Enforcing Database Recoverability on Disks that Lack Write-Through

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

Abstract

Most database systems use ARIES-like logging and recovery scheme to recover from failures and guarantee transactional consistency. ARIES relies on Write-Ahead Logging (WAL) protocol which requires that log records be durably written prior to the corresponding data changes. Database systems use the write-through capability of the storage media to enforce write-ahead logging. While SCSI disks that are commonly deployed in enterprise servers support write-through, commodity hard drives do not. While database systems were mostly limited to enterprise servers in the past, today they are being heavily deployed in large-scale internet services and personal information management systems. In order to minimize costs, these systems use commodity hard drives that have controller caches but lack write-through. These drives delay and reorder the writes thereby breaking the WAL assumption on which recovery is based. Here we present a solution to enforce WAL, hence recoverability of the database on these drives. We also present performance measurements demonstrating that our approach does not cause any overhead in the path of the transaction.

1. Introduction

Living in an era of information explosion, the need for medium to large scale data management is no longer limited to enterprise servers. With exponential growth of markets like e-commerce, packaged applications, mobile computing, and web-based appliances and devices; the need for databases on desktops, laptops and mobile devices is felt stronger than ever before. 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, information search engines etc 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.

In addition to the need for data as a service, need for embedded databases is also on the rise. The packaged applications market witnesses a need for simple and easy to manage tools and applications with complex data management capabilities. The fast paced growth of distributed and mobile computing has created the demand for intelligent applications to power cell phones, PDAs and other mobile devices. These applications have data management needs similar to those of the packaged applications. They are further fueling the demand for embedded databases.

Although the need for data management in many of the low end applications described above is largely similar to those for high end enterprise scenarios, the high end server and low end client environments are significantly different. Applications on the low end require databases with high-performance, small-footprint as well as self-tuning and self-managing capabilities. While some of the differences in the environment and application requirements can provide an opportunity for improving throughput, there are others that require additional infrastructure in the database engine. Traditional

database systems developed with the high end enterprise servers in mind aren't sufficiently client-friendly to adequately address requirements of the low end application scenarios.

Large-scale internet services provide for another unique deployment environment for database systems ([6], [8]). These services supported by fee-for-service or advertising revenue are heavy users of database technology. Several services such as email, social networking, blogs, instant messaging, photo sharing, maps, shopping, and classifieds have requirements for large-scale complex data management. These services have requirements for very high computing power, which is expensive to acquire and operate. If we consider a high-end commodity server (1-4 processors, 2-24GB RAM, and 12-24 SCSI disk drives) that is used to run a database system currently costs upwards of $20K. Multiplying that by the number of servers required to run the large-scale internet service, we see that the system can cost tens or hundreds of millions of dollars. The size and technology of the disk system is a strong determinant of overall system price. Given the cost of the premium servers and storage media, one of the approaches to reduce overall cost of operation is to use commodity servers with commodity hard drives ([6], [7]) i.e. using several low-end machines in place of a high-end machine for the database deployment.

Although these upcoming non-traditional usage scenarios for databases are quite diverse in several aspects, they share something in common; namely the use of commodity hardware, in particular commodity storage media. In this paper we talk about a problem that arises due to the use of commodity (IDE/SATA) disk drives. These hard drives are predominant on low-end machines.

Most traditional database systems use ARIES-like logging and recovery scheme to recover from failures and guarantee transactional consistency [2]**.** ARIES relies on Write-Ahead Logging (WAL) protocol which requires that log records be durably written prior to the corresponding data changes. IDE drives have controller caches and do not correctly support write-through. They delay the writes by caching them in the controller and writing them to the disk at a later point in time. More so write ordering is not guaranteed. The caching and subsequent re-ordering of write requests can break the WAL protocol on which recovery is based. In this paper we present a solution to enforce WAL by using a flush-cache command at the disk controller. We also present performance measurements demonstrating that our approach does not cause any overhead in the path of the transaction.

The rest of the paper is organized as follows. In section 2 we illustrate the problem addressed in this paper. We present the overview of our solution in Section 3. 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 algorithm, performance analysis and conclusion respectively.

2. Problem Definition

Most industrial-strength transaction processing systems including databases use ARIES [2] (Algorithm for Recovery and Isolation Exploiting Semantics) for logging and recovery in order to guarantee ACID properties of transactions and recover from crashes. ARIES supports partial rollbacks of transactions, fine-granularity (record-level) locking and recovery using write-ahead logging (WAL). The WAL protocol ([1], [2]) asserts that the log records representing changes to some data must already be on stable storage before the changed data is allowed to replace the previous version of that data on nonvolatile storage. That is, the system is not allowed to write an updated page to the nonvolatile storage version of the database until log records which describe the updates to the page have been written to stable storage.

To enable the enforcement of this protocol, systems using the WAL method of recovery, store in every page the LSN of the log record that describes the most recent update performed on that page. Before the page is written out, the system ensures that the log up to this LSN has been made durable.

Most database systems use write-through write requests in order to guarantee that the log is synchronously written to stable storage before writing the data changes. SCSI drives that are

predominantly used in enterprise server deployments of database systems, support write-through capability by means of the ForceUnitAccess (FUA) flag.

ForceUnitAccess is however not supported by most of the IDE drives (ATA/SATA drives) ([3], [4], [5] and [10]). IDE drives have a controller cache where write requests are cached before they are written to the physical disk. In the absence of FUA, the write call returns to the user-mode process when the data still may be in the volatile disk controller cache and can potentially be lost in a crash. The writes from the controller cache to the disk platter are not performed in the same order as the writes from the operating system to the controller cache. As a result of the re-ordering, although the database system writes the log, waits for the write request to complete, before writing the data, the actual writes to the disk need not be in the same order. In a system crash, with write re-ordering, the data write could have gone through while the log write is lost in the crash. This results in the violation of the WAL protocol which can result in data inconsistency, loss of data and worst of all, loss of recoverability thereby rendering the database inaccessible.

This problem is not limited to database systems alone. ARIES logging and recovery is used for other transactional systems, recoverable file systems etc. The lack of write-through guarantees poses similar problems to these systems.

One other facet of the problem relates to durability (D in the ACID properties) of transactions. In order to guarantee durability of transactions, database systems issue a synchronous I/O to write the log for each transaction commit. This would achieve durability in case of disks that support write-through. However on IDE disks, the transaction may not be durable if the log records written during commit are lost during a cache while they are still in the volatile cache. Many applications especially those mentioned in section **Error! Reference source not found.** do not require instant durability of the transactions. These applications maintain state and redundancy to be able to re-create effects of the lost transactions so long as the transactions that can be potentially lost are bounded.

In this paper we propose a solution to this problem using the FLUSH_CACHE command at the disk controller level. The scheme proposed in the paper makes judicious use of the FLUSH_CACHE command to guarantee recoverability. Although addressing recoverability is the primary goal of the paper, we also suggest performant ways for providing an upper bound on the delay in transaction durability.

3. Overview of the solution

We solve the problem by using the FLUSH_CACHE command at the disk controller. Most single disk IDE drives support the FLUSH_CACHE command at the controller that flushes all the hardware caches between the host and the disk media. Even if the intent of the call is to flush dirty data for a given file, since the disk controller is not aware of the logical association between the dirty blocks in the cache and those contained in a particular file; all the dirty blocks in the cache are written to the disk. Since the FLUSH_CACHE command is expensive, we use the command only when it is absolutely required for write ordering guarantees. Furthermore our scheme does not introduce any overhead in the path of the transaction.

Some operating systems allow the user to enable/disable write-caching on the disk. While disabling write-caching can be a potential solution, it is not desirable for the following reasons:

- There is no guarantee that write-caching will actually be disabled [5].
- Since this is a setting at disk-granularity, it would affect performance of all applications. On low-end machines, applications requiring write-through guarantees are relatively few. Disabling write-caching will result in poor overall throughput.
- On systems with write caches that are battery-backed, disabling write-caching degrades performance of all the applications without any added reliability for applications requiring write ordering guarantees.

On the other hand, our solution enforces write-through only when it is absolutely required for write ordering guarantees.

There is no support in the operating system to reliably detect if the disk supports write-through. A database system that implements our solution may incur the overhead of occasional FLUSH_CACHE command when not necessary especially in the case of disks with battery-backed controller caches. We address this by providing the ability for the database administrator (user) to specify if the underlying storage hardware already provides the write-ordering guarantee.

4. Background

Before we look at the details of the solution, lets us briefly review the architecture the database system in which this solution has been implemented. The database system 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 the ARIES algorithm [2] 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 applied to a page, the log record's LSN is stored in the page. This is known as the pageLSN for the data page.

When a dirty page is written out from the buffer pool to the disk, the buffer manager ensures that the log records up to the pageLSN have been made durable on the media before writing out the dirty page. This ensures that the write-ahead logging protocol is obeyed.

5. Algorithm

As we saw in the previous section, the mechanism to guarantee write-ahead logging is to record the LSN of the last log record that updated a data page and making sure that all the log records up to that LSN have been durably flushed ahead of writing the data page itself.

5.1 FlushLSN and DurableLSN

The log manager maintains several LSN values for a database to keep track of the progress of the log. We describe the relevant values here below.

FlushLSN: In each database we keep track of a Flush LSN which is the sequence number of the log record which was flushed from the in-memory buffers to the storage media and the server had received the I/O completion confirmation for the same. Whenever an in-memory log buffer is written to the log file on stable storage the FlushLSN is updated to the last log record written out. On disks that do not honor write-through, an I/O completion confirmation is received even when the data is still in the

controller cache. It is therefore possible some of the log records up to FlushLSN may be lost in case of a power loss.

DurableLSN: In addition, we also track a DurableLSN for each database. DurableLSN is the sequence number of the log record up to which all the log records have been made durable on stable storage.

Based on their definition, the following relation always holds for the LSNs in each database

$durableLSN \leq flushLSN \leq currentLSN$

5.2 Durably written log

The above LSNs are used to detect how much of the log is durably written. Before writing a dirty page to disk, we check if the disk controller cache needs to be flushed.

```
if (pageLSN <= durableLSN)
{
     /* WAL requirement already satisfied, no need to flush the disk
     cache.*/
     WriteDataPage();
} else if (pageLSN <= flushLSN)
{
     /* the log records could be in the volatile disk cache, we must
     flush the disk cache. */
     flushLSNBeforeCacheFlush = flushLSN
     FLUSH_CACHE

     /* Flushing the disk cache makes all the previous writes to the
     disk, durable. After FLUSH_CACHE call returns, all the log
     records up to flushLSNBeforeCacheFlush are durable. */
     durableLSN = flushLSNBeforeCacheFlush;

     WriteDataPage();
} else
{
     /* the log records are still in the in-memory buffers; we
     should make them durable on the disk.*/
     WriteLogToDisk(); // Updates flushLSN

     flushLSNBeforeCacheFlush = flushLSN
     FLUSH_CACHE
     durableLSN = flushLSNBeforeCacheFlush;

     WriteDataPage();
}
```

5.3 Initial values

When the database starts, the startup logic scans the log and determines the end of log (EndOfLogLSN). FlushLSN is initialized to the value of EndOfLogLSN. While it may seem that contents of the log up to EndOfLogLSN are durable on disk, in case of database crash restart, we do not know if the entire log read by the startup logic was read from the stable storage or a part of it was still residing in the disk cache. Therefore DurableLSN is initialized to its valid value only after the log comes online and the first set of flushed log records have been made explicitly durable.

5.4 Checkpoint

Checkpoint is the primary mechanism to insure that database changes are periodically written to disk. This helps reduce the active portion of the log that must be processed during database recovery. Crash recovery always begins with the last known good checkpoint. Write-ordering guarantees during the checkpoint are critical for successful crash recovery. During a checkpoint the following steps are carried out in the database [11].

1. *Write log record marking the start of the checkpoint:* After writing the begin check point record, we wait for the log to be durably flushed up to the current LSN.
2. *Record the Minimum Recovery LSN (MinLSN) in the log:* This is the LSN of the first log record that must be present for consistent recovery after the current checkpoint is successful. The MinLSN is the minimum of the LSN of the start of the current checkpoint and the begin LSN of the oldest active transaction.
3. *Flush dirty data pages in the buffer pool:* All the dirty pages in the buffer pool are written to the disk. The algorithm described in section 5.2 is used to guarantee WAL before each dirty data page is written out to the disk. After the log flush in step 1, we should require very few (if any) additional durable log writes.
4. *Make flushed dirty data durable:* After a successful checkpoint, log records before MinLSN won't be processed during the subsequent crash recovery. It is therefore necessary to ensure that the dirty pages written during the checkpoint are made durable. To achieve this, a FLUSH_CACHE command is issued once for each of the disks to which the dirty buffers were written.
5. *Write a log record marking the end of the checkpoint:* After writing the end check point record, we wait for the log to be durable up to the end checkpoint LSN.
6. *Write the begin checkpoint LSN to the database boot page:* We need to make sure that the boot page points to the latest checkpoint. We issue a FLUSH_CACHE to make the changed boot page durable before the log truncation logic can truncate the active log.

A checkpoint operation is infrequent and is only triggered automatically after sufficient database activity [11]. As a part of the checkpoint dirty pages from the buffer pool are written to the storage media. The size of the buffer pool (which is in the order of available main memory) is typically a few hundred megabytes to a few gigabytes while disk controller caches are only 8-16MB in size. In comparison to the amount of data being flushed from main memory to stable storage during a checkpoint, the data flushed from the disk cache is negligible. Therefore adding the overhead of FLUSH_CACHE commands to the checkpoint logic does not hurt throughput since they are amortized over the entire workload.

5.5 Flush only when necessary

As described above; only if the buffer manager needs to ensure that the log up to a pageLSN is durable, do we cause a FLUSH_CACHE. During normal processing, especially on the workloads that are typical on low end client scenarios, paging out a dirty page is not a frequent operation. Although the FlushLSN is updated every time a log block is flushed to disk, the DurableLSN is updated once for

several updates to the FlushLSN (only when sufficient updates have been made to fill up the buffer pool requiring a page to be written out). Thus the overhead of this scheme in the path of the transaction is negligible.

5.6 End of Flushed Log

End of flushed log signifies the last LSN that is known to have been made durable. This value is not maintained separately, but is computed using the other LSN values.

Utilities such as backup use the end of flushed log to determine the extent of a log backup. A log backup chain consists of contiguous, non-overlapping ranges of the log. A log restore simply restores the log from the log backup chain and replays the log records to bring the database to a consistent state as of a point in time in the past [11].

Let us consider the following scenario with respect to log backups. The n^{th} log backup (L_n) backs up the range of the log between LSNs 100 and 500. When it reads the log from the disk, the log records between LSNs 100 and 400 are on the disk platter but the remaining log records (400 to 500) are in the disk cache. A system crash results in the loss of log in the disk cache (LSN 400 to 500). New log records fill up the log between LSN 400 (end of log after crash-restart) and 800. The next log backup (L_{n+1}) backs up the range between LSN 501 and 800. When this log chain is restored, the log records between 400 and 500 will be from the log that was over-written as opposed to the ones in the original database, thus causing inconsistency.

In the presence of the disk controller cache, we do not know about the durability of log records between the DurableLSN and the FlushLSN. In order to avoid scenarios like the one described above the DurableLSN is treated as the end of the flushed log.

In the scenario above, DurableLSN will be less than or equal to 400. Therefore log backup L_n will at most backup log up to LSN 400. All of the log included in this backup will survive a system crash avoiding inconsistency.

5.7 Durability of transactions

As explained in section **Error! Reference source not found.**, in the presence of a disk controller cache, the durability of transactions is not guaranteed in the event of a system crash. Although the database system synchronously writes the log up to the commit log record, there is an interval during which the log record may still be residing in the disk cache. We refer to this time interval as delay in the durability of the transaction

While the primary goal of our scheme is to address recoverability, we also provide an upper bound on the delay in transaction durability. If we issue a FLUSH_CACHE command periodically with an appropriate frequency, we can provide an upper bound on the maximum time between a transaction commit and the data being made durable.

6. Performance Analysis

In this section, we describe performance measurements to show that our scheme does not introduce a significant overhead in the path of the transaction.

6.1 Hardware

All the performance measurements are performed on a Intel® Pentium® 4 3.4 GHz HT with 1GB RAM and Western Digital WDCWD2500JD-75HBC0 250GB disk rated at 5400 RPM with Seek Time 8.9 ms (average) / 21 ms (max) and a 8MB disk controller cache.

6.2 Personal Information Management Systems

For evaluating our scheme for personal information management systems, we simulated a work load that would be handled by a rich email client application that uses a database to store email messages. One of the common operations performed by the email client is syncing the local mail box

with that on the server by downloading email messages, calendar items and contacts. The table below compares the performance of the simulated email client with and without our scheme enforced by its backend database.

Number of entities	Time (in seconds)	
	No WAL Enforcement	WAL Enforced using our scheme
100	5.967	6.326
1,000	8.057	8.079
200,000	636.285	645.641

The results in the table show that the overhead incurred by our scheme for providing recoverability is minimal (1-5%). Further we observe that as the number of entities that are being processed increases, the amount of overhead incurred by our scheme decreases. For instance while downloading 1000 or 200K entities the overhead is negligible (1-2%).

6.3 Modified TPC-C benchmark

With the same hardware as described in section 6.1, we used a scaled-down version of the TPC-C benchmark [12] to measure the performance of our solution on an OLTP workload. At steady state, the scaled-down workload consisted of a database with a single database file roughly 1.6GB in size and the log file roughly 2.9GB in size. The following table shows the results

Scenario	Configuration		tpmC
	WAL Enforced using our scheme	Disk controller caching	
A	No	Enabled	4083
B	**Yes**	**Enabled**	**3994**
C	No	Disabled	3673

The table compares our solution for providing recoverability to the option of disabling the disk controller cache. As seen from the table above, our solution (Scenario B) only incurs a small overhead (roughly 2%) compared to the baseline (A) while providing recoverability. On the other hand, we perform better than disabling the disk controller cache (Scenario C) which causes roughly 10% decrease in performance.

Disabling disk controller caching forces every disk write to be written to the physical disk. Our scheme can take advantage of the disk caching for most of the writes while incurring the cost of flushing only when write-ordering guarantees are required. This explains why our scheme performs better than disabling caching at the disk controller.

7. Conclusion

In this paper, we motivate the problem of enforcing WAL to guarantee database recoverability in the presence of IDE drives that do not correctly honor write-through. We propose a solution that makes judicious use of the FLUSH_CACHE command at the disk cache controller whenever write ordering guarantees are required. In addition to recovery, we also propose a performant solution for providing an upper bound on delay in transaction durability. We also showed that our algorithm, does not introduce any overhead in the path of the transaction.

8. Acknowledgements

We thank Eric Christensen for his contributions to the algorithm and insightful guidance during the implementation. We thank our colleagues Gayathri Venkataraman, Avi Levy and Sherry Li for testing our implementation.

9. References

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

[2] 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, March 1992, pp94-1

[3] Anderson, D., Dykes, J., and Riedel, E. SCSI vs. ATA – More than an interface. In the Proceedings of the 2nd USENIX Conference on File and Storage Technologies, (FAST' 03) (San Francisco, CA, 2003)

[4] L. Chung, J. Gray, B. Worthington, R. Horst, Windows 2000 Disk IO Performance, MSR-TR-2000-55, September 2000

[5] Disk Subsystem Performance Analysis for Windows http://www.microsoft.com/whdc/device/storage/subsys_perf.mspx , March 2004.

[6] Philip A. Bernstein, Nishant Dani, Badriddine Khessib, Ramesh Manne, David Shutt. Data Management Issues in Supporting Large-Scale Web Services IEEE Data Eng. Bull. 29(4), pp. 3-9 (2006).

[7] Data Center Knowledge. www.datacenterknowledge.com

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06), November 2006.

[9] Jim Gray and Catherine van Ingen. Empirical measurements of disk failure rates and error rates. Technical Report MSR-TR-2005-166, December 2005.

[10] Dominguez, R. and Coligan, T. "SCSI vs. ATA: Interface Comparison" Technology Brief, Dell Computer, December 1999.

[11] SQL Server 2005 Books Online. http://msdn2.microsoft.com/en-us/library/bb418498.aspx

[12] Transaction Processing Performance Council, TPCC Benchmark Specification, http://www.tpc.org/tpcc.