Deferred Updates for Flash Based Storage

Biplob Debnath *, Mohamed F. Mokbel *, David J. Lilja *, David Du *

*Department of Electrical and Computer Engineering
*Department of Computer Science and Engineering
University of Minnesota, Twin Cities, USA.
Email: biplob@umn.edu, mokbel@cs.umn.edu, lilja@umn.edu, du@cs.umn.edu

Abstract—The NAND flash memory based storage has faster read, higher power savings, and lower cooling cost compared to the conventional rotating magnetic disk drive. However, in case of flash memory, read and write operations are not symmetric. Write operations are much slower than read operations. Moreover, frequent update operations reduce the lifetime of the flash memory. Due to this asymmetric behavior, flash based storage is particularly attractive for the read-intensive workloads due to the faster read performance, while it can produce poor performance when used for the update-intensive workloads due to excessive random write operations. This paper aims to improve write performance and lifetime of flash based storage for the update-intensive workloads. In particular, we propose a new hierarchical approach named as deferred update methodology that significantly improves the update processing overhead. Instead of directly updating the data records, we buffer the changes due to update operations as logs in two intermediate in-flash layers; and finally apply multiple logs in bulk to the data records. Experimental results show that our proposed methodology significantly improves update processing time and the longevity of the flash based storages.

I. INTRODUCTION

NAND flash memory is increasingly adopted as main data storage media in mobile devices, such as PDAs, MP3 players, cell phones, digital cameras, embedded sensors, and notebooks due to its superior characteristics such as smaller size, lighter weight, lower power consumption, shock resistance, lesser noise, non-volatile memory, and faster read performance [8], [11], [14], [16], [20]. Recently, to boost up I/O performance and energy savings, flash based Solid State Disks (SSDs) are also being increasingly adopted as a storage alternative for magnetic disk drives by laptops, desktops, and enterprise class servers [2], [11], [14], [15], [17], [19], [20], [23]. Due to the recent advancement of the NAND flash technology, it is expected that NAND flash based storages will greatly impact the designs of the future storage subsystems [2], [4], [14], [20].

A distinguishing feature of flash memory is that read operations are very fast compared to magnetic disk drive. Moreover, unlike disks, random read operations are as fast as sequential read operations as there is no mechanical head movement. However, a major drawback of the flash memory is that it does not allow in-place updates (i.e., overwrite). Figure 1 gives an overview of a flash-based storage device. In flash memory, data are stored in an array of flash blocks (as shown in Figure 1). Each block spans 32-64 sectors, where a sector is the smallest unit of read and write operations. Sector write operations in a flash memory must be preceded by an erase operation. Within a block, sectors must be written sequentially (in low to high address order) [2]. The in-place update problem becomes complicated as write operations are performed in the sector granularity, while erase operations are performed in the block granularity. The typical access latencies for read, write, and erase operations are 25 microseconds, 200 microseconds, and 1500 microseconds, respectively [2]. In addition, before the erase is being done on a block, the live (i.e., not over-written) sectors from that block need to be moved to pre-erased blocks. Thus, an erase operation incurs lot of sectors read and write operations, which makes it a performance critical operation. Besides the asymmetric read and write latency issue, flash memory exhibits another limitation: a flash block can only be erased for limited number of times (e.g., 10K-100K) [2]. As a result, frequent block erase operations reduces the longevity of the flash memory.

Faster read performance of the flash memory will be particularly good to speed up the read-intensive type workloads, e.g., decision support systems (DSS). However, flash memory can produce poor performance when used for other workloads that require frequent random update operations. Examples of such workloads include online transaction processing (OLTP), mobile applications, and spatio-temporal applications. Such update-intensive applications perform a lot of small-to-moderate size random write operations that are much smaller than the flash sector size. This would be very problematic for the flash memory as once data is written in a flash sector, no further data can be written in the unused portion of that sector unless the entire block containing that sector is erased. That would be very costly as an erase operation is much

![Diagram of Flash Memory Based Storage](https://example.com/fig1.png)

Fig. 1. Flash Memory Based Storage
slower than a write operation. Moreover, frequent erasure of the blocks will decrease the lifetime of the flash memory. As shown in Figure 1, the flash translation layer (FTL) is an intermediate layer that hides the internal details of flash memory and allows existing disk-based application to use flash memory without any significant modifications. However, recent studies show that existing FTL-based schemes can not efficiently handle random write operations [11], [13], [16]. Since update-intensive workloads are random in nature, current FTL-based schemes will not be very effective for them. It is the goal of this paper to improve write performance and longevity of the flash memory by overcoming the limitations of the current FTL-based schemes to efficiently support update-intensive applications.

In this paper, we propose a novel hierarchical update processing strategy, named deferred update methodology, which significantly improves the in-place update processing overhead and longevity of the flash memory. Our goal is to reduce the number of expensive erase operations due to the processing of in-place update operations through two intermediate flash storage layers. The main idea is that we always write the changes due to newly incoming updates as logs to the first intermediate layer. Once first layer is full, to make free space we populate the logs from the first to the second layer. The first layer acts as a scratch area for the second layer. Finally, when the second intermediate layer is also full, we populate its contents into their actual locations in the flash erase units. These two layers help to batch a set of update logs for the same erase unit together. Finally, we can apply them at once. This results in a huge saving of erase operations where an erase unit is erased only once for a set of bulk updates. Since erase is the most expensive operation, therefore reduction of the number of erase operations helps to improve write performance. On the other hand, this will also help to increase the lifetime of the flash memory due to the limited number of erase operations allowed per block.

The deferred update methodology raises the challenge of data retrieval from the flash memory as a certain record may exist in three different places, i.e., the original record is stored in its original erase unit, then an update of this record is stored as log in the first intermediate layer. Finally, another update of that record may be stored as log in the second intermediate layer. It is nontrivial to retrieve data in an efficient way that achieves a trade-off between the faster data retrieval and the compaction of processing data updates. We use a flash-friendly index to speed up the data retrieval process.

The deferred update methodology needs the following information of the flash memory: block size, sector size, and the block boundaries, which can be easily obtained by querying the flash driver or FTL [20]. Our experimental results show that deferred update methodology performs significantly improve the write processing time and incurs fewer number of erase operations compared to the state-of-the-art in-page logging (IPL) technique [14]. It scales very well with the increase of data size and total number of update operations. The summary of the key contributions of this work are listed as follows.

- A novel update processing methodology named as deferred update methodology to improve slow write performance and lifetime of the NAND flash based storages. The cost of achieving such improvements is only few flash memory blocks. Furthermore, we propose a new technique that complements the deferred update methodology to efficiently retrieve data.
- A thorough theoretical analysis of the trade-offs in terms of erase operations, space overhead, and data retrieval overhead for different alternative designs compared to the deferred update methodology. This analysis can be used as a preprocessing step before using flash based storage for a certain workload.

The remainder of the paper is organized as follows: Section II describes the related work. Section III describes the deferred update methodology in detail. Section IV gives analytical models of the different alternative designs. Section V explains our experimental results. Finally, Section VI concludes the discussion.

II. RELATED WORK

There is substantial recent interest in utilizing flash memory for non-volatile storage in applications including databases [1], [3], [6], [9], [13]–[15], [18], [22]–[24] and sensor networks [20], [21], [25]. Here, we are discussing the works that are related to the writing issues of the flash memory. The existing works can be classified into two main categories. (1) designing flash-friendly data structures. For example, Micro-Hash [25], FlashDB [21], random sampling data structure [20], FD-tree [18], and Lazy-Adaptive Tree [1] propose new or modified index structures for flash based storage. However, these works cannot be directly extended to improve flash memory’s update processing problem as they mainly target specific index structures. In contrast, our goal is to design a generic solution which helps database table spaces as well as index structures. (2) Improving update processing performance for the flash based database servers. This includes in-page-logging (IPL) technique [14]. Our work also falls into this category. In the rest of this section, we discuss IPL in detail and distinguish our work from it.

The state-of-the-art technique of handling updates in flash memory is the in-page logging (IPL) approach [14]. The main idea of IPL is to reserve one of data pages in a flash erase unit as log page for storing the update logs. Each page consists of multiple flash sectors. When a data page becomes dirty, the changes are recorded as update logs in an in-memory update log sector. Once the log sector becomes full or dirty data page is evicted from the buffer, then the in-memory update log sector is written to the corresponding log page. Whenever a log page becomes full, the update logs in the log page are combined with the original data records in the data pages in that erase unit. The first problem of IPL is that if the data pages have very little update locality, then in-memory log sectors will contain small amount of data, as a result log page space will be under-utilized. This space under-utilization accelerates
frequent erase operations, which will slow down performance and affects the lifetime of the flash memory. Second problem with IPL is that if power goes off, the update logs stored in the memory will be lost, thus data inconsistency problem will arise. Our work in this paper targets to develop an efficient update processing methodology that (1) reduces the number of erase operations, (2) increases the space utilization, and (3) avoids data consistency problem.

III. DEFERRED UPDATE METHODOLOGY

In this section, we present the deferred update methodology that aims to improve write performance and increase lifetime of the flash-based database systems, by minimizing the number of erase operations. The outline of this section is as follows. Section III-A gives an overview of our methodology. Section III-B gives a brief description of the update log, timestamp, and index structure. Section III-C explains in detail how update operations (i.e., INSERT, UPDATE, and DELETE) are processed. Section III-D explains how queries are processed.

A. System Overview

The main idea of deferred update methodology is to process updates through an intermediate two-level storage hierarchy consisting of an update memo and log page(s). This intermediate layer helps to reduce the number of expensive erase operations. Conceptually, we group the database pages by the erase unit containing them and named as data blocks, while update memo is a set of erase units which is used as a scratch space. In each data block, some data pages are also reserved for storing update logs. We name these reserved pages as log pages. Figure 2 represents a logical view of an NAND flash memory that employs our proposed deferred update methodology. The boundary of flash memory is depicted by the solid rectangle. In this example, flash memory has eight erase blocks, which are depicted by the fine dotted rectangles. Out of the eight erase blocks, two blocks are reserved as update memo blocks, which are bounded by the solid dotted rectangles; while the remaining six blocks are used as data blocks. In each data block, there are four data pages. One of the data pages will be reserved as log page, which is marked in shaded in gray. Each data page consists of four flash sectors as depicted by the smallest solid rectangles.

The left side of Figure 3 gives an overview of the update processing methodology through update memo and log pages. The update processing has three steps. Step 1: when an update transaction (i.e., INSERT, UPDATE, or DELETE) occurs, the changes made by the transactions are stored as update logs in the available sectors of the update memo blocks. Step 2: when update memo is full, the latest update logs are flushed to the log pages of the corresponding data blocks. A timestamp counter is used to identify the latest update logs. In addition, an index is used to speed up the flushing process. Step 3: when the log pages of a data block are also full, the update logs are stored in-place with the old data records. Without update memo and log pages, for every in-place update operation, we would need to erase a flash block. However, with the help of update memo and log pages, processing of the in-place update operations are deferred. The update memo acts as a buffer and supplies multiple update logs at once to the log pages. These logs are stored compactly in the log pages. Thus, update memo provides the opportunity to compact more update logs and to bulk updates once for each block, which internal log pages cannot do. Overall, update memo and log pages helps to reduce total number of block erase operations due to in-place updates by amortizing cost of single data block erase operation among multiple update operations.

The right side of Figure 3 gives an overview of the query processing steps in deferred update methodology. Queries are processed in the reverse order of the update processing method. It also consists of three steps. Step 1: a raw query result set is generated from the data records stored in the data pages using the traditional query processing techniques. Step 2: initial raw query result set is modified through the update logs stored in the relevant log pages. Step 3: the new result set is modified further by processing the latest update logs stored in the update memo. A flash-friendly hash index is used to expedite this step. The second and third steps ensure that query result is correct.

There is a trade-off between the gain in the update processing performance and query processing overhead. Keeping lot of update logs in the update memo and log pages improves update processing performance. However, this strategy increases the overhead of query processing as we have to scan larger update memo and more log pages to generate correct results. The number of blocks in the update memo and the number log pages in a data block, are the two tuning parameters to control the performance gain of the deferred update methodology.

B. Update Log, Timestamps, and Indexing

In this subsection, we describe the structure of the update log, the various timestamp values, and the indexing data structure.

Update Log. An update transaction represents any INSERT, UPDATE, or DELETE operation. The changes made by the update transaction are stored as update logs. Every update log has five fields: rid, opcode, timestamp, updateLogContents, and previousLogEntryLocation. The rid field indicates the ID of original record for which the update log is intended for.
Thus, this timestamp helps to identify the non-flushed update specific logs. To quickly find the relevant MemoIndex.

Fig. 3. Overview of the deferred update methodology

The **opcode** field indicates the operation code, which denotes INSERT, UPDATE, and DELETE operations, respectively. The **timestamp** field indicates when this update log is inserted in the update memo. The **updateLogContents** field contains the summary of the changes that are supposed to be applied to the original record. In case of DELETE operation, the **updateLogContents** field contains NULL value. The **previousLogEntryLocation** field helps to maintain an index over the update log. In particular, this field helps to build MemoIndex, which will be described later.

**Timestamps.** There can be multiple update logs for the same data record. Therefore, to generate correct result for a query, a mechanism is needed to identify the latest update log of a data record. To this end, each update log is timestamped, when it is stored in the update memo. The timestamp is assigned by a global counter that is monotonically increasing. Once assigned, the timestamp is never changed. The timestamp implicitly places a temporal relationship among the update logs. In addition to timestamp in the update log, one timestamp value, named as logTimestamp, is maintained per data block in the master catalog table. The logTimestamp keeps track of the global timestamp counter value at the time update logs from the update memo are flushed to log pages of that block. Thus, this timestamp helps to identify the non-flushed update logs.

**MemoIndex.** To quickly find the relevant update logs for a specific data block from the update memo, a hash index is maintained. In the rest of this paper, this hash index is referred as MemoIndex. This index speeds up the query processing and flushing of the update memo contents. For each hash entry in the MemoIndex, the key is a data block ID and value is the location of the very recent update log for the corresponding data block in the update memo. Whenever an update log is added in the update memo, from the rid of the log, the corresponding destination data block ID is determined, and MemoIndex is checked to find if there is any hash entry for that data block. If an entry is found, the current value of that entry is copied in the previousLogEntryLocation field of the update log and the in-memory hash entry will be updated with the current location of the update log in the memo. If no entry is found, that means this log is the only entry for the corresponding data block. In this case, NULL is stored in the previousLogEntryLocation field of the update log; and finally, a new hash entry will be added in MemoIndex with data block ID as key and the current location of update log in the memo as value.

The **previousLogEntryLocation** field acts as a backward pointer and helps to avoid updating in the previous update logs, which makes MemoIndex a flash-friendly data structure. The main idea of MemoIndex is easily seen as inspired by the intuition behind the B-File design [20]. The MemoIndex requires very small amount of memory. For example, for 100MB data, which requires 400 flash blocks (assuming a 256KB block size), needs 400 hash entries. Assuming, each hash entry taking eight bytes, in total we need only 3200 bytes (3.12 KB) of additional memory. In case of power failure, the MemoIndex can be rebuilt by scanning update logs stored in the update memo and log pages.

C. Update Transaction Processing

An update transaction including INSERT, UPDATE, or DELETE operation is processed in the same fashion. When update transactions are executed, instead of performing in-place updates, the corresponding changes due the update transactions are first stored as update logs in the update memo. When memo is full, update logs are flushed to the corresponding log pages. Once update memo is full, we need to flush it so that future incoming update logs can be temporarily stored there. The simple way of flushing the update memo would be to flush the entire memo at once by moving all the update logs to the corresponding log pages. However, it will be a very time consuming task. To improve performance, instead of the flushing the entire update memo, we flush one victim block from the memo. There can be various policies to select a victim block. In this paper, we are using a simple round-robin policy. After selecting a victim block, its contents (i.e., update logs) are flushed to the corresponding log pages. Finally, the log pages are also full, the update logs are merged in bulk with the original records in the data blocks. During update processing, within a flash block, we always perform write operations sequentially, which complies with the flash memory’s physical restriction of writing (i.e., in a block sectors need to be written sequentially [2]).

When flushing the update logs from the victim block of the update memo to log pages of data block, the related logs for the same data block from the entire memo are flushed simultaneously. The MemoIndex described in Section III-B helps to quickly find the related update logs. Flushing all relevant logs simultaneously helps to reduce the number of erase operations, as this strategy provides the opportunity to compactly store more update logs in lesser space in the log pages. However, this complicates the flushing scheme of a victim block, as there may be some non-flushed update logs and some flushed logs that are already stored in the log pages or merged with the data blocks contents. Clearly, only the
Algorithm 1 Algorithm for Processing Update Transactions

Procedure UpdateProcessor(rid, opcode, updateLogContents)
1:  timeStamp = timeStampCounter value
2:  Increment timeStampCounter value
3:  Find previousLogEntryLocation from the MemoIndex
4:  newUpdateLog = (rid, opcode, timeStamp, updateLogContents, previousLogEntryLocation)
5:  if (Free space in the update memo) then
6:      Insert newUpdateLog in the update memo
7:  else
8:    // update memo is full
9:      Select a victim update memo block using round robin policy
10:     for (Each update log (u) in the victim block) do
11:       blockId = ID of update log’s destination block (B_{dest})
12:      end if (timestamp of u > logTimestamp of B_{dest}) then
13:     // This update log needs to be flushed
14:     end if (MemoIndex entry found for the key blockId) then
15:    // There can be more non-flushed entries for the same block
16:     location = MemoIndex entry’s value
17:     while (location is not NULL) do
18:       Read update log (u’) from location
19:       if (timestamp of u’ > logTimestamp of B_{dest}) then
20:           Add u’ to the flushList
21:       end if
22:       location = previousLogEntryLocation of u’
23:     end while
24:     Flush list to the log pages in B_{dest}
25:     if (Enough free space in the log pages of B_{dest}) then
26:       Store the flushList compactly in the log pages
27:       Set the logTimestamp of B_{dest} with the timeStampCounter value
28:     else
29:       // log pages are full
30:       Apply all the latest update logs in log pages and flushList in bulk to the data pages in B_{dest}
31:     end if
32:     Remove the hash entry with the key blockId from the MemoIndex
33:     Increment timeStampCounter value
34:     end if
35:     end if
36:     end for
37:     Erase victim memo block
38:     Insert newUpdateLog in the free space in update memo
39:  end if
40:  Update MemoIndex with the location of the newUpdateLog

non-flushed logs need to be stored in the log pages. With the help of logTimestamp value, the non-flushed logs can be easily identified. If an update log’s timestamp value is less than the corresponding data block’s logTimestamp value, then update log has not been flushed yet the log pages in an earlier block eviction phase. Now, we describe the detail of the update processing algorithm.

Algorithm. Algorithm 1 gives the pseudocode to process an update transaction. When an update log reaches the update memo, a timestamp value is assigned to the update log and timestamp counter is incremented (Lines 1-2 in Algorithm 1). We use MemoIndex to find the previousLogEntryLocation field of the update log (newUpdateLog) by using block ID as key (Line 3 in Algorithm 1). The rid field of an update log provides the block ID information. If an entry is found in MemoIndex, the previousLogEntryLocation will be value of that entry; otherwise, we set it to NULL. The newUpdateLog contains the rid, opcode of operation (i.e., INSERT, UPDATE, and DELETE), and the summary of changes by the update transaction, and previousLogEntryLocation (Line 4 in Algorithm 1). Now, we check if there are enough free space left in update memo to store the newUpdateLog. If this is the case, we store newUpdateLog in the next available free space of the update memo (Lines 5-6 in Algorithm 1). If there are not enough free space in the update memo, which means update memo is full. To make free space, we select a victim block in the round robin fashion for flushing (Line 9 in Algorithm 1). For each update log in the victim block, we determine whether this log has already flushed or not (Line 12 in Algorithm 1). If an update log is not flushed before, with the help of MemoIndex, we read all other non-flushed logs from the entire update memo, which also correspond to the same data block as the current log, and flush them in bulk to the corresponding log pages (Lines 14-24 in Algorithm 1). The logTimestamp value of a block helps to identify the non-flushed logs for that block. We again check, if there are enough free space left in the log pages of the respective data block to fit the flushed logs. If log pages have enough free space, we compactly store the flushed logs in the free space of the log pages and update the logTimestamp (Lines 25-27 in Algorithm 1). If log pages are full, we merge the flushed logs from update memo and current logs in the log pages with the old records stored in the corresponding data block (Line 30 in Algorithm 1). The procedure for combining the old data records and update logs is similar to the algorithm described in the IPL approach [14] and it incurs only one block erase operation. After flushing is done, we remove the hash entry corresponding to current processing update log from MemoIndex to remember that for its destination block all logs have been flushed (Line 32 in Algorithm 1). This helps to speed up look-up of the non-flushed logs in the later eviction phases. When all update logs from the victim block and other relevant logs are written to the corresponding log pages, we erase the victim update memo block to make it ready to store new update logs (Line 37 in Algorithm 1). Now, we store the newUpdateLog in the victim update memo block (Line 38 in Algorithm 1). Finally, we update (or insert a new entry, if there is no previous entry found) hash entry in the MemoIndex to remember the location of newUpdateLog (Line 40 in Algorithm 1).

Example. Figure 4 gives an example of processing update transactions. For simplicity, we ignore indexing and the timestamp values. Furthermore, we use full records instead of update logs. In Figure 4, there are three data blocks A, B, and C, depicted by fine dotted rectangles. Each data block has one log page (shaded in gray) and three data pages. Each page consists of four flash sectors. Sectors are depicted by smallest
rectangles. Each sector can contain at most three records. This means that a data page contains up to 12 records, and thus a data block contains up to 36 records. Records 1-12 correspond to first data page, similarly records 13-24 correspond to second data page, and so on. In Figure 4, the number $x$ next to a block denotes the $x$-th record in that block. For example, $A_1$ means the first record of block $A$. In block $A$, there are four records, $A_1$, $A_2$, $A_3$, and $A_4$ in the first data page, two records $A_{13}$ and $A_{14}$ in the second data page, and two records $A_{25}$ and $A_{26}$ in the third data page; the fourth page is the log page which has one used sector and three free log sectors. In block $B$, there are four records and all four log sectors are used. Similarly, in block $C$, there are five records and one log sector is used. In Figure 4, update memo consists of one block and is depicted by solid lines. It has 16 flash sectors. All the update memo sectors are used i.e., update memo is full. As flash memory is write once and write operations are performed in sector unit, that is why each memo sector contains only one record. However, in the data pages, there are multiple records in some sectors. This happens as with the help of update memo and log pages, we can compact multiple records in a sector and write them at once. In Figure 4, the first sector of the update memo contains updated record $A_2$. Similarly, in the second sector contains $B_3$, and so on.

Suppose, in Figure 4, we want to update one more record, $A_{27}$. However, as update memo is already full, we need to clean it to store new records. We partition the update memo records into three groups based on the blocks: \{$A_2, A_3, A_7$, $A_2, A_3, A_7$, $B_2, B_3, B_6, B_7, B_8, B_9, B_{13}, B_{25}$\}, \{$C_3, C_5, C_{16}$\}. The partitioned records are stored in the corresponding blocks’ log pages. If we do not have log pages, we have to perform three immediate erase operations. However, log pages helps to save these erase operations. We store \{$A_2, A_3, A_7$\} and \{$C_3, C_5, C_{16}$\} compactly in the log pages of data block $A$ and $C$, respectively. However, as all the log sectors of the block $B$ are full, we combine current log page records \{$B_1, B_2, B_3, B_4, B_6, B_{10}, B_{14}, B_{26}$\} and memo records \{$B_2, B_3, B_5, B_6, B_7, B_8, B_9, B_{13}, B_{25}$\} and with the old records \{$B_1, B_2, B_{13}, B_{25}$\}, and write them compactly into three data pages. Now, Block $B$ contains \{$B_1, B_2, B_3, B_5, B_6, B_7, B_8, B_9, B_{10}$\}, \{$B_{13}, B_{14}$\}, \{$B_{25}, B_{26}$\} in order in three data pages. Figure 5 gives the new state of the data blocks and update memo. We erase the memo block to make the flash sectors ready for writing. Finally, we store $A_{27}$ in the first sector of update memo.

**Discussion.** At a first glance, it seems that deferred update methodology generates skews of the number of erase operations for flash blocks in the different areas in the flash memory, which would cause performance bottlenecks. For example, update memo blocks are erased more frequently than the data blocks. However, current flash based devices use various wear leveling techniques to even out the erase count of all flash block [5], [8]. Most of the existing algorithms use hot and cold block classification [10]. In deferred update methodology, update memo blocks are relatively hot compared to data blocks. Existing wear leveling algorithms can efficiently handle hot and cold blocks. Therefore, the skewness introduced by deferred update methodology will not pose any performance bottlenecks in the existing flash-based devices.

**D. Query Processing**

Since the update transactions are not immediately reflected in data records, the query processing needs special care for the updated records whose most recent values still reside on either the update memo or log pages, i.e., not updated yet to the actual data pages. A query is processed in the reverse order to the update processing order. At first, we generate raw query result from the the data pages using the traditional query processing techniques. Next, we modified this raw result by update logs stored in the relevant log pages and update memo, in order. These additional modifications ensure that the generated query results are correct, i.e., include the most recent record values.

**Algorithm.** Algorithm 2 gives the pseudocode to process a query, $q$. We generate a raw result set $R_{data}$ from the records...
stored in the data pages using the traditional query processing techniques (Line 1 in Algorithm 2). Now, we modify the $R_{data}$ using update logs in the log pages. We need to scan the update logs stored in log pages of the data blocks which contain any log related to query $q$ (Lines 4-10 in Algorithm 2). The data blocks that we need to scan depends on the query selectivity. For every update log $(u)$, stored in the log pages of related data blocks, we check whether $u$ satisfies query $q$. If $u$ qualifies the query criteria, we add the record related to $u$ with updated value in the result set $R_{data}$. If $u$ does not satisfy query criteria, we further check whether the record related to $u$ is included in the raw result set $R_{data}$. If the related record is in $R_{data}$, we remove that record from $R_{data}$ as updated value of that stored in the log pages does not satisfy $q$. After processing the update logs in the log pages, we again modify the result set $R_{data}$ by the update logs stored in the update memo (Lines 12-28 in Algorithm 2). The MemoIndex and logTimestamp value help to speed up finding of the latest log entries. For every update log $(u')$, related a block stored in the update memo, we check whether $u'$ satisfies query $q$. If $u'$ satisfies the query criteria, we add the record related to $u'$ with the updated value in the result set $R_{data}$. If $u'$ does not satisfy the query criteria, we further check whether the record related to $u'$ is included in the result set $R_{data}$. If that record is in $R_{new}$, we remove it from $R_{data}$, as current value of that record does not satisfy $q$. Finally, after processing all relevant logs both in the log pages and update memo, we return $R_{data}$ as query result.

Example. Figure 6 gives an example of processing query, SELECT * FROM $R$ WHERE $Col2 > 15$. Similar to previous example, for simplicity, we ignore the timestamp values and use full records instead of update logs. Table R has three fields: $Col1$, $Col2$, and $Col3$ and contains nine records $R1$ to $R9$. Some of the updated record values are stored in the log pages and memo blocks. For example, in Figure 6, log pages contain the updated value of records $R1$, $R4$, $R5$, and $R8$. While update memo contains updated values of records $R4$, $R6$, and $R9$. The values of a record stored in the update memo is more recent than the values stored in the log pages, while the record values stored in the log pages are most recent than the values stored in the data pages. When we execute the query over the records stored in the data pages, we get the raw query result set $R_{data}$, which contains $\{R1, R2, R7\}$. To generate correct output, we modified $R_{data}$ by updated records stored in the log pages. From the updated records stored in log pages, $R8$ satisfies the query criteria $Col2 > 15$. However, the updated value of record $R1$ does not satisfy this criteria. Therefore, the new intermediate result set $R_{data}$ contains $\{R2, R7, R8\}$. To generate accurate output, we further modify $R_{data}$ by the updated records stored in the update memo. From the update memo, $R4$ satisfies the query criteria. Therefore, the final query result set $R_{data}$ contains $\{R2, R4, R7, R8\}$.

IV. ANALYSIS OF LOG PAGES AND MEMO

In this section, we analyze deferred update methodology that uses both update memo and log pages compared to the other three alternative approaches including no log and no memo, log only, and update memo only to process $N_u$ update transactions. Our measure of analysis are (a) space overhead ($S_{overhead}$), (b) query processing overhead ($Q_{overhead}$), (c) number of update logs stored in the update memo and each overwrite operation in the flash memory incurs an erase operation. Table I introduces various symbols used in this analysis. The next four subsections discuss the four alternative approaches using log pages and update memo. Finally, in the following subsection, we compare all four approaches.

A. No Log and No Memo Approach

This is existing design with no change for flash memory i.e., this model has no log pages and no update memo. Space Overhead. There is no space overhead in this scheme as there are no log pages and no update memo blocks to hold the update logs. The number of data blocks ($N_D$) to fit all $DB_{size}$ data is $DB_{size} \times B$ blocks. As this is the least possible number of blocks to hold the data, we consider this scheme as no space overhead model and use it as a baseline for comparison with other alternative designs.

Query Processing Overhead. As there is no space overhead, there is no query processing overhead. This is the best possible possible scheme in terms of query processing overhead. So, we
consider this scheme as no query processing overhead model and use it as a baseline for comparison with other alternatives. Erase Operations. In this model, to process every update transaction, in the worst case, we would need to perform one erase operation of the data block containing the corresponding data record per update transaction. Therefore, to process $N_u$ update transactions, the total number of erase operations ($N_E$) would be $N_u$.

B. Log Page Only Approach

In this model, each data block contains $n_1$ number of log pages. However, no update memo is maintained. This is similar to the model adopted by the IPL technique [14], when setting $n_1 = 1$.

Space Overhead. In this scheme, out of $\frac{B}{2}$ pages of a data block, $n_1$ number of log pages are used to temporarily hold the update logs. Therefore, we need additional data blocks to fit all data. The space for holding data in every block is $B - n_1 \times P$ bytes. Therefore, to fit $DB_{size}$ data, the required number of data blocks ($N_D$) would be $\frac{DB_{size}}{B - n_1 \times P}$ blocks. Now, the space overhead ($s_{overhead}$) compared to the baseline no space overhead model would be $\left( \frac{DB_{size}}{B - n_1 \times P} - \frac{DB_{size}}{B} \right) = \frac{DB_{size}}{B} \times \frac{1}{B - n_1 \times P}$ blocks.

Query Processing Overhead. To process a query, in addition to the traditional query processing techniques, we have to scan the update logs stored in the log pages of the data blocks to generate correct query output. This extra scanning contributes to the query processing overhead ($q_{overhead}$). We assume that, out of all data blocks, the $N_q$ number of data blocks contain the data records that satisfy the query criteria. In these data blocks, log pages can be full or partially full, or empty. Thus, we can assume that on average the log pages are half full. The $N_q$ data blocks have in total $N_q \times n_1$ log pages. On average, we have to scan half of these pages during the query processing. Therefore, $q_{overhead}$ compared to the base line no query processing overhead model, would be the time to scan $N_q \times \frac{B}{2}$ pages of data.

Erase Operations. In this model, each data block contains $n_1$ number of log pages and each page has $\frac{B}{2}$ sectors. Therefore, in total a data block has $n_1 \times \frac{B}{2}$ log sectors. Every update log consumes one sector of the log pages of data blocks. Therefore a data block hold $n_1 \times \frac{B}{2}$ update logs before it is full. Once full, we need to erase data blocks to make free space in the log pages to hold future update logs. Now, to process $N_u$ update transactions, the total number of erase operations ($N_E$) would be $\frac{N_u \times S}{n_1 \times P} = N_u \times \frac{S}{nP}$.

C. Update Memo Only Approach

This model has only update memo. However, no log pages are maintained. There are $N_M$ blocks in the update memo. At first, update logs are stored in the memo. When memo is full, update logs are merged with the old data records. The functionality of this approach is quite similar to space efficient FTL design [12].

Space Overhead. In this model, to fit the $DB_{size}$ data, the required number of data blocks ($N_D$) would be $\frac{DB_{size}}{B}$. In addition, we need space to fit $N_M$ number of update memo blocks. Therefore, the space overhead ($s_{overhead}$) compared to the baseline no space overhead model is $N_M$ blocks.

Query Processing Overhead. To generate correct query result, we have to scan the logs stored in the update memo in addition to the traditional query processing techniques. This additional scanning of the update memo contributes to the query processing overhead ($q_{overhead}$). The update memo can be full or partially full or empty. We assume that on average it is half full. Therefore, on average we have to scan half of the update memo during the query processing. Now, the $q_{overhead}$ would be the time to scan $\frac{1}{2} \times \frac{B}{2}$ pages of data.

Erase Operations. In this model, erase operations occur due to cleaning of the update memo blocks and erasure of the data blocks.

Update Memo Block Erases. As we assume that update log ($u_s$) size is less than the flash sector size ($S$), each update log consumes one sector from the update memo. Each memo block contains $\frac{S}{P}$ sectors. The $N_M$ memo blocks has in total $N_M \times \frac{S}{P}$ sectors. Therefore, before full, update memo can hold $N_M \times \frac{S}{P}$ update logs. Once update memo is full, we need to clean it to store future incoming logs. To process $N_u$ update transactions, in total, we need to clean the update memo blocks $N_u \times \frac{S}{P}$ number of times. During cleaning, we have to erase $N_M$ memo blocks. Therefore, the total number of erase operations in the update memo would be $\frac{N_u \times S}{P} \times N_M = N_U \times \frac{S}{P} \times N_M$.

Data Block Erases. Each time the update memo blocks need to be cleaned, we have to merge the update logs with the old data blocks records. During, cleaning each data block receives $n_{val}$ number of update logs. To merge these logs, we need to erase the data blocks. To process $N_u$ update transactions, the total number of data block erase operations would be $\frac{N_u \times S}{P} \times n_{val}$.

By summing the update memo block erase operations and data block erase operations, the total number of erase operations ($N_E$) would be, $\frac{N_u \times S}{P} + \frac{N_u \times S}{P} = N_u \left( \frac{S}{P} + \frac{1}{n_{val}} \right)$.

D. Log page and Update Memo Approach

This is the model adopted by the deferred update methodology. In this model, we have $N_M$ number of update memo blocks and each data block has $n_1$ number of log pages. In fact, this model is a combination of Log Page Only (as described in Section IV-B) and Update Memo Only (as described in Section IV-C) approaches.

Space Overhead. In this model, we need additional space to accommodate log pages and update memo. In every data block, $n_1$ pages are used to temporarily store update logs. Therefore, we need to increase the number of data blocks to fit all $DB_{size}$ of data. The required number of data blocks ($N_D$) would be $\frac{DB_{size}}{B - n_1 \times P}$. In addition, for update memo, we need $N_M$ blocks. The total number of blocks required is $N_D + N_M$. Therefore, the space overhead ($s_{overhead}$) of this model compared to the baseline no space overhead model would be $N_D + N_M - \frac{DB_{size}}{B} = \frac{DB_{size}}{B - n_1 \times P} + N_M - \frac{DB_{size}}{B} = \frac{DB_{size}}{B - n_1 \times P} + N_M$ blocks.
Query Processing Overhead. To process a query, in addition to the normal query processing techniques, we have to scan the logs stored in the both update memo and log pages to generate correct query result. These additional scanings contribute to the query processing overhead ($q_{overhead}$). Similar to the Update Memo Only approach (as described in Section IV-C), in this scheme, on average the update memo is also half full. Thus during the query processing we have to scan $\frac{1}{2} * N_M$ blocks, which is equivalent to $\frac{1}{2} * N_M * \frac{B}{T}$ pages of update data. In addition, similar to the Log Page Only approach (as described in Section IV-B), we need to scan $N_q * \frac{n_1}{T}$ pages of data. Therefore, the query processing overhead $q_{overhead}$ is the time to scan $\frac{1}{2} * (N_M * \frac{B}{T} + N_q * n_1)$.

Erase Operations. In this model, erase operations occur due to update memo cleaning and data block erases during the log pages cleaning.

Update Memo Block Erases. Similar to the Update Memo Only approach (as described in Section IV-C), the total block erase operations on the update memo in this scheme would be, $\frac{N_u * N_M}{N_M + \frac{B}{T}} = \frac{N_u * S}{B}$.

Data Block Erases. Each time when the update memo blocks are cleaned, $n_{ul}$ number of update logs are stored in the log pages of each data block. The total size these update logs is $n_{ul} * u_a$. To fit these update logs, we need $\lceil \frac{n_{ul} * u_a}{N} \rceil$ sectors of a log page. We need to take the ceiling, as once some data is written in a sector, without erasing we cannot rewrite any data in that sector. A data block contains $n_1$ log pages, which has in total $n_1 * \frac{B}{T}$ sectors. Therefore, before full, the log pages of a data block can contain $\lceil \frac{n_{ul} * u_a}{N} \rceil * n_{ul}$ update logs. Once log pages are full, the data block needs to be erased to accommodate new logs. Now, to process $N_u$ update transactions, the total number of data block erase operations performed would be $\frac{N_u}{\lceil \frac{n_{ul} * u_a}{N} \rceil * n_{ul}}$.

Summing up the update memo block erases and data block erases, the total number of block erase operations ($N_E$) would be $\frac{N_u + S}{B} + \frac{N_u}{\lceil \frac{n_{ul} * u_a}{N} \rceil * n_{ul}} = N_u \left( \frac{S}{B} + \frac{1}{\lceil \frac{n_{ul} * u_a}{N} \rceil * n_{ul}} \right)$.

E. Comparison

Table II summarizes the total number of erase operations performed, space requirements, and query processing overhead for different alternative designs using log pages and update memo.

In terms of space overhead and query processing overhead, No Log and No Memo approach is the best. For the same number of log pages ($n_1$), Log Page and Update Memo approach i.e., deferred update methodology has more space overhead and query processing overhead compared to the Log Page Only approach because of the additional update memo. In this analysis, we have not considered indexing of the update logs stored in the update memo. This index improves the query processing overhead of the deferred update methodology. Compared to the Update Memo Only approach, for the same number of memo blocks ($N_M$), deferred update methodology has more space overhead and query processing overhead due to the additional log pages. Our experimental results show that for same space overhead, Update Memo Only approach and deferred update methodology incur comparable query processing overhead, while the Log Page Only approach incurs less query processing overhead compared to both of these approaches.

In terms of erase operations, No Log and No Memo approach incurs the largest number of erase operations. As erase is the most expensive operation and it incurs lot of flash sector read and write operations, therefore update operations will be slow compared to other three approaches. In addition, due to excessive erase operations, flash memory will be worn-out in a relatively faster rate. For the other three approaches, the number of erase operations depend on the values of the different parameters. To get a general idea, we put the following parameter values to the total number of erase operations ($N_E$) calculation formula: $B = 256$ KB, $P = 8$ KB, $S = 4$ KB, $n_{ul} = 2$, and $u_a = 64$ bytes. The parameters value of the flash memory are taken from the SSD design project [2]. Since, deferred update methodology uses both log pages and update memo, while Log Page Only approach uses only log pages, we use $n_1 = 1$ in case of deferred update methodology, while we use $n_1 = 2$ in case of Log Page Only approach. As deferred update methodology uses half number of log pages with respect to the Log Page Only approach, the extra space taken by the update memo will be offset by the smaller
value of $n_l$. Now, by substituting the parameter values, we get $N_E = N_u \times \frac{1}{2}$ for the Log Page Only approach, $N_E = N_u \times \frac{43}{64}$ for the Update Memo Only approach, and $N_E = N_u \times \frac{5}{64}$ for the deferred update methodology. In this hypothetical example, the number of erase operations performed by the Log Page Only and Update Memo Only approaches are almost same, while deferred updates performs 6.2 times fewer number of erase operations compared to the Log Page only approach. Our experimental results shows that, deferred update methodology significantly reduces the total number of erase operations. As erase is most expensive operation and a block can only be erased for limited number of times, therefore reduction of the block erase operations helps to improve update processing performance as well as prevents early worn-out of the flash memory.

V. Experimental Results

We compare deferred update methodology with in-page logging (IPL) technique [14] and Update Memo Only (described in Section IV-C) approach to handle update transactions. IPL is the state-of-the-art technique for handling update transactions for the flash based storage. It is a special case of Log Page Only approach (described in Section IV-B) with one log page per erase unit. We do not consider No Memo and No Log approach (described in Section IV-A) in the comparison, as it is not suitable for processing update transactions [14]. In the rest of section, at first, we describe the simulator, traces, and performance calculation formulas. Next, we estimate the internal parameters values: number of log pages per block ($n_l$) and number of update memo blocks ($N_M$). Finally, we demonstrate the scalability of the deferred update methodology compared to other two approaches by varying the number of update transactions and database size.

Simulator and Traces. To evaluate deferred update methodology, similar to IPL technique [14], we have implemented a standalone event-driven simulator in the C language on the Linux platform. This simulator mimics the behavior of both update memo and log pages as described in Sections III and IV. We use synthetic traces to evaluate deferred update methodology. Synthetic traces are generated by a standalone trace generation program written in C language. This program takes database size, page size, and number of update transactions as input and generates a trace file as output. The update transactions in the output trace file are uniformly distributed over all database pages and there is almost no temporal locality (less than 1%). This trace emulates one of worst writes access patterns for the flash memory. The online transaction processing (OLTP) type applications exhibit quite close behavior to this trace. The size of an update log in each transaction lies in between 20-100 bytes. We vary the number of update transactions from one million to 100 millions, while we vary database size from 100 MB to 10000 MB. We assume that each database page size is 8 KB.

Performance Calculation Formulas. To calculate the update processing time, we use the following formula: $\text{total number of erase operations } \times \text{ erase time} + \text{total sector read operations} * \text{sector read time} + \text{page register access time} + \text{total sector read operations} * \text{sector read time} + \text{page register access time}$.

During the update processing, the simulator takes a trace file as input and as output it provides the total number of erase operations performed, total number of sector read operations, and total number of sector write operations. To calculate the query processing overhead, we use the following formula: the numbers of extra flash sectors read due to query processing * (sector read time + page register access time).

During query processing, the simulator takes query selectivity and returns the numbers of extra flash sectors to process that query. Table III gives the various flash parameters values used to in the experiments. These values are taken from the SSD design project [2].

A. Parameter Value Selection

In-page logging (IPL) technique [14] uses number of log pages per block ($n_l$) and Memo Only approach uses number of update memo blocks ($N_M$), while deferred update methodology uses both $n_l$ and $N_M$. The space overhead and query processing overhead depend on the value of $n_l$ and $N_M$. We vary the $n_l$ from 1 to 31, and $N_M$ from 1 to 400, for a database of size 100 MB processing one million update transactions. Since, a block size is 256 KB and a page size is 8 KB, therefore a block can contain 32 pages. Now, maximum number of log pages in a block ($n_l$) is 31 as we need at least one data page per block. On the other hand, to fit 100 MB database, we need 400 blocks. We restrict maximum number of update memo blocks ($N_M$) to the number of blocks need to fit the original data, that is why the maximum value of $N_M$ is 400. Since, three different approaches uses different number of parameters, to make a fair evaluation, we estimate the average

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block size</td>
<td>256 KB</td>
</tr>
<tr>
<td>Sector size</td>
<td>4 KB</td>
</tr>
<tr>
<td>Data Register Size</td>
<td>4 KB</td>
</tr>
<tr>
<td>4KB-Sector Read to Register Time</td>
<td>25 µs</td>
</tr>
<tr>
<td>4KB-Sector Write Time from Register</td>
<td>200 µs</td>
</tr>
<tr>
<td>Serial Access time to Register (Data bus)</td>
<td>100 µs</td>
</tr>
<tr>
<td>Block Erase Time</td>
<td>1500 µs</td>
</tr>
</tbody>
</table>

Table III

<table>
<thead>
<tr>
<th>Parameters values for NAND flash memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space Overhead</td>
</tr>
<tr>
<td>6.8%</td>
</tr>
<tr>
<td>10.5%</td>
</tr>
<tr>
<td>14.5%</td>
</tr>
<tr>
<td>18.8%</td>
</tr>
<tr>
<td>23.3%</td>
</tr>
<tr>
<td>28.3%</td>
</tr>
<tr>
<td>33.5%</td>
</tr>
<tr>
<td>39.3%</td>
</tr>
</tbody>
</table>

Table IV

Parameters values for Log Pages and Memo Blocks for a 100 MB database processing one million update transactions
Deferred Update methodology improves average update processing time by 1%-17%. The average update processing time decreases with the increase of the space, as we can buffer more update logs and apply them in bulk. This helps to decrease the total number of erase operations. This trend is shown by Figure 7(c). Since erase operations are performed in the block-level and before erasing a block, we need to move data and write them back. Thus, erase operations incurs huge overhead in terms of latency. With the decrease in the number of erase operations, this latency overhead decreases, which helps to improve the update processing performance. In IPL, increasing the space helps to hold more update logs \((n_{ul})\), which reduces total number of erase operations and consequently improves the update processing performance. In contrast, increasing \(N_M\) introduces overhead in query processing due to additional processing of the larger number of update logs stored in the larger update memo. According to Table II, in deferred update methodology, increasing both \((n_l)\) and \((N_M)\) contribute to the increased space overhead. However, larger \(n_l\) and \(N_M\) help to hold larger number of update logs \((n_{ul})\) and process them in bulk, which helps in reducing the total number of erase operations. Thus, helps to improve update processing time. On the other hand, query processing overhead increases with the increased space overhead due to processing of large number of update logs.

For the rest of the paper, we keep the space overhead same for all three approaches. Since with the increase of space overhead, query processing overhead also increases, we keep the space overhead as minimum as possible. With this goal, we set the value of \(n_l\) to two in IPL and calculate the space overhead. In this case, space overhead is 6.8%. Now, we select a value of \(N_M\) in the Update Memo Only approach such that space overhead becomes 6.8%. Similarly, for the deferred update methodology, we set the value of \(n_l\) to one and select a value for \(N_M\) such that overall space overhead becomes 6.8%.

The parameter values for different database size are listed in Table V. These values are used in the scalability demonstration experiments.

### B. Scalability

Now, we demonstrate the scalability of the deferred update methodology with the increase in the number of update transactions and size of a database.

1) Varying Update Transactions: To evaluate the scalability, we vary the number of update transactions from one million to 100 millions in a 100 MB database. Figure 8
gives the average update processing time, query processing overhead, and total number of erase operations. From the figure, it is clear that for the same space overhead, deferred update methodology scales very well with the increased number of update transactions. Compared to IPL, deferred update methodology improves average processing time by 41%, while Update Memo Only approach performs almost same. To understand the query processing overhead, we vary the query selectivity form 1%-11%. With the increase of query selectivity, the query overhead increases in all three approaches. This is expected as with the increase of query selectivity, we need to process more update logs to generate query results. Compared to IPL, deferred update methodology incurs up to 30% more query processing overhead, while Update Memo Only approach incurs up to 41% more overhead. Usually, update-intensive workloads, for example, OLTP type applications, exhibit significant number of (i.e., 20%-40%) write operations [7], therefore query processing overhead will not be a bottleneck for the update-intensive workloads. In terms of erase operations, to process the same number of update transactions deferred update methodology always outperforms IPL and Update Memo Only approach. Compared to IPL, deferred update methodology incurs 39% fewer erase operations, while Update Memo Only approach incurs almost same number of erase operations. Since, with the decrease of erase operations, the lifetime of the flash memory also increases, therefore compared to other two approaches, deferred update methodology will significantly help to improve lifetime of the flash based storage.

2) Varying Database Size: We vary database size and in proportion we vary the number of update transactions. We process one million update transactions in 100 MB database, ten millions update transactions in 1000 MB database, and 100 millions update transactions in 10000 MB database. Figure 9 gives the average update processing time, query processing overhead, and total number of erase operations. It shows that the deferred update methodology scales very well with the increase in database size. Compared to IPL, deferred update methodology improves average update processing time by 42%, while Update Memo Only approach performs almost same. Figure 9(b) shows that the query processing overhead increases with the increase in database size in deferred update methodology and Update Memo Only approach. This reason behind this trend is that with the increase in database size, the number of buffered update logs also increases, and we need to process increasing number of logs to generate query result. Compared to IPL, deferred update methodology incurs up to 24% query processing overhead, while Update Memo Only approach incurs up to 30% overhead. In terms of total erase operations, compared to IPL, deferred update methodology performs 41% fewer erase operations, while Update Memo Only approach performs 3% fewer erase operations.

VI. CONCLUSION

In this paper, we have proposed a flash-friendly hierarchical update processing technique, named as deferred update methodology, for the NAND flash based storages. In this methodology, update transactions are processed through two intermediate in-flash layers. The first layer is named as update memo. The second layer consists of few reserved pages, named as log pages, in each flash erase unit. The update memo layer acts as a buffer for the log pages and helps to collect multiple update logs for the same flash erase units. The buffered update logs are stored compactly in the log pages. When the log pages are full, update logs are applied in bulk to the erase units. This strategy significantly improves the update processing overhead. It also helps to reduce the number of erase operations, which consequently helps to improve the lifetime of the flash memory. During data retrieval, in addition to the traditional data retrieval technique, we process the update logs stored in the log pages and update memo, to get the recent copy of the data. We use a flash-friendly index to expedite processing of the update logs. We have also developed an analytical model to study the effect of update memo and log pages in the update processing performance, data processing overhead, and space overhead. Our experimental results shows that deferred update methodology improves average update processing time by approximately 40% compared to the state-of-the-art in-page logging (IPL) technique [14] for the uniformly distributed update transactions. Experimental results also shows that, compared to IPL, deferred update methodology incurs approximately 40% fewer number of erase operations and scales very well with the increase in the number of update transactions and data size.
Fig. 9. Scalability with varying database size. Here, ‘Memo’ stands for Update Memo Only approach.

REFERENCES