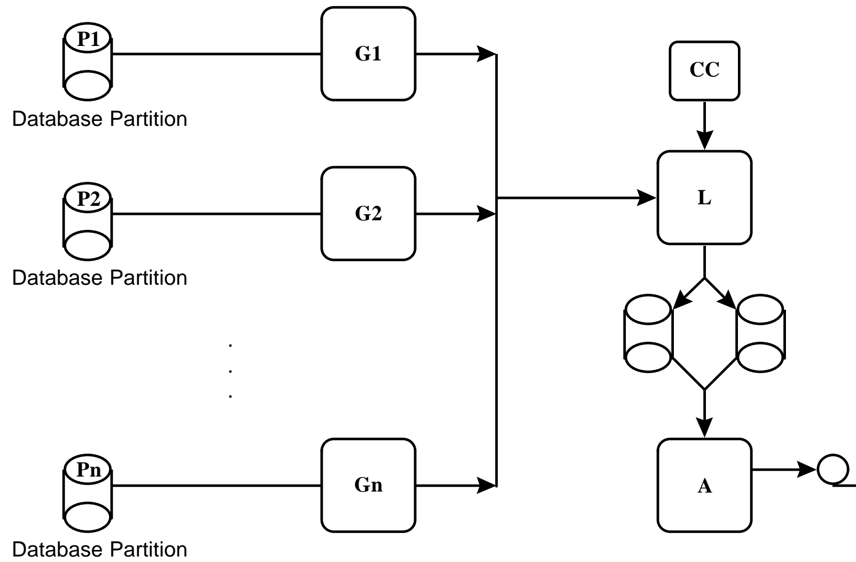

Figure 2

3. Vertical Log Partitioning[2]

A large number of Log Generators (or even a small number of active Log Generators) can overwhelm a single Log Manager. Vertical log partitioning helps alleviate this bottleneck by increasing the number of Log Managers and redirecting the log data from some of the Log Generators to these new Log Managers. With a shared-nothing, multiprocessor architecture, each Log Manager can run in a separate CPU and manage a separate pair of duplexed disks. Thus, log throughput can scale linearly with the number of Log Managers.[3]

Figure 3 below illustrates vertical log partitioning. It shows four Log Generators (including the Commit Coordinator); the Commit Coordinator and Log Generator $\mathbf{G_1}$ send their log data to Log Manager $\mathbf{L_1}$ while Log Generators $\mathbf{G_2}$ and $\mathbf{G_3}$ send their log data to Log Manager $\mathbf{L_2}$. If all four Log Generators could overwhelm a single Log Manager, but no two Log Generators could do so, splitting the log traffic among two Log Managers removes the bottleneck.

[2] Vertical log partitioning corresponds to auxiliary audit trails in Tandem parlance.

[3] Tandem's current implementation allows for up to 16 vertical log partitions.

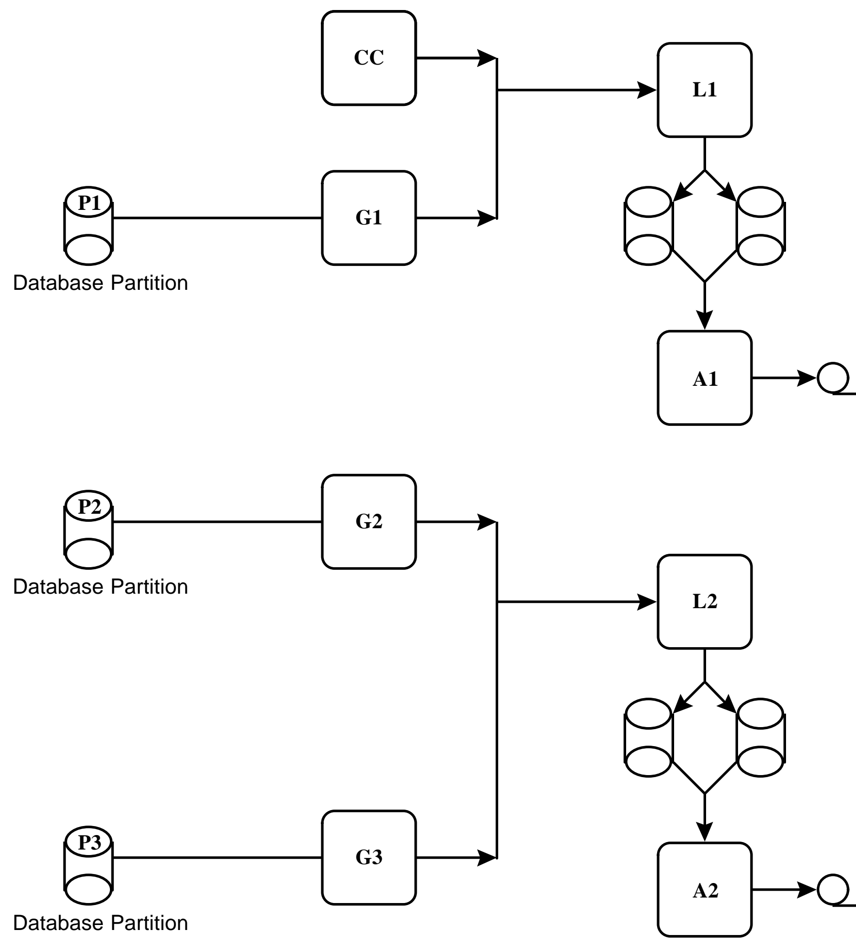

Figure 3

Vertical log partitioning does incur some extra message overhead. The "Write Ahead Log Protocol" (See [GRAY78] section 5.8.3.2 and [BERN87] pages 177ff.), requires the Commit Coordinator to ensure that all log records associated with a particular transaction reside on disk before it can write a commit record for that transaction. With vertical log partitioning, this means the Commit Coordinator must send an extra message asking each Log Manager (other than its own) which has received log records for the transaction in question to write those log records to disk. By "boxcarring" commit records (see [HELL87]), however, the Commit Coordinator can amortize this extra message cost over multiple transactions.

4. Horizontal Log Partitioning[4]

Immediately after a log file rollover, disk head contention arises between the Log Manager and the Archiver. This contention continues until the Archiver has completed backing up the old log file to tape (or other archival media). During this period of contention, Log Manager throughput can degrade appreciably.

Horizontal log partitioning removes this contention between the Log Manager and Archiver by distributing log files among multiple pairs of duplexed disks (in a round robin fashion). Each pair of duplexed disks has its own Log Manager and Archiver, but at any single point in time only one Log Manager accepts new log data from Log Generators, and only one Archiver backs up an old log file to archival media. In a shared-nothing, multiprocessor architecture, the active Log Manager and Archiver can each run in separate CPUs and access different pairs of duplexed disks, so no CPU or disk head contention arises, and Log Manager throughput does not degrade. (See [GRAY93] section 9.6.4. for an interesting alternative to horizontal log partitioning.)

Figure 4 below illustrates horizontal log partitioning. Log files get distributed among Log Managers $\mathbf{L_1}$ and $\mathbf{L_2}$ in a round robin fashion. All Log Generators send their log records to the current Log Manager $\mathbf{L_1}$ while Archiver $\mathbf{A_2}$ backs up an old log file from a different duplexed disk. As indicated by the dotted lines, Log Manager $\mathbf{L_2}$ and Archiver $\mathbf{A_1}$ remain dormant.

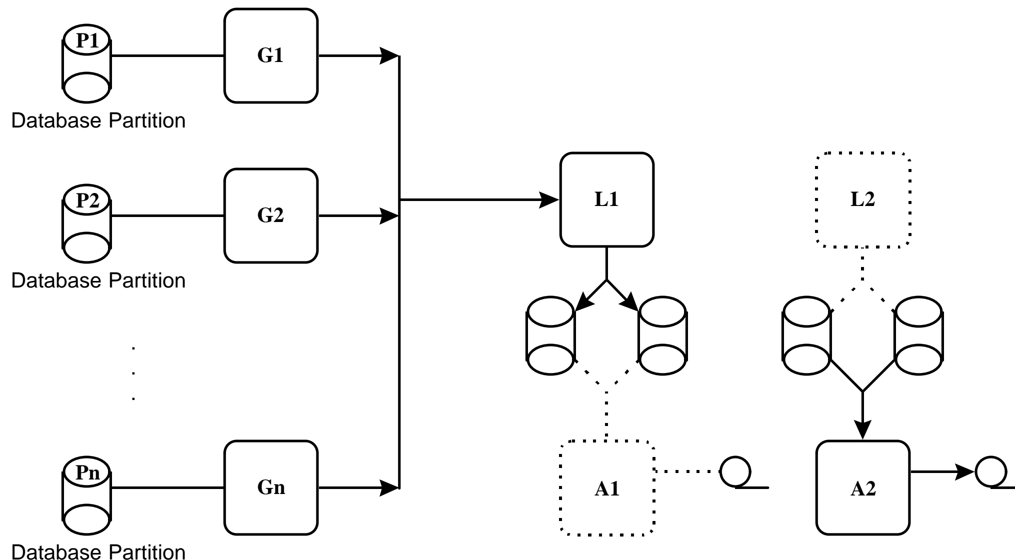

Figure 4

[4] Horizontal log partitioning corresponds to mult-volume audit trails in Tandem parlance.

As soon as Log Manager $\mathbf{L_1}$ fills the current log file, it informs Log Manager $\mathbf{L_2}$ about the rollover. Log Manager $\mathbf{L_2}$ becomes current, Log Manager $\mathbf{L_1}$ tells its associated Log Generators to redirect their log data to Log Manager $\mathbf{L_2}$, and Archiver $\mathbf{A_1}$ begins backing up the formerly current log file to archival media. At this point, Log Manger $\mathbf{L_1}$ and Archiver $\mathbf{A_2}$ become dormant. Figure 5 below illustrates the situation after the rollover has completed.

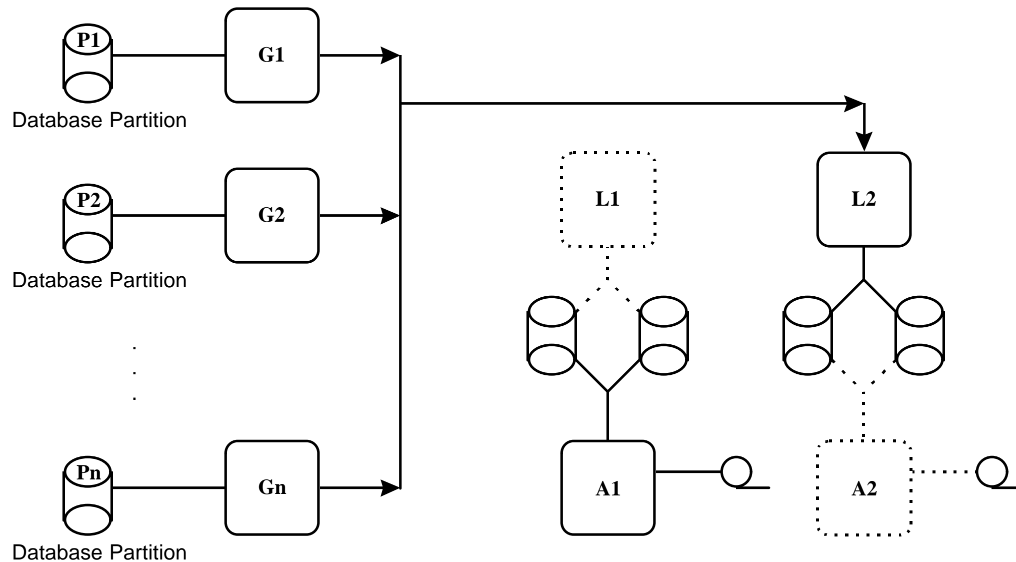

Figure 5

Horizontal partitioning also has some practical administrative benefits as well. For example, if archiving requires manual intervention (to mount a tape, for example), horizontal partitioning a log among many pairs of duplexed disks[5] can effectively delay when manual intervention needs to take place.

Horizontal log partitioning, as with vertical log partitioning, has some extra message overhead. During a log file rollover, the formerly current Log Manager must tell a dormant Log Manager to become current. (We colloquially refer to this as “baton passing”). The formerly current Log Manager must also notify its associated Log Generators about the log file rollover so that they can redirect their log data to the newly current Log Manager. Since rollovers occur relatively infrequently, however, this extra message overhead typically has little impact.

System administrators can configure horizontal and vertical log partitioning simultaneously. Specifically, a system administrator who configures a set of vertically

[5] Tandem’s current implementation allows for up to 16 horizontal log partitions.

partitioned Log Managers to handle large log generation rates can further partition those Log Managers horizontally to reduce contention with Archivers.

5. Analytical Model

Log throughput depends on how much log data the various Log Generators generate and how much log data a Log Manager can accept during a given time interval. Let us begin with log record generation.

Consider database partitions $\mathbf{P_1}$, $\mathbf{P_2}$, … $\mathbf{P_n}$ and transaction types $\mathbf{T_1}$, $\mathbf{T_2}$, … $\mathbf{T_m}$.[6] We can define $\mathbf{G_{ij}}$ as the amount of log data (in bytes) generated by the Log Generator associated with partition $\mathbf{P_i}$ on behalf of transaction type $\mathbf{T_j}$. (Of course $\mathbf{G_{ij}}$ equals zero if transactions of type $\mathbf{T_j}$ do not involve partition $\mathbf{P_i}$.)

Let us now define $\mathbf{R_j}$ as the desired transaction throughput (transactions per second) for transactions of type $\mathbf{T_j}$ and $\mathbf{R}$ as overall transaction throughput. $\mathbf{R} = \Sigma_{j=1..m} (\mathbf{R_j})$.

Given $\mathbf{G_{ij}}$ and $\mathbf{R_j}$, we can now calculate $\mathbf{L_i}$, the log generation rate (bytes per second) for the Log Generator associated with database partition $\mathbf{P_i}$. $\mathbf{L_i} = \Sigma_{j=1..m} (\mathbf{R_j} * \mathbf{G_{ij}})$.

Given the size (in bytes) of a commit record (call it $\mathbf{c}$), we can now calculate the overall log generation rate $\mathbf{L}$ for all Log Generators including the Commit Coordinator.

$$\mathbf{L} = \Sigma_{i=1..n} (\mathbf{L_i}) + \mathbf{R} * \mathbf{c}.$$

Let us now consider how much log data a Log Manager can accept. We define $\mathbf{L_{max}}$ as the maximum amount of log data a Log Manager can accept for a given transaction profile (i.e. mix of transaction types) if it doesn't encounter any interference from an Archiver. The value of $\mathbf{L_{max}}$ depends on a variety of factors including buffering algorithms used by the Log Manager and the performance characteristics of the underlying hardware.[7]

$\mathbf{L_{max}}$ also depends on the transaction profile. More specifically, $\mathbf{L_{max}}$ depends on amount of log data generated per transaction ($\mathbf{L/R}$) for a given transaction profile. The "Write Ahead Log Protocol" (see [GRAY78] section 5.8.3.2 and [BERN87] pages 177ff.) requires the Log Manager to force commit records to disk, so a large amount of log data generated per transaction results in a higher $\mathbf{L_{max}}$ because the log data has proportionally

---

[6] Transaction types here correspond roughly to the notion of transaction types in the TPC-C Specification [TPCC92].

[7] One can experimentally observe the value of $\mathbf{L_{max}}$ for a given transaction profile by increasing the number of Log Generators associated with a given Log Manager until the log data generation rate reaches a plateau.

fewer commit records and thus requires proportionally fewer forced writes. In short, with small amounts of log data per transaction, the Commit Coordinator (rather than the Log Manager) can become the limiting factor for the value of $\mathbf{L_{max}}$.

Let us also define γ $(0 < \gamma < 1)$ [8] as the degradation factor immediately after a log file rollover due to contention between the Log Manager and the Archiver. We can now calculate $\gamma * \mathbf{L_{max}}$ as the maximum amount of log data a Log Manager can accept immediately after a rollover has begun. Since $0 < \gamma < 1$, we have $\gamma * \mathbf{L_{max}} < \mathbf{L_{max}}$.

A comparison of the overall log generation rate, **L,** with $\gamma * \mathbf{L_{max}}$ and $\mathbf{L_{max}}$ gives three possible cases listed in Table 1 below:

Case	Partitioning Strategy
1) $\mathbf{L} < \gamma * \mathbf{L_{max}}$	No log partitioning necessary
2) $\gamma * \mathbf{L_{max}} < \mathbf{L} < \mathbf{L_{max}}$	Horizontal log partitioning
3) $\mathbf{L_{max}} < \mathbf{L}$	Vertical log partitioning

Table 1

Case 1 indicates no need for either vertical or horizontal log partitioning because the maximum amount of log data a Log Manager can accept always exceeds the overall log generation rate.

Case 2 indicates a need for horizontal partitioning because only during the period of degradation following a rollover does the overall log generation rate exceed the maximum amount of log data a Log Manager can accept.

Case 3 indicates a need for vertical partitioning because the overall log generation rate always exceeds the maximum amount of log data a single Log Manager can accept.

The use of vertical log partitioning also requires deciding how many Log Managers to configure and assigning Log Generators to those Log Managers. If all Log Generators generate roughly the same amount of log data, a system administrator can simply calculate

---

[8] As with $\mathbf{L_{max}}$, γ depends on a variety of factors, but one can experimentally observe its value by comparing log data generation rates before and immediately after a rollover for a particular transaction profile (i.e. mix of transaction types).

the number of Log Managers as $\mathbf{L} / \mathbf{L_{max}}$ rounded upwards to the next whole number and assign an equal number of Log Generators to each Log Manager.[9]

If only a small number of Log Generators generate the bulk of the log data, a system administrator needs to examine each individual $\mathbf{L_i}$ and assign Log Generators to Log Managers making sure that the sum of all $\mathbf{L_i}$'s for Log Generators associated with a given Log Manager does not exceed $\mathbf{L_{max}}$. In the worst case, the largest $\mathbf{L_i}$ value alone will exceed $\mathbf{L_{max}}$. When this happens, the system administrator has no other recourse except to reorganize the database partitions to redistribute the generated log records more equally among the various Log Generators.

6. Experiment Results for Vertical Log Partitioning

We conducted a set of experiments to asses the impact of vertical log partitioning on log throughput and transaction response time. The experiments ran on a 16 CPU Tandem NonStop Himalaya K20000 running the D40 version of the transaction management and database software. The system had 64 data disks spread evenly over all CPUs.

Application processes running in all CPUs generated transactions which inserted rows into a large SQL database partitioned over all available disks. The processes generated transactions as fast as possible (i.e., with zero think time) and spread the inserts evenly over all the disk partitions.

Each experiment ran using one of three transaction types identified as $\mathbf{T_1}$, $\mathbf{T_2}$ and $\mathbf{T_3}$ respectively. Transactions of type $\mathbf{T_1}$ did 10 inserts and generated 1.8 Kilobytes of log data; transactions of type $\mathbf{T_2}$ did 100 inserts and generated 14 Kilobytes of log data; transactions of type $\mathbf{T_3}$ did 200 inserts and generated 27.6 Kilobytes of log data. By using exactly one type of transaction in each experiment, we could carefully control the amount of log data generated per transaction. By increasing the log data generation rate per transaction in each successive experiment, we sought to find an upper bound for $\mathbf{L_{max}}$. We also wanted to quantify the benefits (e.g. increase in aggregate log throughput) and possible costs (e.g. transaction CPU cost and response time) of adding a second Log Manager.

Tables 2 and 3 below show observed values for variables presented in the analytical model of the previous section. Table 2 shows the results of experiments with a single Log Manager; table 3 shows the results of experiments with two Log Managers.

---

[9] If the resulting number of Log Managers does not evenly divide the number of Log Generators, the system administrator needs to add an additional Log Manager.

Each experiment involved single type of transaction, so for a given transaction type $\mathbf{T_j}$, $\mathbf{G_{ik} = 0}$ and $\mathbf{R_k = 0}$ for $\mathbf{k <> j}$. Each experiment spread its inserts evenly over all disks, so $\mathbf{G_{i1} = G_{i2} = . . . = G_{i64}}$ and $\mathbf{L_1 = L_2 = . . . = L_{64}}$.

Single Log Manager	**Experiment with Transactions of type T_1**	**Experiment with Transactions of type T_2**	**Experiment with Transactions of type T_3**
G_{i1} i = 1.. 64 (bytes/transaction)	28.73	0	0
G_{i2} i = 1.. 64 (bytes/transaction)	0	223.04	0
G_{i3} i = 1.. 64 (bytes/transaction)	0	0	441.25
R_1 (transactions/second)	332.90	0	0
R_2 (transactions/second)	0	67.08	0
R_3 (transactions/second)	0	0	34.37
R (transactions/second)	332.90	67.08	34.37
L_i i = 1.. 64 (bytes/second)	9,563	14,962	15,166
c (bytes)	50	50	50
L (bytes/second)	628,703	960,892	972,327

Table 2

The last row of table 2 shows aggregate log throughput rising with each successive experiment and finally reaching a plateau at approximately 972,000 bytes/second.

Two Log Managers	**Experiment with Transactions of type T_1**	**Experiment with Transactions of type T_2**	**Experiment with Transactions of type T_3**
G_{i1} i = 1.. 64 (bytes/transaction)	29.36	0	0
G_{i2} i = 1.. 64 (bytes/transaction)	0	224.98	0
G_{i3} i = 1.. 64 (bytes/transaction)	0	0	434.45
R_1 (transactions/second)	412.34	0	0
R_2 (transactions/second)	0	117.22	0
R_3 (transactions/second)	0	0	68.31
R (transactions/second)	412.34	117.22	68.31
L_i i = 1.. 64 (bytes/second)	12,107	26,372	29,677
c (bytes)	50	50	50
L (bytes/second)	795,442	1,693,652	1,902,774

Table 3

The last row of table 3 shows that a second Log Manager increased aggregate log throughput for each type of transaction. In the case of $\mathbf{T_3}$, a second Log Manager actually doubled the aggregate log data throughput. We believe aggregate throughput would also

have doubled for $\mathbf{T_2}$, but we didn't have enough Log Generators to reach a sufficiently large transaction load. The experiment running transactions of type $\mathbf{T_1}$ resulted in a (comparatively) modest increase in aggregate log throughput because the low rate of log data generated per transaction (1.8 Kilobytes) made the Commit Coordinator (not the Log Manager) the limiting factor.

With two Log Managers, average CPU utilization exceeded 90%, so the system had nearly reached its capacity to generate log data. Therefore, we did not perform any experiments with three Log Managers.

The chart in Figure 6 below summarizes the data from the two previous tables by showing aggregate log throughput as a function of the amount of log data generated per transaction. The lower line (with data points as diamonds) shows aggregate log throughput for a single Log Manager (last row of Table 2) while the upper line (with data points as squares) shows aggregate log throughput for two Log Managers (last row of Table 3).

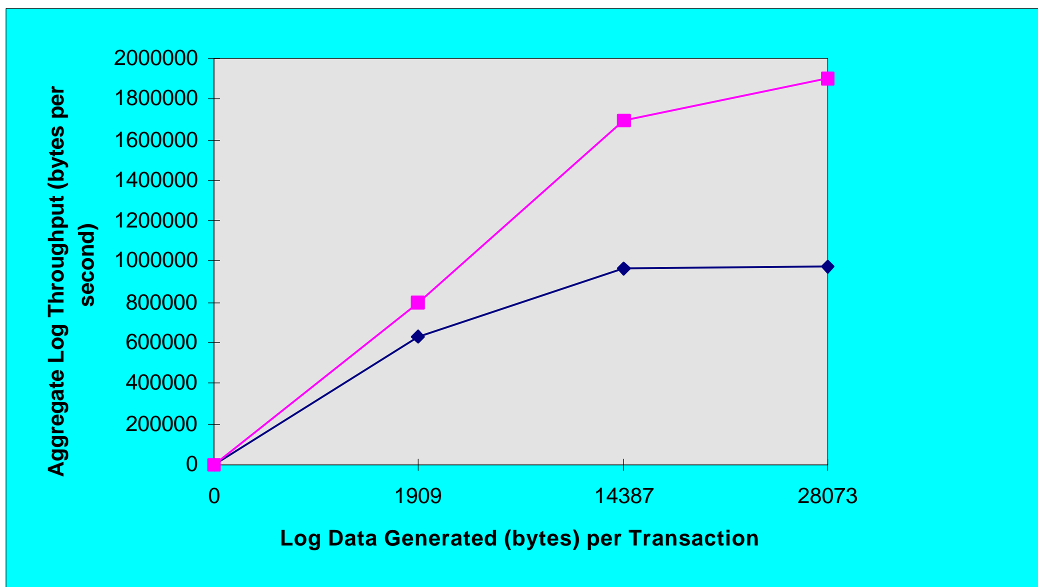

Figure 6

Table 4 below shows transaction CPU cost and response time for each type of transaction with both one and two Log Managers.

Transaction Type	CPU Cost/Tx with One Log Manager (ms.)	CPU Cost/Tx with Two Log Managers (ms.)	Response Time/Tx with One Log Manager (sec.)	Response Time/Tx with Two Log Managers (sec.)
T_1	22.26	22.64	0.37	0.29
T_2	115.49	116.28	1.84	.95
T_3	220.63	221.5	3.55	1.66

Table 4

The addition of a second Log Manager resulted in a very small increase in CPU cost (due to the extra message overhead associated with vertical log partitioning). At the same time, it significantly reduced response time, especially in the cases of transaction types T_2 and T_3 which suffered severe bottlenecks in the single Log Manager case. With high log throughput rates, long queues built up at the single Log Manager resulting in high response times. The addition of a second Log Manager removed the bottleneck and greatly reduced queuing at both Log Managers, thus lowering response times.

7. Summary

In this paper, we presented two complimentary techniques for improving Log Manager throughput for transaction processing systems which run on a shared-nothing, multiprocessor platform. Vertical log partitioning lessens contention among the various Log Generators. Horizontal log partitioning removes contention between the Log Manager and Archiver.

We also presented an analytical model for determining when to use the log partitioning techniques, and we illustrated the aspects of the model pertaining to vertical log partitioning with results from actual experiments.

8. Future Research

In the future we hope to conduct experiments which measure γ, the rollover degradation factor, and assess the impact of horizontal log partitioning on log throughput and transaction response time. We also hope to assess both vertical and horizontal log partitioning for a wider variety of transaction profiles. Finally, we hope to evaluate how the use of RAID [PATT88] technology compares with vertical and horizontal partitioning.

9. Acknowledgments

Andrea Borr and Glenn Peterson invented and implemented vertical partitioning in the early 1980's. In the late 1980's, Bert van der Linden, Gary Smith and Sunil Sharma made significant performance improvements. Mike Skarpelos and Jim Carley rewrote the implementation in the early 1990's as part of a larger development effort.

Mike Skarpelos, Bert van der Linden, Jim Carley, Jim Lyon and Matt McCline collectively invented horizontal partitioning in the early 1990's. Mike Skarpelos and Jim Carley did the actual implementation soon thereafter.

The authors wish to thank Robert Hou and David Eicher for their helpful comments on earlier drafts of this paper.

10. References

[BERN87] P. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[BORR81] A. Borr, "Transaction Monitoring in Encompass [TM]: Reliable Distributed Transaction Processing." Proceedings of the Seventh International Conference on Very Large Databases (1981), 155-165.

[BORR84] A. Borr, "Robustness to Crash in a Distributed Database: A Non Shared-Memory Multi-Processor Approach." Proceedings of the Tenth International Conference on Very Large Databases (1984), 445-453.

[DEWI92] D. Dewitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems." Communications of the ACM, June 1992, Vol. 35. No. 6.

[HELL87] P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett and A. Reuter, "Group Commit Timers and High-Volume Transaction Systems." Tandem Technical Report 88.1, March 1987, Part Number 12522.

[GRAY78] J. Gray, "Notes on Database Operating Systems", in *Operating Systems--An Advanced Course*. Springer-Verlag, 1978.

[GRAY93] J. Gray and A. Reuter *Transaction Processing: Concepts and Techniques*, Morgan Kauffman, 1993.

[PATT88] D. A. Patterson, G. Gibson and R. Katz. "A Case for Redundant Arrays of Inexpensive Disks (RAID)" ACM SIGMOD (1988), 109-116.

[TPCC92] The Transaction Processing Performance Council. *TPC Benchmark C Specification.* (1992).

[XOPE91] X/Open DTP. *X/Open Common Application Environment; Distributed Transaction Processing: The XA Specification*. X/Open Ltd. (1991).