A Logging Scheme for Reducing Update Workloads in Flash Storage

S Seong-Chae Lim, Hyuck Han, Chang-Sup Park

Abstract - By caching dirty pages in memory space of a buffering pool, a database system can reduce expensive physical I/O's required in page updates. If any data page cached has constant updates on itself, it seems to stay long in the buffering pool without flushing-out. Although the existence of such aged dirty pages can reduce the amount of physical updates in storage, it is apt to prolong time taken for recovery procedure after system failure. To prevent such a delayed recovery time, database systems usually take an approach of flushing aged dirty pages in a background mode. Even though the approach may be beneficial in the case of HDD storage, this may not be the case for flash storage because of its high update costs. To solve this problem, we proposed a new logging scheme and a recovery algorithm running with it. Since aged dirty pages in our method are written into a dedicated log file, rather than into data area in storage, we can evade frequent updating of them. To reduce the amount of log data written for that purpose, our logging scheme uses a small size of snapshot log. Since the write of a snapshot log record can put the redo start point forwards, we can guarantee the fast recovery procedure, while reducing the number of page updates. Due to reduced update workloads, our method can improve the overall throughput of flash storage.

Keywords: flash memory, database recovery, logging algorithm, storage system

1. INTRODUCTION

Thanks to a distinguished I/O advantage in random reads, flash memory has been regarded as a promising storage media, superseding the hard disk drive (HDD) (Baumann, Nijs, Strobel & Sattler, 2010; Colgrove et al., 2015). In particular, as the price per bit gets cheaper at a fast rate, flash storage seems to be used for large-scale database systems in the future. In this light, many researches have been done for the purpose of adopting NAND flash for those systems. Most of those researches are devoted to efficiently handling the inherent poor performance of random updates in flash memory (Ganim, Mihaila, Bhattacharjee, Ross & Lang, 2010; Do, Zhang, Patel, DeWitt, Naughton & Halverson, 2011; Gupta, Kim & Urgaonkar, 2009; Jeong, Kim & Lim, 2015; Moon, Lim, Park & Lee, 2011; Lee & Moon, 2007; Lim, 2016; Wang, Goda & Kitsuregawa, 2009; Wu, Kuo & Chang, 2007)

To reduce the occurrences of costly update I/O's in flash storage, the previous researches take two different approaches, that is, a logging based approach (Gupta, Kim & Urgaonkar, 2009; Moon et al., 2011; Wang, Goda & Kitsuregawa, 2009) and a buffering based approach (Do et al., 2011; Ganim et al., 2010; Jeong, Kim & Lim, 2015; Li, He, Yang, Luo Y Yi, 2010; Lim, 2016; Xu et al., 2010;). In the former approach, histories of updates in databases are recorded as log data without physical reflections on storage directly. Then, a collection of updates are flushed into storage at once before their involved log data are cleaned. Since the I/O cost for storing log data and doing batch-style updates is usually cheaper than that for executing individual updates, that approach gains I/O advantages in flash storage. In the case of the latter approach, extra space is reserved in main memory for the use of a buffering area. By updating data within the buffering memory, the approach diminishes the number of I/O's needed for hot pages. Although the buffering based approach demands the usage of extra memory, it is reported to be very efficient for handling updates in B-trees stored in flash storage (Li et al., 2010; Xu et al., 2010;).

Unlike the earlier approaches, for less update workload we pay attention to the way a recovery algorithm works. This research point has not been a major interest in previous researches. Modern recovery algorithms are usually based on the WAL (write-ahead-logging) protocol and the NO-FORCE policy in buffer management (Kornacker, Mohan & Hellerstein, 1997; Mohan & Levine, 1992; Lim, 2016). Therefore, hot pages are repeatedly updated in a buffering pool, without being written (or flushed) to storage. The existence of such dirty pages long staying in a buffering pool is apt to delay the recovery time after system failure because of more redo actions performed. To avoid such a delay of the recovery time, periodic flushing of dirty pages is acceptable. Note that flushing of dirty pages normally arises at buffer replacement times. Although such background-mode flushing does not impair the performance of HDD-based database systems, it can be not true in flash storage because of asymmetric performance between updates and other sorts of I/O's (Baumann et al., 2010; Colgrove et al., 2015; Do et al., 2011; Lee & Moon, 20017; Ganim et al., 2010; Wang, Goda & Kitsuregawa, 2009)

To solve this problem, we propose new logging and recovery schemes that can guarantee the fast recovery time without flushing of aged dirty pages. To this end, we made some modifications to the conventional logging scheme so that it creates snapshot log for aged dirty pages and uses those log data for physical redos during the recovery time. By storing the snapshot log in a log file, we can put forward the redo start point of the redo process.

Revised Manuscript Received on December 22, 2018.

S Seong-Chae Lim, Department of Computer Science, Dongduk Women's University, Seoul, South Korea.(E-Mail: sclim@dongduk.ac.kr)

Hyuck Han, Department of Computer Science, Dongduk Women's University, Seoul, South Korea.(E-Mail: hhyuck96@dongduk.ac.kr)

Chang-Sup Park, Department of Computer Science, Dongduk Women's University, Seoul, South Korea.(E-Mail: cspark @dongduk.ac.kr)

Since the writing of snapshot log does not incur updates in storage, we can reduce the number of page update operations needed for flushing dirty pages. Since the snapshot log size recorded for an aged dirty page is less than half its page size, we can save overall I/O bandwidth during the normal processing time. Additionally, because the physical redos are used for restoring aged dirty pages, our method can enhance the redo time. Those benefits are evaluated based on a storage cost model that is proposed in the paper.

This paper is organized as follows. In Section 2, we introduce some background knowledge about the conventional logging and recovery schemes. In Section 3, we sketch the idea of the proposed method and address the detailed algorithms needed for writing log data and recovering a failed system. The performance advantages of the proposed method are estimated in Section 4. Then, we conclude this paper in Section 5.

2. PRELIMINARIES.

In modern database systems, a logging mechanism is vital for preserving the ACID properties of transactions. For this, the WAL protocol is usually implemented along with a buffering scheme used for caching update operations in memory (Do et al., 2011; Li et al., 2010, On, Hu, Li & Xu, 2010). Since the buffering scheme works with the NO-FORCE policy for fast processing of transactions, some dirty pages may not be written for a long period of time because of their frequent references. Although the existence of such aged dirty pages can be useful for reducing I/O workloads, it adversely affects the recovery time in face of system failure. This is because we have to pay more time for redoing abrupt updates involved with the aged dirty pages.

To understand that, one needs to know about the redo process during a recovery time. Note that a recovery algorithm performs a redo process for redoing failed updates after it has analyzed a log file for recovery. Most of recovery algorithms like ARIES (Kornacker, Mohan & Hellerstein, 1997; Mohan & Levine, 1992). make periodically checkpoints to capture the states of a buffering pool and in-progress transactions. As checkpoint data for dirty pages in the buffering pool, ARIES-style logging schemes save page ID's of dirty pages and LSNs (Log Sequence Numbers) of recovery log records of them. Note that the log file location of the recovery log record is used as its LSN (Mohan & Levine, 1992).

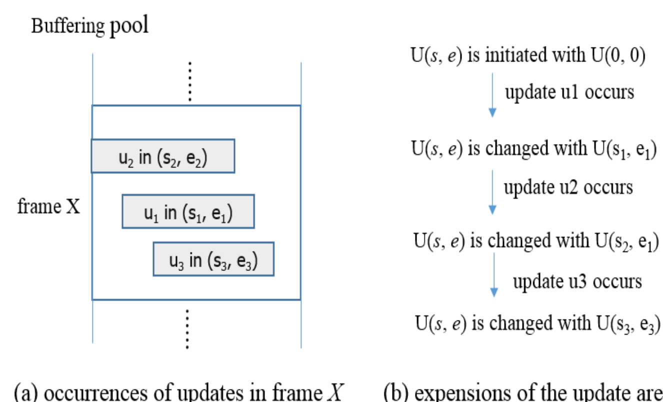
To see the notion of the recovery log record, consider a page X that was buffered at time t1. Then, if page X is first updated at time t2 (t2 • t1), then a log record recording the update at t2 is referred to the recovery log record for X. If page X is written to storage, then it is removed from a dirty page table (DPT) along with the information about its recovery log record. Because every log record R preceding the recovery record at time t2 has been already reflected on storage, there is no need for redoing log R. In other words, the redo process is performed from the recovery log record. Therefore, we can move the redo start point forwards by flushing a dirty page whose recovery log record is oldest. Since the redo time accounts for a majority of the recovery time, it is fundamental to put forwards the redo start point.

To this end, previous recovery algorithms usually flush aged dirty pages in a background mod. Such background flushing of dirty pages may not be problematic in the case of traditional HDD storage. However, this is not the case for flash storage because of its high update costs. To prevent delay of the redo process without recovery-purpose flushing, we devise a new logging scheme that can put forward the redo start point without flushing of dirty pages.

3. PROPOSED METHOD

3.1. Proposed Logging Scheme

To prevent update workloads caused by background-mode flushing of dirty pages, as described before, we store physical redo logs for aged dirty pages, rather than flushing those dirty pages. The physical redo log for such a purpose is called the snapshot log. In general, a log record made in a database contains both of redo log data and undo log data for each update. In the case of redo log data, physiological redo data is common for the considerations of log sizes and locking granularity (Lee & Moon, 2007; Kornacker, Mohan & Hellerstein, 1997; Mohan & Levine, 1992). Unlike the traditional logging scheme, we save only redo log data in a snapshot log record, since it is not used for doing undos. In addition, log data in a snapshot log record is physical log data that saves the after-image covering updated data area in a page.



(a) occurrences of updates in frame *X* (b) expensions of the update area

Figure 1. Use of an update area for covering updates in a buffer frame

To explain the notion of the snapshot log record, we use Figure 1. In the figure, the notation of U[s, e] represents the existence of updates in an area of a buffer frame. Here, integral numbers s and e are offsets within a buffer frame caching involved updates. By increasing the update area of U[s, e], we can locate a wider range of updated data. To see this, let us assume that three times of updates have occurred in frame X of Figure 1. When the i-th update is denoted by ui, and its update area is expressed with U[si, ei]. In the figure, the update area is finally set to U(s2, e3), which covers the update ranges of u1, u2, and u3.

At that case, we save the binary image between U[s2, e3] in a snapshot log record for frame X. When update positions of u1, u2, and u3 are scattered, area U[s, e] inevitably includes many non-updated areas, thereby increasing the sizes of snapshot records.

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

To reduce such an undesirable increase of snapshot log, we use two different update areas for each frame, that is, one for updates in the data area and the other for updates in the metadata area. By saving log data for two different areas each, we can reduce the sizes of snapshot log records being stored.

Algorithm 1: Procedure *PeriodicFlushingInBackground()*

```
Input : Buffer = object of the buffer pool;
        DT = object of the dirty page table;
        lsn = LSN of the latest log record;
        max_lag = maximum lag of the redo-start-point;
1  foreach e ← DT do // for every entry in the dirty page table
2    if e.rec-lsn > max_lag + lsn then // flushing or logging
3      frame ← Buffer.Frame(e.frame_ID); // access to the
         buffer frame of the current aged dirty page
4      ua_size ← Σ_{i=1}^{2}(frame.u_i.e − frame.u_i.s);
5    if update_size ≥ frame_size/2 then // flushing is decided
6      Buffer.Write(frame); // write of in-frame image
7      DT.RemoveEntry(e); // deletion of an entry
8    else // logging is decided
9      logdata ← Buffer.Copy(frame, e_1, e_2); // log data copy
10     Create a snapshot log record Rec containing logdata;
11     Save Rec in the log file and let its LSN be new_lsn;
12     (e.rec-type, e.rec-lsn) ← ('snapshot', new_lsn);
13     DT.UpdateEntry(e); // DPT update
```

Figure 3. Algorithm for saving snapshot log records for aged dirty pages

To elaborate the usage of the snapshot log record, we use Figure 2, where a buffering pool contains three dirty pages of X, Y, and Z. In the DPT, the LSNs of their recovery log records are saved as rec-lsn. Here, the value of 'normal' means the corresponding dirty page has no snapshot log record. In this figure, the redo start point equals the LSN of X's recovery log record. If any traditional recovery scheme attempts to put the redo start point forwards, then it flushes page X and deletes it from the DPT. Then, LSN z is used as new restart point.

Unlike that, our logging scheme saves a snapshot log record for page X, At the same time, we update its values of rec-type and rec-lsn with 'snapshot' and the LSN of the newly created snapshot log record, respectively. Note that the new rec-lsn is always greater than y of Figure 2. In our scheme, the page X is not deleted from the DPT, because its in-frame image has not been written to storage. Therefore, if page X has to be evicted from the buffering pool at a buffer replacement time, then its in-frame image will be written to storage and it is removed from the DPT.

Figure 3 shows the algorithm that is periodically executed to put forwards the redo start point. When the algorithm is invoked, it receives the LSN of the latest log record, that is, lsn of Figure 3. Then, for each dirty page existing in the DPT its rec-len is compared with lsn in line 2. For exposition convenience, we denote that dirty page in comparison by N. If the LSN gap is larger than max_lag, then page N needs either flushing or logging for the fast redo process.

Algorithm 2: Procedure *RecoverSysFailure()*

```
Input : FLog = object of the log file;
        Buffer = object of an empty buffer pool;
        DT = object of an empty dirty page table;
1  /* begin of a log analysis */
2  Read the lastest checkpoint record and initiate DT using that;
   while FLog.IsEOF() ≠ true do // log reads for analysis
3    rec ← FLog.GetNextLog() ; // read of a log record
4    if rec is a snapshot log record then
5      Update DT for the page of rec.page_ID based on rec;
6    else
7      Update the undo transaction list based on rec;
8  /* begin of a redo process */
9  foreach e ← DT.GetEntry() do // action for every page
   that was found to have a snapshot log record
10   if e.rec-type = 'snapshot' then
11     frame ← Buffer.Read(e.page_ID); // page loading
12     Buffer.DoPhysicalRedo(frame, e.rec-lsn);
13 redo_start_point ← the minimum LSN among rec-lsn's in DT;
14 LogF.SetOffset(redo_start_point); // file pointer setting
15 while FLog.IsEOF() ≠ true do // traditional redo loop
16   rec ← FLog.GetNextLog() ; // read of a log record
17   Perform the physiological redo using the log data in rec;
18 /* begin of an undo process */
19 Perform the undo process for the transactions in the undo list;
```

Figure 4. Algorithm for the recovery after system failure

For that decision between flushing or logging, the overall size of two update areas is calculated in line 4. If that size is greater than half the frame size, then node N is flushed as in lines 6 to 7; otherwise, a snapshot log record is made for N and its DPT entry is updated as in lines 9 to 13. Then, two update areas of N are copied into a snapshot log record. After this logging, the value of rec-lsn of N is modified with the LSN of a newly created snapshot record. With this algorithm, we can put the redo star point forwards for very cheap I/O costs.

3.2. Recovery Algorithm

To recover from system failure, our recovery procedure begins with reading the latest checkpoint record in a log file. After rebuilding a DPT using the log data saved in that checkpoint record, log analysis is conducted by scanning the log file. Then, the processes for doing redos and undos are performed in this order. Since our method for the undo process is very similar to the traditional one (Jeong, Kim & Lim, 2015; Kornacker, Mohan & Hellerstein, 1997; Lee & Moon, 2007; Mohan & Levine, 1992). , we mainly address the other processes for log analysis and redo actions.

The proposed recovery algorithm is given in Figure 4. In the log analysis process of lines 2 to 7, the algorithm sequentially reads log records and looks up the log types of them. If a log record retrieved has a type of 'snapshot', then the DPT is updated to reflect its log information. More specifically, its data of rec-lsn and page ID are saved as a DPT entry; otherwise, it is used for updating an undo transaction list.

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

The way for manipulating the undo transaction list is not different from the previous ways in (Kornacker, Mohan & Hellerstein, 1997; Mohan & Levine, 1992). Those conditional actions depending on log types are performed in lines 4 to 7.

At the beginning of a redo process, the recovery algorithm performs physical redos for the pages having snapshot log records. For this, the recovery algorithm reads each DPT entry and check its rec-type. If that type is the same as 'snapshot', then a physical redo is executed to restore the up-to-date image of the associated page P. Such a redo action is done in line 12, where the function DoPhysicalRedo() for the physical redo overwrites the data logged in the snapshot log record within the in-frame image of P. Then, the up-to-date image of P is written to storage within the function. Since the LSN of page P is modified with that of its recovery log record, redo log records preceding the snapshot log record will be ignored during the redo process later. Therefore, the proposed recovery algorithm can save the time taken for traditional redo actions.

After the physical redo actions using snapshot log records, the traditional undo actions are executed in lines of 13 to 17. For this, the redo start point is decided and the file pointer of the log file is set with that start point as in lines 13 and 14. Then, by reading each redo log record the recovery algorithm performs the remaining steps of the redo process. During those steps, physiological redos are performed as in the traditional recovery schemes. Finally, to restart the database system, undo actions are performed to roll back update effects from aborted transactions.

4. PERFORMANCE ANALYSIS & RESULTS

As a dirty page with constant updates seems to be a winner at buffer replacement times, it is apt to stay long in the buffer pool without being flushed. This situation seems to increase the time for system recovery. For this reason, the mechanism of background-mode flushing is usually employed in HDD-based database systems. Since the background-mode flushing easily incurs the number of update operations, however, it can harm storage performance in the case of SSD-based database.

Differently from such background-mode flushing used for fast recovery, the proposed method based on snapshot logging provides two advantages in the I/O respect. First, our method does not incur any update operations since snapshot log records are stored into clean pages. When clean pages of a log file are exhausted, storage space of that file is freed to a file system via the I/O calls of TRIM or UNMAP (Wu & He, 2012). Through those kinds of I/O calls, we can largely save I/O costs because of the reduction of updates, while guaranteeing fast recovery. Second, since the size of snapshot log record is less than half the page size, our method can reduce the overall amount of data written. Thanks to those two characteristics, the proposed method can support the reliable I/O performance of a flash-based database system against heavy update workload.

To assess the performance advantages in more detail, we introduce a I/O cost model based on the parameters of Table 1. In that table, the value of P_fm relies on the two factors, that is, real I/O request patterns in the system and the internal mapping mechanism of an FTL (Flash Translation Layer) used in flash storage. Since the hybrid addressing for the FTL is usually accepted nowadays (Gupta, Kim, Urgaonkar, 2009; Moon et al., 2011; Wang, Goda & Kitsuregawa, 2009; Wu & He, 2012), we also assume the same addressing mechanism for our cost model. That is, a portion of storage is dedicated for the use of the log block area, and that area is used for saving updated pages until it becomes full. Under this assumption, actions for full-merging arises when log blocks are all consumed for saving updated pages in flash storage (Moon et al., 2011; Wang, Goda & Kitsuregawa, 2009). From this, we can compute approximately the probability P_{fm} such that $P_{fm} = 1/(N_p \times N_{lb})$.

Table 1. Notations and meanings of I/O parameters

Notations	Meanings	Values
N_p	#. of pages made in a flash block	64 - 128
N_{lb}	# of logging blocks used for page address	30 - 50
N_{fm}	Averaged # of flash blocks involved in a full-merge	20 - 30
P_{fm}	Probability of a full-merge occurrence w.r.t an update	$1/(N_p \times N_{lb})$
S_b	Size of a flash block in Kbytes	128 - 256
C_w	I/O cost for a single write to an empty page	200 – 250 us
C_e	I/O cost for erasing a flash block	200 – 250 ms

While a page is being updated according to a page addressing mechanism of an FTL, its new image is first written to a clean page existing in a log block. If there is no clean page in the log block area, that update operation leads to a full-merge for reclaiming some space in the log block area. By considering that hidden cost for full-merging, the I/O cost for updating a page can be estimated as follows:

$$C_u = C_w + P_{fm} \times N_{fm} \times (C_e + N_p \times 0.7 \times C_w)$$

Here, the fraction of 0.7 is the storage utilization rate of the data block. Therefore, 70% of pages within a block are rewritten for saving valid pages. Since the size of a snapshot log record is less than half the page size, we can compute the performance benefit as follows:

$$C_{benift} = C_u - 0.5 \times C_w = P_{fm} \times N_{fm} \times \left(C_e + N_p \times 0.7 \times C_w\right) + C_w/2$$

In literature (Do et al., 2011; Lee & Moon, 2007; On et al., 2010; Wu, Kuo & Chang, 2007), it was reported that the black erase cost of C_e is at least 100 times higher than the write cost of C_w. When the number of log blocks is the same as 20 and a block contains 64 pages, the rates of C_benift/C_u becomes around 20%. Although this performance may vary according to the real values of P_fm,

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

the proposed method can improve the flash storage performance by reducing update workloads in database systems.

5. CONCLUSIONS

In this paper, we propose a new logging scheme that can evade the necessity of periodic flushing of dirty pages, thereby reducing the number of costly update operations. For this, we made some modification to a dirty page table used for bookkeeping the state of dirty pages, and proposed an algorithm used for making a decision between flushing and logging for an aged dirty page. Since the proposed logging scheme should be valid for recovering a failed system, we also proposed a recovery algorithm that works correctly with the proposed snapshot logging scheme. With the proposed recovery algorithm, we can restore the dirty pages to an up-to-date state. Since the redo actions are done through physical redos, we can improve the time for recovery. To show performance advantages, we introduced a simple I/O cost model. The results from the cost model show that the proposed method can improve the overall I/O performance for flash storage.

REFERENCES

1. Agrawal, D., Ganesan, D., Sitaraman, R., Diao, Y., Singh, S. (2009). Lazy-Adaptive Tree: an optimized index structure for flash devices. In Proceedings of VLDB. VLDB.
2. Baumann, S., Nijs, G., Strobel, M., & Sattler, K. (2010). Flashing databases: Expectations and limitations. In Proceedings of ACM DaMon. ACM.
3. Colgrove, J., Davis, J., Hayes, Miller, E., Sandvig, C., Sears, R., Tamches, A., Vachharajani, N., Wang, & Purity, F. (2015). Building fast, highly-available enterprise flash storage from commodity components. In Proceedings of SIGMOD. ACM.
4. Do, J., Zhang, D., Patel, J., DeWitt, D., Naughton, J., & Halverson, A. (2011). Turbo-charging DBMS buffer pool using SSDs. In Proceedings of ACM SIGMOD. ACM.
5. Ganim, M., Mihaila, G., Bhattacharjee, B., Ross, K., & Lang, C. (2010). SSD bufferpool extensions for database systems. In Proceedings of VLDB. VLDB.
6. Gupta, A., Kim, Y. & Urgaonkar, B. (2009). DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In Proceedings of ASPLOS. ACM.
7. Jeong, K., Kim, S., & Lim, S. (2015). A flash-aware buffering scheme using on-the-fly redo. In Proceedings of ACM CIKM. ACM.
8. Kornacker, M., Mohan, C., & Hellerstein, J. (1997). Concurrency and recovery in generalized search trees. In Proceedings of SIGMOD. ACM.
9. Lee, S. & Moon, B. (2007). Design of flash-based DBMS: An in-page logging approach. In Proceedings of ACM SIGMOD. ACM.
10. Li, Y., He, B., Yang, R., Luo, Q., & Yi, K. (2010). Tree indexing on solid state drives. In Proceedings of VLDB. VLDB.
11. Lim, S. (2016). A new flash-based B+-tree with very cheap update operations on leaf node. In Proceedings of ETBDA. IIENG.
12. Mohan, C. & Levine, F. (1992). ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. In Proceedings of SIGMOD. ACM.
13. Moon, S., Lim, S., Park, D., & Lee, S. (2011). Crash recovery in FAST FTL. In Proceedings of software technologies for embedded and ubiquitous systems. Springer.
14. Na, G., Lee, S. & Moon, B. (2012). Dynamic in-page logging for B+-tree index. IEEE Transactions on Knowledge and Data Engineering, 24(7). 1231-1243.
15. On, S., Hu, H., Li, Y., & Xu, J. (2010). Flash-optimized B+-tree. Journal of Computer Science and Technology, 25(3). 509-522.
16. Wang, Y., Goda, K., & Kitsuregawa, M. (2009). Evaluating Non-In-Place update techniques for flash-based transaction processing systems. In Proceedings of DEXA. ACM.
17. Wu, C., Kuo, T., & Chang, L. (2007). An efficient B-tree layer implementation for flash-memory storage systems. ACM Transactions on Embedded Computing Systems, 6(3).
18. Wu, G. & He, X. (2012). Delta-FTL: Improving SSD lifetime via exploiting content locality. In Proceedings of European conference on computer systems. ACM.
19. Xu,C., Show, L., Chen, G., Yan, C., & Hu, T. (2010). Update migration: An efficient B+ tree for flash storage. In Proceedings of DASFAA. DASFAA.