Write Amplification: An Analysis of In-Memory Database Durability Techniques

Jaemyung Kim
Cheriton School of Computer Science
University of Waterloo
j368kim@uwaterloo.ca

Kenneth Salem
Cheriton School of Computer Science
University of Waterloo
ken.salem@uwaterloo.ca

Khuzaima Daudjee
Cheriton School of Computer Science
University of Waterloo
kdaudjee@uwaterloo.ca

ABSTRACT

Modern in-memory database systems perform transactions an order of magnitude faster than conventional database systems. While in-memory database systems can read the database without I/O, database updates can generate a substantial amount of I/O, since updates must normally be written to persistent secondary storage to ensure that they are durable. In this paper we present a study of storage managers for in-memory database systems, with the goal of characterizing their I/O efficiency. We model the storage efficiency of two classes of storage managers: those that perform in-place updates in secondary storage, and those that use copy-on-write. Our models allow us to make meaningful, quantitative comparisons of storage managers' I/O efficiencies under a variety of conditions.

1. INTRODUCTION

As memory prices continue to drop and memory sizes continue to increase, it has become possible to fit terabyte OLTP databases entirely in memory. Modern database systems can leverage these memory trends to deliver high performance [3, 5]. In-memory database systems [2, 8, 4] can execute transactions an order of magnitude faster than conventional database systems.

While keeping the database in memory can greatly reduce read latencies, writes need to be made durable to ensure that the system can recover to a consistent state after a failure. Although in-memory database systems usually perform less write I/O due to the absence of disk-resident B-trees and heap files, high performance transaction processing generates a correspondingly high rate of update I/O.

For restart recovery, disk-resident database systems read log records generated since the last checkpoint so as to recover to a consistent database state. On the other hand, in-memory database systems have to read the entire database as well as the log, and rebuild indices on database start-up. These steps may result in a long restart recovery time. Limiting the amount of data that must be read during recovery can be valuable for controlling the recovery time.

In-memory database systems generally use one of two paradigms for persisting updates. One simple method is to write in-memory data to disk by using a direct mapping of an in-memory representation to persistent storage [4]. We refer to this method as as *Update-In-Place* (UIP). Physical data independence implies that the positions of updated objects in persistent storage do not necessarily match the order of the updates, leading to random writes. To address this problem, systems such as H-Store [5] and SiloR [9] checkpoint not only updated objects but also clean objects. This form of UIP using snapshotting (UIP-S) optimizes performance through the conversion of writes from random to sequential.

Unlike UIP and UIP-S, in-memory database systems such as Hekaton [2] follow a different paradigm, under which all updates are appended to a series of checkpoint files on disk. This method, which we refer to as *Copy-On-Write* (COW), is advantageous in that it converts random I/O into sequential I/O. In addition, to alleviate the disruptive impact of checkpoints, some COW systems such as Hekaton spill out I/Os continuously. However, COW systems incur overhead due to the need to garbage collect outdated objects in checkpoint files. Systems such as RAMCloud [7] avoid this overhead by using copy-on-write both in memory and on persistent storage. Garbage collection can then be performed on the in-memory representation, and then persisted afterwards. We refer to this variant as COW-M, and to the original as COW-D.

In this paper, we study these different persistent storage management techniques for in-memory database systems. Our goal is to characterize the *I/O efficiency* of these techniques, which we quantify as *write amplification*. Intuitively, write amplification is the ratio of the rate with which data are written to persistent storage to the rate at which the database is updated. Techniques with high write amplifications have poor I/O efficiency; They must be able to perform a lot of I/O to sustain a given database update rate.

We formulate a simple analytical model for database I/O and use it to measure the write amplication factor for each storage management scheme shown in Table 1. We study the write amplification factor under both a random update model and a skewed update model. Our model also allows us to distinguish between random and sequential I/O, so that we can weigh random I/O more heavily than sequential I/O when calculating write amplification.

Our model allows us to estimate to compare the write amplification factors of various storage management techniques, which is valuable for several reasons. First, the write amplification characteristics provide us with some insight into the different natures of update-in-place and copy-on-write storage managers, and to the settings in which each may be most appropriate. Second, improved I/O efficiency can contribute to lower cost for operating a database management system, since such a system must be configured with enough I/O capacity to sustain its anticipated performance. Third, in situations in which I/O capacity is constrained, improved I/O efficiency may lead to better system performance.

It is also worth pointing out what our model does *not* do. Most

IMDM '15, August 31 2015, Kohala Coast, HI, USA

DOI: http://dx.doi.org/10.1145/2803140.2803141

Storage Scheme	Description	Examples
UIP	persistent DB plus log, checkpoint dirty pages	Shore-MT
UIP-S	persistent DB plus log, checkpoint entire DB	H-Store, SiloR
COW-D	log-structured persistent DB	Hekaton
COW-M	log-structured persistent and in-memory DB	RAMCloud

Table 1: Storage Management for In-Memory Database Systems

importantly, our model does not attempt to directly predict the performance (e.g., transaction throughput or latency) of an in-memory DBMS. Such systems are complex and difficult to model in their full generality, and performance depends on many factors in addition to I/O efficiency. In particular, many systems try to hide as much I/O latency as possible by performing I/O operations outside of the critical transaction execution path. Thus, we expect that it would be extremely difficult to analytically model the effect of I/O on, say, transaction latency. Even if this were possible, the resulting model would be tied closely to a particular DBMS. Instead, we have developed a more abstract model that focuses on I/O efficiency, and that tries to capture only the key characteristics of copy-on-write and update-in-place storage managers, ignoring many implementation-specific details.

The rest of this paper is organized as follows: Section 2 describes the systems from Table 1; Section 3 presents our write amplification analytical model for database I/O, and Section 4 presents the results of our analysis using this model. In Section 5, we generalize the model from Section 3 to account for update skew. Section 6 concludes the paper.

2. MANAGING PERSISTENT STORAGE

Most OLTP DBMSs append updates to persistent logs, which provides durability without overwriting the persistent database. Since the storage capacity of the log is bounded, UIP and COW systems periodically or continuously *clean* their logs. This log cleaning operation in UIP systems is accomplished by *checkpointing* while in COW systems it is accomplished by *log cleaning*, a form of garbage collection. When done in batches (of updates), log cleaning is efficient though lazy log cleaning can result in longer restart recovery time due to the possibility of having a greater number of updates to process from the log. Thus, the frequency of log cleaning can impact normal system operation, I/O efficiency, and restart recovery time. This tradeoff between recovery time (due to log size) and I/O per update (IOPU) is present when providing in-memory DBMS persistence. Next, we describe how UIP and COW schemes (outlined in Table 1) manage their persistent storage.

2.1 Update-In-Place Scheme

In UIP, persistent storage holds a copy of the in-memory database and the log to which committed updates are written. Restart recovery time is proportional to the size of the log, so its size is capped by checkpointing. During checkpointing, the DBMS copies updated objects from its in-memory database to its persistent database. In addition to this copy operation, the DBMS inserts a "checkpoint" record in the log to indicate the position where the last log records were checkpointed. After copying updated objects, the DBMS can delete its corresponding log records.

In conventional disk-based DBMSs, the database consists of pages where each page contains multiple objects. Updated pages in memory are copied to persistent storage through checkpointing or when the buffer eviction policy is applied – only some pages are buffered in memory because the size of memory space is limited. I/O resulting from checkpointing or buffer evictions may be random writes.

In contrast, two representative UIP in-memory systems, H-Store [1] and SiloR [9], copy the entire in-memory database including both dirty and clean objects. We call this type of checkpoint a snapshot checkpoint under the *Update-In-Place-Snapshot* scheme (UIP-S). The advantage of UIP-S is in its simplicity and its ability to convert random writes to sequential database writes. The DBMS does not need to keep track of which objects are dirty, and checkpointing does not result in random writes. An additional benefit is that UIP-S can easily support the creation of multiple historical backups of the database, providing the ability to revert to a previous backup version in case of an unrecoverable failure. However, because a snapshot checkpoint copies both dirty and clean objects, the amount of checkpoint I/O is proportional to the database size, regardless of the update frequency.

2.2 Copy-On-Write Scheme

In the copy-on-write scheme, persistent storage holds only a log, which is itself the database (this is sometimes referred to as log-structured storage). Like UIP, the log size needs to be bounded through log cleaning so as to prevent the log from growing indefinitely as the database is updated. In COW implementations, objects in the log are usually grouped together into a segment. When the log size reaches a set threshold, "live" objects from segments with the fewest live objects are re-inserted into the log while the rest are removed to reduce the total size of the log.

Some COW-based systems may implement additional features. For example, Hekaton holds an additional shared log for both memory and disk DBMSs so as to work seamlessly with legacy disk-based DBMSs [2]. Hekaton checkpoints continuously by dumping updated objects to data files. In addition, Hekaton appends IDs of deleted (invalidated) objects to delta files. Each data file has a corresponding delta file that can be used to avoid loading deleted objects to the in-memory database during restart recovery. The specifics of log cleaning also vary among COW systems. Hekaton merges two adjacent data files and writes the merged data file into the same position from where the two data files were located. This is to ensure that segments are placed in epoch order in the system. In contrast, RAMCloud [7] merges multiple segments from arbitary locations, and inserts the merged segment into the head of the log as merged objects are treated the same as the latest objects updated by the application.

RAMCloud maintains its in-memory COW representation also on disk (we call this the COW-M scheme). Instead of updating in-place in memory, RAMCloud appends updates without overwriting existing objects. Appending valid objects from old segments to new segments during log cleaning allows RAMCloud to avoid reading all of the old segments. A COW scheme would perform best when persistent storage is more than twice the size of the database in general, though the memory may not reach this size. To address this, RAMCloud compacts segments that contain the highest percentage of invalid objects, thereby avoiding immediate log cleaning on disk when memory space is limited.

3. WRITE AMPLIFICATION MODELS

To allow us to compare the I/O efficiency of the persistent storage management schemes shown in Table 1, we have developed a simple analytic database I/O model that is general enough to capture all

parameter	definition	description
D	input	database size
α	input	storage scale factor
S	αD	persistent storage capacity
P	input	page size
L	$S-D$	persistent log capacity (UIP)
P_{hot}	input	update skew parameter
D_{hot}	$P_{hot}D$	database hotspot size

Table 2: Model Parameter Summary

of the schemes. Our model predicts the *write amplification* that will occur under each of the storage management schemes. Intuitively, write amplification is the ratio of the I/O rate to the database update rate. For example, a write amplification of two means that the I/O to persistent storage will occur at twice the rate with which the (in-memory) database is updated. Write amplification is a measure of I/O inefficiency, i.e., lower values are better. A storage management scheme with low write amplification will place less demand on persistent storage than one with higher write efficiency.

In general, we expect write amplification to depend on the amount of persistent storage space that is available. For this reason, our model parameterizes the amount of persistent storage, and all comparisons that we make among the storage management schemes are made on a constant-space basis. Specifically, we model the database as a set of objects, with cardinality D, and we assume that the capacity of persistent storage is αD objects, where $\alpha \geq 1$. Thus, α controls how fully the database utilizes the persistent storage space. Larger values of α result in lower utilization of persistent storage.

Another reason for parameterizing persistent storage space in our model is that the amount of persistent storage is related to recovery time. Under all of the storage management schemes that we consider, it is necessary to read the contents of persistent storage to fully recover from a failure that results in the loss of the in-memory copy of the database. Thus, to the extent that I/O is the performance bottleneck during system recovery, αD (the persistent storage capacity) can serve as an indirect measure of recovery time, and our constant space comparisons of storage management schemes are also constant recovery time comparisons.

Our model assumes that the in-memory database is updated at a fixed rate of object updates per unit time, and that objects are selected randomly for update. For the purposes of the analyses in this section, we assume that all objects are equally likely to be updated. In Section 5, we generalize our analyses to account for update skew, i.e., settings in which some objects are more likely to be updated than others.

Table 2 summarizes our model parameters, including α and D. The remaining parameters will be introduced as we present our analyses.

3.1 Update-In-Place (UIP)

To estimate the write expansion for UIP storage managers, we use a simple "stop-and-copy" checkpointing model. Under this model, updates are logged without checkpoint until the available log space is filled. Then the database temporarily halts update processing, performs a checkpoint, clears the persistent log, and then resumes update processing. Let $L = \alpha D - D = (\alpha - 1)D$ represent the amount of log space in persistent storage. Under the stop-and-copy model, the DBMS must checkpoint at least once every L updates since each update consumes one unit of log space.

The UIP storage manager divides the database into pages of size P objects. Assuming a uniform distribution of updates over pages, the probability of a database page receiving at least one update during a checkpoint interval, which we refer to as P_{dirty}, is given by

$$P_{dirty} = 1 - \left(\frac{D-P}{D}\right)^{L} = 1 - \left(\frac{D-P}{D}\right)^{(\alpha-1)D} \quad (1)$$

Thus, the expected amount of data that will need to be written to the persistent database during each checkpoint interval is $P_{dirty}D$. In addition, the DBMS will write L objects to the log during the interval. The write amplification factor for UIP, which we refer to as $IOPU_{UIP}$ (I/O Per Update) is the ratio of the number of objects written to persistent storage during each interval to the number of updates to the in-memory database:

$$IOPU_{UIP} = \frac{P_{dirty}D + L}{L} = \frac{P_{dirty}}{\alpha - 1} + 1 \quad (2)$$

The UIP-S storage manager is similar to UIP, except that it writes the entire database to persistent storage during each checkpoint. Thus, the write amplification factor for UIP-S is

$$IOPU_{UIP-S} = \frac{D + L}{L} = \frac{\alpha}{\alpha - 1} \quad (3)$$

3.1.1 Accounting for Random I/O

An important property of UIP storage managers is that I/O for database checkpoints is random. In contrast, all I/O generated by UIP-S and COW storage managers is sequential. Indeed, this is one of the key factors that has motivated COW storage management. Thus, we would like to adjust the UIP model to account for random I/O. To do this, we apply a penalty factor $\rho \geq 1$ to the I/O cost of UIP checkpoints. Intuitively, if the I/O operations performed by the UIP checkpoint are, for example, 3 times as expensive as sequential I/O operations, then we want ρ to be three.

We expect the relative cost of a UIP checkpoint's I/O operations to (ρ) to depend on two factors. The first is P, the size of the database pages, since writing larger pages may be more efficient than writing smaller pages. The second is P_{dirty}, which determines the number of pages written during each checkpoint. Because of I/O scheduling, I/O performed by checkpoints that write many pages can be more efficient than I/O performed by checkpoints that write fewer pages. Thus, we define ρ as follows:

$$\rho = P_{dirty} + (1 - P_{dirty}) \cdot \text{Penalty}(P) \quad (4)$$

where Penalty(P) is a device-specific function that characterizes the cost of random writes of pages of size P, relative to the cost of sequential writes of pages of the same size. Thus, as P_{dirty} approaches one, and each checkpoint writes more pages, ρ will approach one. Conversely, as P_{dirty} approaches zero, and the checkpoint writes few pages, ρ will approach Penalty(P).

With the introduction of ρ, the revised model of the write amplification of UIP becomes

$$IOPU_{UIP} = \frac{\rho P_{dirty}D + L}{L} = \frac{\rho P_{dirty}}{\alpha - 1} + 1 \quad (5)$$

We can estimate Penalty(P) for specific types of I/O devices using simple calibration tests which measure sequential and random I/O bandwidth at different page sizes. We then take Penalty(P) to be the ratio of measured sequential write bandwidth to measured random write bandwidth for pages of size P. For example, Figure 1 shows the Penalty(P) determined using this method for two devices, an Intel SSDSC2BA200G 200GB SATA SSD, and a Seagate ST1000NM0033 1TB 7200RPM SATA disk drive. For the

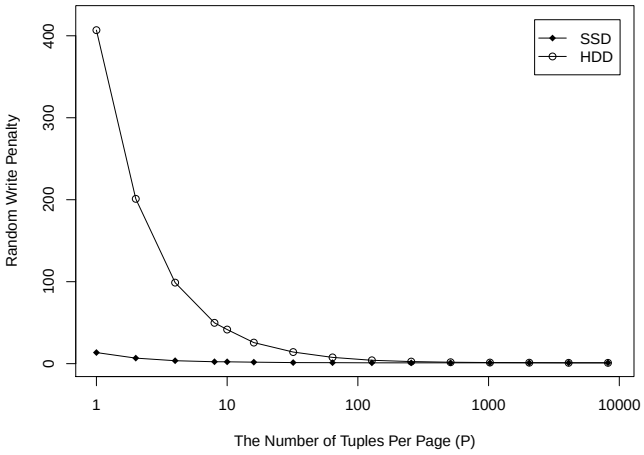


Figure 1: Random Write Penalty for Two Devices

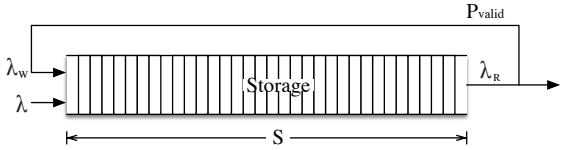


Figure 2: Copy-on-write model

UIP results presented in the rest of this paper, we have used these two penalty functions to model UIP write amplification on SSDs and disk drives (HDDs).

3.2 Copy-On-Write (COW)

COW storage managers use all of persistent storage as a log. We begin by modeling the behavior of COW-D, and then show how this model can be revised to model COW-M.

To estimate write amplification for COW-M, we model its use of persistent storage as shown in Figure 2. λ refers to the rate of database updates in the in-memory database. As these updates occur, the storage manager appends a log record for each update to one end of the log - the left end in Figure 2. These log insertions occur at rate λ. To ensure that the log does not overflow persistent storage, the storage manager removes entries from the other end of the log, at rate λ_R. It checks each removed entry to determine whether the object version that it describes is still valid, or is garbage. If the version is still valid, the removed log record for that version is re-inserted at the head of the log. Otherwise it is discarded. We use P_{valid} to represent the probability that a removed object version is still valid, and λ_W to refer to the *re-insertion rate*, i.e., the rate with which valid versions are re-inserted, as shown in Figure 2.

With these definitions, we can define the write amplification factor for COW storage managers, $IOPU_{COW-D}$, as:

$$IOPU_{COW-D} = \frac{\lambda + \lambda_R + \lambda_W}{\lambda} \tag{6}$$

Note that, unlike UIP storage managers, the I/O for a COW-D storage manager includes reads as well as writes, since the log must be read for garbage collection. We include both reads and writes to persistent storage in the write amplification factor.

In steady state, it must be the case that $\lambda_R = \lambda + \lambda_W$.

$$IOPU_{COW-D} = \frac{2(\lambda + \lambda_W)}{\lambda} \tag{7}$$

From our definitions, λ_W depends on λ and P_{valid}. Thus, we can re-write Equation 7 as

$$IOPU_{COW-D} = \frac{2}{1 - P_{valid}} \tag{8}$$

To estimate P_{valid}, consider an arbitrary log entry inserted into the log by the storage manager. After $\alpha D - 1$ subsequent insertions, that log entry will be at the tail of the log, about to be cleaned. Let U represent the number of those subsequent insertions that are *not* re-insertions. We can write

$$P_{valid} = \frac{\lambda_W}{\lambda_R} = \frac{\lambda_W}{\lambda + \lambda_W} = \frac{\alpha D - 1 - U}{\alpha D - 1} \tag{9}$$

Since only one valid version of each database record exists in the log at any time, the maximum number of re-insertions that can occur before our arbitrary log entry reaches the end of the log is $D-1$. Thus, we also know that $(\alpha - 1)D \leq U \leq \alpha D - 1$. For arbitrary update distributions, these lower and upper bounds on U give corresponding upper and lower bounds on $IOPU_{COW-D}$.

For the special case in which updates are uniformly distributed over the database objects, we can also relate P_{valid} to U by the following expression.

$$P_{valid} = \left(\frac{D-1}{D}\right)^U \tag{10}$$

This represents the probability that a newly logged update is not invalidated before it reaches the tail of the log. Using our general bounds on U, we can solve Equation 10 numerically for P_{valid}, and hence obtain $IOPU_{COW-D}$ from Equation 8.

3.2.1 Modeling COW-M

The behavior of COW-M is similar to that of COW-D, except that it does not need to read the log from persistent storage to determine which log entries require re-insertion. Thus, we can write:

$$IOPU_{COW-M} = \frac{\lambda + \lambda_W}{\lambda} \tag{11}$$

Following the same arguments used for the COW-D model, we can write:

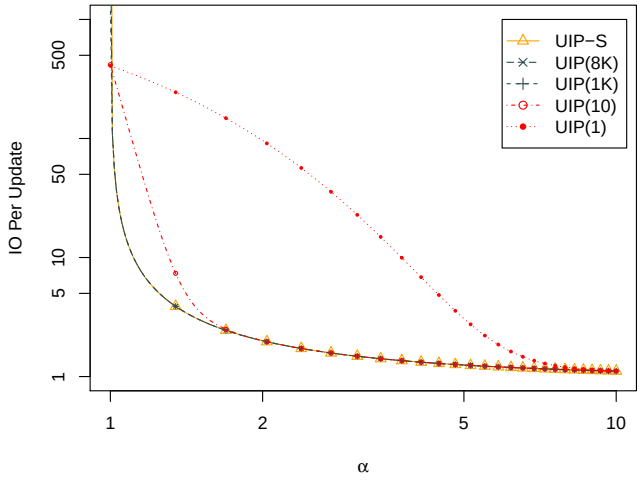
$$IOPU_{COW-M} = \frac{1}{1 - P_{valid}} \tag{12}$$

To calculate $IOPU_{COW-M}$ for the special case of a uniform update distribution, we calculate P_{valid} using Equation 10, and use this value to determine $IOPU_{COW-M}$ from Equation 12.

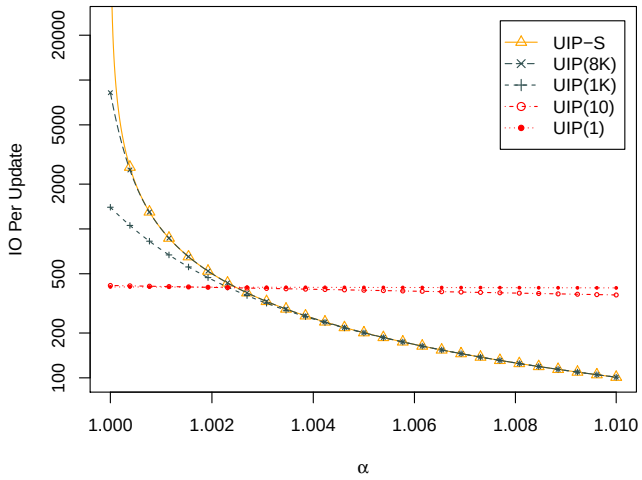
4. EVALUATION

Before comparing UIP and COW storage managers, we begin with an analysis of the effect of page size on UIP. Figure 3 shows the write expansion factor (IOPU) for UIP storage managers as functions of α and the page size, P. Figure 3(b) shows the same data as Figure 3(a), but focused on a range of small values of α. For Figure 3, we have assumed that persistent storage is implemented with hard disks, and the write expansion factors have been adjusted with a random I/O penalty as described in Section 3.1.1. In this and all other figures presented in this paper, we have fixed the database size, D, at 10^6 objects.

The message from this figure is clear. Unless the log is very small, UIP-S is the preferred update-in-place storage manager. Recall that the available log size under our model is $(\alpha - 1)D$. Thus,



(a) Large α



(b) Small α

Figure 3: UIP: Uniform Random Updates, HDD

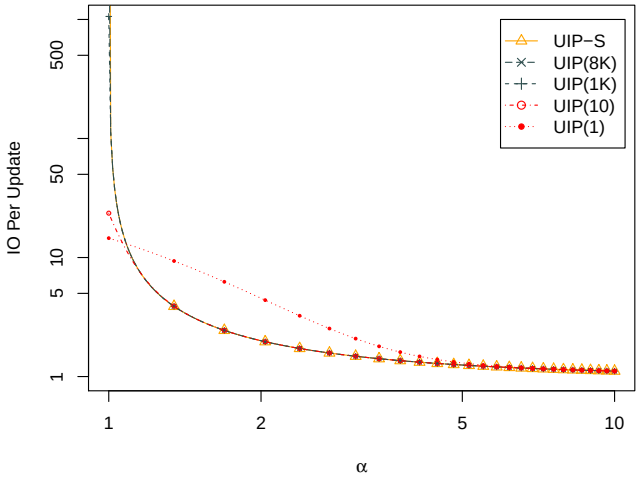


Figure 4: UIP: Uniform Random Updates, SSD

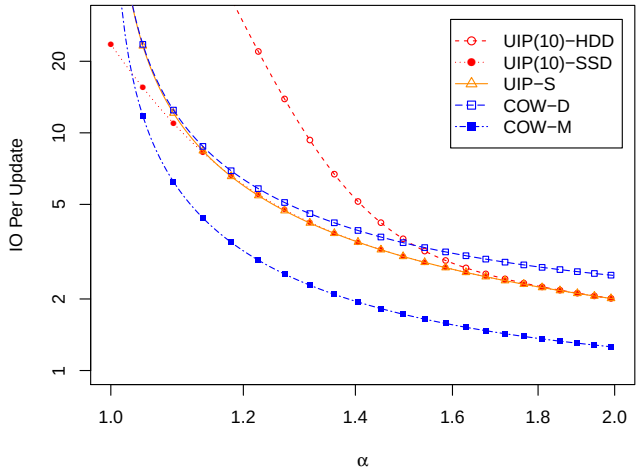


Figure 5: UIP vs. COW, Uniform Random Updates

small values of α correspond to storage managers configured to use little log space – typically to allow fast recovery from failures. All storage managers have high write amplifications in such small-α configurations, but UIP may be not as bad as UIP-S for extremely small α.

Figure 4 shows the same comparison, but under the assumption that persistent storage is implemented with SSDs. Here, UIP (with small pages) beats UIP-S for a wider range of (small) *alpha* values, since the penalty for non-sequential updates is smaller on SSDs. However, as was the case with HDDs, larger values of α drive write amplification down quickly because checkpointing can occur less frequently, and UIP quickly converges to UIP-S under those conditions.

4.1 UIP vs. COW

Figure 5 shows the IOPU of the two COW storage managers, as well as several of the UIP managers. COW-M has lower write amplification than COW-D, and converges to the ideal write amplification factor of 1 as α grows. The important distinction between COW-M and COW-D that accounts for this is that COW-D must read from persistent storage to perform cleaning, while COW-M does not. Thus, for large α, $IOPU_{COW-D}$ converges to 2, rather than 1, since every object update will result in one object being written (to commit the update) and one being read (to clean the committed update from the log). Of course, the disadvantage of COW-M is that it requires a log-structured in-memory data representation that mirrors the representation in persistent storage.

For all α, UIP-S write-amplification is either comparable to, or slightly better than, that of COW-D. Both types of storage managers perform only sequential I/O. However, on an equal-space basis, UIP checkpointing is a more I/O-efficient way to limit log size than COW-D's log cleaning. For very large values of α, $IOPU_{UIP-S}$ converges to the ideal value of 1. Finally, we note that in situations where fast recovery is critical (small α), UIP(10) is competitive with even COW-M, provided that there is not a large penalty for random I/O.

5. ACCOUNTING FOR SKEW

In Section 3, we presented write amplification models that were developed under an assumption of uniformly distributed database updates. In this section, we generalize those models to account for skewed updates. We use a Pareto skew model, in which P_{hot}% of the database receives $(100 - P_{hot})$% of the updates. We vary the

parameter P_{hot} to control the skew.

5.1 Update-In-Place (UIP)

The write amplification model for the update-in-place storage managers is the same with skew as without, except for the calculation of P_{dirty}. With the introduction of skew, we assume that hot objects and cold objects are clustered, resulting in hot pages and cold pages. The probability that a page is dirty will depend on whether it is hot or cold. With skew, we replace our previous calculation of P_{dirty} (Equation 1) with the skew-aware calculation given by Equation 13.

$$
\begin{aligned}
P_{dirty} &= P_{dirty|hot}P_{hot} + P_{dirty|cold}(1-P_{hot}) \\
P_{dirty|hot} &= 1 - \left(\frac{D_{hot}-P}{D_{hot}}\right)^{(1-P_{hot})L} \\
P_{dirty|cold} &= 1 - \left(\frac{(D-D_{hot})-P}{(D-D_{hot})}\right)^{P_{hot}L}
\end{aligned}
\tag{13}
$$

5.2 Copy-On-Write (COW)

Incorporating skew into the copy-on-write model is more challenging. The write amplification for copy-on-write storage managers depends on P_{valid}, the probability that determines the log re-insertion rate. Equation 9 for P_{valid} holds for all update distributions. However, Equation 10 does not. With skewed updates, cold objects are more likely to be re-inserted than hot objects since they are less likely to be over-written. Thus, P_{valid} is a complex function of P_{hot} and the log size.

Rather than solving analytically for $IOPU_{COW-D}$ and $IOPU_{COW-M}$ when updates are skewed, we instead use discrete-event simulations of the COW model (Figure 2) to estimate both write amplification factors.

5.3 Evaluation of Skew's Impact

Figures 6 compares the I/O per update (IOPU) for both UIP and COW storage managers at several different levels of skew.

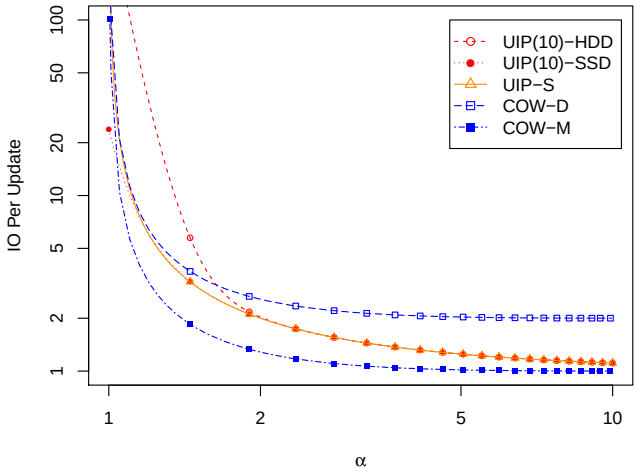
The UIP-S storage manager write amplification is insensitive to skew, since it always checkpoints the entire database. However, the write amplification of the other UIP storage managers is very sensitive. Higher skew is beneficial to UIP storage managers because it reduces the amount of data that must be written during each checkpoint. When α is small, UIP to SSDs is a winning strategy.

In contrast, skew has a mildly negative effect on the COW storage managers. Cold objects have relatively long lifetimes, and thus are more likely to have to be salvaged and re-inserted during log cleaning, increasing the I/O cost of the cleaner. Some techniques have been proposed to address this issue [6], but they are not included in our analysis.

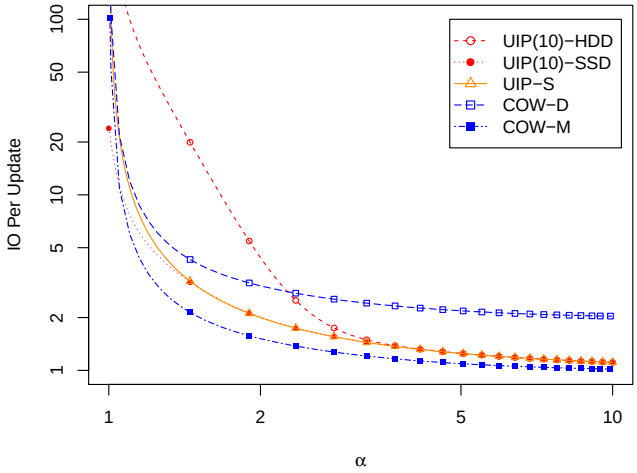
6. CONCLUSION

In this paper, we have presented a write amplification model for in-memory database storage managers. Our model allows us to quantify and compare the I/O efficiency of two broad classes of storage managers: those that use update-in-place, and those that use copy-on-write. Our model is parameterized by space efficiency, which makes it easy for us to consider the write amplification of different storage managers on a constant-space basis.

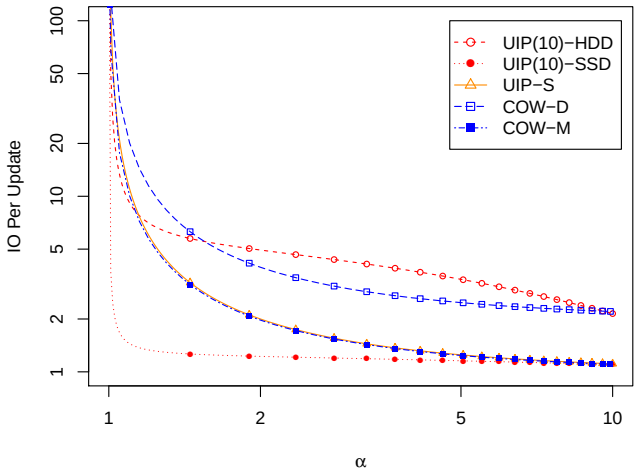
Among the storage managers we tested, COW-M was the most I/O efficient (lowest write amplification) in many settings. However, unlike the other storage managers, COW-M requires a log structure in memory that mirrors the log structure in persistent storage, which may be a significant limitation. For example, COW-M



(a) Light Skew (60-40)



(b) Medium Skew (80-20)



(c) High Skew (99-1)

Figure 6: Write Amplification Under Skew Distribution

may preclude the storage manager from using a cache-friendly in-memory data organization that is optimized for efficient reads.

Among the storage managers that do not constrain the in-memory organization of the database, UIP-S is an appealing option in many settings. It is always at least as I/O efficient as COW-D, and in most situations is also at least as good as page-based UIP alternatives. UIP-S is also insensitive to update skew, and it requires only sequential writes to persistent storage.

Page-level UIP schemes, which are commonly used in disk-based database systems, make sense for in-memory databases only in very specific situations. For example, they can be effective when the capacity of persistent storage (or recovery time) is tightly constrained, provided that persistent storage can support random writes efficiently. They may also be effective if database updates are highly skewed.

7. REFERENCES

[1] *H-Store Manual http://hstore.cs.brown.edu/documentation/*.

[2] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proc. ACM SIGMOD*, pages 1243–1254, New York, New York, USA, June 2013. ACM.

[3] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. Oltp through the looking glass, and what we found there. In *Proc. ACM SIGMOD*, pages 981–992, New York, NY, USA, 2008. ACM.

[4] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *Proc. EDBT*, pages 24–35, New York, NY, USA, 2009. ACM.

[5] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *The VLDB Journal*, 1(2):1496–1499, 2008.

[6] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM SIGOPS OSR*, 1991.

[7] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured memory for DRAM-based storage. In *Proc. USENIX FAST*. USENIX Association, Feb. 2014.

[8] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proc. SOSP*. USENIX Association, Nov. 2013.

[9] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *Proc. SOSP*, pages 465–477. USENIX Association, 2014.