The Journal of The Institute of Internet, Broadcasting and Communication (IIBC)
Vol. 19, No. 4, pp.85-91, Aug. 31, 2019. pISSN 2289-0238, eISSN 2289-0246

https://doi.org/10.7236/JIIBC.2019.19.4.85
JIIBC 2019-4-13

스냅샷 로그를 사용한 SSD 기반 데이터베이스 복구 기법

A Recovery Scheme of SSD-based Databases using Snapshot Log

임성채*

Seong-Chae Lim*

요 약 논문에서는 플래시 스토리지 기반의 고성능 트랜잭션 처리시스템을 구현할 때 유용한 스냅샷을 사용한 로깅 및 데이터베이스 복구 기법을 제안한다. 제안된 기법은 플래시 메모리의 I/O 특성인 페이지 갱신/읽기 비용 간의 비대칭성에 기반한다. 즉, I/O 비용이 큰 페이지 갱신을 대신하여 스냅샷 로그라는 페이지 단위의 물리적 redo를 위한 로그를 기록하고 이를 실시간으로 적용할 수 있게 하였다. 이를 통해 로깅의 목적인 빠른 시스템 복구란 목적 외에도 더티 페이지를 재기록 없이 버퍼풀에서 삭제할 수 있게 하였다. 이런 방식은 페이지 갱신 비용과 읽기 비용 간에 차이가 없는 기존 HDD(Hard Disk Drive)에서는 성능 개선을 기대할 수 없다. 하지만 플래시 메모리인 SSD에 적용할 때는 페이지 갱신 횟수의 감소에 따른 성능 향상과 빠른 시스템 복구를 기대할 수 있다. 제안된 기법은 스냅샷 로그와 기존의 로그가 서로 섞여 기록된 상황에서 기존 REDO 알고리즘의 간단한 변경만으로 적용될 수 있기 때문에, 향후 구현될 SSD 기반 데이터베이스 시스템의 성능 개선에 사용될 수 있을 것이다.

Abstract In this paper, we propose a new logging and recovery scheme that is suited for the high-performance transaction processing system base on flash memory storage. The proposed scheme is designed by considering flash's I/O characteristic of asymmetric costs between page update/read operations. That is, we substitute the costly update operation with writing and real-time usage of snapshot log, which is for the page-level physical redo. From this, we can avoid costly rewriting of a dirty page when it is evicted form a buffering pool. while supporting efficient revery procedure. The proposed scheme would be not lucrative in the case of HDD-based system. However, the proposed scheme offers the performance advance sush as a reduced number of updates and the fast system recovery time, in the case of flash storage such as SSD (solid state drive). Because the proposed scheme can easily be applied to existing systems by saving our snapshot records and ordinary log records together, our scheme can be used for improving the performance of upcoming SSD-based database systems through a tiny modification to existing REDO algorithms.

Key Words : flash memory; SSD storage; datagbase systems; recovery algorithm;

*정회원, 동덕여자대학교 컴퓨터학과
접수일자 2019년 5월 27일, 수정완료 2019년 7월 3일
게재확정일자 2019년 8월 2일

Received: 27 May, 2019 / Revised: 3 July, 2019 /
Accepted: 2 August, 2019
*Corresponding Author: sclim@dongduk.ac.kr
Dept. of Computer Science, Dongduk Women's University, Korea

Ⅰ. 서 론

플래시 메모리는 낮은 수준의 전력 사용, 작은 크기, 외부 충격에 대한 저항성 등의 장점으로 인해 모바일 기기의 주 저장장치로 이용된다. 최근 낮아진 비트 당 가격과 기존 HDD(hard disk drive) 대비 빠른 임의(random) 읽기 속도로 인해 고사양 서버의 데이터 저장장치에도 사용되는 추세이다[1, 2, 13, 14, 15]. 특히, 여러 장의 플래시 메모리로 구성된 SSD(solid state drive) 장치를 고성능 트랜잭션 시스템의 주 저장장치로 사용하여, 트랜잭션 처리 속도를 크게 향상시킬 수 있는 방법에 대한 연구가 최근 활발히 진행되고 있다[1, 5-8, 12-15].

SSD 저장장치에 대한 초기 연구들은 SSD 장치를 메모리와 HDD 사이에 두고, 접근 빈도가 높은 데이터 객체를 캐시해 사용하는 것들이 주를 이루었다. 예를 들어, SSD와 HDD 간의 데이터 일관성 유지 기법, 캐시할 핫(hot) 데이터 선택 기법, 시스템 고장 시에 캐시된 데이터를 빠르게 데이터베이스에 적용시키는 기법 등이 이에 속한다[1, 6, 8, 11, 12, 15]. 이 방식에서는 HDD의 부가적인 저장장치로 SSD를 사용되기 때문에, SSD에 있는 데이터를 결국 HDD로 재저장(write-back)하는 추가비용이 발생한다. 또한 시스템 고장에 대비한 데이터 일관성 유지 비용이 커지기 때문에 기존 HDD 기반 시스템의 I/O 병목현상을 근본적으로 해결하기에는 한계가 있었다[1, 8, 15].

이런 문제를 피하기 위해서는 HDD 대신 SSD를 주 저장장치로 사용하는 것이 필요하다. 즉, 데이터베이스의 테이블 파일은 물론이고 레코드 접근 속도를 높이기 위한 색인 파일과 시스템 복구에 필요한 로그 파일 등도 SSD에 모두 저장해야 한다. 이런 상황에서는 SSD에 갱신 연산이 빈번히 발생하게 되므로 기존 캐시 용도와는 다른 문제를 갖게 된다[6, 9]. 플래시 메모리는 페이지의 제자리 갱신이 가능하지 않기 때문에, 페이지 갱신 시 빈 페이지에 갱신 페이지를 기록하고 기존 페이지는 가비지 회수(garbage collection)를 통해 빈 공간으로 편입시켜야한다[5, 13]. 이와 같은 out-of-place 갱신과 가비지 회수는 매우 큰 추가 비용이 있기 때문에 SSD를 주 저장장소를 사용할 때 기대했던 성능향상이 실제로는 가능하지 않을 수 있다. 특히 가비지 회수 작업이 진행될 때는 다른 I/O 연산이 중단되는 현상이 있기 때문에, 실시간성을 보장해야 하는 트랜잭션 처리 시스템에 심각한 문제를 초래할 수 있다[2, 6, 7]. 따라서 페이지 갱신을 최소화시킬 수 있는 다양한 기법이 요구되며, 본 논문에서도 이에 관련된 기법을 다룬다.

기존 HDD 기반의 시스템은 I/O 성능 향상을 위해 버퍼풀(buffer-pool)을 사용하며 페이지 갱신을 버퍼풀 안에서 수행한다[3, 9]. 버퍼풀이 사용될 때 HDD로의 물리적 갱신은 버퍼풀에 있던 더티(dirty) 페이지가 HDD로 재기록될 때이며, 두 가지 경우 발생한다. 첫째는 버퍼풀 공간 부족을 해결하기 위해 수행된 버퍼풀 재배치(replacement) 알고리즘에 의해서이며, 희생자(victim) 페이지는 버퍼풀에서 삭제 전 저장장치에 기록된다. 둘째는 시스템 복구(recovery) 시간을 줄이기 위해 수행되는 재기록이다. 이를 통해 시스템 복구에 필요한 REDO 절차의 시작 시점을 앞당길 수 있다[3, 10]. 후자의 경우 해당 더티 페이지는 버퍼풀에 그대로 존재하며 더티 비트 만 초기화된다. 제안하는 기법은 이런 두 가지 경우에 발생하는 페이지 갱신 연산을 줄임으로써 SSD 기반 시스템의 성능을 안정적으로 높일 수 있다.

논문에서는 더티 페이지의 재기록(즉 페이지 갱신)을 스냅샷(snapshot) 로그 레코드를 기록으로 대체하는 기법을 제안한다. 스냅샷 로그는 미리 확보된 빈 공간에 저장되며 이를 통해 페이지 갱신 없이도 REDO 시작점을 앞당길 수 있다. 논문의 스냅샷 로그는 물리적 redo를 위한 로그이며, 갱신된 페이지의 변경 이미지 정보만을 저장한다. 스냅샷 로그 레코드의 기록 후에는 연관된 더티 페이지는 재기록없이도 버퍼풀에서 삭제될 수 있다. 이를 통해 로그 기록과 읽기 연산으로 페이지 갱신 연산을 대신할 수 있으며, 줄어든 갱신 연산으로 인해 SSD를 주 저장장치로 사용하는 시스템의 성능을 크게 개선할 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구로서 시스템 복구에 대해서 알아보고 논문의 기본 아이디어를 정리한다. 3장에서는 스냅샷 로그 기록 및 시스템 복구 절차에 필요한 알고리즘을 기술한다. 4장에서는 성능상의 개선점을 제시하고, 5장에서 결론을 맺는다.

Ⅱ. 관련 연구

1. 시스템 복구를 위한 REDO 절차

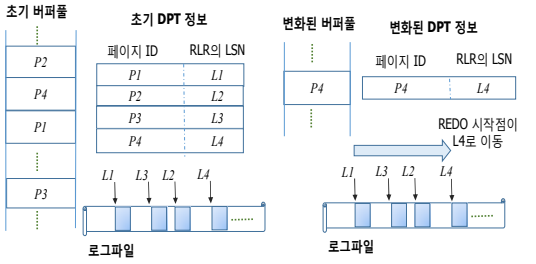
현대 트랜잭션 처리 시스템은 트랜잭션의 ACID 특성을 구현하기 위해 로깅 기법을 사용한다. 로깅은 WAL(write-ahead logging) 프로토콜에 따라 수행되며 이를 통해 더티 페이지의 재기록을 트랜잭션의 종료 시점에 강제하지 않는다[3, 10]. 즉, 보다 유연하고 성능이 좋

은 NO-FORCE 버퍼링 정책을 지원한다. WAL 프로토콜과 NO-FORCE 정책은 디스크 I/O 횟수를 줄여주지만 더티 페이지가 재기록 없이 버퍼풀에 오래 존재할 수 있어 시스템 복구 시간을 길게 하는 효과도 가진다.

시스템 고장 발생 후의 복구 작업은 일반적으로 로그 분석(log analysis) 절차, REDO 절차, UNDO 절차를 거친다[3, 10]. 수행 시간의 측면에서 볼 때 REDO 절차의 비중이 가장 크며 REDO는 로그 파일의 특정 시작 위치(REDO 시작점)에서부터 전체 로그 레코드를 순차적으로 읽으면서 redo 연산을 수행한다. REDO 절차는 데이터 페이지를 버퍼로 읽어 들이는 I/O 시간과 redo를 위한 CPU 시간으로 인해 로그분석이나 UNDO 절차에 비해 그 비중이 크다. REDO 시간을 줄이기 위해서는 REDO 시작점을 가급적 뒤로 하는 것이 필요하다[1, 10].

REDO 절차 시작점이 되는 로그 레코드의 결정에는 체크포인트 시점에 기록된 더티 페이지 테이블이 사용된다[10]. 정의에 따르면 임의의 더티 페이지 P의 RLR(recovery log record)을 R이라고 하면, 로그 레코드 R이 기록한 갱산 연산은 페이지 P가 버퍼풀에 적재된 후 최초 발생한 갱신 연산이다. 다른 관점으로는, 로그 R 이전에 생성된 로그 레코드들이 기록한 갱신 연산들은 페이지 P에 이미 반영되어 있다고 볼 수 있다. 따라서 R 이전의 로그 레코드는 redo 연산에 사용되지 않는다. 이런 RLR의 정의에 의해 따라 REDO 시작점은 체크포인트된 모든 더티 페이지의 RLR 중에서 가장 작은 LSN(log sequence number) 값을 가지는 RLR이 된다[3, 10].

설명을 위해 그림 1을 사용한다. 그림에서 버퍼풀에는 4개의 더티 페이지가 존재한다고 가정하며, 더티 페이지는 더티 페이지 테이블(DPT: Dirty Page Table)[10]을 통해 관리된다. DPT는 버퍼풀에 있는 페이지 중 어떤 페이지가 더티 페이지이며, 각 더티 페이지의 RLR의 LSN, 즉 로그 파일내의 위치를 기록한다. 만약 그림 1(a)의 DPT을 가지고 시스템 복구가 시작된다면, REDO 시작점은 가장 앞쪽에 위치한 L1이 된다. 시간이 흘러 P1, P2, P3가 재기록되어 DPT 가 1(b)로 변경된다면 REDO 시작점은 보다 시간이 지난 L4로 변경된다. 그림에서 보이듯이 오래 전에 생성된 RLR을 가진 더티 페이지를 재기록함으로써 REDO 시작점을 최근 시점으로 앞당길 수 있다.



(a) 네 개의 더티 페이지와 이를 위한 DPT 정보의 예. (b) 페이지 *P1, P2, P3*가 재기록된 후의 변화.

그림 1. 더티 페이지 테이블과 RLR 정보의 변화 예.
Fig. 1. Example of the dirty page table and information of RLRs.

2. 제안 아이디어

더티 페이지의 재기록은 REDO 시작점을 뒤로 이동시키기 위해서나 버퍼 공간 부족을 해결하기 위해서 필요함을 앞서 설명하였다. 하지만 이에 따른 잦은 페이지 갱신 연산이 SSD 기반 시스템의 성능 저하를 야기할 수 있다. 논문에서는 이를 해결하기 위해 SSD의 빠른 읽기 연산을 이용한다. 즉, SSD의 I/O 비용 비대칭성에 착안하여 더티 페이지 P를 재기록 하는 대신 P에 대한 스냅샷(snapshot) 로그를 저장하고, 필요시 이를 읽는 기법을 고안한다. 스냅샷 로그 레코드는 빈 SSD 공간에 기록되기 때문에 페이지 갱신을 발생시키지는 않는다. SSD에서 순수(pure) 쓰기 연산은 갱신 연산에 비해 낮은 I/O 비용을 가지며 스냅샷 로그를 읽는 비용 역시 매우 작음을 고려한다[2, 4, 6, 7].

제안한 기법은 버퍼 재배치 알고리즘에 따라 더티 페이지 재기록이 발생하는 상황에서도 스냅샷 로그를 기록함으로써 재기록 없이 해당 페이지를 버퍼풀에서 삭제할 수 있다. 그림 2는 이를 위해 버퍼 할당 테이블(buffer allocation table)이 어떻게 사용되는지 보인다. 사용된 버퍼 할당 테이블은 기존 테이블 구조에 스냅샷 비트를 추가한다. 이 비트는 해당 페이지가 재기록 없이 버퍼풀에서 삭제된 페이지인지 여부를 표시한다. 만약 더티 페이지 P가 재기록 없이 버퍼풀에서 삭제된 것이라면 P의 스냅샷 비트 값은 1이다. 또한 P를 위한 DPT의 RLR 정보도 그림 2의 하단과 같이 기록된 스냅샷 로그 레코드의 LSN 값으로 변경된다.

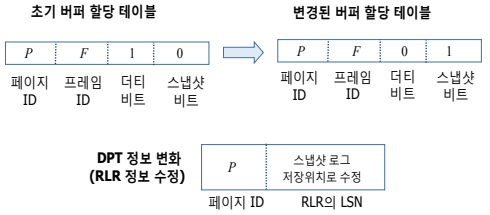


그림 2. 스탭샷 로그 기록을 위한 버퍼 할당 테이블 정보의 변경.

Fig. 2. Management of buffer allocation table for reflecting snapshot log being saved.

그림 2와 같이 페이지 P가 버퍼풀에서 삭제된 후 다시 참조될 때를 가정해보자. 이 경우 버퍼풀 관리자는 P의 스냅샷 비트 값을 보고 P가 재기록 없이 버퍼 삭제되었음을 알 수 있다. 이에 따라 페이지 P를 SSD에서 다시 버퍼풀로 읽은 후 스냅샷 로그를 적용하여(즉, 물리적 redo 연산 수행) P의 삭제 전 이미지를 올바르게 생성할 수 있다. 다른 경우는 더티 페이지를 버퍼풀에서는 삭제하지 않고 단지 REDO 시작점을 뒤로 하기 위해 스냅샷 로그 레코드를 기록하게 되는 경우이다 이 경우에도 스냅샷 비트가 1로 변경되며 더티 페이지 비트가 0으로 됨으로써 더티 페이지는 아님을 기록하면 된다.

이처럼 제안한 기법은 스냅샷 로그를 기록함으로써 페이지 갱신 없이 버퍼 공간을 확보하거나 REDO 시작 시점을 앞당길 수 있다. 더티 페이지가 버퍼에서 삭제된 경우라면 다시 적재할 때 해당 페이지를 다시 SSD로부터 읽어야 하고 스냅샷 로그도 읽어야 한다. 이 때문에 페이지 갱신 횟수를 줄이는 대신 페이지 읽기와 스냅샷 로그 읽기를 위해 2번의 I/O가 필요하다. 만약 저장장치가 HDD라고 한다면 노드 갱신과 읽기 연산과의 비용차가 없기 때문에 제안된 방식은 사용할 수 없다. 하지만 SSD의 특성으로 인해 성능상의 이점을 기대할 수 있다.

Ⅲ. 스냅샷 로그 기반의 복구기법

1. 스냅샷 로그 레코드

트랜잭션의 원자성과 지속성(durability)을 위한 로깅 기법은 undo와 redo 연산을 위한 로그 데이터를 하나의 레코드에 저장한다. 이때 undo 로그는 크기를 줄이기 위해 논리적 undo가 적용되며, redo 로그의 경우는 수행 속도를 빠르게 하기 위해 유사물리(physiological) 로그가 작용된다[10]. 본 연구도 이를 따르며 단지 스냅샷 로그라는 새로운 종류의 로그를 사용한다.

제안된 스냅샷 로그는 undo에 사용되지 않기 때문에 redo 로그만을 가지며 물리적 redo 로그이다. 저장된 (물리적) redo 로그는 갱신된 페이지의 수정 후 이미지(즉, after image), after-image의 크기와 페이지 내 위치 정보(즉, 시작 오프셋)로 표현된다. 즉, 로그 데이터는 (offset, size, after-image)의 형태를 가진다. 페이지 안에서 여러 번의 갱신이 발생한 경우라면 갱신 영역의 합집합(union) 영역에 대해서 로그 데이터가 저장된다. 합집합 영역이 N개의 영역으로 구성되었다면 해당 스냅샷 로그 레코드는 [LSN, (offset-1, size-1, after-image-1), … , (offset-N, size-N, after-image-N)]의 형태로 기록된다. 스냅샷 레코드에 기록된 after-image들을 읽어 갱신 전의 페이지에 덮어쓰기 함으로써 물리적 redo를 수행한다. 스냅샷 레코드의 크기를 일정 이하로 두기위해 after-image의 합이 전체 페이지 크기의 50%를 넘지 않도록 한다. 이 비율을 초과하는 갱신이 발생할 때 해당 더티 페이지를 기존 방식과 같이 저장장치에 재기록 한다.

2. 제안하는 시스템 복구 절차

제안하는 방법도 기존 HDD 기반 시스템의 3단계 복구 절차를 따른다. 즉, 로그 분석 절차, REDO 절차, UNDO 절차를 순서대로 수행하며, 특히 UNDO 절차는 기존 방식이 그대로 적용된다. 이것은 추가된 스냅샷 로그가 undo 연산과는 관련이 없기 때문이다.

그림 3은 제안하는 복구 절차를 위한 유사 코드이다. 라인 2-10는 로그 분석 절차이며 로그 파일에 저장된 마스터 로그 레코드의 읽기부터 시작된다. 마스터 로그 레코드는 체크포인트 생성 시에 저장되는 데이터이며, 가장 최근의 DPT의 저장 위치를 기록한다[10]. DPT 저장 위치로부터 로그 레코드를 순차적으로 읽어 작업취소(rollback)할 트랜잭션 리스트를 생성한다. 라인 6-8에서는 읽혀진 로그가 스냅샷 로그 레코드인 경우 해당 정보에 맞춰 더티 페이지 정보를 생성한다. 만약 이미 더티 페이지로 기록되어 있던 페이지라면 RLR 정보만을 최신 정보로 갱신한다.

라인 12에서부터 REDO 절차가 시작되며 가장 작은 LSN을 가진 RLR을 찾음으로써 REDO 시작점을 구한다. 이에 맞춰 라인 13에서와 같이 로그 파일 포인터가 수정된 후 프로시저 *DoRedoAct()*가 수행된다. 이 프로시저에 대해서는 다음 절에서 설명한다. 마지막 절차인 UNDO 절차는 라인 16-17에서 수행되며, 이를 통해 undo시킬 트랜잭션의 갱신 연산이 취소된다.

Algorithm 1: Procedure *RestartSystem(Log, BufPool)*

```
Input  : LogFile = 로그 파일 접근을 위한 자료구조;
         BufPool = 버퍼풀 접근을 위한 자료구조;
1  /* 1단계 : 로그 분석 수행 */
2  LogFile에 저장된 마스터 로그 레코드로부터 더티 페이지
   테이블 위치를 찾아 해당 페이지를 DT로 읽어들임;
3  DT에 저장된 active 트랙잭션 정보를 이용하여 undo 트랙잭션
   집합을 생성;
4  while LogFile.IsEnd() ≠ true do // 파일 스캔
5      Rec ← LogFile.GetNextLog() ; // 로그레코드 fetch
6      if Rec 로그의 type이 스냅샷 로그 레코드 then
7          테이블 DT에 더티페이지로 Rec.page_ID를 삽입.
           Rec.LSN 값을 해당 페이지 RLR의 LSN 값으로
           둠으로써 redo 시작점을 앞으로 당김;
8          이미 DT에 있던 페이지라면 해당 페이지의 RLR LSN
           값을 Rec.LSN으로 갱신;
9      else
10         Rec이 Commit 레코드이면 Rec을 만든 트랜잭션을
           undo 트랙잭션 집합에서 삭제. Rec을 생성한 트랜잭션이
           undo 트랜잭션 집합에 없다면 집합에 삽입;
11 /* 2단계 : REDO 단계 수행 */
12 테이블 DT에 저장된 더터페이지의 RLR 중 가장 앞선 LSN를
   가진 RLR의 LSN 값을 REDO_START로 사용;
13 LogFile.SetOffset(REDO_START); // 파일포인터 이동
14 redo 연산을 위해 DoRedoAct(LogFile, DT, BufPool) 수행;
15 /* 3단계 : UNDO 단계 수행 */
16 저장된 undo 트랙잭션 집합을 참조하여 undo 연산을 수행;
17 시스템 복구 완료를 알리고 트랜잭션 처리를 시작;
```

그림 3. 시스템 복구를 위한 유사 알고리즘 *RestartSystem*.
Fig. 3. Pseudo-algorithm *RestartSystem* used for system recovery.

3 REDO 절차 알고리즘

앞 절에서 언급한 *DoRedoAct()*를 통해 REDO 절차가 수행된다. 이 절차에서는 스냅샷 로그를 가진 페이지에 대해서 물리적인 redo를 수행한 후, 다시 REDO 시작점으로 이동하여 기존 REDO 절차에 따라 redo 연산을 수행한. 이에 대한 설명을 위해 그림 4를 사용한다.

그림 4에서는 제안한 REDO 절차를 위한 유사코드를 보인다. 라인 1-8의 단계를 통해 더티 페이지 중 스냅샷 로그 레코드를 RLR로 가지는 모든 페이지에 대해서 물리적 redo 연산을 수행한다. 이를 위해 우선 해당 페이지와 스냅샷 로그 레코드를 버퍼로 읽어 들인 후 라인 4에서 redo 연산을 수행한다. 만약 스냅샷 로그를 가지지 않는 더티 페이지의 경우에 대해서는 redo 연산을 수행하지 않는다. 이런 페이지는 다음에 이어지는 라인 10-15에서의 redo 연산을 통해 복구된다.

Algorithm 2: Procedure *DoRedoAct(LogFile, DT, BufP)*

```
Input  : LogFile = 로그파일 접근을 위한 자료구조;
         DT = 더티 페이지 테이블 접근을 위한 자료구조;
         BufP = 버퍼풀 접근을 위한 자료구조;
1  foreach dp ∈ DT do // 스냅샷 로그를 우선적으로 읽음
2      (db, sb) ← BufP.GetBits(dp.page_ID);
3      if (db, sb) = (0, 0) then // 스냅샷 로그를 가짐
4          frame ← BufP.Read(dp.page_ID); // 페이지 로딩
5          log ← LogFile.ReadSnapshot(dp.rec_LSN); // 스냅샷
           로그 레코드 로딩
6          BufP.DoSnapshot(frame, log); // redo 수행
7      else
8          Do nothing; // 일반 로그 레코드는 다음 단계에서 처리
9  LogFile의 파일 포인터를 REDO 시작점으로 다시 위치시킴;
10 while LogFile.IsEnd() ≠ true do // 파일 스캔
11     log ← LogFile.GetNextLog(); // 로그 레코드 fetch
12     if ! BufP.Exist(dp.page_ID) then // 버퍼에 없음
13         BufP.Read(log.page_ID); // 해당 페이지 로딩
14     if rec.lsn < BufP.LSN(dp.page_ID) then // 로그
       레코드와 데이터 페이지 간의 LSN 비교
15         BufP.DoRedo(dp.page_ID, log); // redo 연산 수행
```

그림 4. REDO 절차를 위한 유사 알고리즘 *DoRedoAct*.
Fig. 4. Pseudo-algorithm *DoRedoAct* for REDO procedure.

Ⅳ. 성능 분석

버퍼풀에 적재된 페이지 X가 빈번히 참조되고 갱신된다면 X는 오랜 시간 재기록없이 버퍼풀에 존재하게 된다[1, 6, 8, 10]. 이를 통해 물리적인 I/O 횟수를 줄일 수 있지만 갱신 연산이 오랜 시간 데이터베이스에 반영되지 않음으로 해서 REDO 수행 시간을 늘린다. REDO 시간이 길어지는 것을 막기위해 기존 HDD 기반 시스템에서는 백그라운드 모드로 더티 페이지를 재기록 하였다.

제안된 기법에서는 이런 시스템 복구를 고려한 더티 페이지의 재기록이나, 버퍼풀 공간을 만들기 위해 수행되던 더티 페이지 재기록 대신 스냅샷 로그를 사용하였다. 스냅샷 로그에 페이지의 최신 이미지인 after-image를 저장하여 REDO 시작점을 앞당기는 동시에 페이지 갱신을 피할 수 있었다. 사용되는 after-image의 최대 크기를 한정 시킬 수 있으며 이를 페이지 크기의 50% 크기로 정하고 있다. 로그 레코드는 한번의 I/O로 여러 개가 동시 저장되는 것이 보통이기 때문에, 스냅샷 로그의 실제 쓰기 비용은 이 보다 매우 작다[4, 5, 9]. 이런 장점에 더해 제안된 기법은 시스템 복구 과정에서 수행되는 redo 연산의 수를 줄일 수 있다는 장점이 있다. 즉, REDO 시작 시점과 RLR인 스냅샷 로그 레코드 사이에 존재하는 redo 로그 레코드는 모두 무시할 수 있기 때문에 redo 연산 횟수를 줄일 수 있다.

하지만 제안된 기법은 스냅샷 로그가 기록된 페이지가 버퍼풀에서 삭제된 후에 다시 참조된다면 온라인 redo를 수행해야 하는 추가비용이 발생한다. 이 경우 재기록 없이 삭제된 페이지를 다시 읽어 들어야 하고 스냅샷 로그도 함께 읽어야 한다. 또한 after-image를 사용한 물리적 redo 연산 비용이 추가 비용이 된다. 이때 redo를 위한 메모리 복사 비용은 무시할 수준이기 때문에 스냅샷 로그의 읽기/쓰기 비용과 해당 페이지를 다시 읽기 위한 비용을 추가 비용으로 둘 수 있다.

이와 같은 비용 모델에 따라 제안된 기법을 통해 얻을 수 있는 I/O 측면에서의 성능향상을 에서 평가할 수 있다. 페이지의 갱신비용(C_u), 읽기 비용(C_r), 그리고 페이지 쓰기 비용(C_w)을 사용하여 다음의 성능평가 식 (1)을 구할 수 있다. 식에서 C_{gain}은 한 번의 페이지 갱신을 줄였을 때 기대할 수 있는 I/O 측면에서의 성능 향상 분이다. C_u는 평균적인 페이지 갱신비용이며 이 값은 사용되는 SSD의 특성에 따라 다양한 값이 존재할 수 있다[1, 4, 15]. 뒤의 $0.5\times(C_w+C_r)$은 스냅샷 로그를 읽고 쓰는 비용이며 마지막의 C_r은 삭제되었던 더티 페이지를 다시 읽어 들이는 비용이다.

$$C_{gain} = C_u - 0.5\times(C_w + C_r) - C_r \quad (1)$$

$$= C_w + P_{full}\times C_{erase} - 0.5\times(C_w + C_r) \quad (2)$$

$$\geq P_{full}\times C_{erase} - C_r \quad (3)$$

C_u는 (2)에서와 같이 빈 페이지에 갱신된 내용을 기록하는 비용(C_w)와 가비지 회수에 따른 비용이 고려되어야 한다. 가비지 회수에 따른 I/O 비용은 full-merge가 발생할 때의 확률(P_{full})과 이때 필요한 블록 삭제 비용(C_{erase})에 따른다[1, 5, 7, 13]. 그리고 SSD의 경우 C_w의 크기는 C_r보다 크기 때문에 부등식 (3)을 얻을 수 있다.

위의 식에서 C_r은 SSD에서 매우 작고 반면 가비지 수집의 비용은 매우 크기 때문에 성능상의 잇점을 가진다[1, 4, 15]. 식에서 보이는 성능 개선은 하나의 더티 페이지를 갱신하지 않을 때의 개선이다. 일반적으로 버퍼풀에는 수 천 개의 페이지가 존재하기 때문에 제안한 방식을 통해 얻을 수 있는 I/O 성능 향상은 매우 큰 값이 된다고 할 수 있다. 이에 대한 값은 데이터베이스 저장 상황과 사용된 SSD 장치에 따라 다르지만, 5%-10%의 성능 향상을 기대할 수 있다.

V. 결 론

논문에서는 고성능 트랜잭션 처리 시스템의 주 저장장치로 플래시 메모리를 사용할 때 필요한 스냅샷을 사용한 로깅 및 데이터 복구 기법을 제안하였다. 제안한 기법은 플래시 메모리가 가지는 특성인 페이지 갱신과 읽기 비용간의 비대칭성에 기반한다. 즉, 페이지 갱신을 유발하는 더티 페이지 재기록을 줄이기 위해 스냅샷 로그라는 하는 페이지 단위의 물리적 redo를 위한 특별한 로그를 사용한다. 이 로그 데이터는 기존 디스크 기반 시스템이 가지는 로그 데이터의 목적, 즉 시스템 복구라는 목적 외에 더티 페이지가 버퍼풀로부터 제거될 때에도 저장장치에 재기록 되지 않도록 한다. 이런 방식은 페이지 갱신 비용과 읽기 비용 간에 차이가 없는 기존 HDD 저장 장치에서는 성능상의 이점이 전혀 없다. 하지만 SSD와 같은 플래시 메모리 저장 장치에서는 성능상의 잇점을 갖는다.

논문에서는 제안한 스냅샷 로그를 사용한 데이터베이스 복구를 위해 스냅샷 로그와 기존의 일반적 로그 레코드가 서로 섞여 저장되는 상황에서 REDO 알고리즘의 간단한 변화를 통해서 시스템 복구가 가능하도록 하였다. 이를 통해 기존 트랜잭션 시스템에서 사용하고 있는 버퍼풀 구현 모듈과 로깅 시스템에 작은 변경을 줌으로써 제안한 기법이 구현될 수 있도록 하였다. SSD 저장 장치를 고성능 트랜잭션 처리 시스템에 적용하려고 하는 요즘의 추세에 비추어 제안한 기법이 매우 유용하게 사용될 수 있을 것을 기대한다.

References

[1] Seong-Chae Lim, “A Flash-based B+-Tree using Sibling-Leaf Blocks for Efficient Node Updates and Range Searches,” JIIBC Vol. 8, No. 3, pp. 12-24, August 2016.
DOI: http://dx.doi.org/10.7236/IJIBC.2016.8.3.12

[2] Eunji Lee, Hyokyung Bahn, “NVM-based Write Amplification Reduction to Avoid Performance Fluctuation of Flash Storage,” The Journal of the Institute of Internet, Broadcasting and Communication, Vol.16 No.4, pp. 15-20. 2016.
DOI: http://dx.doi.org/10.7236/JIIBC.2016.16.4.15

[3] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne, Operating System Concepts, 9th edition, Wiley Publication, 2016.

[4] Samsung SSD 850 Evo Specification, Web

URL:https://www.cnet.com/products/samsung-ssd-850-evo/specs/

[5] Gap-Joo Na, Sang-Won Lee, and Bongki Moon, "Dynamic In-Page Logging for B+-tree Index," IEEE Trnas. on Knowledge and Data Engineering, Vol. 24, No. 7, pp. 1231-1243, 2012. DOI: 10.1145/1645953.1646152

[6] Mustafa Ganim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lan, "SSD bufferpool extensions for database systems," *In Proc.* of VLDB, pp. 1435-1446, 2010. DOI: 10.14778/1920841.1921017

[7] Colgrove, J., Davis, J., Hayes, Miller, E., Sandvig, C., Sears, R., Tamches, A., Vachharajani, N., Wang, & Purity, F. , "Building fast, highly-available enterprise flash storage from commodity components," *ACM SIGMOD*, pp. 1683-1694, June 2015. DOI: 10.1145/2723372.2742798

[8] Do, J., Zhang, D., Patel, J., DeWitt, D., Naughton, J., & Halverson, A., " Turbo-charging DBMS buffer pool using SSDs," *ACM SIGMOD*, pp.1113-1124, June 2011. DOI: 10.1145/1989323.1989442

[9] Sang-Won Lee and Bongki Moon, "Design of flash-based DBMS: An in-page logging approach", *ACM SIGMOD*, pp. 55-66, June 2007. DOI: 10.1145/1247480.1247488

[10] C. Mohan and Frank E. Levine, "ARIES/IM: An efficient and high concurrency index management method using write-ahead logging," ACM SIGMOD, 1992. DOI: 10.1145/130283.130338

[11] Sungup Moon, Sang-Phil Lim, Dong-Joo Park, and Sang-Won Lee, "Crash recovery in FAST FTL," *In Proc.* of Software Technologies for Embedded and Ubiquitous Systems. pp. 13-22, 2011

[12] Yongseok Son, Sunggon Kim, and Heon Young Yeom, "High-Performance Transaction Processing in Journaling File Systems," In Proc. of USENIX, pp. 224-240, Feb. 2018.

[13] Wu, G. & He, X. (2012). "Delta-FTL: Improving SSD lifetime via exploiting content locality," *In Proc.* of European conference on Computer Systems. April 2012. DOI: 10.1145/2168836.2168862

[14] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson, "KAML: A Flexible, high-performance key-value SSD," *In Proc.* of IEEE HPCA, Feb. 2017. DOI: 10.1109/HPCA.2017.15

[15] Puyuan Yang, Peiquan Jin, and Shouhong Wan, "HB-Storage: Optimizing SSDs with a HDD Write Buffer," *In Proc.* of International Confebrence on Web-Age Information Management, pp. 28-39, June 2013. DOI: 10.1007/978-3-642-39527-7_5

저 자 소 개

임 성 채(정회원)

• Seong-Chae Lim received his BS degree at Seoul National University, and then received MS and Ph.D degrees at KAIST. He currently works for the department of computer science at Dongduk Women's University.

※ 이 논문은 2018년도 동덕여자대학교 학술연구비 지원에 의하여 수행된 것임