Transaction Isolation and Lazy Commit

Vishal Kathuria, Robin Dhamankar, Hanuma Kodavalla
Microsoft Corporation,
One Microsoft Way, Redmond WA 98052
{vishalk, robindh, hanumak}@microsoft.com

Abstract

In order to guarantee durability of transactions (D in the ACID properties) database systems issue a synchronous log write on transaction commit. However, in many scenarios such as queue processing and Personal Information Management systems, durability on commit is not required as the transactions rolled back due to a crash can be reprocessed using the application state. Avoiding such synchronous I/O improves system throughput and response time. Also, when the database is running on a mobile device or a laptop, reduced I/O consumes less power and increases battery life – a significant consideration for mobile users. Transactions that do not require durability can be "lazily committed", i.e. the log is not written synchronously upon commit. If another transaction reads such lazily committed data and performs an independently committed external action based on it, then an inconsistency may result if the effects of the lazily committed transaction are lost in a crash. We present a solution that provides the efficiency of lazily committed transactions and the flexibility for other transactions to read only durably committed data. We demonstrate that even in the presence of a large number of readers requesting durably committed data our approach reduces I/Os significantly.

1. Introduction

Historically, disk performance has not kept pace with processor capability. Therefore, software systems must minimize the number of I/Os for better performance. This has added significance in the case of Personal Information Management (PIM) systems such as email clients. Many PIM systems run on laptops, where the hard disk is a significant power consuming component and increased I/O increases power consumption by preventing the disk from spinning down. Also, PIM systems typically share resources with other applications on the same machine; therefore minimizing I/O is crucial for improving system responsiveness.

In order to guarantee durability of transactions, database systems issue a synchronous I/O to write the log to disk on transaction commit. There are many applications where durability on commit is not required because the transactions that are lost in case of a crash can be reprocessed using the application state.

For example, typical queue processing applications begin a transaction, dequeue an entry, perform a set of updates to process that entry, delete it and commit the transaction. If some of the committed transactions were lost due to a crash, after recovery the queue entries would still be available for this application to process. Other examples include bulk loading data from a file, replication of transactions and data warehousing applications.

Relaxing the durability requirement for transactions reduces I/O by not forcing the log to disk — i.e. doing a *lazy commit*. A lazy commit is identical to a normal commit except that a lazily committed transaction's log records may not yet have been durably written to the disk. In this paper, we use the term *durable commit* to refer to normal (non-lazy) commit and to emphasize its difference from *lazy commit. Non-durable data* refers to the data that has been modified by a transaction that performed a lazy commit. This is the data that might revert to its original value in the event of a crash. *Durable data* refers to data that has been modified by a transaction whose log records are known to be written to disk.

If a transaction performs an external action based on another transaction's *lazily committed* but not yet durable data, then there is a potential for inconsistency. The inconsistency is caused when the lazily committed changes are lost in a crash but the effects of the external action have committed independently. Although the performance benefits of lazy commit are desirable, the potential inconsistency is unacceptable.

In this paper we present a scheme that achieves the performance benefits of lazy commit without incurring the risk of inconsistency. Further we demonstrate that this scheme can lead to a significant reduction in I/O even in

1-4244-0803-2/07/$20.00 ©2007 IEEE

the presence of a large number of readers requesting only durably committed data.

The rest of the paper is organized as follows. In Section 2, we take an in-depth look at the application scenarios that motivated the use of lazy commit. In section 3, we review related work, prior state of art and its limitations. In section 4, we describe the overall architecture of the database system in which our scheme is implemented. In sections 5, 6 and 7, we present the algorithm, its analysis and performance results respectively.

2. Motivation

As discussed in the previous section, there are many applications where relaxing the durability property improves throughput and response time without losing data integrity. Using lazy commit for such applications provides better response time as the transactions do not wait for the log records to go to disk. Further, since the transactions commit faster, they hold locks for shorter duration. This reduces wait for other transactions, thus improving concurrency and throughput. The next few subsections describe various application scenarios that can benefit from lazy commit.

2.1 Bulk Load

Bulk load utility reads rows from a data file and inserts them into a table in batches where each batch inserts a set of rows in a single transaction. Increasing the batch size decreases the number of commits and hence the number of synchronous I/Os. The transaction batch size specifies the number of rows to be loaded before committing the transaction. If the batch size is too small, it slows down the load because of the forced log IO at the end of each commit. If the size is too large, the transaction takes too many lock resources and adversely affects concurrency. With lazy commit we can keep the batch size small limiting the number of resources held by the transaction without incurring the forced log I/O at the end of each transaction commit. In case of a system crash, upon restart, the application can query the database to determine the data row in the input file from where the processing should be resumed.

2.2 Queue Processing

Queue processing applications typically process requests from a transactional queue. Consider an application that begins a transaction, dequeues a request, then updates other tables as a part of processing the request and subsequently commits this transaction. If the dequeuing and processing of the request does not involve an external action that is outside the sphere of the executing transaction's control, then it is not necessary to durably commit the transaction. If the transaction's log records are not durably written, in case of a system crash, the transaction's results may be lost. However, the database will be consistent because all of the transaction's effects would have been undone. When the application resumes after crash recovery, it will find the request in the queue and will reprocess it in a new transaction.

2.3 Personal Information Management (PIM)

With more and more personal information created, stored and exchanged in the digital form, the need for Personal Information Management systems such as rich email clients and information search engines. has been ever increasing. These PIM systems have requirements for complex data management. They use databases systems for storage and efficient retrieval of data and metadata.
It is common for laptop and mobile devices to run these PIM applications; hence consumption of battery power is an important consideration. Spinning the disk and keeping it running drains the battery. To save power, it is preferable to spin the disk down and keep the system running off cached data. Rather than durably committing every transaction, applications can use lazy commit and force the log to disk infrequently thereby providing a chance to spin the disk down and increase battery life. Next we describe PIM applications that can benefit from lazy commit.

File Metadata Indexing: For efficient file lookup, many PIM systems store file metadata indexes in a database. Since the data in the database is duplicated from the data in the file system, updates done to store the metadata in the database can be lazy committed.

Synchronization with Servers: Many PIM systems have data that is a cache of the data from the server. An example of that is an e-mail client—as a user receives e-mails, they are synchronized to the client from the server. These email updates can be lazy committed because, in case of a crash, they can be resynchronized from the server.

2.4 Sequence number generation

Many applications use the database to generate sequence numbers for uniquely identifying rows—employee ids, invoice numbers, product keys and so on. By nature, sequence number generation is a hot spot; one solution is to generate the sequence number in an autonomous transaction that is committed independent of the parent transaction. Lazy commit can be used for committing the autonomous transaction. Since the sequence generation is logged before the rows that refer to it, a crash would never lead to a state where the database still has the rows but the sequence number rolled back.

3. Prior Art and Problem Definition

We briefly review some of the existing database systems that support functionality similar to lazy commit.

Applications could request an asynchronous commit in IMS/FastPath [1], which results in the commit record

being written to stable storage as a deferred task separate from the commit request.

In Informix [2], a database can be created with the option of buffered log, whereby transactions in that database will not force their commit log records synchronously to stable storage.

When disk-based logging is enabled, Oracle TimesTen [4] provides a configuration attribute (DurableCommits) that specifies whether the log buffer is synchronously written to disk at transaction commit or the write is deferred. When configured for non-durable commits by default, individual transactions can override the behavior if they require durability guarantee.

Oracle [5], in its 10g release, supports asynchronous and deferred commit through the COMMIT_WRITE clause. This clause can be used to specify commit behavior for an individual transaction and a default behavior can be configured for a session or the complete instance.

Lazy commit in the above systems does not provide for isolation from the lazily committed updates to those transactions that require stricter isolation semantics. Consider the following example to illustrate this problem: A transaction T1 updates row R in a database D1 and subsequently commits lazily. Another transaction T2 reads the modified contents of the row R from database D1, performs an update in database D2 based on the contents of R and subsequently commits *durably* in D2. Although a crash could undo T1's changes, T2's updates in D2 based on these changes would have committed independently.

Inconsistency results in the example above because T2 the durable reader transaction (a transaction that expects to read durably committed data) is not isolated from the changes performed by T1, a lazily committed transaction. Although the benefits of lazy commit are desirable, the lack of stricter guarantees of isolation from these lazily committed transactions and the resulting inconsistencies are not.

We describe a scheme that retains the benefits of lazy commit while providing isolation required by durable read transactions from the changes made by transactions that commit lazily. To the best of our knowledge, no database or transaction processing system has this facility.

4. Background

We have implemented our solution for a future version of Microsoft SQL Server. The SQL Server architecture is similar to that of System R. Its storage engine consists of various managers—index manager, lock manager, buffer manager, transaction manager, log manager and recovery manager—and uses an ARIES like algorithm [3] for logging and recovery. Before describing our solution, we set the context by first describing the flow of a request in the storage engine:

To read or update a row, the query processor module calls the index manager to find and optionally update the relevant row in a table. The index manager finds the page the row is present in and requests the buffer manager to retrieve the page for read or write access.

The buffer manager retrieves the page from disk into the buffer pool if it is not already in the pool, latches the page in share or exclusive mode based on the intended access and returns the page.

The index manager finds the required row in the page and acquires shared or exclusive lock on the row. If this is an update, the index manager generates a log record and applies the change to the page. If this is a read, the row is copied from the page into private memory. Then the page is unlatched.

When the transaction commits, the transaction manager generates a commit log record and requests the log manager to flush the contents of the log up to and including the commit log record to disk. Only after those log records are written to disk is the transaction declared committed and its locks released.

The log manager and the buffer manager use log sequence numbers (LSNs) to keep track of changes to the pages. Log records in the log have monotonically increasing LSNs assigned to them. Whenever a log record is generated for an update to a page, the log record's LSN is stored in the page. This is known as the pageLSN for the data page.

5. Algorithm

With the above database system architecture in view, we introduce the elements of our algorithm that provides isolation to durable read transactions from the effects of lazily committed transactions.

The transactions in our system are durable by default, i.e. they read only durable data and perform durable commit. A session level option called "LAZY COMMIT" marks all transactions in the session for lazy commit. These transactions can read non-durable data and commit lazily.

5.1 Durable Read after Lazy Commit

In its simplest form, a transaction that is committed lazily, logs the commit log record in the in-memory log buffer; however the log buffer is not synchronously written to stable storage. The locks held by this transaction are released immediately after the commit log record has been written to the log buffer.

In order to provide stricter isolation, we keep track of changes made by lazily committed transactions and make sure they are durably committed before a reader requiring durability guarantee can consume them. We achieve this as follows:

1. Whenever a row on a data page is modified by a lazy commit transaction, the page is marked **DIRTIED_BY_LC_XACT** by setting a bit on the page.
2. In addition, each database maintains a counter called **LazyCommitLSN** which is the LSN of the commit log record for the most recent lazily committed transaction. When a transaction commits lazily, in addition to writing the commit log record, the transaction manager sets LazyCommitLSN to the LSN of the commit log record for the transaction.
3. After a durable read transaction latches a page, it checks if DIRTIED_BY_LC_XACT is set on the page. If the bit is not set, then the reader proceeds with the page read. However, if the bit is set, it is an indication that the page may contain data from a lazily committed transaction. Before reading the page, the reader flushes the transaction log up to the LazyCommitLSN for the database. This ensures that any transactions that modified this page and subsequently committed lazily, will have all their effects committed durably.

This avoids the situation where a durable reader reads non-durable data from a previously lazily committed update transaction.

As illustrated in Figure 1, T1 is a lazy commit transaction and T2 is a durable read transaction requiring read committed isolation. T1 updates the row R on page P from value 6 to 7 in database D1 and subsequently commits lazily. As part of the update, the page P is marked with the DIRTIED_BY_LC_XACT. When T1 commits, the transaction manager sets the LazyCommitLSN to the LSN of T1's commit log record. Subsequently T2 comes along to read row R from page P. After T2 latches P, it finds that P is marked with the DIRTIED_BY_LC_XACT. This triggers a log flush up to the LazyCommitLSN. Since LazyCommitLSN is set to T1's commit log record, all the effects of T1 are made durable. As a result, the external update that T2 performs in another database D2 based on the modified contents of R, is consistent even in the event of a system crash.

5.2 Concurrent Lazy Commit and Durable Transactions

If the two transactions T1 and T2 mentioned in the previous section are executing concurrently, there is a possibility of a race condition that allows the durable read transaction to read non-durable data. The race condition

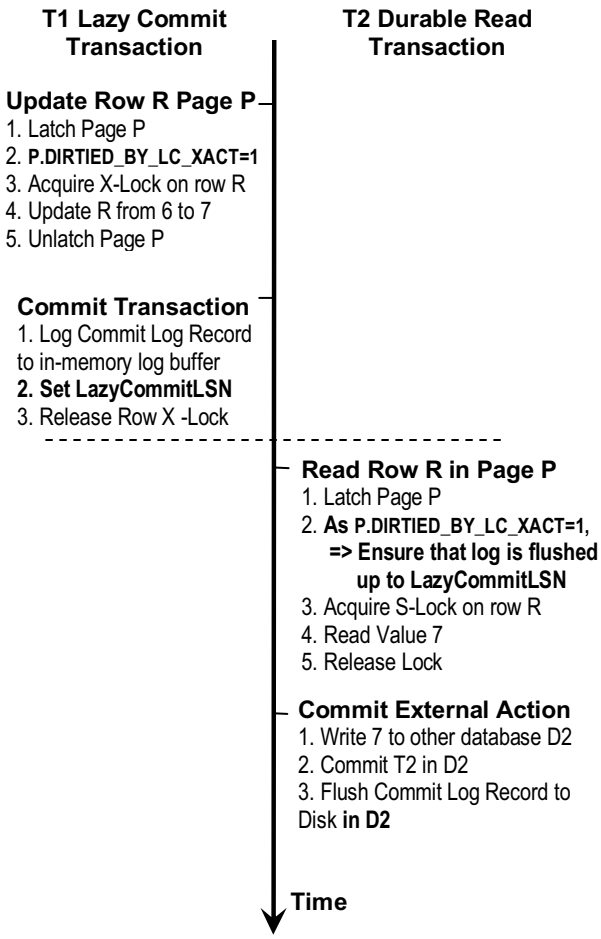
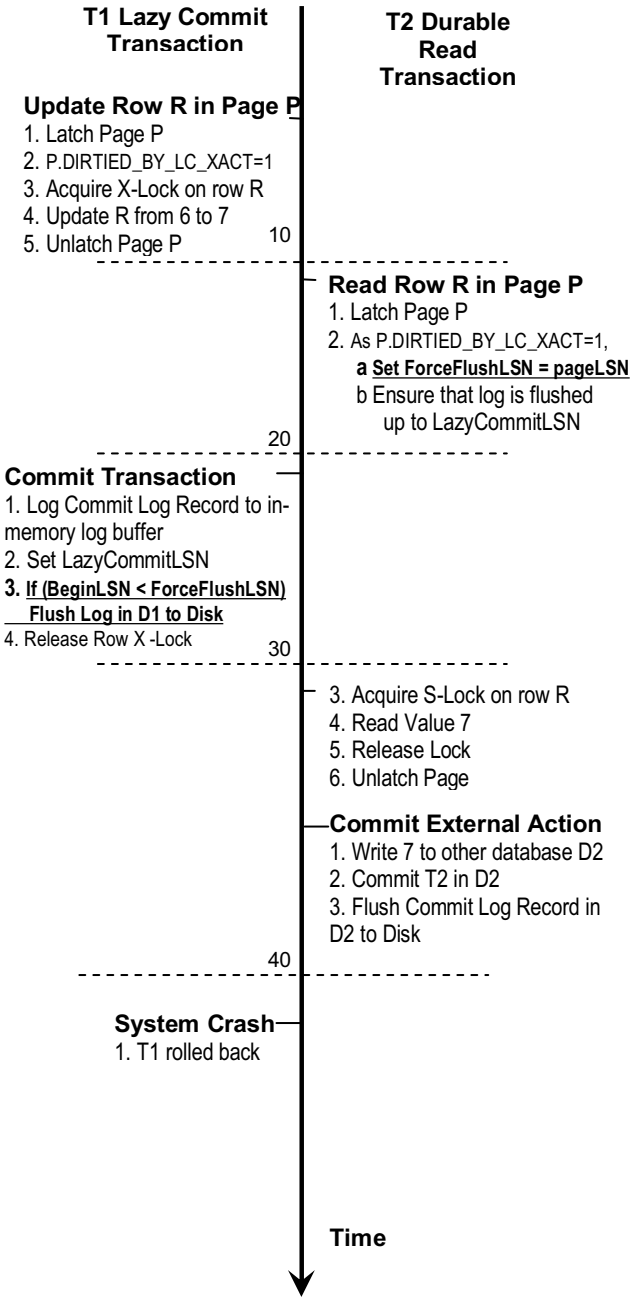


Figure 1: Durable Read Following a Lazy Commit

exists because the locking is done at the row level but the DIRTIED_BY_LC_XACT bit is maintained at the page level. We decided not to maintain the bit at the row level because of the storage and runtime overhead as well as implementation complexity. The sequence of operations that leads to the race condition is illustrated in Figure 2.

As in the previous example, T1 updates a column in the row from value 6 to 7 in database D1 and afterwards, T2 reads that new value 7 and performs the external action of writing it to another database D2. The key point here is that T2 tries to read the row before T1 commits. By time 10, T1 has modified the row, marked the page DIRTIED_BY_LC_XACT and released the latch but is still holding the lock on the row. By time 20, T2 latches the page and since the page is marked DIRTIED_BY_LC_XACT, it flushes the log until the LazyCommitLSN.



Legend: The race condition is eliminated by introducing the underlined steps

Figure 2: Race Condition With Concurrent Transactions

Since T1 has not yet committed, this does not persist T1's commit log record. Between time 20 and time 30, T1 commits the transaction, updates the LazyCommitLSN and releases the row lock. After time 30, T2 acquires the row lock and reads the updated value 7. Then it updates database D2 on the basis of this new value 7 and commits its transaction. Since T2 is a durable transaction, its commit log record in database D2 is flushed to disk. If there is a crash after time 40, T1 will roll back as its commit log record was not flushed, and databases D1 and D2 will end up being inconsistent.

To resolve this race condition, each database maintains a counter called **ForceFlushLSN**. This value is set by a durable read transaction to signal to the lazy commit transactions that might have modified a page, to force their commit log record to disk, whenever they commit.

Lazy commit transactions whose BeginLSN is less than ForceFlushLSN must flush the log on commit.

We describe below how both the transactions use this value:

Durable Read Transaction: To ensure that the lazily committed transaction (T1) that modified the page forces its commit log record to disk whenever it commits, the durable read transaction (T2) sets the ForceFlushLSN to the pageLSN of this page (unless it is already set to a higher value). Since T1 modified the page, its BeginLSN must be less than the PageLSN and hence setting ForceFlushLSN to PageLSN will ensure that T1 commits durably whenever it commits.

Lazy Commit Transaction: Before releasing its locks at the commit time, a lazy commit transaction checks if its BeginLSN is less than the ForceFlushLSN and if it is, it flushes the log like a durable transaction.

5.3 Optimizing Log Flushes

In our algorithm, when a durable reader ensures that the contents of a page are durable, it ends up ensuring that the contents of all pages with pageLSN less than the current page's LSN are also durable. These pages still have DIRTIED_BY_LC_XACT bit set and a subsequent durable read of these pages can cause unnecessary log flushes.

To reduce unnecessary log flushes, the system maintains a counter called **DurableReadLSN**. Any page with pageLSN less than or equal to DurableReadLSN is guaranteed to have all its changes durably committed.

When a durable read transaction visits a page with DIRTIED_BY_LC_XACT bit set, it flushes the log only if the pageLSN is greater than the DurableReadLSN and sets the DurableReadLSN to the pageLSN.

To ensure it is reading durable data, the durable reader transaction performs as follows:

1) If PageLSN <= DurableReadLSN, then the page contains durably committed changes. The

transaction clears the DIRTIED_BY_LC_XACT and then reads the data.

2) However if PageLSN > DurableReadLSN then the log records of the lazily committed transactions that modified this page may not have been written to disk. In this case, this transaction performs the following steps:
   i) Ensures that the contents of the page being read are durable using the steps outlined in sections 5.1 and 5.2.
   ii) Sets the DurableReadLSN to the PageLSN of this page to indicate that all pages with LSN < this PageLSN have durable data and then clears the DIRTIED_BY_LC_XACT flag on the page.

5.4 Algorithm Summary

The algorithm introduces three in-memory counters for each database:

1. **LazyCommitLSN:** The LSN of the commit log record of the last lazy commit transaction.
2. **ForceFlushLSN:** Any lazy commit transaction with LSN less than ForceFlushLSN must commit durably.
3. **DurableReadLSN:** Any page with LSN less than DurableReadLSN is known to be durable.

The pseudo code describing changes to the database engine is as follows:

Durable Read Transaction

```
ModifiedGetPage () {
   LatchPage;
   If (DIRTIED_BY_LC_XACT) {
      If (PageLSN <= DurableReadLSN) {
         Clear DIRTIED_BY_LC_XACT bit;
      }
      else {
         ForceFlushLSN = PageLSN;
         Flush Log To LazyCommitLSN;
         DurableReadLSN = PageLSN;
         Clear DIRTIED_BY_LC_XACT bit;
      }
   }
}
```

Although ForceFlushLSN and DurableReadLSN always get assigned to PageLSN, they are updated at different points in the above function.

ForceFlushLSN must be updated *before* the log flush; otherwise, a lazy commit transaction that updated the current page might commit right after the log flush and before the update of ForceFlushLSN. The lazy commit transaction would not become durable and durable read transaction would end up reading non-durable data.

The DurableReadLSN must be updated after the log flush; otherwise, another durable reader might read the page, assuming it is durable, even before the log is flushed.

Lazy Commit Transaction

```
ModifiedGetPageForWrite() {
   LatchPage;
   Set DIRTIED_BY_LC_XACT bit;
}

ModifiedCommit () {
   Write commit log record to log buffer;
   If (Transaction Marked for Lazy Commit) {
      Set LazyCommitLSN;
      If (BeginLSN < = ForceFlushLSN) {
         Flush log buffer;
      }
   }
   else {
      Flush log buffer;
   }
   Release Locks;
}
```

5.5 Implementation Notes

Our technique requires a durable read transaction to clear the DIRTIED _BY_LC_XACT bit on the pages it reads. Normally, an update to a page requires a log record to be generated and the page to be marked dirty so that it is written to disk when it is evicted from the buffer pool. Strictly following this approach in case of clearing the bit would incur additional I/Os and hence undesirable.

To avoid this problem, we do not mark the page dirty, nor do we generate a log record after clearing the DIRTIED_BY_LC_XACT bit. It is quite possible that after clearing the bit, the page gets removed from the buffer pool without being written to the disk and the bit will remain set on the page. When the page is brought into the buffer pool the next time, the buffer manager clears the bit if the pageLSN is less than the DurableReadLSN. Doing so ensures that the future durable updates to the page do not look like lazy commit updates and durable readers don't incur unnecessary log flushes.

6. Analysis

In this section, we present an analysis of the advantages and limitations of this algorithm.

6.1 Advantages

1. Requires little bookkeeping: This algorithm maintains one bit per page and three additional LSN values per database.

2. Adds little overhead when no lazy commit transactions in the system: One of the goals of this algorithm is to add minimal overhead to the durable commit code path. This algorithm requires:

i. For every page latched one extra comparison to check whether that page has been dirtied by a lazy commit transaction. In the absence of lazy commit transactions, this is the only overhead.
ii. One comparison for every commit. At commit time, a transaction checks if it is a lazy commit transaction to decide whether to flush the log to disk.
iii. One comparison every time a page is marked dirty. Whenever a page is marked dirty, it is checked if the current transaction is a lazy commit transaction, in which case DIRTIED_BY_LC_XACT flag is set.

3. Never performs more I/Os than durable commit: Regardless of the workload and the mix of lazy commit and durable read transactions, this algorithm does not lead to more I/Os than a database system that supports only durable commit. Consider N lazy commit transactions that committed in succession. Each time a transaction commits, the LazyCommitLSN is updated. During these N transaction commits, the LazyCommitLSN could have had at most N distinct values. A durable reader that forces log flush always flushes up to the LazyCommitLSN only if the log has not been previously flushed beyond that point. Hence no matter how many durable readers are active in the system, the log maybe flushed up to a distinct value of LazyCommitLSN at most once. Since LazyCommitLSN has at most N distinct values for N lazily committed transactions, our scheme will never perform more synchronous log writes than the number of transaction commits.

6.2 Limitations of the algorithm

When a page modified by a lazy commit transaction is read by a durable read transaction, it causes this page to be made durable even if the row the reader is interested in is different from the row that was modified by the lazily committed transaction. This limitation exists because the algorithm's book keeping is at the page level, instead of at the row level.

7. Performance Measurements

We describe several experiments to measure the performance benefits of our algorithm. Section 7.3 demonstrates the reduction in I/O because of lazy commit and how it is influenced by the presence of durable read transactions. Section 7.4 compares lazy commit with batched updates, another technique commonly used to reduce the number of log flushes.

7.1 Hardware

All the performance measurements are performed on a Toshiba Portégé 3500 Tablet PC that has Intel Pentium III processor-M at 1.33 GHz and Toshiba MK6022GAX 60GB disk rated at 5400 RPM with 13msec seek time.

7.2 Workload

All the experiments are performed with a queuing workload. We created an accounts table with 200 rows, which has the account number, balance and an account description. The queue table, which is created in the same database, contains the debits or credits that are to be applied to the accounts. The queue processing application dequeues the entries from the queue and applies the corresponding debit or credit to the account. The readers query the account balance of the accounts with random distribution. The updates are distributed according to the 80/20 rule - 80% of the updates happen to 20% of the rows.

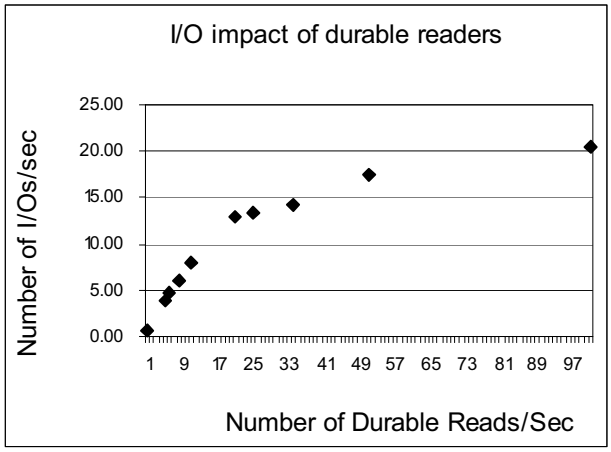


Figure 3: I/Os vs. Number of Durable Readers

7.3 Improvement in I/O

In this experiment, we simulate an application that is receiving and processing requests at a constant rate while there is a varying rate of concurrent durable read operations.

The application is throttled to process 20 queue entries per second. The I/O rate and the rate of durable read operations are related as shown in Figure 3.

With few durable reads per second (0-5), lazy commit significantly reduces the I/O rate. The I/O rate is as low as 0.59 I/Os per sec when there are no durable readers. The I/O rate increases with increase in the durable reads. This

reaches a maximum of 20/second which is also the number of commits/second demonstrating that the number of I/Os with lazy commit is never more than the number of I/Os without. This experiment also demonstrates that the use of lazy commit can yield significant reduction in the number of I/Os in the presence of a moderate number of durable read operations.

7.4 Throughput and Response Time

In this experiment, we compare the throughput and response time of the traditional approach of batching multiple updates in a single transaction with using lazy commit and not forcing the application to do multiple updates in a single transaction. The application processes 20000 queue entries using durable commit with varying batch sizes and using lazy commit with no batching. Figure 4 shows the variation in number of updates per second (throughput) with variation in batch size.

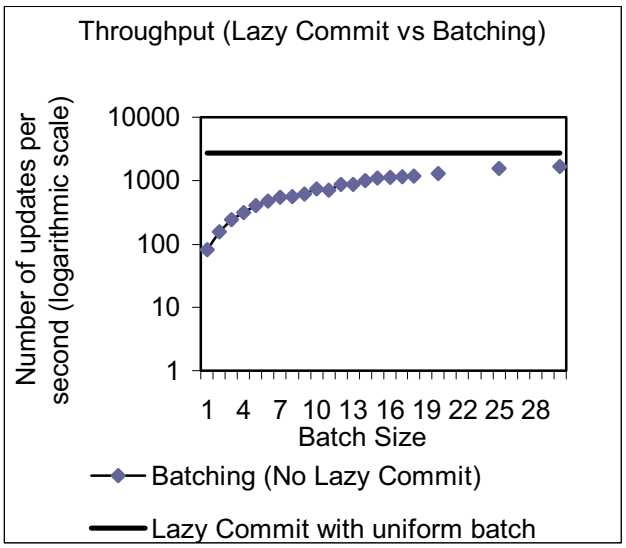


Figure 4: Throughput: Lazy Commit vs. Batching

In the absence of batching, the throughput with lazy commit is 2710.6 updates/sec which is significantly better than that with durable commit (82.5 updates/sec).

As the batch size increases, the throughput for durable commit increases, gradually approaching the performance of lazy commit. However with increased batch size comes longer duration row locks, reduced concurrency and increased reader response time. On the other hand, lazy commit achieves high throughput while retaining high concurrency and low reader response time.

Figure 5 demonstrates how the average reader response time for durable commit degrades with higher batch sizes. Lazy commit makes it possible for the application to achieve high throughput with better reader response time than durable commit.

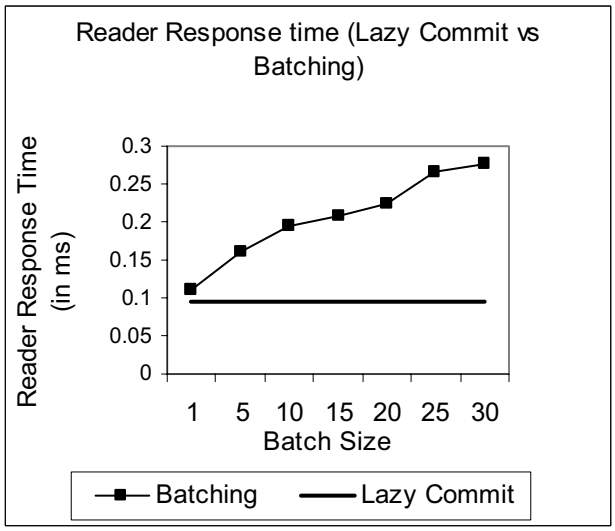


Figure 5: Reader Response Time (Lazy Commit vs. Batching)

8. Conclusion

In this paper, we presented an algorithm to isolate durable read transactions from lazy commit transactions and eliminate the possibility of data inconsistencies. We demonstrated that lazy commit, compared with batching of updates, yields higher throughput and lower response time. We also showed that lazy commit, even in the presence of durable readers, significantly reduces the I/O requirements of the application.

9. Acknowledgements

We thank our colleagues - Michael Zwilling for his contributions to the algorithm, Avi Levy and Rajesh Iyer for testing the implementation and Atul Adya for a timely review.

10. References

[1] Gray, J., and Reuter, A., *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[2] IBM Informix Guide to SQL: Syntax, Version 9.4 (G251-1243-0). http://publibfi.boulder.ibm.com/epubs/pdf/ct1sqna.pdf

[3] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P. *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*, **ACM Transactions on Database Systems**, Vol. 17, No. 1, pp94-162, March 1992.

[4] Oracle TimesTen Architecture Overview: Release 6.0. http://download-west.oracle.com/otn_hosted_doc/timesten/603/TimesTen-Documentation/arch.pdf

[5] Oracle Database 10g Release 2 (10.2) Documentation. http://technet.oracle.com