Designing an Optimized Transaction Commit Protocol

By Peter M. Spiro, Ashok M. Joshi, and T. K. Rengarajan

Abstract

Digital's database products, VAX Rdb/VMS and VAX DBMS, share the same database kernel called KODA. KODA uses a grouping mechanism to commit many concurrent transactions together. This feature enables high transaction rates in a transaction processing (TP) environment. Since group commit processing affects the maximum throughput of the transaction processing system, the KODA group designed and implemented several grouping algorithms and studied their performance characteristics. Preliminary results indicate that it is possible to achieve up to a 66 percent improvement in transaction throughput by using more efficient grouping designs.

Introduction

Digital has two general-purpose database products, Rdb/VMS software, which supports the relational kernel called KODA. In addition to other database services, KODA provides the transaction capabilities and commit processing for these two products.

In this paper, we address some of the issues relevant to efficient commit processing. We begin by explaining the importance of commit processing in achieving high transaction throughput. Next, we describe in detail the current algorithm for group commit used in KODA. We then describe and contrast several new designs for performing a group commit. Following these discussions, we present our experimental results. And, finally, we discuss the possible direction of future work and some conclusions. No attempt is made to present formal analysis or exhaustive empirical results for commit processing; rather, the focus is on an intuitive understanding of the concepts and trade-offs, along with some

data model, and VAX DBMS software, which supports the CODASYL (Conference on Data Systems Languages) data model. Both products layer on top of a database

empirical results that support our conclusions.

Designing an Optimized Transaction Commit Protocol

Commit Processing

To follow a discussion of commit processing, two basic terms must first be understood. We begin this section by defining a transaction and the "moment of commit."

A transaction is the execution of one or more statements that access data managed by a database system. Generally, database management systems guarantee that the effects of a transaction are atomic, that is, either all updates performed within the context of the transaction are recorded in the database, or no updates are reflected in the database.

The point at which a transaction's effects become durable is known as the "moment of commit." This concept is important because it allows database recovery to proceed in a predictable manner after a transaction failure. If a transaction terminates abnormally before it reaches the moment of commit, then it aborts. As a result, the database system performs transaction recovery, which removes all effects of the transaction. However, if the transaction has passed the moment of commit, recovery processing ensures that all changes

For the purpose of analysis, it is useful to divide a transaction processed by KODA into four phases: the transaction start phase, the data manipulation phase, the logging phase, and the commit processing phase. Figure 1 illustrates the phases of a transaction in time sequence. The first three phases are collectively referred to as "the average transaction's CPU cost (excluding the cost of commit)" and the last phase (commit) as "the cost of writing a group commit buffer." [1]

made by the transaction are
permanent.

Transaction Profile

The transaction start phase involves acquiring a transaction identifier and setting up control data structures. This phase usually incurs a fixed overhead.

The data manipulation phase involves executing the actions dictated by an application program. Obviously, the time spent in this phase and the amount of processing required depend on the nature of the application.

At some point a request is made to complete the transaction. Accordingly in KODA, the transaction enters the logging phase which involves updating the database with the changes and writing the undo/redo information to disk. The amount of work done in the logging phase is usually small and constant (less than one I/O) for transaction processing. Finally, the transaction enters the commit processing phase. In KODA, this phase involves writing commit information to disk, thereby ensuring that the transaction's effects are recorded in the database and now visible to other users.

For some transactions, the have to fetch and modify every employee/salary record in the company database. The commit processing phase, in this example, represents 0.2 percent of the transaction duration. Thus, for this class of transaction, commit processing is a small fraction of the overall cost. Figure 2 illustrates the profile of a transaction modifying 500 records.

In contrast, for transaction processing applications such as hotel reservation systems, banking applications, stock market transactions, or the telephone system, the data manipulation phase is usually short (requiring few I/Os). Instead, the logging and commit phases comprise the bulk of the work and must be optimized to allow high transaction throughput. The transaction profile for a transaction modifying one record is shown in Figure 3. Note that the commit processing phase represents 36 percent of the transaction duration, in this example.

Group Commit

Generally, database systems must force write information to

data manipulation phase is very expensive, possibly requiring a large number of I/Os and a great deal of CPU time. For example, if 500 employees in a company were to get a 10 percent salary increase, a transaction would

disk in order to commit transactions. In the event of a failure, this operation permits recovery processing to determine which failed transactions were active at the time of their termination and which ones had reached

their moment of commit. This information is often in the form of lists of transaction identifiers, called commit lists.

Many database systems perform an optimized version of commit processing where commit information for a group of transactions is written to disk in one I/O operation, thereby, amortizing the cost of the I/O across multiple transactions.

So, rather than having each transaction write its own commit list to disk, one transaction writes to disk a commit list containing the commit information for a number of other transactions. This technique is referred to in the literature as "group commit." [2]

Group commit processing is essential for achieving high throughput. If every transaction that reached

the commit stage had to actually perform an I/O to the same disk to flush its own commit information, the throughput of the database system would be limited to the I/O rate of the disk. A magnetic disk is capable of performing 30 I/O operations per second. Consequently, in the absence of group commit, the throughput of the system is limited to

There are several variations of the basic algorithms for grouping multiple commit lists into a single I/O. The specific group commit algorithm chosen can significantly influence the throughput and response times of transaction processing. One study reports throughput gains of as much as 25 percent by selecting an optimal group commit algorithm. [1]

At high transaction throughput (hundreds of transactions per second), efficient commit processing provides a significant performance advantage. There is little information in the database literature about the efficiency of different methods of performing a group commit. Therefore, we analyzed several grouping designs and evaluated their performance benefits.

Factors Affecting Group Commit

Before proceeding to a description of the experiments, it is useful to have a better understanding of the factors affecting the behavior of the group commit mechanism. This section discusses the group size, the use of timers to stall transactions, and the relationship between these

30 transactions per second (TPS). Group commit is essential to breaking this performance barrier.

two factors.

Group Size. An important factor affecting group commit is the number of transactions that participate in the group commit. There must be

several transactions in the group in order to benefit from I/O amortization. At the same time, transactions should not be required to wait too long for the group to build up to a large size, as this factor would adversely affect throughput.

It is interesting to note that the incremental advantage of adding one more transaction to a group decreases as the group size increases. The incremental savings is equal to 1/(G x (G + 1), where G is the initial group size. For example, if the group consists of 2 transactions, each of them does one-half a write. If the group size increases to 3, the incremental savings in writes will be (1/2 - 1/3), or 1/6 per transaction. If we do the same calculation for a group size incremented from 10 to 11, the savings will be (1/10 - 1/11), or 1/110 of a write per transaction.

In general, if G represents the group size, and I represents the number of I/Os per second for the disk, the maximum transaction commit rate is I x G TPS. For example, if the group size is 45 and the rate is 30 I/Os per second to disk, the transaction throughput of the transaction processing system.

Use of Timers to Stall Transactions. One of the mechanisms to increase the size of the commit group is the use of timers. [1, 2] Timers are used to stall the transactions for a short period of time (on the order of tens of milliseconds) during commit processing. During the stall, more transactions enter the commit processing phase and so the group size becomes larger. The stalls provided by the timers have the advantage of increasing the group size, and the disadvantage of increasing the response time.

Trade-offs. This section discusses the trade-offs between the size of the group and the use of timers to stall transactions. Consider a system where there are 50 active database programs, each repeatedly processing transactions against a database. Assume that on average each transaction takes between 0.4 and 0.5 seconds. Thus, at peak performance, the database system can commit approximately 100 transactions every second, each program actually completing two transactions

maximum transaction commit rate is 30 x 45, or 1350 TPS. Note that a grouping of only 10 will restrict the maximum TPS to 300 TPS, regardless of how powerful the computer is. Therefore, the group size directly affects the maximum in the one-second time interval. Also, assume that the transactions arrive at the commit point in a steady stream at different times.

If transaction commit is stalled for 0.2 seconds to allow the commit group to build up, the group then consists of about 20 transactions (0.2 seconds x 100 TPS). In this case, each transaction only incurs a small delay at commit time, averaging 0.10 seconds, and the database system should be able to approach its peak throughput of 100 TPS. However, if the mechanism delays commit processing for one second, an entirely different behavior sequence occurs. Since the transactions complete in approximately 0.5 seconds, they accumulate at the commit stall and are forced to wait until the one-second stall completes. The group size then consists of 50 transactions, thereby maximizing the I/O amortization. However, throughput is also limited to 50 TPS, since a group commit is occurring only once per second.

Thus, it is necessary to balance response time and the size of the commit group. The longer the stall, the larger the group size; the larger the group size, the better the I/O amortization that is achieved. However, if the stall time is too long, it is possible to limit transaction throughput because of wasted CPU

The concept of using commit timers is discussed in great detail by Reuter. [1] However, there are significant differences between his group commit scheme and our scheme. These differences prompted the work we present in this paper.

In Reuter's scheme, the timer expiration triggers the group commit for everyone. In our scheme, no single process is in charge of commit processing based on a timer. Our commit processing is performed by one of the processes desiring to write a commit record. Our designs involve coordination between the processes in order to elect the group committer (a process).

Reuter's analysis to determine the optimum value of the timer based on system load assumes that the total transaction duration, the time taken for commit processing, and the time taken for performing the other phases are the same for all transactions. In contrast, we do not make that assumption. Our designs strive to adapt to the execution of many different transaction types under different system loads. Because of the complexity introduced by allowing variations in

cycles.

Motivation for Our Work

transaction classes, we do not attempt to calculate the optimal timer values as does Reuter.

Cooperative Commit Processing

In this section, we present the stages in performing the group commit with cooperating processes, and we describe, in detail, the grouping design currently used in KODA, the Commit-Lock Design.

Group Committer

Assume that a number of transactions have completed all data manipulation and logging activity and are ready to execute the commit processing phase. To group the commit requests, the following steps must be performed in KODA:

1. Each transaction must make its commit information available to the group committer.
2. One of the processes must be selected as the "group committer."
3. The other members of the group need to be informed that their commit work will be completed by the group committer. These processes must wait until the commit information is written to disk by the group committer.
4. Once the group committer has written the commit information to stable

The Commit-Lock Design uses a VMS lock to generate groups of committing transactions; the lock is also used to choose the group committer.

Once a process completes all its updates and wants to commit its transaction, the procedure is as follows. Each transaction must first declare its intent to join a group commit. In KODA, each process uses the interlocked queue instructions of the VAX system running VMS software to enqueue a block of commit information, known as a commit packet, onto a globally accessible commit queue. The commit queue and the commit packets are located in a shared, writeable global section. Each process then issues a lock request for the commit lock. At this point, a number of other processes are assumed to be going through the same sequence; that is, they are posting their commit packets and making lock requests for the commit lock. One of these processes is granted the commit lock. For the time being, assume the process that currently acquires the lock acts as the group committer.

storage, it must inform the other members that commit processing is completed.

Commit-Lock Design

The group committer, first, counts the number of entries on the commit queue, providing the number of transactions that will be part of the group commit. Because of the VAX interlocked queue instructions, scanning

to obtain a count and concurrent queue operations by other processes can proceed simultaneously. The group committer uses the information in each commit packet to format the commit block which will be written to disk. In KODA, the commit block is used as a commit list, recording which transactions have committed and which ones are active. In order to commit for a transaction, the group committer must mark each current transaction as completed. In addition, as an optimization, the group committer assigns a new transaction identifier for each process's next transaction. Figure 4 illustrates a commit block ready to be flushed to disk.

Once the commit block is modified, the group committer writes it to disk in one atomic I/O. This is the moment of commit for all transactions in the group. Thus, all transactions that were active and took part in this group commit are now stably marked as committed. In addition, as explained above, these transactions now have new transaction identifiers. Next, the group committer sets a commit flag in each commit packet for all recently committed transactions, removes all commit packets from the commit queue, and, finally, releases the commit lock. Figure 5 illustrates a committed group with new transaction identifiers and with commit flags set.

At this point, the remaining processes that were part of the group commit are, in turn, granted the commit lock. Because their commit flags are already set, these processes realize they do not need to perform a commit and, thus, release the commit lock and proceed to the next transaction. After all these committed processes release the commit lock, a process that did not take part in the group commit acquires the lock, notices it has not been committed, and, therefore, initiates the next group commit.

There are several interesting points about using the VMS lock as the grouping mechanism. Even though all the transactions are effectively committed after the commit block I/O has completed, the transactions are still forced to proceed serially; that is, each process is granted the lock, notices that it is committed, and then releases the lock. So there is a serial procession of lock enqueues /dequeues before the next group can start.

This serial procession can be made more concurrent by, first, requesting the lock in a shared mode, hoping that all processes lock request is mastered on a different node in a VAXcluster system, the lock enqueue/dequeues are very expensive.

Also, there is no explicit stall time built into the algorithm. The latency associated with the lock enqueue/dequeue requests allows the commit queue to build up. This stall is entirely dependent on the contention for the lock, which in turn depends on the throughput.

Group Commit Mechanisms- Our New Designs

To improve on the transaction throughput provided by the Commit-Lock Design, we developed three different grouping designs, and we compared their performances at high throughput. Note that the basic paradigm of group commit for all these designs is described in the Group Committer section. Our designs are as follows:

Commit-Stall Design

In the Commit-Stall Design, the use of the commit lock as the grouping mechanism is eliminated. Instead, a process inserts its commit packet onto the commit queue and, then, checks to see if it is the first process on the queue. If

committed are granted the lock in unison. However, in practice, some processes that are granted the lock are not committed. These processes must then request the lock in an exclusive mode. If this so, the process acts as the group committer. If not, the process schedules its own wake-up call, then sleeps. Upon waking, the process checks to see if it has been committed. If so, the process proceeds to its

next transaction. If not, the process again checks to see if it is first on the commit queue. The algorithm then repeats, as described above.

This method attempts to eliminate the serial wake-up behavior displayed by using the commit lock. Also, the duration for which each process stalls can be varied per transaction to allow explicit control of the group size. Note that if the stall time is too small, a process may wake up and stall many times before it is committed.

Willing-to-Wait Design

As we have seen before, a delay in the commit sequence is a convenient means of converting a response time advantage into a throughput gain. If we increase the stall time, the transaction duration increases, which is undesirable. At the same time, the grouping size for group commit increases, which is desirable. The challenge is to determine the optimal stall time. Reuter presented an analytical way of determining the optimal stall time for a system with transactions of the same type. [1]

Ideally, we would like to automatically, because the database management system cannot judge which is more important to the user in a general customer situation-the transaction response time or the throughput.

The Willing-to-Wait Design provides a user parameter called WTW time. This parameter represents the amount of time the user is willing to wait for the transaction to complete, given this wait will benefit the complete system by increasing throughput. WTW time may be specified by the user for each transaction. Given such a user specification, it is easy to calculate the commit stall to increase the group size. This stall equals the WTW time minus the time taken by the transaction thus far, but only if the transaction has not already exceeded the WTW time. For example, if a transaction comes to commit processing in 0.5 second and the WTW time is 2.0 seconds, the stall time is then 1.5 seconds. In addition, we can make a further improvement by reducing the stall time by the amount of time needed for group commit processing. This delta time is constant, on the order of 50 milliseconds (one I/O plus some computation).

devise a flexible scheme that makes the trade-off we have just described in real time and determines the optimum commit stall time dynamically. However, we cannot determine the optimum stall time

The WTW parameter gives the user control over how much of the response time advantage (if any) may be used by the system to improve transaction throughput. The choice of an abnormally high value

of WTW by one process only affects its own transaction response time; it does not have any adverse effect on the total throughput of the system. A low value of WTW would cause small commit groups, which in turn would limit the throughput. However, this can be avoided by administrative controls on the database that specify a minimum WTW time.

Hiber Design

The Hiber Design is similar to the Commit-Stall Design, but, instead of each process scheduling its own wake-up call, the group committer wakes up all processes in the committed group. In addition, the group committer must wake up the process that will be the next group committer.

Note, this design exhibits a serial wake-up behavior like the Commit-Lock Design, however, the mechanism is less costly than the VMS lock used by the Commit-Lock Design. In the Hiber Design, if a process is not the group committer, it simply sleeps; it does not schedule its own wake-up call. Therefore, each process is guaranteed actually schedule the wake-up call after a delay. Such a delay allows the next group size to become larger.

Experiments

We implemented and tested the Commit-Lock, the Commit-Stall, and the Willing-to-Wait designs in KODA. The objectives of our experiments were

- To find out which design would yield the maximum throughput under response time constraints
- To understand the performance characteristics of the designs

In the following sections, we present the details of our experiments, the results we obtained, and some observations.

Details of the Experiments

The hardware used for all of the following tests was a VAX 6340 with four processors, each rated at 3.6 VAX units of performance (VUP). The total possible CPU utilization was 400 percent and the total processing power of the

to sleep and wake up at most once per commit, in contrast to the Commit-Stall Design. Another interesting characteristic of the Hiber Design is that the group committer can choose to either wake up the next group committer immediately, or it can

computer was 14.4 VUPs. As the commit processing becomes more significant in a transaction (in relation to the other phases), the impact of the grouping mechanism on the transaction throughput increases. Therefore, in order to accentuate the performance differences

between the various designs, we performed our experiments using a transaction that involved no database activity except to follow the commit sequence. So, for all practical purposes, the TPS data presented in this paper can be interpreted as "commit sequences per second." Also, note that our system imposed an upper limit of 50 on the grouping size.

Results

Using the Commit-Lock Design, transaction processing bottlenecked at 300 TPS. Performance greatly improved with the Commit-Stall Design; the maximum throughput was 464 TPS. The Willing-to-Wait Design provided the highest throughput, 500 TPS. Using this last design, it was possible to achieve up to a 66 percent improvement over the less-efficient Commit-Lock Design. Although both timer schemes, i.e., the Commit-Stall and Willing-to-Wait designs, needed tuning to set the parameters and the Commit-Lock Design did not, we observed that the maximum throughput obtained using timers is much better than that obtained with the lock. These results were similar to those of Reuter.

by the formula: number of servers x 1000/WTW = maximum TPS. For example, our maximum TPS for the WTW design was obtained with 50 servers and 90 milliseconds WTW time. Using the formula, 50 x 1000/90 = 555. The actual TPS achieved was 500, which is 90 percent of the maximum TPS. This ratio is also a measure of the effectiveness of the experiment.

During our experiments, the maximum group size observed was 45 (with the Willing-to-Wait Design). This is close to the system-imposed limit of 50 and, so, we may be able to get better grouping with higher limits on the size of the group.

Observations

In the Commit-Stall and the Willing-to-Wait designs, given a constant stall, if the number of servers is increased, the TPS increases and then decreases. The rate of decrease is slower than the rate of increase. The TPS decrease is due to CPU overloading. The TPS increase is due to more servers trying to execute transactions and better CPU utilization. Figure 6 illustrates how TPS varies with the number of servers, given a constant stall WTW

For our Willing-to-Wait Design, the minimum transaction duration is the WTW time. Therefore, the maximum TPS, the number of servers, and the WTW stall time, measured in milliseconds, are related

time.

Again, in the stalling designs, for a constant number of servers, if the stall is increased, the TPS increases and then decreases. The TPS increase is due to better grouping

and the decrease is due to CPU underutilization. Figures 7 and 8 show the effects on TPS when you vary the commit-stall time or the WTW time, while keeping the number of servers constant.

To maximize TPS with the Commit-Stall Design, the following "mountain-climbing" algorithm was useful. This algorithm is based on the previous two observations. Start with a reasonable value of the stall and the number of servers, such that the CPU is underutilized. Then increase the number of servers. CPU utilization and the TPS increase. Continue until the CPU is overloaded; then, increase the stall time. CPU utilization decreases, but the TPS increases due to the larger group size.

This algorithm demonstrates that increasing the number of servers and the stall by small amounts at a time increases the TPS, but only up to a limit. After this point, the TPS drops. When close to the limit, the two factors may be varied alternately in order to find the true maximum. Table 1 shows the performance measurements of the Commit-Stall Design. Comments are included in

the table to highlight the performance behavior the data supports.

Table 1

Commit-Stall Design Performance Data

Number of Servers	Commit Stall (Milliseconds)	CPU Utilization (Percent)*	TPS	Comments
50	20	360	425	Starting point
55	20	375	454	Increased number of servers, therefore, higher TPS
60	20	378	457	Increased number of servers, therefore, CPU saturated
60	30	340	461	Increased stall, therefore, CPU less utilized
65	30	350	464	Increased number of servers, maximum TPS
70	30	360	456	"Over-the-hill" situation, same strategy of further increasing the number of servers does not increase TPS
70	40	330	451	No benefit from increasing number of servers and stall
65	40	329	448	No benefit from just increasing stall

*Four processors were used in the experiments. Thus, the total possible CPU utilization is 400 percent.

The same mountain-climbing algorithm is modified slightly to obtain the maximum TPS with the Willing-to-Wait TPS with this design is inversely proportional to the WTW time, while CPU is not fully utilized. The first four rows of

Design. The performance measuremenmts of this design are presented in Table 2. As we have seen before, the maximum Table 2 illustrate this behavior. The rest of the table follows the same pattern as Table 1.

Table 2

Willing-to-Wait Performance Data

Table 2 (Cont.)

Willing-to-Wait Performance Data

Number of Servers	Willing-to-Wait Stall (Milliseconds)	CPU Utilization (Percent)*	TPS	Comments
45	100	285	426	Starting point, CPU not saturated
45	90	295	466	Decreased stall to load CPU, CPU still not saturated
45	80	344	498	Decreased stall again
45	70	363	471	Further decreased stall, CPU almost saturated
50	80	372	485	Increased number of servers, CPU more saturated
50	90	340	500	Increased stall to lower CPU usage, maximum TPS
55	90	349	465	"Over-the-hill"situation, same strategy of further increasing number of servers does not increase TPS
50	100	324	468	No benefit from just increasing stall

*Four processors were used in the experiments. Thus, the total possible CPU utilization is 400 percent.

The Willing-to-Wait Design performs slightly better than the Commit-Stall Design by adjusting to the we expect the Willing-to-Wait Design to perform much better than the Commit-Stall Design.

variations in the speed at which different servers arrive at the commit point. Such variations are compensated for by the variable stalls in the Willing-to-Wait Design. Therefore, if the variation is high and the commit sequence is a significant portion of the transaction,

Future Work

There is scope for more interesting work to further optimize commit processing in the KODA database kernel. First, we would like to perform experiments on the Hiber Design and compare it to the other

designs. Next, we would like to explore ways of combining the Hiber Design with either of the two timer designs, Commit-Stall or Willing-to-Wait. This may be the best design of all the above, with a good mixture of automatic stall, low overhead, and explicit control over the total stall time. In addition, we would like to investigate the use of timers to ease system management. For example, a system administrator may increase the stalls for all transactions on the system in order to ease CPU contention, thereby increasing the overall effectiveness of the system.

Conclusions

We have presented the concept of group commit processing as well as a general analysis of various options available, some trade-offs involved, and some performance results indicating areas for possible improvement. It is clear that the choice of the algorithm can significantly influence performance at high transaction throughput. We are optimistic that with some further investigation an optimal commit sequence can be incorporated into considerable gains in transaction processing performance.

Acknowledgments

We wish to acknowledge the help provided by Rabah Mediouni in performing the experiments discussed in this paper. We would like to thank Phil Bernstein and Dave Lomet for their careful reviews of this paper. Also, we want to thank the other KODA group members for their contributions during informal discussions. Finally, we would like to acknowledge the efforts of Steve Klein who designed the original KODA group commit mechanism.

References

1. P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett, and A. Reuter, "Group Commit Timers and High Volume Transaction Processing Systems," High Performance Transaction Systems, Proceedings of the 2nd International Workshop (September 1987).
2. Gawlick and D. Kinkade, "Varieties of Concurrency Control in IMS/VS Fast Path," Database Engineering (June 1985).

Rdb/VMS and VAX DBMS with