Consistency of Databases on Commodity Disk Drives

Robin Dhamankar, Hanuma Kodavalla, Vishal Kathuria

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
USA
{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 the Write-Ahead Logging (WAL) protocol which requires that log records be written durably prior to the corresponding data changes. In order to enforce WAL, database systems rely on the write-through capability of the storage media. While SCSI disks that are commonly deployed in enterprise servers support write-through, commodity hard drives do not. In the past, database systems were mostly limited to enterprise servers; however, 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 and lack write-through. These drives delay and reorder the writes thereby breaking the WAL protocol recovery is based on. Here we present a solution to enforce WAL and guarantee recoverability of the database on these drives. We also present performance measurements demonstrating that our approach does not adversely impact transaction throughput.

1. Introduction

Database management systems have traditionally been designed and tuned to be deployed on high-end on-premise enterprise servers. 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, 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 use of Personal Information Management (PIM) systems such as rich email clients and desktop search engines has increased. In order to meet their requirements for complex data management, these systems use databases for storage and retrieval of data and metadata. As they are primarily used on desktops and laptops, they run on commodity hardware.

Large-scale internet services provide another unique deployment environment for database systems ([6], [8]). 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 that 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, it 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 the technology of the disk system significantly affect the overall system cost. Given the high cost of the premium servers and the storage media, one of the approaches to reduce overall cost is to use commodity servers with commodity hard drives ([6], [7], [16]), i.e. replace a single high-end machine with several low-end machines.

While data management is similar across these diverse user segments, the desired performance characteristics, cost considerations and hence hardware choices vary significantly between low-end applications, large-scale cloud computing and high-end enterprise servers. High-end servers deploy specialized expensive hardware while low-end applications and cloud computing use commodity hardware. In particular, high-end servers use expensive SCSI disks for storage; desktop and cloud environments use much cheaper IDE or SATA disk drives.

15th International Conference on Management of Data
COMAD 2009, Mysore, India, December 9–12, 2009

Most traditional database systems use ARIES-like logging and recovery scheme to recover from failures and guarantee transactional consistency [2]. ARIES relies on the Write-Ahead Logging (WAL) protocol which requires that log records be durably written prior to the corresponding data changes. Commodity disk drives (SATA/IDE) have caches in the controller 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 time. In addition, writes to the disk platter may not be in the same order in which they were issued by the database system. The caching and subsequent re-ordering of write requests can break the WAL protocol on which database recovery is based. This re-ordering can also affect other mechanisms such as checkpoint and incremental log backups that depend on write-ordering. In this paper, we assess the impact of lack of write-through on database consistency and propose enhancements to the database engine to ensure consistency. This paper builds on the work we presented in [13]. In summary, we make the following contributions:

1. We discuss the two aspects of database consistency affected by lack of write-through, namely recoverability and transaction durability. We argue that while recoverability is critical, it is sufficient to ensure that the delay in durability is bounded.
2. We present a solution to ensure recoverability, whereby write-ordering is enforced by judicious use of the FLUSH_CACHE command at the disk controller.
3. Further, we show how our scheme can be extended to provide an upper bound on the delay in transaction durability.
4. We present performance measurements to demonstrate that our approach does not adversely impact transaction throughput.

The rest of the paper is organized as follows. Sections 2 and 3 illustrate the problem and the solution. Section 4 describes the overall architecture of the database system in which our solution is implemented. Sections 5 through 9 present the details of the algorithm. Section 10 presents performance analysis and Section 11 presents conclusions.

2. Recoverability and Durability

In order to guarantee ACID properties of transactions and to recover from crashes, most industrial-strength database systems use ARIES [2], which relies on the write-ahead logging (WAL) protocol. The WAL protocol ([1], [2]) asserts that the log records representing changes to some data must already be on non-volatile storage before the changed data replaces the previous version of that data on non-volatile storage. In order to enforce this protocol, within each data page, the Log Sequence Number (LSN) of the log record that describes the most recent update to that page is maintained. Before the page is written out, log records up to this LSN are made durable by issuing a write-through log write. While enterprise-class SCSI disks support write-through capability by means of the ForceUnitAccess (FUA) flag, most of the commodity drives (ATA/SATA drives) do not ([3], [4], [5] and [10]). These 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 may still be resident 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 and 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 without the corresponding log write. This results in the violation of the WAL protocol which can cause data inconsistency, loss of data or loss of recoverability thereby rendering the database inaccessible.

Another 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 a group of transactions to commit. This would achieve durability in case of disks that support write-through. However, on commodity disks, the transaction may not be durable if the log records written during commit are lost in a system crash while they were still in the controller cache. Many applications, especially those mentioned in the previous section, do not require durability of the transactions ([14], [6]). In some cases, these applications maintain redundant copies of the data and have the ability to re-create effects of the lost transactions so long as the transactions that can be lost are bounded ([6], [8]). In other cases, as long as the amount of activity lost can be detected, the application has the ability to resume processing from the last successfully committed change [14]. A bounded loss in transaction activity provides for a bounded recovery time and hence guarantees predictable availability of the system. If recoverability is compromised, the entire database is lost and re-creating the database from redundant copies may be expensive and time consuming. Therefore recoverability is essential, durability is not; bounded loss of durability is acceptable.

In this paper we propose a solution to this problem using the FLUSH_CACHE command at the disk controller level. The proposed scheme makes judicious use of this command to guarantee recoverability. Although addressing recoverability is our primary goal, we also suggest an efficient way 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 that flushes all the hardware caches between the controller and the disk media. Most single disk SATA and IDE drives support this command. 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. As the FLUSH_CACHE command is expensive, we use the command only when it is required for write-ordering guarantees. Furthermore, our scheme does not introduce any overhead in the path of the transaction.

Some operating systems expose knobs to allow the user to disable write-caching at the disk controller. Although this solves the write-ordering problem, it has the following drawbacks:

- 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 and disabling write-caching will result in poor overall throughput.

There is no support in the operating system to reliably detect if the disk supports write-through. Our solution introduces the overhead of an occasional FLUSH_CACHE command when it may not be necessary for disks with battery-backed controller caches where data in the cache survives a crash. We address this by providing the ability for the database administrator to specify if the underlying storage hardware already provides the write-ordering guarantee.

4. Background

In this section, we briefly review the database system architecture in which this solution is implemented. The system is modeled after System R and its storage engine consists of various managers: index manager, lock manager, buffer manager, transaction manager, log manager and recovery manager. It uses an ARIES [2] like algorithm for logging and recovery.

The flow of a request in the storage engine can be summarized as follows:

- To read or update a row, the **query processor** calls the **index manager** to find and optionally update the relevant row. The index manager finds the corresponding page 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 present, 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 before it 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.
- The **log manager** and the **buffer manager** use monotonically increasing log sequence numbers (LSNs) associated with log records to keep track of changes to the pages. 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. Before a dirty page is written to disk, the buffer manager ensures that the log records up to the pageLSN are durable.
- When transaction commit is requested, the transaction manager generates a commit log record and log manager writes 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 to the client and its locks released.

5. Write-Ahead Logging

5.1 FlushLSN and DurableLSN

The mechanism to guarantee write-ahead logging is to ensure that all the log records up to the pageLSN have been *durably* flushed ahead of writing the page itself. The pageLSN is compared with several LSN values maintained by the log manager to determine if it is safe to be written to non-volatile storage. The most pertinent to this discussion are the following:

FlushLSN: Whenever an in-memory log buffer is written to the log file on non-volatile storage, the FlushLSN is updated to the LSN of the last log record in that buffer. 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. In case of a power loss, some of the log records within the range of FlushLSN may be lost.

DurableLSN: DurableLSN is the sequence number of the log record up to which all the log records have been persisted on non-volatile storage.

Based on these definitions, the following relation always holds:

$$durableLSN \leq flushLSN \leq currentLSN$$

5.2 Durably written log

The LSNs described above 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. Figure 1 shows the algorithm to enforce write-ahead logging using these LSNs.

```
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();
}
```

Figure 1: Algorithm for Enforcing Write Ahead Logging.

5.3 Initialization of DurableLSN

When the database starts, the startup logic scans the log, determines the end of log (EndOfLogLSN) and sets the FlushLSN to that. The reason for not setting DurableLSN to that value is that some part of the log may still be in the controller cache. Therefore, DurableLSN is initialized only after the first set of newly written log records have been made explicitly durable by issuing a FLUSH_CACHE command.

5.4 Flush only when necessary

FLUSH_CACHE is an expensive command and hence should be used judiciously. As described in Figure 1, only if the buffer manager needs to ensure that the log up to a pageLSN is durable, will we issue the FLUSH_CACHE command. The FlushLSN is updated every time a log buffer is written to disk. A FLUSH_CACHE command makes all the previous log writes durable; therefore the DurableLSN is updated once for several updates to the FlushLSN.

Typically the buffer pool is large enough to accumulate updates made by several transactions before a dirty page would have to be evicted. The cost of the FLUSH_CACHE command will be amortized over several transactions, thereby making the overhead per transaction negligible.

6. Checkpoint

Checkpoint ([2], [11]) is the primary mechanism to ensure that updated database pages 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. Next we outline the steps involved in a checkpoint and also describe how ordering of writes is guaranteed in the absence of write-through:

1. *Write log record marking the start of the checkpoint:* After writing the begin check point record, we issue a FLUSH_CACHE to make the log durable up to the current LSN. The LSN associated with the begin checkpoint log record is the CheckpointLSN.
2. *Flush dirty data pages in the buffer pool:* All the dirty pages in the buffer pool are written to the disk. Before each dirty data page is written out to the disk, the algorithm described in Figure 1 is used to guarantee write-ahead logging. After the log flush in step 1, step 2 requires very few (if any) additional durable log writes.
3. *Make flushed dirty data durable:* Once the checkpoint has been successfully recorded, log records before CheckpointLSN won't be redone during the redo phase of 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 to each of the disks to which the dirty buffers were written during step 2.
4. *Record the state of active transactions in the log:* For each of the active transactions, information that will be required during recovery is recorded in the log using special log records.
5. *Write a log record marking the end of the checkpoint:* The end checkpoint log record makes a note of the MinRecoveryLSN, i.e. the LSN of the first log record that must be present for consistent recovery. A FLUSH_CACHE command is then issued to make the log durable up to the end checkpoint log record.
6. *Write the CheckpointLSN to the database boot page:* We need to make sure that the boot page points to the latest checkpoint. The boot page is updated with the CheckpointLSN and made durable by issuing a FLUSH_CACHE. Only after the boot page that is pointing to the new checkpoint is persisted, the log truncation process can safely truncate the re-usable portion of log prior to this checkpoint.

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 non-volatile storage, the data flushed from the disk cache is negligible. Therefore adding the overhead of FLUSH_CACHE commands to the checkpoint process does not adversely impact throughput.

7. Log Backups

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].

During each individual log backup, the engine uses the end of flushed log to determine the extent of log to archive. End of flushed log signifies the last known durable LSN. This value is not maintained separately, but is computed using the other LSN values that are tracked by the log manager. In the presence of write-through, the engine can use the FlushLSN as the end of the flushed log.

In the absence of write-through, Figure 2 illustrates the consequences of using the FlushLSN as the last LSN for a log backup:

a. Let us suppose that the current value of FlushLSN and DurableLSN are 500 and 400 respectively, i.e. the log records between LSN 100 and 400 are on disk but the remaining log records (401 to 500) are potentially still in the disk cache.
b. The n^{th} log backup ($\boldsymbol{L_n}$) backs up the range of the log between LSN 100 and 500 (FlushLSN).
c. A system crash occurs and results in the loss of log in the disk cache (LSN 401 to 500).
d. The database restarts and new log records fill up the log between LSN 400 (end of log after crash-restart) and 800.
e. The next log backup ($\boldsymbol{L_{n+1}}$) backs up the range between LSN 501 and 800.

When this log chain is restored, the log records from 401 to 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 inconsistency, DurableLSN must be treated as end of the flushed log. In the example above log backup $\boldsymbol{L_n}$ will at most backup log up to LSN 400 and the entire log included in this backup will survive a system crash. $\boldsymbol{L_{n+1}}$ will archive the newly written log. The phantom portion that was lost in the crash is not included in the log backup chain.

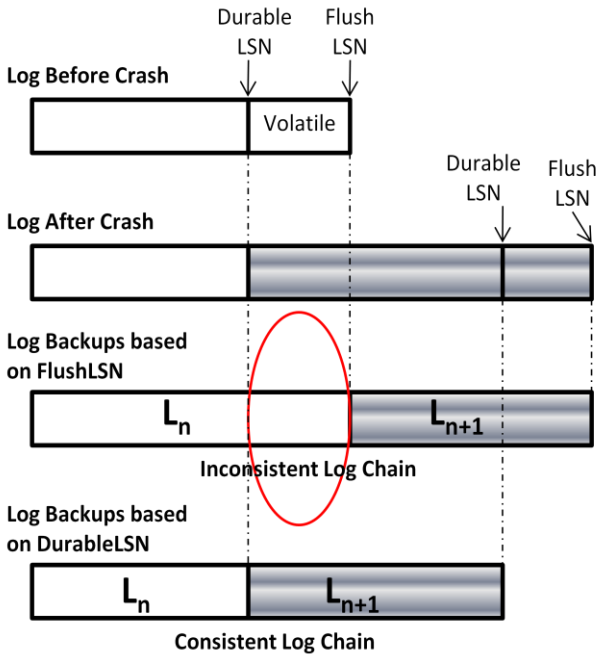

Figure 2: Log Backups.

8. Minimal logging

In order to support high-volume data loading scenarios, SQL Server implements minimally logged operations [11]. Unlike fully logged operations, where the transaction log is used to keep track of every row change, minimally logged operations only track page allocations and metadata changes in the log. If the minimally logged transaction needs to be rolled back, the database engine will undo the metadata changes and allocation log records to mark the pages as unallocated. Since the data itself is not logged, atomicity of such a transaction cannot be achieved through a log flush and replay. Instead, when this transaction commits the database engine flushes all the affected data pages to disk.

If all the affected data pages are not persisted on the disk before committing the transaction, there is a risk of violating atomicity. The following example illustrates the problem. Consider a minimally logged operation that re-builds a clustered index on a table. The operation involves the following steps, carried out in a single transaction:

a. Create a new index by populating rows from the old index. This operation is minimally logged, so the log does not have all the data that is being added.
b. Update the metadata to point to the new index location (this operation is fully logged).
c. Drop the old index.

If the transaction is marked as committed before all the pages affected in step a) are persisted, then a crash can cause a corrupt index after step b) has been replayed. The metadata would point to the new index, some of whose pages may be stale or missing.

If the underlying disk does not support write-through, in addition to writing the minimally logged data pages, they must be made durable before acknowledging the commit to the client. This is achieved as follows:

1. The transaction maintains a bitmap tracking all the disks that the transaction has affected.
2. When a data page is updated by the transaction, in addition to keeping track of the page itself, the bitmap also records the associated disk.
3. While processing a commit, after the data pages are written, the bitmap is consulted to issue a FLUSH_CACHE command for each of the disks that the transaction has affected.

9. Transaction Durability

As explained in Section 2, when the disk has a 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 solution is to address recoverability, we also provide an upper bound on the delay in transaction durability by issuing FLUSH_CACHE commands at a user-specified frequency.

10. 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.

10.1 Hardware

All the performance measurements are performed on a low-end machine having an Intel® Pentium® 4 3.4 GHz HT CPU 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 an 8MB disk controller cache.

10.2 Personal Information Management Systems

For evaluating our scheme for personal information management systems, we simulated the workload of a rich email client application that uses a database to store messages. One of the common operations performed by the email client is synchronizing the local mail box with that on the server by downloading email messages, calendar items and contacts. Table 1 compares the performance of the simulated email client with and without write-ahead logging.

Number of Entities	Time (in seconds)	
	WAL not enforced	WAL enforced using our scheme
100	5.967	6.326
1,000	8.057	8.079
200,000	636.285	645.641

Table 1: Email client workload.

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 being downloaded increases, the overhead decreases. For instance, the overhead is 5% for 100 entities, 2% for 1000 entities and 1% for 200K entities.

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

Table 2: Modified TPC-C benchmark.

10.3 Modified TPC-C benchmark

With the same hardware as described in section 10.1, 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 1.6GB data file and a 2.9 GB log file. Table 2 shows the results of these experiments.

The table compares our solution for providing recoverability with the option of disabling the disk controller cache. Compared to the baseline (Scenario A) where recoverability is not ensured, our solution (Scenario B) incurs only a small overhead (roughly 2%). On the other hand, compared to disabling the disk controller cache (Scenario C), our solution performs roughly 10% better.

Disabling disk controller caching forces every write 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.

11. Conclusion

In this paper, we illustrate the problem of enforcing WAL to guarantee database recoverability on commodity (SATA/IDE) disk 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. We also show that our algorithm does not adversely impact transaction throughput.

12. Acknowledgements

We thank Eric Christensen for his participation in the initial design discussion, Peter Byrne, Eric Christensen, Steve Lindell and Steve Schmidt for valuable guidance during implementation, Neeraj Joshi and Shirley Wang for setting up the scaled-down TPC-C benchmark, and Vitaly Akulov, Sonia Parchani, Avi Levy, Sherry Li and Gayathri Venkataraman for testing the scheme.

13. References

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

[2] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh and Peter Schwarz. *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] Dave Anderson, Jim Dykes and Erik Riedel. *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] Leonard Chung, Jim Gray, Robert Horst and Bruce Worthington. *Windows 2000 Disk IO Performance*, Technical Report MSR-TR-2000-55, September 2000.

[5] Microsoft Corporation. *Disk Subsystem Performance Analysis for Windows.* Windows Hardware Developer Central (http://www.microsoft.com/whdc), March 2004.

[6] Philip A. Bernstein, Nishant Dani, Badriddine Khessib, Ramesh Manne and 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] R. Dominguez and T. Coligan. *SCSI vs. ATA: Interface Comparison.* Technology Brief, Dell Computer, December 1999.

[11] Microsoft Corporation. *SQL Server 2008 Books Online*. http://msdn.microsoft.com/en-us/library/ms130214.aspx.

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

[13] Robin Dhamankar, Hanuma Kodavalla and Vishal Kathuria. *Enforcing Database Recoverability on Disks that Lack Write-Through.* Technical Report MSR-TR-2008-36, March 2008.

[14] Vishal Kathuria, Robin Dhamankar and Hanuma Kodavalla. *Transaction Isolation and Lazy Commit.* In Proceedings of the 23rd International Conference on Data Engineering (ICDE '07), April 2007.

[15] Robin Dhamankar and Hanuma Kodavalla. *InProcDiskSim: Testing Database Recovery on Commodity Disk Drives*. In Proceedings of the Second International Workshop on Testing Database Systems (DBTest '09), Providence, RI, June 2009.

[16] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica and Matei Zaharia. *Above the Clouds: A Berkeley View of Cloud Computing*. Technical Report, EECS Department, University of California, Berkeley, UCB/EECS-2009-28, February 2009.