Accommodating Logical Logging under Fuzzy Checkpointing in Main Memory Databases[†]

Seungkyoon Woo Myoung Ho Kim Yoon Joon Lee
Department of Computer Science
Korea Advanced Institute of Science and Technology
373-1 Kusung-Dong, YuSung-Gu, Taejon, 305-701, Korea
{skwoo,mhkim,yjlee}@cs.kaist.ac.kr

Abstract

This paper presents a simple and effective method to reduce the size of log data for recovery in main memory databases. Fuzzy checkpointing is known to be very efficient in main memory databases due to asynchronous backup activities. By this feature, most recovery works in the past have used only physical logging schemes. Since the size of physical log records is quite large, physical logging schemes cause the recovery time to be much longer than that using logical logging schemes. In this paper, we propose a hybrid logging method that can accommodate logical logging under fuzzy checkpointing. This method significantly decreases the size of log data, and hence speeds up the recovery time. We also propose a reapplying rule in segmented MMDB, which reduces the number of log records for recovery. We evaluate the performance of the proposed methods through analytic analyses. The results show that we can reduce the size of log data to more than half, compared with those that use only physical logging.

Keywords: database recovery, main memory databases (MMDB), fuzzy checkpointing, hybrid logging method, shadow updating, delayed backup

1. Introduction

In Main Memory Databases (MMDB) data reside permanently in main memory. By the significant decrease of memory cost with the fast growth of memory capacity, the importance of MMDB has been increasingly recognized [3]. There are several database systems that use MMDB as a part of databases [6, 9, 15]. Though MMDB can provide faster response time and higher throughput than Disk-Resident Databases (DRDB), all the contents of memory can be lost in system failures due to volatility of main memory. To preserve data against crashes, we need a recovery method that restores MMDB to a consistent state. Many researches have been done in the past for recovery methods in MMDB [2, 4, 7, 8, 10, 11, 14].

MMDB has a goal of high-performance transaction processing. Thus a recovery method must be efficiently processed with little synchronization and fast recovery. Among several tasks in recovery methods, checkpointing is the most important one for these requirements. Since MMDB has no buffering activities for data stored in the memory permanently, an efficient checkpointing scheme is required. Fuzzy checkpointing in MMDB, which has been shown to be effective in DRDB, was introduced by Hagmann [5]. As fuzzy checkpointing flushes dirty pages asynchronously against transaction activities, the quiescence of transactions is not required. However, only physical logging scheme was considered in fuzzy checkpointing due to asynchronous flushing in the past MMDB recovery works [5, 11, 12]. Physical logging causes a recovery time of MMDB to be longer than logical logging. This is because a large amount of log data are produced in physical logging [14]. Logical logging can make a long physical log into a record of only a few words, which can reduce the size of log data significantly. To achieve faster recovery, we need a method to reduce the size of log data by accommodating logical logging.

There are some researches that attempt to reduce the size of log data. DeWitt *et al.* [2] and Hagmann [5] present a log compression method. These methods are to eliminate the "redo" part of log records for aborted transactions and the "undo" portion of log records for committed transactions. In Lehman and Carey [8] and Jagadish *et al.* [7], redo log records are flushed to disks and undo log records are discarded, when a transaction is committed. Salem and Garcia-Molina [14] use a shadow updating policy to record only

[†]This material was supported in part by Ministry of Information and Communication, 1996.

redo log records. However, two former methods are not based on fuzzy checkpointing, and in [14] it has been shown that fuzzy checkpointing still has longer recovery time than other checkpointing methods with logical logging. If secondary index modifications are logged, the recovery time of fuzzy checkpointing would be much longer.

In DRDB there are some works on adapting logical logging under fuzzy checkpointing. Bernstein *et al.* [1] describe a penultimate fuzzy checkpointing method with logical logging. This checkpointing scheme, however, requires the quiescence of transaction processing. The quiescence time is determined by duration of active transactions and the number of dirty pages. Mohan *et al.* [13] introduce a recovery method, called ARIES which is based on a fuzzy checkpointing and WAL logging scheme. ARIES supports logical logging, which is, however, restricted to objects with increment or decrement kinds of operations, e.g., garbage collection and changes to the amount of free space. The two above-mentioned fuzzy checkpointing methods are for DRDB, not MMDB. This means that flushing of dirty pages is based on buffering activities. Because MMDB has data in main memory permanently, the penultimate checkpointing idea in [1] and the scheme of un-flushing dirty page in [13] cannot be applied for MMDB.

In this paper, we propose a simple and efficient logging method that uses both physical and logical log records under fuzzy checkpointing in MMDB. The basic idea of the method is that a transaction writes physical log records during only checkpointing, but uses logical log records in other cases. Shadow updating and private log buffers are used together with hybrid logging to make a transaction consistent checkpoint. We apply the proposed method to segmented MMDB in order to support consecutive checkpointings. We also present an effective method that can significantly reduce the *number* of log records used for recovery.

The rest of the paper is organized as follows. In Section 2, we present the system configurations and the proposed fuzzy checkpointing method. Then we apply the method to segmented MMDB. Section 3 describes the recovery processing and an effective reapplying method. In Section 4, we evaluate the impact and performance of the proposed method through analyses. Finally Section 5 has conclusions and describes some further works.

2. Fuzzy Checkpointing Accommodating Logical Logging

2.1. Basic Concept

Under fuzzy checkpointing [5] in MMDB, the checkpointer flushes dirty pages without considering on transaction activities. Thus only physical logging must be used during checkpointing. With physical log records, we can restore a fuzzy checkpoint to the same physical state as it was. After then, the logical log records can be applied to the checkpoint because the checkpoint is in exactly the same state when the log records were generated. This is the basic idea of our approach that can accommodate logical logging in fuzzy checkpointing. We refer to the approach as *Hybrid Logging Rule*:

> **Hybrid Logging Rule** *Write physical log records during only checkpointing, and use logical log records in other cases.*

By using shadow updating and a private log buffer of each transaction, we can make a checkpoint into a transaction consistent checkpoint. Under shadow updating, only committed transactions can update MMDB, which prevents the partial undo of a transaction and generates only redo log records. Using both shadow updating and private log buffers causes log records of a transaction to be written consecutively to log. Since these features enable to apply redo log records by the unit of a transaction, a fuzzy checkpoint can be restored to a transaction-consistent checkpoint.

2.2. System Configurations

Our hardware system is based on general system configurations, so we do not consider a special stable memory.[1] Main memory is partitioned into two areas: system and database area. The system area is used by an operating system to control the system. The database area is controlled by a database management system (DBMS) and consists of MMDB area, log buffer, and shadow area. We assume that the entire database can be stored in the MMDB area. The log buffer has several log pages in which log records of transactions are stored. A log page is flushed into disk when it is full. The shadow area is used for shadow updating and keeps updated data. During commit works of transactions, the data are moved to appropriate locations in the MMDB area.

In disks, two backup databases are maintained and *ping-pong* backup policy [14] is used. For each checkpoint, one of two backup databases is used alternatively. During a checkpoint, only portions of the database that have been updated are written out to their corresponding position on the backup database according to ping-pong policy. This increases the number of pages to be flushed, but prevents *the violation of WAL*. The violation of WAL is a situation that an updated page is flushed to a backup database before the corresponding log records are flushed to disk. The violation may occur when fuzzy checkpointing is used carelessly. When the system crashes in this situation, the log records have to be applied to the page in order to redo or undo actions issued by the corresponding transaction. However, as

[1] The stable memory is not an important factor on this work.

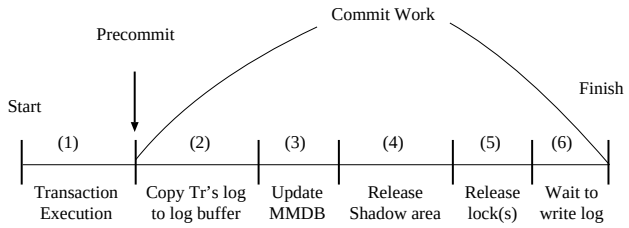

Figure 1. Transaction Processing Model

the log records are not stored in stable log, the page cannot be recovered. This problem can be solved by ping-pong backup policy.

With one backup database, the violation of WAL can be avoided. To do this, the quiescence of transaction processing is required during checkpointing. It is not adequate to MMDB because of high-performance transaction processing. If we use a stable memory as the log buffer, the WAL violation can be avoided without two backup databases and the quiescence. However, most conventional systems are not equipped with stable memory in standard configuration and a method without stable memory can be easily applied to systems with it. So, in this paper, we do not consider the usage of stable memory.

2.3. Transaction Processing

Under shadow updating, updated data of a transaction are first written to the shadow area, not to the MMDB area in place. The updated data are reflected on the MMDB area after normal operations of the transaction are finished. Figure 1 shows a transaction processing model for our recovery method. We use the *pre-commit* scheme that has been presented in [2]. For locking policy a strict two-phase locking protocol is used.

At step 1, operations of a transaction are executed by locking required objects. After that, the transaction mode is changed to pre-commit mode. A transaction in pre-commit mode first writes its log records to the current log page (step 2). Next, MMDB is updated through the updated data of the transaction in the shadow area (step 3). And then, the used pages in the shadow area and all acquired locks of the transaction are released (step 4 and 5). Finally, at step 6 the transaction compares the current log page with the log page that has its log records. If two pages are the same, the transaction waits the current log page to be flushed; otherwise, it just commits. Then the user is notified that the transaction has committed.

According to our hybrid logging rule, the type of log records to be written at step 2 is determined by the activity of the checkpointing. During step 1, a transaction makes logical log records for its operations. These log records are stored in a private log buffer of the transaction. After finishing its operations, the transaction tries to lock the current log page in order to write its log records. After locking the log page, the transaction determines the type of log. If the checkpointer is at work, the transaction writes physical log records to the log page; the log records are constructed from the shadow area. Otherwise, the transaction writes logical log records that are stored in its private log buffer. As the lock of current log page is a synchronization point between transactions and the checkpointer, the checkpointer cannot finish its work while a transaction does logging work.

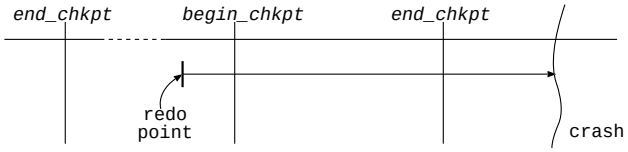

Figure 2. Redo Point of Normal Checkpoint

2.4. Checkpointing

We use ping-pong backup scheme with fuzzy checkpointing. When a new fuzzy checkpoint begins, the checkpointer writes a checkpoint beginning mark, *begin_chkpt*, to the current log page, and then flushes dirty pages to one database backup in disks without considering on locks and other transaction activities. When finishing the backup work, the checkpointer write a checkpoint ending mark, *end_chkpt*, to current log page and flushes the log pages to log disks. After the log page with the *end_chkpt* is written to disk, the checkpointer records the position of *begin_chkpt* at a known location on disk. After that, the next checkpointing can begin and backup dirty pages to another backup database. This is a normal fuzzy checkpointing process [5, 14].

There are two problems in using logical logging under the normal fuzzy checkpointing. One is the WAL violation, but it can be avoided by ping-pong backup scheme. Another problem is related to the redo point of recovery.[2] Figure 2 shows the redo point under the normal checkpointing process. As we do not consider the quiescence of database, some transactions may be updating pages at *begin_chkpt* and partially updated pages can be flushed to backup databases. If the system crashes as shown in Figure 2, the redo point for recovery is determined as the smallest number among log page numbers of these transactions. Thus the redo point is before the *begin_chkpt* in log.

The problem related to the redo point is that the type of log records written by transactions updating some pages at

[2] As using shadow updating policy, we do not consider a transaction undo.

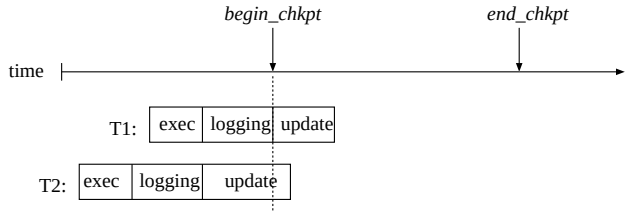

Figure 3. Example of Redo Point Problem

begin_chkpt is of logical log. This is because the hybrid logging rule obeys logical logging during non-checkpoint duration. As an example, consider a situation in Figure 3. $T1$ and $T2$ record their logical log records to a log page according to the hybrid logging rule. With physical log records, we can restore after-images without worrying about the current state of the data. With logical log records, however, the redo log records may only be applicable to a data item when it is in exactly the same logical state as when the log records were created [1]. Thus, the logical log records cannot be applied to the partially updated pages of $T1$ and $T2$.

To solve this problem, we have to correspond the redo point with *begin_chkpt* in the last complete checkpoint. In other words, the checkpointer must begin its backup processing after all transactions updating MMDB at *begin_chkpt* finish their updating works. By delaying the backup beginning point, the checkpointer can do backup dirty pages on which all updated data of transactions with logical log records are reflected. Therefore, we can avoid applying logical log records to partially updated pages. Figure 4 shows this delayed backup mechanism. T_{backup} is the beginning point of backup work. Some pages of $T3$ and $T4$ may be flushed to a backup database during updating, which generates partially updated pages. However, these pages can be recovered because the type of log records of $T3$ and $T4$ is of physical log.

This delayed backup is easily implemented by using a variable. We define the variable as *num_active_tr[ ]*, which represents the number of transactions updating MMDB in

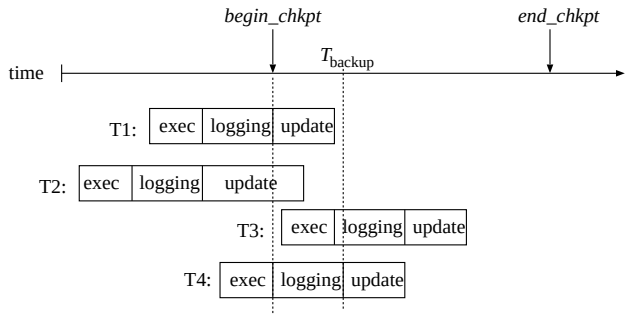

Figure 4. Delayed Backup Concept

```
int num_active_tr[N];
  /* N is the number of log pages. */
  /* num_active_tr[N] are initialized by zero. */
int tail;
  /* tail is an index of num_active_tr[],
     which points the oldest log page with updating
     transaction(s). */
. . .
Checkpointing() {
      Write begin_chkpt to a current log page i;
      Change the current log page
        to the next log page;
      Sleep until tail ≤ i;
      Backup dirty pages;
      Write end_chkpt to a current log page j
        and flush it;
      Change current log page to the next log page;
      Wait the log page j and its previous all log
        pages to be written to log disks;
      Record the position of begin_chkpt to a known
        location on disk;
}
```

Figure 5. Fuzzy Checkpointing with Delayed Backup Timing Point

each log page. After writing its log records to a log page i, a transaction increases *num_active_tr[i]* by one, and decreases *num_active_tr[i]* by one after updating MMDB. When *num_active_tr[i]* is zero, it means that transactions that had written their log records to the log page i finished their updating works. By using this variable, the checkpointer can determine the backup timing point.

Figure 5 shows a fuzzy checkpointing algorithm with the delayed backup timing point. After writing *begin_chkpt* to a current log page i, the checkpointer sleeps until $tail$ comes to log page i. The index $tail$ is moved by transactions. After updating MMDB, the transaction locks the log page j with its log records and decreases *num_active_tr[j]* by one. At that time, when *num_active_tr[j]* is zero and $tail$ points to the log page, the transaction sets $tail$ to the next adequate log page number. If checkpointing is at work, the transaction wakes up the checkpointer.

This approach adds locking overhead to the transaction overhead, because of twice lockings per a log page. However, the first locking is a normal process for transaction processing, and thereby the overhead has little influence on transaction processing. Due to the delayed time added to the checkpointing, the checkpointing interval is increased a little. But updating works at MMDB are processed with-

out disk I/Os, so they are straightforward works. Thus the delayed time may be very small, and the quiescence of the database work does not occur.

2.5. Applying to Segmented MMDB

As fuzzy checkpointing in MMDB has little synchronization with executing transactions, a consecutive fuzzy checkpointing has been considered in some previous MMDB recovery methods [14, 11]. That is, the *end_chkpt* of a checkpoint becomes the *begin_chkpt* of the next checkpoint. In this way the checkpointer is always active, so there is no room to use logical logging.

A way applying our hybrid logging scheme to the consecutive checkpointing is to partition MMDB to several segments and to checkpoint segments circularly in the serial order. A segment consists of one or more pages. Every database object (relation, index, etc) is stored in a segment. When the checkpointer is flushing dirty pages in the i th segment, a transaction uses physical logging for objects in the segment and logical logging for objects in other segments. To do this, we need some information on the relation between a log record and its segment. This information can be stored in the private log buffer of a transaction and the shadow area. The following rule is the hybrid logging scheme in segmented MMDB.

> **Hybrid Logging Rule for Segmented MMDB** *Write physical log records for objects in a segment that is on checkpointing, and use logical log records for objects in other segments.*

The ping-pong backup policy is also used in segmented MMDB. The checkpointer flushes dirty pages in each segment orderly to one of two backup databases in disks. The delayed backup method is used in order to adjust the redo point. Before flushing dirty pages in a segment, the checkpointer delays the backup timing point. A consecutive checkpointing algorithm on segmented MMDB is presented in Figure 6. We use only the beginning mark of a segment *segid*, *begin_chkpt_segid*. The mark indicates the beginning of the checkpoint of a segment *segid* as well as the ending of the checkpoint of just previous segment. Whenever the checkpointer completes the checkpoint of a segment, the checkpointer records the position of *begin_chkpt_segid* at a known location on disk, like the checkpointing in the non-segmented MMDB.

Since we use one log buffer, the delayed time to adjust the redo point at each segment checkpoint is equal to that of non-segmented MMDB. However, the time is so small that it has little influence on transaction processing. We can multiple log buffers, one for each segment. The log records

```
Checkpointing() {
    foreach segment do {
        Write begin_chkpt_segid to a current log
          page i;
        Flush the log page i;
        Change current log page
          to the next log page;
        Sleep until tail ≤ i;
        Record the position of begin_chkpt_segid
          to a known location on disk;
        Backup dirty pages in segment segid;
    }
}
```

Figure 6. Consecutive Fuzzy Checkpointing on Segmented MMDB

of a transaction are stored to the log buffers of corresponding segments. In this case, a global log buffer is required to check the completeness of transactions and checkpoints. In the global log buffer, the marks for beginning and ending of transactions and checkpointings are stored. By using multiple log buffers, the delayed time can be reduced. However, because of having more lock points, the overhead of locking and its contention will be grown.

3. Recovery Processing

As non-segmented MMDB can be regarded as a special case of segmented MMDB with one segment, we generally describe the recovery of segmented MMDB. The recovery for MMDB is composed of two processes: reloading backup database and reapplying log. The reloading process is to move the last complete checkpoint into main memory, and the reapplying process is to apply log data to the reloaded database. Since we use the delayed backup policy for each segment, the redo point of segment i is *begin_chkpt_i*.

For reloading, we have to determine the last complete checkpoint. In the conventional fuzzy checkpointing, the last complete checkpoint has been regarded as $checkpoint_1$ in Figure 7, which shows the checkpointing process of MMDB with 4 segments. The point to note is that the checkpoint of segment 1 after $checkpoint_1$ has a meaningful image. The image is in which all log records of segment 1 generated during the $checkpoint_1$ are applied to the backuped pages of segment 1 in the $checkpoint_1$. This means that the checkpoint of segment 1 after $checkpoint_1$ includes data items updated in $checkpoint_1$. So, we have to reload the pages of segment 1 checkpointed after $checkpoint_1$, not

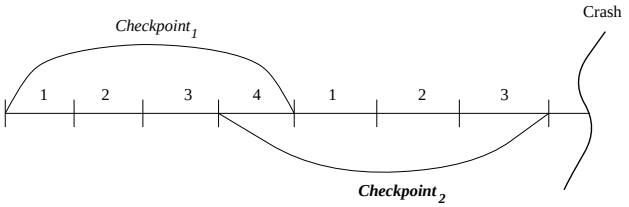

Figure 7. The Last Complete Checkpoint

in $checkpoint_1$. This approach can be applied to segment 2 and 3. Therefore, the last complete checkpoint for recovery is $checkpoint_2$. This approach has been proposed in [12], which reduces the amount of log required for recovery. By applying this approach to our hybrid logging scheme, we can reduce the more size of log data to be read from disks, which reduces the more number of disk I/Os.

After pages backuped during the last complete checkpoint are reloaded into memory, the log records generated during the checkpoint are applied to the pages. However, we do not have to apply all physical and logical log records to the reloaded pages. This is because, according to the delayed backup mechanism, the checkpointing of segment i begins only after all transactions that write their log records before *begin_chkpt_i* finish updating works. This means that backuped pages during the checkpoint of segment i have all after-images of the corresponding log records generated before *begin_chkpt_i*. Thus, when segment i is recovered, we do not have to reapply log records stored before *begin_chkpt_i*.

We focus the reapplying method on the last complete checkpoint. Figure 8 shows components of log records of the last complete checkpoint in Figure 7. The checkpoint of segment 4 begins with *begin_chkpt_4*, and the log records generated during the checkpoint consists of physical log records for objects in segment 4 and logical log records for objects in other segments. When we consider the checkpointed image of segment 1, it includes all updates of logical log records of segment 1 among log records generated during the checkpoint of segment 4; the delayed backup mechanism guarantees this. So, we do not have to apply logical log records of segment 1 in segment 4 to the reloaded database at all. Therefore, we can recover a segment by reapplying only logical log records of the previously recovered segments and physical log records of its segment to the reloaded database.

The reapplying method can be also used in segmented

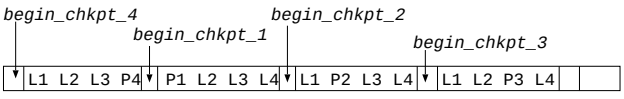

Figure 8. Contents of Log of Figure 7

MMDB under conventional fuzzy checkpointings that permit only physical logging scheme. Following is the general expression of the reapplying policy:

> **Reapplying Rule** *When recovering the last complete checkpoint of segmented MMDB, in each segment reapply log records related to its segment and previous recovered segments in the last complete checkpoint. After recovering the last complete checkpoint, reapply all remain log records to MMDB till log records of the last complete transaction.*

By the last complete transaction, we mean the last transaction whose all log records are stored in log disk. In our configurations, determining whether a transaction is the last complete transaction can be readily checked, because the log records of a transaction are stored in log consecutively. The reapplying rule enables to reduce the number of log records to be applied.

4. Performance Evaluation

This section shows the effect and recovery performance over the hybrid logging scheme through simple analyses. Our metrics are the size of log data generated in a complete checkpoint (*chkpt-log-size*), the size of log data to apply for recovering the last complete checkpoint (*apply-log-size*), and recovery time. *chkpt-log-size* determines the number of disk I/Os and it has great influence on the recovery time. *apply-log-size* is for presenting the influence of the reapplying rule.

We consider consecutive checkpoints on segmented MMDB. MMDB is partitioned to N segments. We assume that transactions access segments with equal access ratio; that is, log records generated during the checkpoint of a segment include S log records for each segment. So, $S \cdot N$ log records are included in a segment checkpoint. Let P be the size of a physical log record and L the size of a logical log record.

Since we assume the equal access ratio to segments, *chkpt-log-size* in only physical logging is the sum of the log size of each segment multiplied by P; that is, $(S \cdot N) \times P \times N = SN^2P$. If hybrid logging scheme is used, the log records generated in a checkpoint consist of physical log records of the checkpointed segment and logical log records of other segment. The size of log records generated during a segment checkpoint is $S \times P + S \times (N-1) \times L$. Thus *chkpt-log-size* is

$$\begin{aligned} & (S \times P + S(N-1) \times L) \times N \\ = \; & S \cdot N(P + L \cdot N - L) \\ = \; & SN^2L + SNP - SNL. \end{aligned}$$

Table 1. Rate of the size of log

Case	Logging	Reapplying Rule	*chkpt-log-size*	*apply-log-size*
1	Physical	X	1	1
2	Physical	O	1	$\frac{1}{2} + \frac{1}{2N}$
3	Hybrid	O	$\frac{L}{P} + \frac{1}{N} - \frac{L}{NP}$	$\frac{L}{2P} + \frac{1}{N} - \frac{L}{2NP}$

Next, we analyze *apply-log-size*. If only physical logging scheme is used and our reapplying rule is not considered, *apply-log-size* in segmented MMDB is SN^2P, like *chkpt-log-size*. When the reapplying scheme is used with physical logging, the *apply-log-size* is

$$\begin{aligned} & S \times P \\ + \quad & S \times P + S \times P \\ & \vdots \\ + \quad & S \times P + S \times P \times (N-1) \\ = \quad & SP(1+2+3+\ldots+N) \\ = \quad & SP \times \frac{N(N+1)}{2} \\ = \quad & \frac{1}{2}SN^2P + \frac{1}{2}SNP. \end{aligned}$$

When hybrid logging scheme is considered, logical log records are applied to the previous segments by using the reapplying rule. The *apply-log-size* thus is

$$\begin{aligned} & S \times P \\ + \quad & S \times P + S \times L \\ + \quad & S \times P + S \times L \times 2 \\ & \vdots \\ + \quad & S \times P + S \times L \times (N-1) \\ = \quad & SPN + SL(1+2+3+\ldots+N-1) \\ = \quad & SPN + SL\frac{(N-1)N}{2} \\ = \quad & \frac{1}{2}SN^2L + SPN - \frac{1}{2}SNL. \end{aligned}$$

To evaluate the impact of physical logging and the reapplying scheme, table 1 shows rates of the log size in hybrid logging to the log size in only physical logging; that is, all above equations are divided by SN^2P. According to [14], we assume that L is 64 words and P is 192 words. The rate of reduced size of log data to varying N is presented at Figure 9. With 4 segments, the size of log data generated during a checkpoint can be reduced to half, compared with that of only physical logging. For *apply-log-size*, we can reduce it to less than half by using hybrid logging and

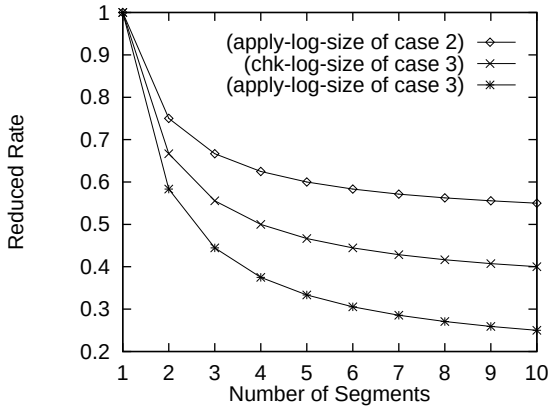

Figure 9. Rate of The Reduced Log

the reapplying scheme, so the recovery process can be performed faster. For simple and straight analyses, we consider only *chkpt-log-size* on next analyses.

Next, we consider a database with hotspots and assume that f_{H} portions of all database pages have $(1 - f_{\mathrm{H}})$ of accesses, e.g., *20-80 rule*. Let H be the *number* of log records generated during a complete checkpoint. If a database is partitioned into two segments: hotspot segment and non-hotspot segment, we can assume that rates of generation of dirty pages and checkpointing time in the hotspot segment are similar to the access rate of hotspot, $(1 - f_{\mathrm{H}})$. Thus, the number of log records generated during the checkpoint of the hotspot segment is $(1 - f_{\mathrm{H}}) \times H$. When considered a uniform distribution over accessed positions, $(1 - f_{\mathrm{H}})$ portions of $(1 - f_{\mathrm{H}}) \times H$ log records are related to the hotspot segment. These portions of log records are physical log records and the remainders are logical log records. We can also apply this idea to the non-hotspot segment. Thus, *chkpt-log-size* with two segments is

$$\begin{aligned} & (1-f_{\mathrm{H}})H[(1-f_{\mathrm{H}})P + f_{\mathrm{H}}L] \\ + \quad & f_{\mathrm{H}}H[f_{\mathrm{H}}P + (1-f_{\mathrm{H}})L] \end{aligned}$$

We expand the above idea to N segments: $\lceil N \times f_{\mathrm{H}} \rceil$ hotspot segments and $(N - \lceil N \times f_{\mathrm{H}} \rceil)$ non-hotspot segments. Let N_{H} be $\lceil N \times f_{\mathrm{H}} \rceil$. We assume every hotspot (or non-hotspot) segments has same access rate. Thus the number of log generated during the checkpoint of a hotspot segment is

$$(1 - f_{\mathrm{H}})H \times \frac{1}{N_{\mathrm{H}}}.$$

Among the above number of the log records, $(1 - f_{\mathrm{H}})/N_{\mathrm{H}}$ portions are only related to the segment that is on checkpointing, and the type of these log records is of physical log. The type of log records for remaining other hotspot segments and for all non-hotspot segments is of logical log.

Thus the size of log data for a hotspot segment is

$$\frac{(1-f_{\rm H})H}{N_{\rm H}} \times \left[\frac{(1-f_{\rm H})}{N_{\rm H}}P + \frac{(1-f_{\rm H})}{N_{\rm H}}(N_{\rm H}-1)L + f_{\rm H}L\right].$$

Similarly, the size of log data for a non-hotspot segment can be given by

$$\frac{f_{\rm H}H}{N-N_{\rm H}} \times \left[\frac{f_{\rm H}}{N-N_{\rm H}}P + \frac{f_{\rm H}}{N-N_{\rm H}}(N-N_{\rm H}-1)L + (1-f_{\rm H})L\right].$$

Since there are $N_{\rm H}$ hotspot segments and $(N-N_{\rm H})$ non-hotspot segments, *chkpt-log-size* is the sum of above equations multiplied by $N_{\rm H}$ and $(N-N_{\rm H})$, respectively. The size of log data in only physical logging is HP. The rate of the log size with hybrid logging to only physical logging thus is

$$\begin{aligned}
&(1-f_{\rm H})\left[\frac{(1-f_{\rm H})}{N_{\rm H}} + \frac{(1-f_{\rm H})}{N_{\rm H}}(N_{\rm H}-1)\frac{L}{P} + f_{\rm H}\frac{L}{P}\right] \\
&\quad + f_{\rm H}\left[\frac{f_{\rm H}}{N_{\rm H}} + \frac{f_{\rm H}}{N_{\rm H}}(N-N_{\rm H}-1)\frac{L}{P} + (1-f_{\rm H})\frac{L}{P}\right] \\
&= \frac{(1-f_{\rm H})^2}{N_{\rm H}} + \frac{f_{\rm H}^2}{N-N_{\rm H}} \\
&\quad + \frac{L}{P}\left[(1-f_{\rm H})^2 + f_{\rm H}^2 + 2f_{\rm H}(1-f_{\rm H})\right] \\
&\quad - \frac{L}{P}\left[\frac{(1-f_{\rm H})^2}{N_{\rm H}} + \frac{f_{\rm H}^2}{N-N_{\rm H}}\right].
\end{aligned}$$

If we consider equal access rate for each segment, $f_{\rm H} = 1/2$, the result of above equation is

$$\frac{1}{N} + \frac{L}{P} - \frac{L}{NP},$$

which is the same to *chkpt-log-size* of case 3 in Figure 1.

Figure 10 shows the rate of reduced size of log data to varying N for hotspot rates, with L of 64 words and P of 192 words. This result says that our hybrid logging method has great impact on reducing the size of log data. When $f_{\rm H}$ is 20%, 45% of the log size are reduced with six segments, compared with that of only physical logging.

Finally, we measure the recovery time of MMDB. Table 2 shows some parameters and their default values. They are derived from [11] and [14]. For the simplicity, the recovery time of the last complete checkpoint is considered. The recovery time consists of the MMDB reloading time, log pages reading time, and log reapplying time. The time to read the backup database, $T_{\rm back}$ is

$$T_{\rm back} = \frac{S_{\rm db}}{S_{\rm page}} \times (T_{\rm seek} + T_{\rm latency} + T_{\rm transfer}).$$

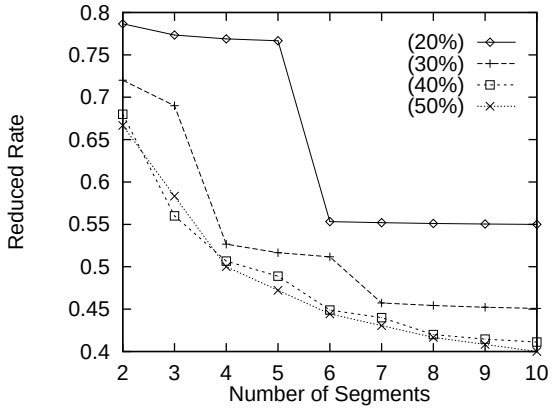

Figure 10. Effects of Log Reduction under hotspots

The size of log data for recovering the last complete checkpointing, $S_{\rm log}$, is

$$S_{\rm log} = (1-P_{\rm abort}) \times T_{\rm rate} \times t_{\rm icp} \times D_{\rm redo}$$

where $t_{\rm icp}$ is an inter-checkpoint interval and $D_{\rm redo}$ is the size of redo log data per transaction. If only physical logging is used, $D_{\rm redo} = S_{\rm init} + S_{\rm rec} \times N_{\rm act}$. When only logical logging is used, we simply assume $D_{\rm redo} = S_{\rm op} + S_{\rm init}$. $D_{\rm redo}$ of the hybrid logging is calculated according to the result of Figure 10.

$t_{\rm icp}$ is a period between the beginnings of checkpoints and is determined by the number of dirty pages generated and the I/O capability. According to [11], the expected number of dirty pages generated during a time t, $N_{\rm dirty}(t)$, is

$$\begin{aligned}
N_{\rm dirty}(t) = &\left[1 - \left(1 - \frac{R_{\rm spa}}{N_{\rm page}}\right)^{f_{\rm hotact} \times N_{\rm act} \times T_{\rm rate} \times t}\right] \\
&\times f_{hot} \times N_{\rm page} \\
+ &\left[1 - \left(1 - \frac{R_{\rm spa}}{N_{\rm page}}\right)^{(1-f_{\rm hotact}) \times N_{\rm act} \times T_{\rm rate} \times t}\right] \\
&\times (1-f_{hot}) \times N_{\rm page}
\end{aligned}$$

where $N_{\rm page}$ is $S_{\rm db}/S_{\rm page}$. Since we use ping-pong backup policy, the number of dirty pages to be flushed during time t, $N_{\rm flush}(t)$, is $N_{\rm dirty}(2 \times t_{\rm icp})$.

According to [14], the number of pages that can be written out to the disks during time t, $N_{\rm io}(t)$ is given by

$$N_{\rm io}(t) = N_{\rm bdisks} \times \frac{t}{T_{\rm seek} \times T_{\rm latency} \times T_{\rm transfer}}.$$

Table 2. Parameters and Their Defaults

Symbol	Meaning	Defaults
$S_{\rm db}$	database size	512 M words
$S_{\rm lpg}$	log page size	1024 words
$S_{\rm page}$	page size	8K words
$S_{\rm rec}$	record size	32 words
$S_{\rm op}$	logical log entry size	32 words
$S_{\rm init}$	log header size	32 words
$T_{\rm rate}$	transaction arrival rate	1000 TPS
$T_{\rm seek}$	average seek time	0.008 sec
$T_{\rm latency}$	average rotation time	0.00417 sec
$T_{\rm transfer}$	average transfer time per page	0.00039 sec
$N_{\rm bdisks}$	number of backup disks	20
$N_{\rm act}$	actions per transaction	5
$P_{\rm abort}$	abort probability	0.05
$f_{\rm H}$	fraction of hotspot	0.2
$f_{\rm hotact}$	fraction of actions to hotspot	1-$f_{\rm H}$
$R_{\rm spa}$	pages per action	1.1

By setting $N_{\rm flush}(t) = N_{\rm io}(t)$, we find the minimum $t_{\rm icp}$.

In general, since disk reading and CPU processing can be overlapped together, and since disk I/O time is much larger than CPU processing time, the log page reading time may be regarded as total log processing time. Due to locality of log that is sequential file [11], the time to read log is

$$\frac{S_{\rm log}}{S_{\rm lpg}} \times (0.3 \times T_{\rm seek} + T_{\rm latency} + T_{\rm transfer}).$$

Figure 11 presents recovery times of three logging scheme. The recovery time of logical logging (Logical) is an ideal case because we cannot use only logical logging with fuzzy checkpointing in MMDB. Compared with the recovery time of physical logging, our hybrid logging approach performs better. As having more segments causes the size of physical log data to be smaller, the recovery time of our approach converges to that of logical logging. With 20 segments, the gap of recovery times between physical logging and hybrid logging is about 20 seconds. This is not small in the high transaction processing rate. At 1000 TPS, 20000 transactions can be processed during the gap.

5. Conclusions and Further Work

Fuzzy checkpointing is an efficient checkpointing method in MMDB, but generates greater size of log data due to physical logging. This causes the recovery time of MMDB to be longer since much disk I/Os are required to read the log. To reduce the size of log data, we have proposed the hybrid logging method accommodating logical

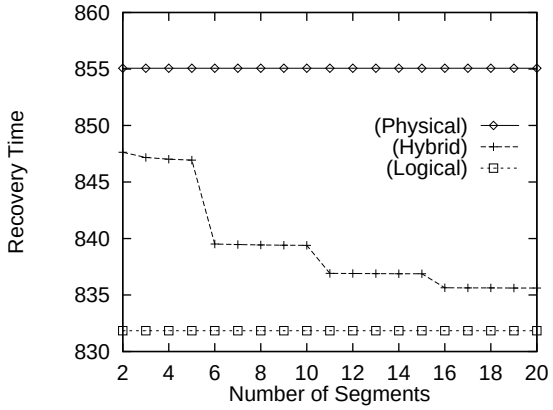

Figure 11. Recovery Time of a Checkpoint

logging under fuzzy checkpointing. By using the hybrid method, the size of log data generated during a checkpoint can be significantly reduced. We have also presented an efficient reapplying rule in segmented MMDB. This rule reduces the *number* of log records applied for recovery.

We have shown through analyses that the hybrid method can reduce the size of log data generated during a checkpoint by more than half, compared with those that use only physical logging. The result of the reapplying rule shows that we can recover the last complete checkpoint with about half of number of log records generated during the checkpoint. For further works, we are investigating possibility of appliance to redo/undo schemes and segmenting methods suitable to the proposed method. Detail analyses through experiments are also a subject.

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Co., 1987.

[2] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. "Implementation Techniques for Main Memory Database Systems". In *Proc. of Intl. Conf. on Management of Data*, pages 1–8. ACM SIGMOD, 1984.

[3] H. Garcia-Molina and K. Salem. "Main Memory Database Systems: An Overview". *IEEE Trans. on Knowledge and Data Engineering*, 4(6):509–516, Dec. 1992.

[4] L. Gruenwald and M. H. Eich. "MMDB Reload Algorithms". In *Proc. of ACM SIGMOD*, pages 397–405. ACM, 1991.

[5] R. B. Hagmann. "A Crash Recovery Scheme for a Memory-Resident Database System". *IEEE Trans. on Computers*, C-35(9):839–843, Sept. 1986.

[6] H. V. Jagadish, D. Lieuwen, R. Rastogi, and A. Silberschatz. "Dalí: A High Performance Main Memory Storage Manager. In *Proc. of the 20th VLDB Conf.*, pages 48–59, 1994.

[7] H. V. Jagadish, A. Silberschatz, and S. Sudarshan. “Recovering from Main-Memory Lapses”. In *Proc. of the 19th VLDB Conf.*, pages 391–404, 1993.

[8] T. J. Lehman and M. J. Carey. “A Recovery Algorithm for A High-Performance Memory-Resident Database System”. In *Proc. of Intl. Conf. on Management of Data*, pages 104–117. ACM SIGMOD, 1987.

[9] T. J. Lehman, E. J. Shekita, and L.-F. Cabrera. “An Evaluation of Starburst’s Memory Resident Storage Component”. *IEEE Trans. on Knowledge and Data Engineering*, 4(6):555–566, Dec. 1992.

[10] E. Levy and A. Silberschatz. “Incremental Recovery in Main Memory Databases Systems”. *IEEE Trans. on Knowledge and Data Engineering*, 4(6):529–540, Dec. 1992.

[11] X. Li and M. H. Eich. “Post-crash Log Processing for Fuzzy Checkpointing Main Memory Databases”. In *Proc. of Intl. Conf. on Database Engineering*, pages 117–124. IEEE, 1993.

[12] J.-L. Lin and M. H. Dunham. “Speedup Recovery From Fuzzy Checkpoints”. Technical Report TR 95-CSE-17, Department of Computer Science and Engineering, Southern Methodist University, Sept. 1995.

[13] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. “ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging”. *ACM Trans. on Database Systems*, 17(1):94–162, Mar. 1992.

[14] K. Salem and H. Garcia-Molina. “Checkpointing Memory-Resident Databases”. In *Proc. of Intl. Conf. on Data Engineering*, pages 452–462, 1989.

[15] K. Salem and H. Garcia-Molina. “System M: A Transaction Processing Testbed for Memory Resident Data”. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):161–172, Mar. 1990.