Adaptive Logging for Mobile Device

Young-Seok Kim, Heegyu Jin, Kyoung-Gu Woo
Samsung Advanced Institute of Technology (SAIT), Samsung Electronics. Co., Ltd.
San #14-1 Nongseo-dong Giheung-gu Yongin-si Gyeonggi-do 446-712, Korea
{ys24.kim, heegyu.jin, kg.woo}@samsung.com

ABSTRACT

Nowadays, due to the increased user requirements of the fast and reliable data management operation for mobile applications, major device vendors use embedded DBMS for their mobile devices such as MP3 players, mobile phones, digital cameras and PDAs. However, database logging is the major bottleneck against the fast response time. There has been a lot of work minimizing logging overhead but no single recovery method provides the best performance to a variety of database workloads. In this paper, we present a novel recovery method called *adaptive logging* which can switch the logging method from ARIES to shadow paging adaptively at a page level according to the update state of each page on run time. Also, we propose a log compaction method called *deferred logging* which removes redundant logs by deferring to create log records until the updated data page is flushed or until the transaction commits. Deferred logging is coupled with adaptive logging seamlessly so that it boosts the performance of adaptive logging by reducing the typical overhead of hybrid methods. We have implemented the proposed approaches to our embedded DBMS which was deployed to more than 10 million mobile devices and evaluated them through a real world application on a mobile device. The result shows that our approaches can reduce logging overhead significantly and consequently can improve the response time of both small update transaction and large update transaction effectively.

1. INTRODUCTION

Recently, major device vendors such as Nokia, Samsung, Apple and LG, use embedded DBMSs for their mobile devices such as MP3 players, mobile phones, digital cameras, and PDAs [1, 2, 3, 4] since the devices should provide fast and reliable data management operation to applications such as phone book, music browser, and photo browser. On music browser, for example, a user may request to list a bunch of songs sorted by title of which artists are “Beatles” and album names are “Hey Jude”. Also he(or she) may insert and delete 1 to 10 song(s) or even all songs at a time to replace old songs with new songs. These are very typical use cases which require fast and rich browsing interface and reliable contents manipulation method.

We developed an embedded DBMS which has been deployed to more than 10 million mobile devices. Logging, however, was the main bottleneck against the fast response time of update transactions, especially for large update transactions since a large amount of log should be flushed to stable storage during commit. In server-side database, a separate log-writer daemon may solve this problem by flushing the large amount of log periodically. This approach, however, is not appealing in embedded DBMSs since they are typically simple libraries and tightly bound with application processes, which make it hard to run independent log-writer daemons. Although we can make a separate log-writer daemon in some cases and it can deal with the problem, the large amount of log writes itself is still a burden to flash memory storages of mobile devices because of wear-out issue and expensive block erase operation of the flash memory [5].

Meanwhile, Cabrera et al. [6] showed that no single recovery method provides the best performance to all transaction types. To minimize logging overhead, they proposed to choose an appropriate recovery method according to the property of each transaction’s workload. For example, ARIES [8] is applied for small update transactions and shadow paging [7] is used for large update transactions, especially for updating large objects. However, this hybrid approach is static since it can’t change the pre-assigned recovery method dynamically on run time even if a transaction doesn’t show the expected workload. This static hybrid approach is applicable only to a situation where all the properties of a transaction are known before the transaction starts. In case the properties can’t be predefined, which is more general situation, this approach is not effective.

In this paper, we propose a novel recovery method called *adaptive logging* which focuses on reducing the update log size in a way that different logging methods[1] are applied dynamically on run time at a page level switching from ARIES to shadow paging according to the update state of for each page. To the best of our knowledge, no work has dealt with switching recovery method dynamically on run time according to the transaction’s update state. Our dynamic hybrid approach overcomes the limitation of both ARIES and shadow paging and consequently accomplishes the fast response time for large update transaction as well as small update transaction. With respect to concurrency control, the inherent property of shadow paging limits the lock granularity at most to the page level, but multi-granularity locking of page- and tuple-level can be used according to the current logging method of each page in the adaptive scheme. In other words, when the logging method of the page is switched to shadow paging, the lock granularity for the page will be escalated from tuple-level to page-level.

Articles from this volume were presented at The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.
Proceedings of the VLDB Endowment, Vol. 3, No. 2

[1] Of course, shadow paging is not log-based recovery method, but we will use the term ‘logging method’ which also includes shadow paging in this paper

Along with adaptive logging, we present a log compaction method called *deferred logging* which can reduce the log size further. It is a variant of techniques such as log folding [10] and semantic compaction [11]. The differences will be explained on Section 6. Deferred logging postpones creating the ARIES-style log record [8] (hereafter simply referred to as log record) until the corresponding data page is flushed or the transaction commits. Instead, when a data page is updated, it creates and manages log information called *log entry* in a memory space separated from the log buffer. If the data page is flushed or the transaction commits, all log entries from the page or all log entries caused by the transaction are converted to log records, respectively. As a result of the deferring, deferred logging has a chance to compact log entries which are created from the adjacent or overlapped area within a same page. Deferred logging can be coupled with adaptive logging seamlessly and furthermore removes the overhead of writing useless log records in adaptive logging.

The contributions of this work are summarized as follows.

- A novel recovery method called adaptive logging is proposed for the first time to overcome the limitations of *static logging* method such as ARIES and shadow paging.
- A log compaction method called deferred logging is proposed not only to reduce log size but also to remove the overhead of writing useless log records in adaptive logging. We do not claim that deferred logging is a very new idea but do show that it is important feature of improving performance of adaptive logging further.
- We implement our approaches and evaluate them through a real world application on a mobile device, which demonstrates that adaptive logging with deferred logging can reduce logging overhead significantly and consequently can improve the response time of both small update transaction and large update transaction effectively.

The rest of this paper is organized as follows. Section 2 clarifies the problem that we handle. Section 3 explains static approaches which are the basic building blocks for understanding the adaptive approach. Section 4 presents the basic concepts and the design of adaptive logging with a simple example. Section 5 explains the rationale of deferred logging and Section 6 surveys the related work. In Section 7, we demonstrate the performance advantage of adaptive logging and deferred logging through a real world application on a mobile device comparing against the static logging approaches. Section 8 summarizes the contributions of this paper with future work.

2. PROBLEM

User requests to the real world application on a mobile device cause large updates as well as small updates to the database as following cases.

- Case 1: If a user copies 1,000 new songs from his (or her) personal computer to a micro SD card and the card is inserted into the user's mobile phone, then the phone's music browser application will extract metadata such as genre, artist, album, title, file path, etc from the songs and reflect each metadata into the corresponding music database, which causes bulk updates to the database. Also, the opposite case such as removing 1,000 songs or even more songs from the songs stored at the micro SD card causes bulk updates to the database.
- Case 2: If a user deletes all songs belonging to a specific album, a specific artist, or a specific folder from the micro SD card through the music browser application on his(or her) mobile phone, then small or large updates may happen to the database depending on how many songs meet the condition.

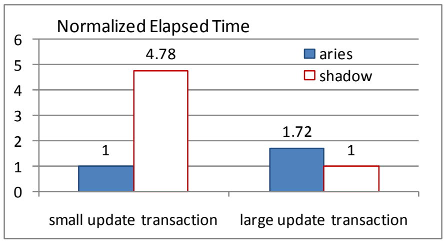


Figure 1. ARIES vs. Shadow Paging

Case 1 shows the situation that file operation is executed in the personal computer and metadata update to the database is conducted in the mobile device after the micro SD card is inserted into the mobile device. Case 2 shows the situation that both file operation and metadata update to the database are executed in the mobile device. The update transaction from the above situations can modify many pages taking a lot of time or can modify just a few pages. But the user always wants his(or her) device to respond as quickly as possible. One interesting thing worth to be noted is that the response time of the transaction varies a lot according to its recovery method.

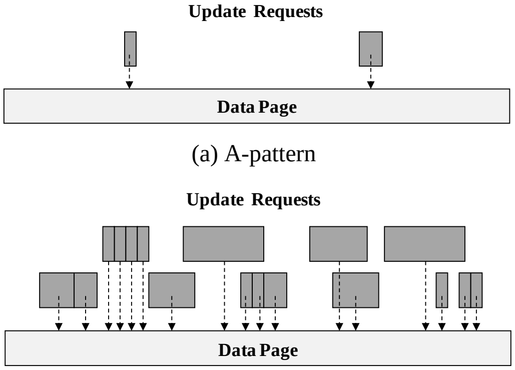
We have implemented two recovery methods, ARIES and shadow paging, to our embedded DBMS. When ran with many real queries of MP3 player device, ARIES clearly outperformed shadow paging in small update transactions while shadow paging was the winner in large update transactions as in Figure 1.[2]

To analyze the reasons behind this interesting phenomenon, we define two update patterns called A-pattern (**A**RIES-favorable update **pattern**) and S-pattern (**S**hadow-paging-favorable update **pattern**). Because page[3] is used as the basic unit of recovery, we examine logging behaviors of recovery method in page level first. We then clarify the problem that we address.

As shown in Figure 2(a), A-pattern updates a small area of a data page in a transaction, which generates a small amount of log that is less than the data page size. If a data page is updated by an A-pattern update operation, it benefits from no-force policy of ARIES since ARIES writes only log records without flushing the data page to disk on commit time. When a transaction is composed of mostly A-pattern update operations, we call such a transaction as small update transaction. Small update transaction is usually found in traditional OLTP applications [14].

[2] In Figure 1, 'small update transaction' consists of 1,000 transactions of inserting single record and 'large update transaction' represents single transaction which updates 10% of records in a database.

[3] There is traditional assumption of that the disk page is the basic unit of recovery although variations such as minipage and segment exist.



(a) A-pattern

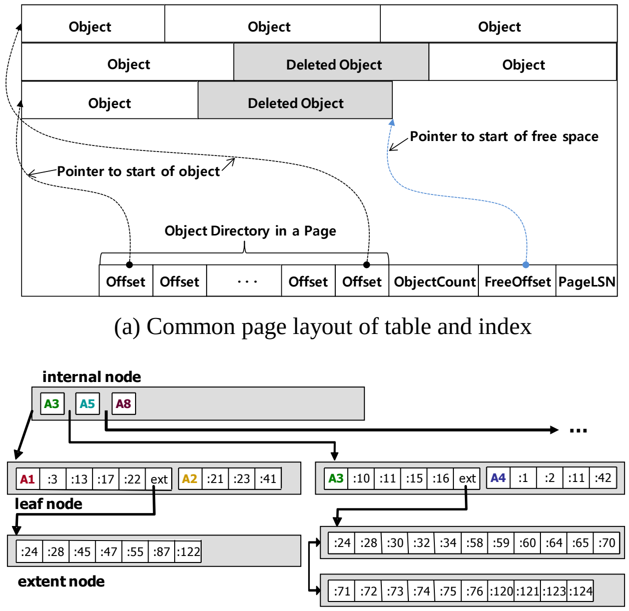
(b) S-pattern

Figure 2. Two Update Patterns

As shown in Figure 2(b), S-pattern updates a large area of a data page or repeatedly updates the same area of a data page in a transaction, which causes a large amount of log that is larger than the data page size itself. If the update pattern of the transaction is S-pattern, it can't leverage the no-force policy because the log size is larger than the data page size. In this situation, shadow paging is better since it doesn't make any log record and its force policy needs less pages to be written: updated data pages are less than log pages by ARIES. We call this type of transaction as large update transaction. Updating large objects is one typical example of the large update transaction, and shadow paging was shown to be more appropriate for it [6].

A transaction, however, doesn't always fall into one of the two transaction types. Rather, as a transaction may have data pages modified as A-pattern and S-pattern together, mixed update transaction is very common. Prior to presenting examples of a mixed update transaction, we describe a typical page layout [16] of the table and the index in order to make it easy to understand how update operation is executed at the page in the perspective of logging. Figure 3(a) shows the common page layout of the table and the index which can handle variable length records and keys, respectively.

If the page layout is used for a table, the object in the figure represents a record. When a new record is inserted into a page, 1) the record image is copied at the offset of the page pointed by the FreeOffset value, 2) the offset is appended to the leftmost slot of Object Directory, 3) ObjectCount value is increased by 1, and 4) FreeOffset value is set to the offset where the record image ends. If physical logging [15] is used, each step will generate the corresponding log record since each physical log record should maintain pre- and post-update image of each update area. On the contrary, if physiological logging [15] is used, generating only one compact log record may be enough since it can manage in the log record all the information necessary for redo/undo operation within the page. For example, a physiological log record for inserting a record into a page maintains information such as transaction id, operation type, page id, record image, image size, etc (without preserving both the pre- and post-update image). Then for redo, the above 4 steps are replayed starting with copying the record image and for undo, a compensation operation such as a record deletion is executed based on the log record. Meanwhile, when a record is deleted, 1) the corresponding offset in Object Directory is removed and the rest of offsets are compacted to handle the slot fragmentation, 2) ObjectCount value is decreased by 1. By the way, records themselves may or may not be compacted depending on the implementation. If they are compacted and physical logging is used, a large amount of log can be generated depending on the deleted record location. However, physiological logging is free from the logging overhead of the compaction since the compaction can be logically executed without storing the pre- and post-update image of the compacted area. Although physiological logging has less log size than physical logging, in order to apply physiological logging to every update operation of the database, it is necessary to design all the corresponding logical redo/undo operation at a page level very carefully. In this perspective, physiological logging is not as simple as physical logging.



(a) Common page layout of table and index

(b) Page layout of index for duplicated key

Figure 3. Page Layout

When the page layout is used for an index, the object in Figure 3(a) represents a key. For the simplicity of explanation, we omitted information on the page such as sibling node pointer, child node pointer, node type, etc. Inserting/deleting a key is executed just as the record operation is done except for following operations. 1) Offsets in Object Directory are sorted according to the key order instead of sorting key images. 2) For duplicated keys, the key image itself is stored only once and the corresponding record IDs (RID) are managed in an array on the leaf node. As the number of the duplicated keys is getting increased and exceeds the capacity of the reserved key space, the overflowed RIDs are managed separately on a special node called *extent node*. The RIDs both on a leaf node and an extent node are managed in a sorted order for efficient retrieval such as binary search. As shown in Figure 3(b), key A1 has 11 duplicated keys so that 4 RIDs of them are managed on a leaf node and the other 7 RIDs are managed on an extent node. When the record is deleted on the extent node, the rest of RIDs are compacted to avoid fragmentation. This compaction also causes a large amount of log in physical logging.

These update aspects in the page level show that even though physiological logging may generate more compact log record than physical logging, still S-pattern as well as A-pattern can happen to both methods depending on the workload of the transaction.

Now, we go back to the mixed update transaction and provide the example. Let's assume that there is a music table composed of ID, Artist, Album, Preference, ThisMonthPlayCount, LastMonthPlayCount, etc, where Artist and Preference has index, respectively. Here are two queries from a music browser which automatically updates user's preference according to the monthly play count of the songs; "update music set preference = preference + 1 where ThisMonthPlayCount >= LastMonthPlayCount + 10", and "update music set Preference = Preference - 1 where ThisMonthPlayCount < LastMonthPlayCount - 10". If we consider the page layout of the table and the index as shown in Figure 3, the above queries have a tendency to generate A-pattern on the table pages and S-pattern on the index pages of the preference field as the record size is getting larger than the field size relatively and the selectivity of the query is getting increased.

Unlike the above example, if the update field is the same field with the predicate field, for example, "update table set x=x+1 where x>10", the tendency will be much more apparent since the qualified keys obviously will exist on the same data page of the index and therefore the index page will show more S-pattern updates.

Although an update transaction shows much different response times as different recovery methods are used (as in Figure 1), it is very hard to tell the optimal recovery method for each transaction in advance. A transaction can consist of both A-pattern updates and S-pattern updates together preventing one single recovery method from showing the best response time. Furthermore, even though the update pattern for each page is known to in advance, different transactions may show different update patterns in the same page. These aspects mean that neither static single recovery mechanism nor static hybrid recovery mechanism (in the manner of Cabrera et al.) is an appropriate solution. This is why we have devised a novel logging method, adaptive logging.

3. STATIC CHOICE

Prior to presenting the rationale of adaptive logging in Section 4, we describe the static approach such as ARIES and shadow paging in this section, which is the basic building block of our implementation details to make it easy to understand how ARIES and shadow paging works adaptively even at the same page in a transaction. Our DBMS is able to guarantee atomic operation of transactions by providing out-of-place update approach or in-place update approach, which is represented by shadow paging and ARIES, respectively and selects the option on library build time. When shadow paging is used, page-level locking is the finest granularity (Coarser levels such a table-level locking also can be used).

To implement out-of-place update approach, just as the original shadow paging scheme [7] does, we manage a data structure called *page table* for valid page mapping. Let's call the slot number of the page table *logical page number* and the value in the slot *physical page number*. We maintain two bitmaps called *free physical page map* and *free logical page map* to manage free physical pages and free logical pages, respectively. For example, when a new data page is required for B+-tree, a new physical page number is assigned from the free physical page map and a new logical page number is assigned from the free logical page map. Then we find the page table slot corresponding to the logical page number and set the new physical page number to the slot. Unlike shadow paging, atomic updates to the free page maps and the page table are guaranteed by ARIES approach. It means that pre- and post-update image of the bitmaps and the page table are preserved by log records. Then the log records are flushed to disk during commit. After the new page is allocated, updates to the new page are reflected in the new page without making another copy of the page in the transaction since there is no pre-update image in the new page. On the contrary, the first update to an existing data page (except for the free page maps and the page table) in a transaction makes a copy of the page at a different physical page. Then updates are reflected to the copied page. By the way, it is necessary to distinguish whether the data page has already been updated or not in the transaction boundary to decide whether the current update to the page has to make a new copy of it or not. To check the occurrence of an update to a specific data page, we maintain a bitmap called *update occurrence map* in main memory. The bitmap has the number of bits which is identical with the number of slots in the page table. If the system crashes, the update occurrence map is useless in the perspective of the recovery process.

With respect to in-place-update approach, we follow WAL (Write Ahead Log) protocol and steal/no-force policy just as ARIES does. Different from the original ARIES scheme [8], we use force policy to newly allocated data pages [13] since updates to the new page do not create log records in our scheme. (Deallocated pages are not reused until the transaction commits. This is also valid for adaptive logging.) Subsequently, the new page doesn't have log-sequence number (LSN) on it, i.e. page LSN, which may cause problem in recovery process. During the recovery, redo and undo operation must be idempotent, i.e. executing operation an arbitrary number of times is equivalent to doing it once [15]. The property is guaranteed by LSN for redo and compensation log record (CLR) for undo in ARIES scheme [8]. Page LSN plays a role of an order indicator such as timestamp at the data page so that by indicating to which effect of the update is applied to the page, more than one execution of the redo operation is not allowed. Therefore the idempotent operation is guaranteed. To handle the problem, that is, the lack of page LSN on the newly allocated data page, we have assigned a LSN to the new page when it is flushed to the stable storage. The LSN is the last LSN which has been issued for the moment of the data page being flushed. It is guaranteed that the last LSN is less than LSNs of upcoming updates to the page. Therefore recovery mechanism can work correctly since it can decide the order between updates to the new page and the upcoming updates to the page after the page is flushed.

4. ADAPTIVE LOGGING

Adaptive logging works based on ARIES as well as shadow paging adaptively according to the update state of each data page, which overcomes the limitations of the conventional static logging approach. To make ARIES and shadow paging works adaptively, we have harmonized the behavior of ARIES and shadow paging based on what we have explained in Section 3. Firstly, every data page is accessed indirectly through the page table even in ARIES

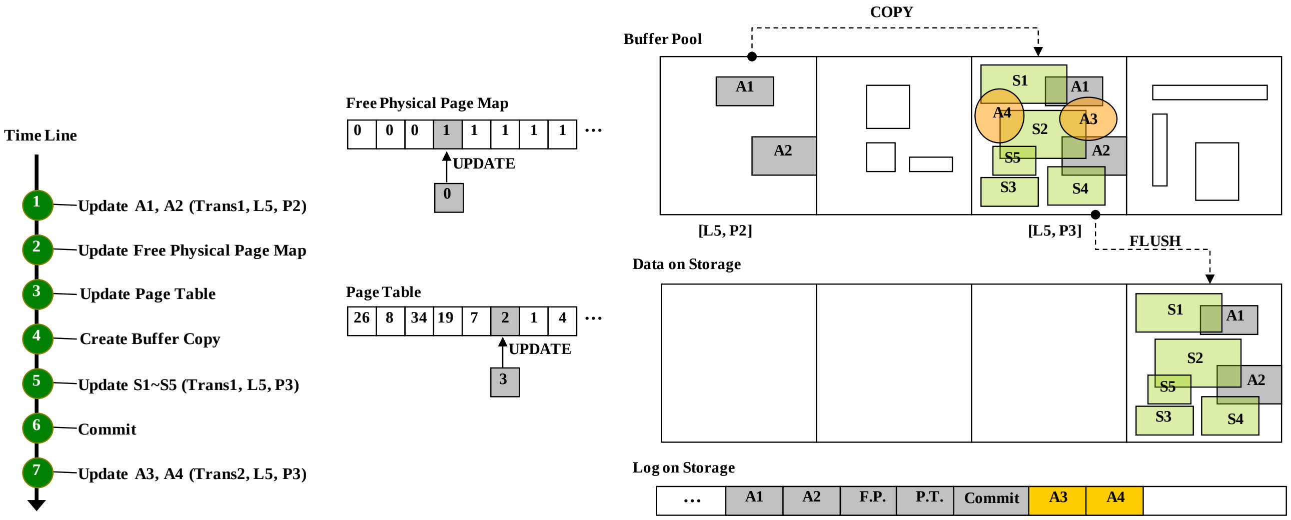


Figure 4. Adaptive Logging Example

scheme, which means that every access to data pages takes a step of converting the logical page number to physical page number through the page table. Every first update to a data page in shadow paging mode changes the value of the corresponding slot of the page table since the update creates the copy of the page into the newly allocated physical page. On the contrary, the update in ARIES mode doesn't change the value of them since it still follows the in-place-update scheme by creating log records instead of copying the page. Secondly, a log record created by an update in ARIES mode contains a physical page number of the data page instead of a logical page number of it. It means that the data page contained at log record is accessed directly without taking the converting step through the page table during recovery or abort. These two treatments with the description in Section 3 about our variants of ARIES and shadow paging are the core building block for understanding the adaptive logging. More detail description with an example will be continued on Section 4.2 through Section 4.6 after presenting the basic concept of adaptive logging.

4.1 Overview

ARIES is applied as a default recovery mechanism. When a transaction starts and updates are occurred, the corresponding log records are generated in ARIES way. Each data page manages the update state by counting the total size of log records generated from the page. If the size exceeds a predefined threshold value, the logging mechanism of the page is switched to shadow paging. (However, if the page lock of the page can not be acquired due to the other transaction's operations, the logging mechanism will stay with ARIES.) When the switch occurs, a new copy of the page is created in the buffer (selectively) as well as in the disk. Then every following update to the page is reflected into the new copy of it without generating log records until the transaction commits. Updates of the same transaction to other data pages of which the threshold is not exceeded still generate log records. When the transaction commits, all new copies of the data pages from the transaction as well as log records are flushed to stable storage. If the next transaction starts, the logging method for updates to every data page is ARIES again. Shadow paging is only applied adaptively to the page of which total size of log records exceeds the threshold value. Abort and recovery are processed by following ARIES scheme. As we have explained in Section 3, the page table and the free page maps are aborted and recovered in the log-based approach. Subsequently, the effect of redo and undo to the pages created by shadow paging method will be reflected through the page table and the free page maps.

4.2 Normal Processing

Consider the simple example illustrated in Figure 4. Initial state is that logical page 5 (L5) of which physical page number is 2 (P2) (6th slot of the page table in the figure indicates the mapping) is loaded on the first buffer page of the buffer pool and physical pages 0, 1, 2 are already allocated (1st, 2nd and 3rd slots of the free physical page map in the figure show the allocations). Then transaction Trans1 changes the value of area A1, A2, and S1~S5 on the page L5 and then commits. Transaction Trans2 then updates the value of area A3 and A4 on the page L5 too. Let's assume that logging method switches at the moment between update A2 and update S1 because the total size of the log records on the page exceeds the threshold value. (Character 'A' stands for an update in ARIES mode and character 'S' stands for an update in shadow paging mode.) Now, we present the detailed behaviors for each step of the above example. When Trans1 updates area A1 and A2, the corresponding log records are created just as ARIES does except for the indirect page accesses through the page table. At the same time, the total size of log records on the page increased. By the way, the log size of the page is maintained in a transaction boundary during the page stays on the buffer pool. This means two things. First, if the transaction commits, the log size of the page on the buffer pool is reset to zero. Second, if the page is evicted by the buffer replacement, the log size is reset to zero. Therefore, the smaller buffer size, the less chance of switching logging method. Trans1 then requests to update area S1. At this moment, the page L5 recognizes that the log size of the page exceeds the threshold value. This brings up the switch of logging method from ARIES to shadow paging.

The switch action is accomplished by the following 3 steps. 1) New physical page is allocated, which generates a log record ('F.P.' stands for the log record from the free physical page map). 2) The page table is updated to reflect the new physical page number, which generates a log record ('P.T.' stands for the log record from the page table). 3) Optionally, new copy of the buffer page is made at another buffer page with a new physical page number. If the buffer page was dirty when the current transaction updates the page for the first time, the new buffer copy should be made. Otherwise, the copy is avoided. The new buffer copy is required for the area of the page updated by the previous committed transaction. Because after switching, updates in shadow paging don't create log record, there is no way to restore the previous image without new buffer copy during the current transaction aborts. Even though the original physical page is reloaded from the disk, the previous image, that is, post-update image of the previous committed transaction may not exist at the page due to the no-force policy of ARIES. At the same abort situation except the buffer page was not dirty so that there is no new buffer copy, the previous image can be restored by reloading the original physical page from the disk since no dirty means the data page which contains the previous image was flushed to the disk. In summary, as a result of the switching, two physical pages (old and new one) for the page exist at disk and one or two buffer instances (old and new one) of the page exist in buffer pool. Also, the page table indicates the new physical page number for the page and ARIES log records of the page created by the transaction contains the old physical page number.

As shown in Figure 4, L5 was loaded on the first page of the buffer pool at initial state. (Let's assume that the page was dirty when the first update of Trans1 to the page is occurred.) After the switch is occurred, the copy of the L5 is made at the third page of the buffer pool. Although page L5 has two instances on the buffer pool temporarily, it is not a problem to access the valid page since they have different physical page number. However, it reduces the available buffer pages. Also, if the buffer pool is not large enough to prevent the old instance of L5 from being flushed due to the page eviction, the old instance increases write overhead. Otherwise, it can be discarded on commit time by managing old buffer instance list of the transaction without causing the write overhead. This explains that the larger the buffer pool size, the more beneficial to adaptive logging. After the switch, the following updates of S1~S5 are reflected to the new copy of the page and these updates don't make any log record. Then transaction1 commits.

The commit action consists of the following steps. 1) Newly allocated data pages and data pages updated in shadow paging scheme are flushed to disk. 2) The log size of each page in the buffer updated by the transaction is reset to zero. 3) Log records including the commit log record are flushed, where the commit log record should be the last one to be flushed. 4) Discard the old buffer instances if they exist. As described above, the old buffer instances which survive from the page eviction until the transaction commits can be discarded without causing write overhead. Nonetheless log records from the data page still should be written to the disk. Actually, these log records are useless as far as the data page is flushed on commit time. However, because the log records are interleaved at log page(s) with another log records which are created from the other data pages updated in ARIES scheme only, it is not practical to remove the useless log records from the log page(s). This is because the removal operation causes the compaction of the remaining log records to reduce the number of the log pages to be written to disk and subsequently the compaction causes the modification of the previous LSN in the log records and the corresponding page LSN on the data pages. This overhead of writing useless log records can be improved with deferred logging to be presented in Section 5. Updates of area A3 and A4 from Trans2 are processed in ARIES mode again with counting the log size of the page. Timeline shown in Figure 4 summarizes the sequence of the events.

There may be a question about how the LSN at a page updated in shadow paging is managed since it doesn't generate any log record so that there will be no LSN at the page and consequently can make adaptive logging work incorrectly. However, LSN at the page does exist and it is handled in the same way that ARIES does. The fact that the switching occurs from ARIES to shadow paging indicates that the page was updated in ARIES so that it updated LSN at the page accordingly until the switching occurs. Although there is no more LSN update after switching, still the last LSN is managed in the buffer instance of the new physical page and applying force policy to the page on commit time can solve the potential problems resulted from the lack of LSN update. During recovery, regardless of the commit of the transaction, redo operation of every ARIES update to the page which is old physical page will be executed according to the result of the comparison between LSN at the page and LSN of the log record. Regardless of the redo operation to the old physical page, if the transaction completed to commit before the crash, the new physical page is guaranteed to contain both the effect of the transaction's operations and the LSN from the old physical page due to the force policy. Otherwise, during the recovery, redo and undo will be applied to the old physical. This is the comparable rationale with handling the LSN at the newly allocated page as described in Section 2.

4.3 Abort

The core of abort operation in adaptive logging is to remove the effect of updating the page table and is to discard all buffer pages which are updated in shadow paging by the aborted transaction. The other miscellaneous treatments are handled naturally by ARIES scheme.

The abort action is accomplished by the following two steps. 1) Buffer pages updated in shadow paging are discarded if any, which can be managed by a certain data structure for each transaction. 2) Log-based undo is executed just as ARIES does, which removes the effect of updating the page table and restores pre-update images of the other data pages at the same time.

Log-based undo is executed as follows. Log records from the transaction are read to backward direction at the log file and undo is executed one by one. For ease of explanation, let's assume there is no other concurrent transaction and only one page is updated as the example in Figure 4. The first undo is applied to the page table and the next to the free physical page map, then to the old physical page. Every log record contains the original physical page number since the log records were created before the switching occurs. When undo is executed to the original physical page, if the new buffer copy was not made during the switching, the old physical page at disk is reloaded to the buffer and then undo is applied to it. This is because the buffer instance of the page in the buffer represents the new physical page after switching. By contrast, if the new buffer copy was made and the old buffer instance still exists on the buffer, undo is applied to the old buffer since the old buffer still represents the old physical page. If the old buffer doesn't exist on the buffer, old physical page is reloaded to the buffer and then undo is applied to it. As described in Section 4.2, the new buffer copy is required to restore the post-update image of the previous committed transaction on the page if any. After completing the abort, every access to the page is directed to the original physical page since the page table was restored to the original one. The new physical page can be used for the next new page request since the free physical page map was restored too.

4.4 Recovery

Recovery is very simple. There is nothing special treatment in the recovery operation of the adaptive logging comparing with ARIES. It just follows the 3 phase recovery steps such as analysis, redo and undo just as ARIES does. If the log-based recovery completes, the atomicity of the updates in the shadow paging mode is guaranteed by the free page maps and the page table which have been just restored from the log-based recovery.

Log-based recovery is executed as follows. Through the analysis phase, information of loser transaction is collected at transaction table and information of dirty page is collected at dirty page table. Then redo is executed from the earliest recovery LSN at the dirty page table to forward direction. Finally, undo is executed from the latest LSN of the loser transaction to backward direction. To notice at the redo phase is that because a log record contains the physical page number, redo to the corresponding physical page can be executed correctly even if the logical page number in the page table is indicating a different physical page number as the result of switching logging method or even if the logical page number is indicating invalid physical page number updated by the loser transaction. Even though the page has the updates in shadow paging as well as ARIES so that the lack of page LSN update was involved during shadow paging scheme, the correct redo/undo operation is guaranteed by managing the last LSN from the old physical page at the new physical page and the force policy to the page as described in Section 4.2.

4.5 Concurrency Control

With respect to the concurrency control, multi-granularity locking of page- and tuple-level can be supported by following two phase locking (2PL) protocol according to the current logging method of each page in adaptive logging. The inherent property of shadow paging limits the lock granularity at most to the page-level. It follows that if more than one transaction access a same page, it is impossible for any transaction to switch from ARIES to shadow paging due to the failure of page lock acquirement. In this situation, every transaction for the same page should work in ARIES scheme until the page lock can be acquired. If the logging method for the page is switched to shadow paging, no other transaction can access the page until the transaction commits.

4.6 Correctness

Adaptive logging works based on ARIES and shadow paging. If there is no large update to a page, adaptive logging works just as ARIES does except indirect page access through the page table and force commit for the newly allocated page. The ARIES only mechanism is clear enough to understand. What we concern is to examine the correctness of the algorithm when shadow paging is used together. Even though shadow paging is used with ARIES, we still follow the golden rule of each recovery algorithm. When data pages are flushed, WAL protocol and out-of-place update are guaranteed for the pages updated in ARIES and shadow paging. It enables to remove the effect of the operation from the non-committed transaction even if the steal policy is applied to the buffer. Steal/no-force policy to pages updated in ARIES scheme (with force policy to log) and force policy to pages update in shadow paging are followed. It enables to make durable the effect of the operation from the committed transaction. Also 2PL protocol is followed for concurrency control. By following these rules for each operation to the database as described in Section 4 so far, we can guarantee the ACID property of the transaction.

5. DEFERRED LOGGING

In this section, we present a log compaction method called deferred logging which can also contribute to reduce logging overhead with adaptive logging scheme. This method defers to write log information to the log buffer by managing the each log information called *log entry* at the separate main memory of for each data page apart from log buffer. When the data page is flushed or the transaction is committed, the corresponding log entries are converted to the log records, written to the log buffer, and then the log buffer is flushed to the disk just before the data page is flushed or the transaction completes to commit. As a result of deferred logging, we have a chance to compact log entries which are created from the adjacent or overlapped area updated to the same page. It is impractical operation to compact log records which have been written to the log page since the compaction makes the field of the previous LSN in the log records modified according to the compaction and also the LSN at the corresponding data page should be modified accordingly. However, compacting log entries is free from the overhead of modifying both the previous LSN and the LSN at the page since log entries do not have the previous LSN.

5.1 Rationale

A log entry consists of 4 fields such as image length, offset of the update area in the page, pre-update image and pointer to the next log entry. Log entries are managed at the separate memory for each data page apart from the log buffer. Figure 5 describes how log entries are created and managed with a simple example. There were 3 updates to a data page, 2 updates from a transaction T1 and 1 update from T5 and another update from T1 is now

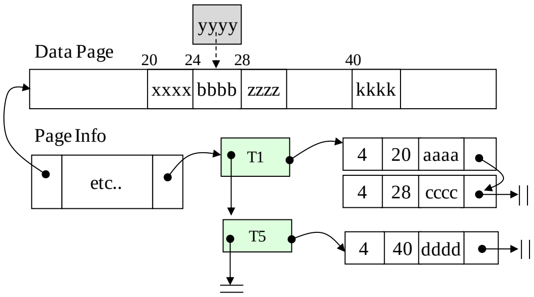


Figure 5. Deferred Logging Example

occurring. Log entries of T1 explain the update history such as 4 bytes of image 'aaaa' is changed into 'xxxx' at offset 20 and 4 bytes of image 'cccc' is changed into 'zzzz' at offset 28. The data page contains the post-update image and the log entry contains the pre-update image. After the update of 'yyyy' is executed, the log entries of T1 will be compacted into a new log entry representing that 12 bytes of image 'aaaabbbbcccc' is changed into 'xxxxyyyyzzzz' at offset 20. In the similar manner, if there is a new update of which area is identical with the existing log entry, copying the post-update image to the data page is the only necessary operation without modifying the existing log entry since the existing log entry already has the pre-update image.

A log entry is converted into a log record when one of two events happens. First, when a transaction commits, data pages updated by the transaction are found at the buffer pool (which can be managed by a certain data structure), log entries for each page are converted into log records, log records are written to a log page and then the log page is flushed to disk. As soon as the log record is written to the log page, the LSN of the data pages is updated accordingly. Second, when a data page is evicted, log entries of the page are converted into log records, and the log records are written to a log page, and then the log page is flushed to disk. The LSN of the page is also updated. There is a difference between two events in the perspective of log entries. When a transaction commits, log entries only from the transaction are converted into log records, by contrast, when a page is evicted, all log entries from the page are converted into log records.

When a transaction aborts, the pre-update images in the log entries of the transaction are restored and then the pre-update images in the log records from the transaction are reflected if any. There is no special treatment for deferred logging during recovery process since the log entry is only managed at the main memory so that if system crashes, we can restore the consistent state of the database by following ARIES recovery scheme.

The rationale described so far can be applied to not only physical logging but also physiological logging as far as 2PL protocol is followed by both logging methods.

5.2 Coupling with Adaptive Logging

Deferred logging in itself is a complete method which can improve logging overhead by compacting log entries. Also deferred logging can be coupled with adaptive logging seamlessly, which can remove the overhead of writing useless log records in adaptive logging.

As described in Section 4.2, without deferred logging, if the switching from ARIES to shadow paging occurs on a data page, log records created from the page before the switching should be written at most on commit time even if the page survives from the eviction. However, with deferred logging, log entries (not log records) created from the data page before the switching can be discarded as soon as the switching is completed. This is enabled by keeping the log entries in the separate memory from the log page. Consequently, the overhead of writing useless log records is disappeared in adaptive logging.

6. RELATED WORK

We survey related work to the problems that we address. First we discuss other approaches with support for hybrid logging and then we discuss approaches related with deferred logging and log compaction.

Cabrera et al. [6] showed that no single recovery method provides the satisfactory performance to all transaction types. They then proposed to choose an appropriate recovery method according to the property of the transactions. For example, ARIES is applied to small update transactions and shadow paging is used for large update transactions, especially for updating large objects. However, this static hybrid approach is applicable only to a situation where all the properties of the transactions are known before the transactions start. Furthermore, although the properties are known in advance, two transactions which show different update patterns to the same page can not be handled effectively. Page-oriented recovery has constraints in handling a record larger than a page and reordering updates to the same page due to the LSNs on pages. Segment-based recovery [17] overcomes these constraints by introducing a segment, which is a set of bytes which may span page boundaries, with LSN-free pages [13]. It also shows how to build a hybrid system allowing ARIES and segments to coexist but it doesn't work adaptively. ARIES [8] also does not prevent the shadow page technique from being used for selected potions of the data to avoid logging of only undo information or both undo and redo and introduces that it may be useful for dealing with long fields.

Log folding technique [10] was proposed to merge logically redundant logs when the transferred updated information, i.e. redo log is reflected to the remote secondary site. Similarly, semantic compaction technique [11] was proposed in the context of the log-structured B-Tree indexing [12]. This technique discards log records having opposite semantics and replaces multiple log records from the repeated update to the identical object with the last log record. Both techniques are different from our approach since we reduce the number of log records before the log records are created. Whereas the log folding technique omits reflecting logically redundant logs after the logs are transferred to the remote site and the semantic compaction occurs during the log garbage collection, which means the compaction occurs after the log records written to the storage.

7. PERFORMANCE EVALUATION

In this section, we examine the performance of adaptive logging and adaptive logging with deferred logging comparing with ARIES and shadow paging. We ignore the effect of the concurrent execution among transactions since it is not critical factor in mobile devices so far. We have implemented to our embedded DBMS adaptive logging coupled with deferred logging based on the physical logging method, i.e. all of what we have described in the paper except for the concurrency level. Although our approach can support multi-granularity locking of page- and tuple-level, we have implemented table-level locking currently. In order to evaluate the performance, we also implemented a benchmark based on the music browser application which generates most likely workloads referring to the queries, the indices, the table schema and the data set of the real world application which has been deployed to more than 10 million mobile devices. The rest of this section is organized as follows. Section 7.1 describes the experimental environment and the benchmark. Section 7.2 through Section 7.4 analyze the performance results.

MusicTable

Column Name	Data Type (bytes)
ID	Integer (4)
Genre	String (256)
Artist	String (256)
Album	String (256)
Title	String (256)
LatestPlayDate	Integer (4)
LastMonthPlayCount	Integer(4)
ThisMonthPlayCount	Integer(4)
Preference	Integer(4)

Index

Type	Indexed Columns
Single Column Index	Every field has single column index.
Multi-Column Index	(Genre, Artist), (Genre, Album), (Artist, Album), (Artist, Album, Title)

Query

Type	SQL (or Description)
Q1 : Insert	Insert a specified number of new records.
Q2 : Update	update MusicTable set Preference = Preference + 1 where ThisMonthPlayCount < ? and LastMonthPlayCount > 0
Q3 : Update	update MusicTable set ThisMonthPlayCount = ThisMonthPlayCount + 1 where ThisMonthPlayCount > ?
Q4 : Delete	delete from MusicTable where ThisMonthPlayCount < ?
Q5 : Select	select * from MusicTable where ID = ?

Figure 6. Table Schema, Indices, and Queries of the Benchmark

7.1 Setup for Experiment

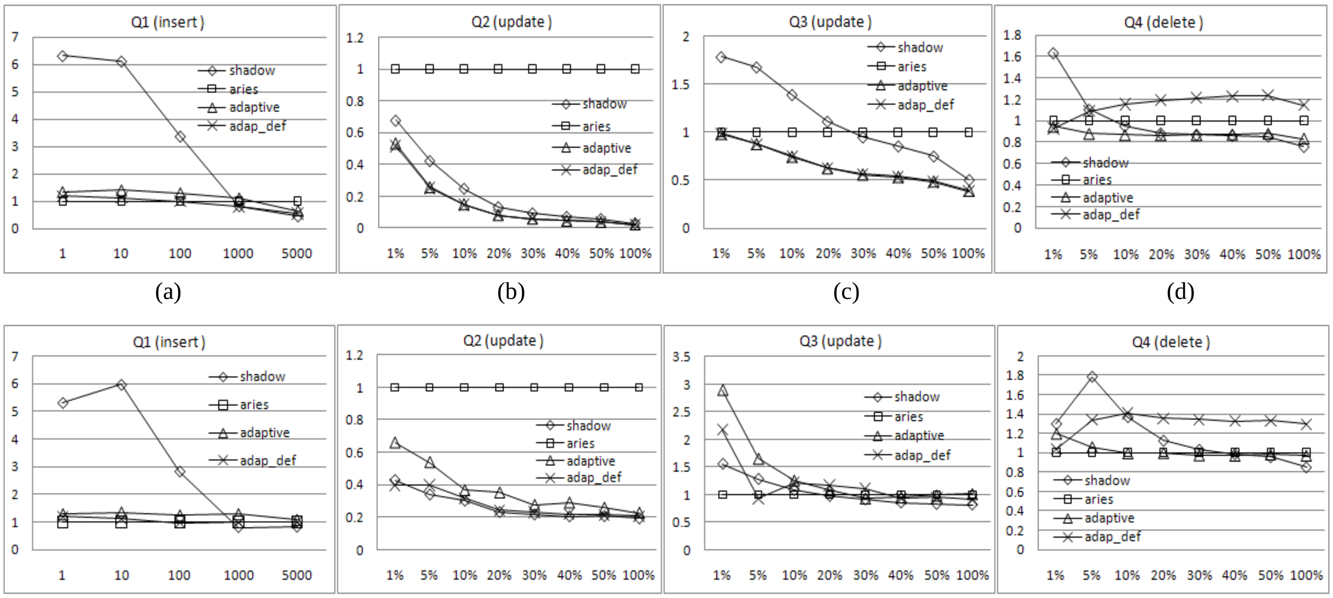
We varied the experimental parameters such as update transaction size, buffer size, and threshold value of adaptive logging to show the causality between the parameters and the performances for each approach. Figure 6 shows the table schema, the indices and the queries of the benchmark used in the experiments. Q2 and Q3 don't make sense in the perspective of the semantic. However, in order to make it easy to vary the update transaction size instead of modifying the play count values, we have modified the preference update queries described in Section 2 into Q2 which still follows the update patterns of them. Also, the query in Section 2 of which predicate field is same with update field is reflected into Q3. With respect to the data set, 10,000 records were used and the average length of the records was around 80 bytes. During the experiments, the database size reached around 10 mega bytes (MB) at maximum. The page size was fixed to 4,096 bytes for data page and log page together and the same size was used as the threshold value of the adaptive logging. 4MB buffer pool was used. The performance was measured by I/O count and elapsed time. For proprietary reason, we provide the normalized value of them. Depending on the experiment, the real elapsed time took less than 10 seconds through less than 1,000 seconds. The experiments were run on ARM11 500MHz MX37 Freescale chip with 8GB MLC NAND flash memory, running Linux 2.6.

7.2 Varying Update Transaction Size

Prior to running the insert/delete/update operation, we made the initial database which had 5,000 records prior to insert operation and had 10,000 records prior to update and delete operations. Insert operation inserted 5,000 records, where the insert operation was executed by varying the number of transaction such as 5,000, 500, 50, 5, and 1, which means that each transaction inserted 1, 10, 100, 1000 and 5,000 records, respectively. The variation of the number of records for each transaction to insert is to make variation of the update pattern of the data pages affected by the insert operation. Figure 7(a) and 7(e) show the result of the insert operation. X-axis shows the number of records for each transaction to insert, which means that the left side stands for the small update transaction and the right side stands for the large update transaction. Y-axis shows the I/O count and the elapsed time of the insert operation, which is normalized by the count and the elapsed time of ARIES, respectively.

As we have expected, ARIES (denoted as aries) outperforms shadow paging (denoted as shadow) for small update transaction, but shadow paging doesn't show the impressive performance gap against ARIES for large update transaction even though it shows a little better result. The latter case results from the fact as follows. As the number of records inserted is increased within a transaction, the more number of new pages are allocated. As described in Section 2, the updates to the newly allocated pages do not create log records and the pages are flushed to stable storage on commit time. The fact is valid for all four approaches, which results in the small performance gap for large update transaction. Adaptive logging (denoted as adaptive) follows gracefully the performance of ARIES approach for small update transaction and shadow paging approach for large update transaction. Adaptive logging with deferred logging approach (denoted as adap_def) shows better performance than adaptive logging as a rule except for Q4. For adaptive logging and adaptive logging with deferred logging, the I/O count always shows better result than the elapsed time throughout all queries. Also, even though the I/O count of them shows better performance than other approaches, the elapsed time of them shows worse performance in some cases. These results from the additional overhead of CPU and memory operation for both adaptive approaches such as indirect page access through the page table, making the new buffer copy if necessary, memory allocation/de-allocation for log entry, and compacting log entries.

Update operation is executed by a transaction which updates the specified percent of records on x-axis shown in Figure 7(b), 7(c), 7(f), and 7(g). Q2 is different from Q3 at two perspectives; the first one is the ratio of the key duplication of the index and the second one is the identity between the field to be updated and the predicate field. The 'Preference' field has only 10 distinct keys among 10,000 records, which means 1,000 records have the same key in average. Whereas the 'ThisMonthPlayCount' field has the key values from 1 to 1,000 randomly, which means every 10 records have the same key in average. In Q3, the field to be updated is the same field with the predicate field, which has more chance to create S-pattern when update occurs since the qualified key of the predicate will be collocated at the same page. As shown



(a) (b) (c) (d)

(e) (f) (g) (h)

Figure 7. I/O Count(a to d) and Elapsed Time(e to h) Normalized by ARIES Varying Update Transaction Size

in Figure 7(b) and 7(f), the effect of the key duplication at B+-tree index has significant impact on the performance of the update operation. The performance results from the sorting effect of the extent page of B+-tree index as described in Section 2. In terms of I/O count, both adaptive logging approaches outperform shadow paging as well as ARIES since they benefit from S-pattern at the extent page of the index and A-pattern at the page of the table. However, the elapsed time is worse than shadow paging due to the CPU and memory operation overhead. The performance analysis of Q2 can be applied to it of Q3. However, the performance superiority of adaptive logging approaches are less than the Q2 case, which means that the effect of the key collocation of the index on the same page is less significant than the effect of sorting on extent page of the index.

Just like the update operation, delete operation is executed by a transaction which deletes the percent of records as specified in x-axis of Figure 7(d) and 7(h). Interestingly, adaptive logging with deferred logging shows the worst performance. This unusual result is caused by the frequent page eviction at the buffer pool. Unlike updating a field of record, delete involves deletion of all corresponding keys from the related indices. As the percent of records to be deleted is increased in a transaction, much more pages should be updated than one field update operation. When a page is evicted, log entries of the page are converted to log records at log page(s), then log page(s) is(are) flushed. Adaptive logging with deferred logging can flush log records only from the evicted page. Whereas ARIES can flush log records from all other updated pages together at the eviction moment. It means that whenever a page is evicted, flushing log records must be involved in deferred logging approach. Therefore, deferred logging approach should be used carefully considering the available buffer space of the system.

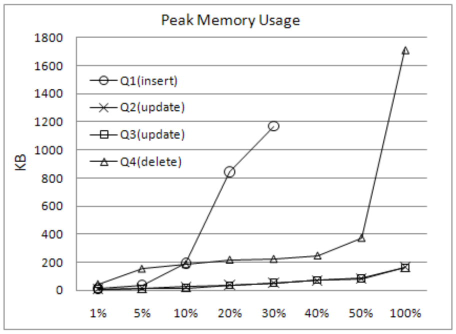


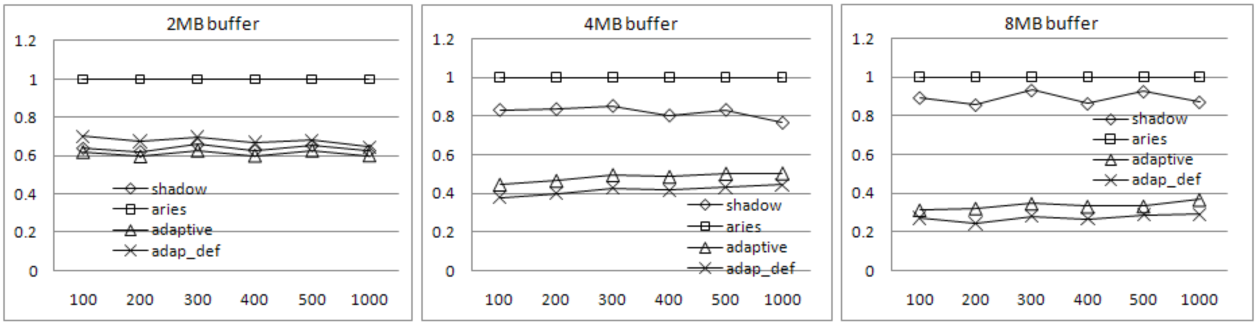
Figure 8. Peak Memory Usage of Deferred Logging

Figure 8 shows the peak of the memory usage used by adaptive logging with deferred logging approach during the each query execution. For Q1, x-axis values 1% through 30% stands for the 1 through 5,000, respectively. Most cases show the moderate memory usage except some cases such as 1,000/10,000 records insertion and 100% deletion. Considering the peak memory usage with the results in Figure7, we can explain that the memory usage can be varied according to the several parameters such as the number of data page updated, the compaction condition for each page, and also the available buffer space.

7.3 Varying Buffer Size

As we have described the impact of the buffer size on adaptive logging in Section 4, we present the result of varying buffer size in this section. We have tried to reflect the real world user's behavior in certain degree at this experiment. The same table schema, indices, and queries shown in Figure 6 were used but the number of records affected by the queries was fixed as follows; Q1 inserts 100 new records. Q2 updates 100 records. Q3 updates only 1 record. Q4 deletes 100 records. Q5 finds only 1 record. Each query consisted of single transaction. We made 10 transactions by combining the 5 queries, which have the sequence as follows;

Q5 → Q5 → Q5 → Q3 → Q3 → Q2 → Q5 → Q4 → Q5 → Q1



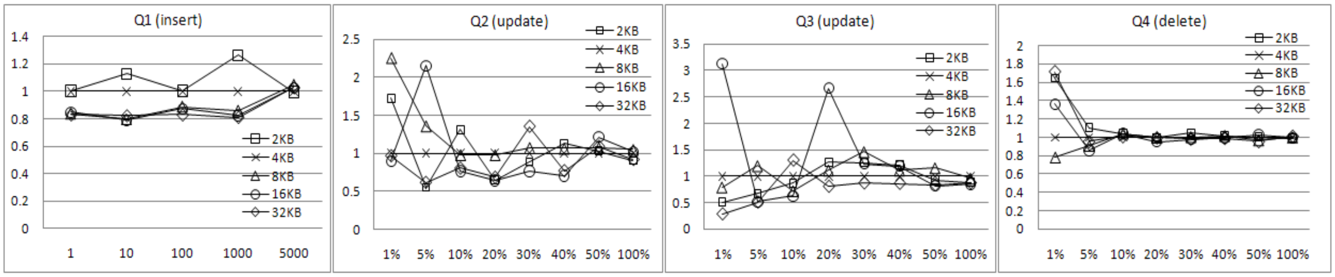
(a) (b) (c)

Figure 9. Elapsed Time Normalized by ARIES Varying Buffer Size

The sequence represents that write operation is usually preceded by select operation. As shown in Figure 9, we changed the buffer size into 2MB, 4MB, and 8MB. X-axis in the figures specifies the number of transaction executed. For example, the number 100 stands for 100 transactions and this means that the above sequence of transactions were executed 10 times and the number 200 means the series of transactions were executed 20 times and so on. Y-axis shows the elapsed time normalized by ARIES. Prior to running each case, the initial database had 10,000 records and it took around 8MB size and it spanned to around 10 MB during the experiments. The result shows that shadow paging and two adaptive approaches outperforms ARIES in all cases. It explains that even though 20% of 1 record update operation and 50% of select operation were included, the elapsed time is dominated by the 30% of large update transactions. The more buffer size, the higher performance gap between two adaptive logging approaches and ARIES. Since two adaptive logging approaches can make a better decision for switching logging method and log compaction when the buffer space is sufficient. However, the tendency is opposite to shadow paging. This shows that shadow paging has less chance to benefit from the large buffer size than both ARIES and two adaptive loggings. For shadow paging, the updated pages should be written to the durable storage on commit time regardless of the buffer size. Whereas, for ARIES and two adaptive logging approaches, more buffer size, less chance for the eviction of updated pages during normal processing. When the buffer size is not sufficient, all approaches may face to frequent page eviction, but ARIES and adaptive loggings suffer from flushing the log page at the same time. These are very typical situation which shows the pros and cons of both steal/no-force policy of log-based approach and force policy of shadow paging. To make matters worse, deferred logging approach suffers the more frequent log converting and log flushing overhead. This is the reason of the worse performance than shadow paging and adaptive logging as described in Figure 9(a). However, the peak memory usage of it doesn't exceed more than 300 KB in all cases.

7.4 Varying Threshold Value

The threshold value of adaptive logging is one of the most critical parameter which may show different tendency of the performance. To show the impact of it, we have applied varying the threshold value of adaptive logging to two experiments run in Section 7.2 and the 4MB buffer case of Section 7.3. The threshold value begins with 2KB (the half of the page size) and ends to 32KB by doubling each value. The result of the first and the second experiment is shown in Figure 10 and in Figure 11, respectively. Y-axis of both figures presents the elapsed time normalized by the 4KB value. In Figure 10, no specific value shows the superior performance than others for all cases but still the performance is sensitive to the threshold value. It means that assigning a threshold value statically can not be the appropriate solution for achieving optimal performance since the update state of each page keeps varying as the percentage of the qualified records is increased. If we can dynamically assign the threshold value for each page according to the update state of it, we can achieve better performance. This will be explored in our future work. Without this work, the page size (4KB) is the reasonable second choice since it is a barometer to decide whether the no-force policy can benefit or not and it generally shows the average performance in all cases. The delete operation is not sensitive to the threshold value from 10% interval due to the frequent page eviction as described in Section 7.2. Unlike the result of the separate operations in Figure 10, the result of the mixed operations in Figure 11 shows a tendency. 8KB shows the best performance in all cases. As the threshold value is increased, the



(a) (b) (c) (d)

Figure 10. Elapsed Time Normalized by 4KB Threshold Varying Threshold Value

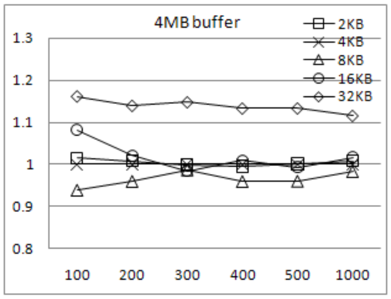


Figure 11. Elapsed Time Normalized by 4KB Threshold Varying Threshold Value

performance is also getting better until reaching to 8KB. After 8KB, however, the result is getting worse. This means that there is a threshold value which can outperform others. However, the best threshold value is unknown in advance for the upcoming workloads, still the page size is the reasonable choice as shown in the result.

8. CONCLUSIONS

We have proposed adaptive logging as well as deferred logging to reduce logging overhead and showed the benefit by evaluating them through a real world application on a mobile device. Adaptive logging works based on ARIES and shadow paging adaptively according to the update patterns of each page, which overcomes the limitation of the conventional static logging approach. Deferred logging provides a chance to reduce log size by compacting redundant log entries and also removes the overhead of writing useless log records in adaptive logging. Dynamic assignment of the threshold value of adaptive logging will be explored in our future work.

Even though we verified our approaches based on physical logging, we believe that the idea also can be applied to physiological logging. Furthermore, although we found the problem from applications on the mobile devices, our approach can be applied to the server-side DBMS as long as the workloads have the update patterns of both small update transaction and large update transaction.

9. ACKNOWLEDGEMENTS

We would like to thank S. Sudarshan and Krithi Ramamritham for their thorough comments. We also thank project members of AceDB and AceAnalytics for their kind support.

10. REFERENCES

[1] Sybase's Ultralite, http://m.sybase.com/products/databasemanagement/sqlanywhere/blackberry

[2] Sqlite, http://www.sqlite.org/

[3] Gye-Jeong Kim, Seung-Cheon Baek, Hyun-Sook Lee, Han-Deok Lee, and Moon Jeung Joe. LGeDBMS: A Small DBMS for Embedded System with Flash Memory. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, Seoul, Korea, September 12-15, 2006

[4] Ki Yong Lee, Hyojun Kim, Kyoung-Gu Woo, Yon Dohn Chung, and Myoung Ho Kim. Design and implementation of MLC NAND flash-based DBMS for mobile devices. *J. Syst. Softw.* 82, 9 (Sep. 2009)

[5] Intel. Understanding the Flash Translation Layer (FTL) Specification. Application Note AP-684, Intel Corporation, December 1998.

[6] Cabrera, L.-F., McPherson, J., Schwarz, P., Wyllie, J. Implementing Atomicity in Two Systems: Techniques, Tradeoffs, and Experience, *IEEE Transactions on Software Engineering*, Vol. 19, No. 10, October 1993

[7] Raymond A. Lorie. Physical integrity in a large segmented database, *ACM Trans Database Syst* 2, 1 (March 1977), 91-104.

[8] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17(1): 94-162(1992)

[9] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. "A History and Evaluation of System R" *Communications of the ACM*, Vol. 24, No. 10, October 1981, pp. 632-646.

[10] Kazuo Goda and Masaru Kitsuregawa. Power-aware remote replication for enterprise-level disaster recovery systems. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference* (Boston, Massachusetts, June 22 - 27, 2008). USENIX Association, Berkeley, CA, 255-260.

[11] Suman Nath and Aman Kansal. FlashDB: dynamic self-tuning database for NAND flash. In *Proceedings of the 6th international Conference on information Processing in Sensor Networks*

[12] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang. An efficient b-tree layer for flashmemory storage systems. In *Proc. 9th Intl. Conf. on Real-Time and Embedded Computing Systems and Applications*, 2003.

[13] Russell Sears and Eric Brewer, Stasis: flexible transactional storage, *Proceedings of the 7th symposium on Operating systems design and implementation*, November 06-08, 2006, Seattle, Washington

[14] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. I/O Reference Behavior of Production Database Workloads and the TPC Benchmarks - An Analysis at the Logical Level. *ACM Transactions on Database System*, 26(1):96–143, 2001.

[15] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[16] Raghu Ramakrishnan and Johannes Gehreke. *Database Management Systems.* McGraw Hill, 1999

[17] Russell Sears and Eric A. Brewer: Segment-based recovery: Write ahead logging revisited. *PVLDB* 2(1): 490-501 (2009)