플래시 메모리 기반의 DBMS를 위한 동적 블록 할당에 기반한 효율적인 로깅 방법

(An Efficient Logging Scheme based on Dynamic Block Allocation for Flash Memory-based DBMS)

하 지 훈 † 이 기 용 †† 김 명 호 †††
(Ji Hoon Ha) (Ki Yong Lee) (Myoung Ho Kim)

요 약 플래시 메모리는 비휘발성이면서도 작고 가벼우며, 전력 소모가 적고 충격에 강하다는 장점 등으로 인해 휴대 기기를 포함한 다양한 기기의 저장매체로 사용되고 있다. 그러나 플래시 메모리는 하드 디스크와는 달리 제자리 갱신이 불가능하고, 읽기 연산에 비해 쓰기 및 지우기 연산이 매우 느리기 때문에, 기존의 하드 디스크를 기반으로 설계된 데이터베이스 시스템은 플래시 메모리 상에서 최적의 성능을 내기 어렵다. 플래시 메모리 상에서 데이터베이스의 성능을 극대화하기 위해, 어떤 데이터에 변경이 발생하면 원래 위치의 데이터를 덮어쓰는 대신, 해당 데이터의 변경 사항에 대한 로그만을 다른 위치에 기록하는 방식들이 제안되었다. 본 논문에서는 플래시 메모리 기반의 데이터베이스 시스템을 위한 효율적인 로깅 방법을 제안한다. 제안하는 방법은 기존 방법들과 달리, 로그만을 저장하는 로그 블록들을 별도로 두고 데이터의 변경에 따라 발생하는 로그를 로그 블록들에 고르게 분포시킨다. 이를 통해 제안하는 방법은 페이지 쓰기 및 블록 지우기 연산의 횟수를 크게 감소시킬 수 있다. 합성 데이터와 TPC-C 벤치마크 데이터를 사용한 실험을 통해, 제안하는 방법은 기존의 방법에 비해 좋은 성능을 나타냄을 보였다.

키워드 : 플래시 메모리 기반 데이터베이스, 로그 블록 로깅

Abstract Flash memory becomes increasingly popular as data storage for various devices because of its versatile features such as non-volatility, light weight, low power consumption, and shock resistance. Flash memory, however, has some distinct characteristics that make today's disk-based database technology unsuitable, such as no in-place update and the asymmetric speed of read and write operations. As a result, most traditional disk-based database systems may not provide the best attainable performance on flash memory. To maximize the database performance on flash memory, some approaches have been proposed where only the changes made to the database, i.e., logs, are written to another empty place that has been erased in advance. In this paper, we propose an efficient log management scheme for flash-based database systems. Unlike the previous approaches, the proposed approach stores logs in specially allocated blocks, called log blocks. By evenly distributing logs across log blocks, the proposed approach can significantly reduce the number of write and erase operations. Our performance evaluation shows that the proposed approaches can improve the overall system performance by reducing the number of write and erase operation compared to the previous ones.

Key words : Flash memory-based DBMS, log block logging

· 이 논문은 2007년도 정부(교육과학기술부)의 재원으로 한국과학재단의 지원을 받아 수행된 연구임(No. R0A-2007-000-10046-0).
· 이 논문은 2009년도 한국과학기술원 BK21 정보기술사업단에 의하여 지원되었음

† 정 회 원 : KT
drha@kt.com
†† 정 회 원 : KAIST 전산학전공 연구조교수
kiyong.lee@gmail.com
††† 종신회원 : KAIST 전산학전공 교수
mhkim@dbserver.kaist.ac.kr

논문접수 : 2009년 3월 31일
심사완료 : 2009년 7월 16일

정보과학회논문지 : 데이타베이스 제36권 제5호(2009.10)

1. 서 론

최근 들어 NAND 플래시 메모리는 MP3 플레이어, PMP(Personal Media Player), 휴대폰 등 휴대 기기의 저장 장치로서 널리 사용되고 있다. 이는 NAND 플래시 메모리가 비휘발성(non-volatility)이면서도 작고, 가벼우며, 전력 소모가 적고, 충격에 강하다는 장점을 가지고 있기 때문이다[1]. NAND 플래시 메모리의 기술 발전으로 집적도가 해마다 평균 두 배씩 증가하는 한편 가격은 점점 낮아지면서, NAND 플래시 메모리는 하드 디스크를 대체할 차세대 저장 매체로서 주목 받고 있다. 플래시 메모리로 구성된 SSD(Solid State Disk)를 탑재한 랩톱 컴퓨터와 데스크톱 컴퓨터가 이미 상용화되었으며, 추후 플래시 메모리가 엔터프라이즈 서버의 저장 매체로서도 사용될 것으로 예상된다[2]. 이렇게 플래시 메모리의 용량이 증가하면서, 플래시 메모리에서의 데이터베이스 시스템의 사용도 크게 늘고 있다[3,4].

그러나 플래시 메모리는 하드 디스크와는 다른 몇 가지 특성을 가지고 있다. 플래시 메모리는 하드 디스크와는 달리 제자리 갱신(in-place update)이 불가능하며, 읽기 연산에 비해 쓰기 연산이 느리고, 플래시 메모리를 구성하는 각 블록(block)에 대한 지우기 연산의 횟수에 제한이 있다는 제약을 가진다. 플래시 메모리는 하나 이상의 블록으로 구성되고, 각 블록은 다시 고정된 수의 페이지(page)로 이루어진다. 플래시 메모리는 제자리 갱신이 불가능하므로, 플래시 메모리에서 어떤 페이지를 갱신하기 위해서는 해당 페이지를 포함하고 있는 블록 전체에 대한 지우기 연산을 수행한 다음, 해당 페이지에 대한 쓰기 연산을 수행하여야 한다. 지우기 연산은 읽기나 쓰기 연산에 비해 매우 느리고 각 블록에 대한 지우기 연산의 횟수에는 제약이 있으므로, 어떤 페이지의 데이터가 변경될 때마다 매번 블록에 대한 지우기 연산을 수행하는 것은 시스템의 성능을 저하시키는 한편 플래시 메모리의 수명을 크게 단축시킨다. 이러한 제약을 극복하기 위해 플래시 메모리에서는 흔히 플래시 변환 계층(Flash Translation Layer, FTL)[5]이라는 소프트웨어 계층이 사용된다. FTL은 어떤 데이터에 대한 쓰기 요청이 들어오면, 해당 데이터를 원래 위치가 아닌 미리 지워놓은 빈 페이지에 쓰는 방식을 통해 플래시 메모리를 마치 제자리 갱신이 가능한 하드 디스크처럼 사용할 수 있게 해준다.

기존의 하드 디스크를 기반으로 설계된 데이터베이스 시스템도 FTL을 사용하면 플래시 메모리에서 별도의 수정 없이 그대로 수행될 수 있다. 그러나 FTL은 순차 쓰기(sequential writes)가 아닌 임의 쓰기(random writes)를 주로 요청하는 데이터베이스 시스템에 대해서는 최적화된 성능을 내기 어렵다고 알려져 있다[4]. 이에 따라, 플래시 메모리 상에서의 데이터베이스 시스템의 성능을 극대화하기 위해 데이터베이스 시스템이 FTL을 사용하지 않고 플래시 메모리에 직접 접근하는 방법들이 제안되었다[3,4]. 이렇게 데이터베이스 시스템이 FTL을 사용하지 않고 플래시 메모리에 직접 접근하는 방법들은, 어떤 데이터에 변경이 발생했을 때 원래 위치의 데이터를 덮어쓰지 않고, 해당 데이터의 변경 사항에 대한 로그만을 다른 위치에 기록하는 방식을 사용한다. 이러한 방법들 중 대표적인 방식인 로그-구조(log-structured) 방식[3,6]은 어떤 데이터에 변경이 발생하면 그에 대한 로그를 항상 데이터베이스의 맨 끝에 순차적으로 기록한다. 이 방법은 변경된 데이터의 위치에 관계없이 쓰기 연산이 항상 데이터베이스의 맨 끝에서만 순차적으로 일어나므로, 임의 쓰기 요청에 대해서도 좋은 성능을 보일 수 있다. 그러나 이 방법은 어떤 데이터의 최신 내용을 얻기 위해 로그 전체를 검색해야 하므로 읽기 성능이 떨어진다는 단점이 있다. 페이지내 로깅(In-page logging) 방식[4]은 이러한 문제를 극복하기 위해 제안된 방식으로, 플래시 메모리의 각 블록 내에 고정된 크기의 영역을 로그 영역으로 할당한다. 이 방법은 어떤 데이터에 변경이 발생하면 그에 대한 로그를 해당 데이터와 같은 블록 내의 로그 영역에 기록한다. 따라서 이 방법은 어떤 데이터의 최신 내용을 얻기 위해 해당 데이터가 저장된 블록 내의 로그 영역에 기록된 로그만 검색하면 되므로 읽기 성능이 향상된다. 그러나 이 방식은 어떤 블록 내의 로그 영역이 로그로 가득 차게 되면, 해당 로그 영역 내의 로그를 블록 내의 데이터에 반영하여 새로 할당 받은 빈 블록에 쓰고 기존 블록을 지우는 합병(merge) 연산을 수행하여야 한다. 이 방식은 모든 플래시 메모리 블록에 대해 동일한 고정된 크기의 로그 영역을 할당하므로, 만약 어떤 블록의 데이터가 다른 블록의 데이터보다 빈번히 갱신되면, 해당 블록의 로그 영역은 로그들로 금방 차게 되어 해당 블록에 대한 빈번한 합병 연산이 발생하게 된다. 이러한 합병 연산은 다수의 쓰기 및 지우기 연산을 유발하므로, 이렇게 특정 블록의 데이터가 다른 블록의 데이터보다 빈번히 갱신되는 경우 IPL 방식은 많은 수의 쓰기 및 지우기 연산을 발생시킬 수 있다.

본 논문에서는 플래시 메모리 기반의 데이터베이스 시스템을 위한 새로운 로깅 방법을 제안한다. 제안하는 방법은 플래시 메모리의 블록을 데이터를 저장하는 데이터 블록과 그에 대한 로그만을 저장하는 로그 블록으로 나누고, 데이터 블록과 로그 블록 간의 N:1 연관 관계를 유지한다. 제안하는 방법은 어떤 데이터 블록 내의 데이터에 변경이 발생하면, 그에 대한 로그를 해당 데이

터 블록과 연관된 로그 블록에 저장한다. 특히 제안하는 방법은 각 데이터 블록의 로그 발생 빈도와 각 로그 블록의 여유 공간을 고려하여 데이터 블록과 로그 블록을 연관시킴으로써, 어떤 특정한 로그 블록만 로그로 금방 가득 차게 되어 빈번한 합병 연산이 발생하는 것을 막는다. 실험을 통해 본 논문은 제안하는 방법이 기존 방법에 비해 쓰기 및 지우기 연산의 발생 횟수를 크게 감소시킴을 보였다.

본 논문의 구성은 다음과 같다. 2장에서 플래시 메모리의 특징 및 관련 연구에 대해 기술한다. 3장에서는 제안하는 방법에 대해 자세히 설명한다. 4장에서 제안하는 방법의 성능을 평가하고, 마지막으로 5장에서 결론을 맺는다.

2. 배경 및 관련 연구

2.1 플래시 메모리의 특징

플래시 메모리는 크게 NAND 플래시 메모리와 NOR 플래시 메모리로 나뉜다. 이 중 NAND 플래시 메모리는 NOR 플래시 메모리에 비해 가격이 저렴하면서도 대용량의 저장 공간을 제공하기 때문에 현재 시장의 대부분을 차지하고 있다[7]. 본 논문에서는 NAND 플래시 메모리만을 고려하며, 별다른 표기가 없는 한 플래시 메모리는 NAND 플래시 메모리를 지칭한다.

플래시 메모리는 고정된 수의 블록으로 구성되며, 각 블록은 다시 고정된 수의 페이지로 이루어진다. 페이지는 플래시 메모리에서의 읽기와 쓰기 연산의 기본 단위이며, 주로 2KB 또는 4KB의 크기를 가진다. 플래시 메모리에서 읽기와 쓰기 연산은 페이지 단위로 수행되는 반면, 지우기 연산은 블록 단위로 수행된다. 각 페이지는 다시 주 데이터 영역(main data area)와 여유 영역(spare area)으로 구성된다. 여유 영역은 주로 에러 수정 코드(error correction code, ECC)나 기타 관리 정보를 저장하는데 사용되며, 주 데이터 영역을 읽거나 쓸 때 부가적인 비용 없이 동시에 같이 읽거나 쓸 수 있다. 그림 1은 플래시 메모리의 구성 예를 보여준다.

플래시 메모리는 하드 디스크와는 다른 몇 가지 특성을 가진다. 첫째, 플래시 메모리에서는 제자리 갱신(in-place update)이 불가능하다. 따라서, 어떤 페이지를 갱신하기 위해서는 해당 페이지를 포함하고 있는 블록 전체에 대한 지우기 연산을 수행한 후, 해당 페이지에 대한 쓰기 연산을 수행해야 한다. 이러한 특성을 쓰기 전 지우기(erase-before-write)라 부른다. 둘째, 플래시 메모리는 하드 디스크와는 달리 기계적인 부분이 없어서 데이터의 물리적 위치에 관계없이 모든 데이터에 동일한 속도로 접근할 수 있다. 따라서, 데이터의 물리적 위치는 시스템의 읽기 또는 쓰기 성능에 큰 영향을 미치지 않는다. 셋째, 플래시 메모리의 각 블록은 지우기 연산의 횟수가 약 10,000~100,000회 정도로 제한되어 있다. 허용된 지우기 연산의 횟수를 초과한 블록은 불량 블록(bad block)이 되어 데이터의 무결성을 보장할 수 없다. 따라서, 플래시 메모리에서는 지우기 연산의 횟수를 줄이거나, 특정 블록에 지우기 연산이 집중되는 것을 막는 마모 평준화(wear-leveling) 기법이 필수적이다[8]. 넷째, 플래시 메모리에서는 읽기 연산에 비해 쓰기 연산이 매우 느리며, 지우기 연산은 쓰기 연산에 비해서도 매우 느리다. 따라서, 시스템의 성능을 높이기 위해서는 쓰기 연산이나 지우기 연산의 횟수를 줄이는 것이 중요하다.

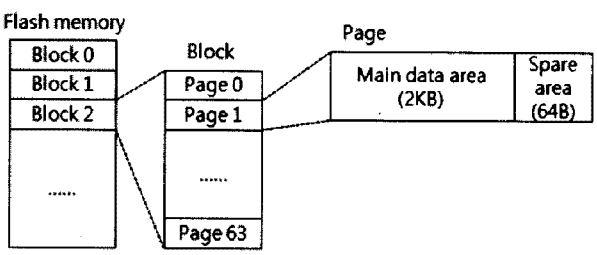


그림 1 플래시 메모리의 구성 예

2.2 관련 연구

1장에서 언급한 바와 같이, 기존의 하드 디스크 기반의 데이터베이스 시스템도 FTL을 사용하면 플래시 메모리 상에서 별도의 수정 없이 수행될 수 있다. FTL은 파일 시스템과 같이 순차 쓰기나 혹은 적은 개수의 페이지에 대한 임의 쓰기만을 주로 요청하는 시스템에 대해서는 좋은 성능을 보인다[9,10]. 그러나 FTL은 데이터베이스 시스템과 같이 대용량의 데이터에 대해 임의적인 쓰기 패턴을 보이는 시스템에 대해서는 최적화된 성능을 내기가 어렵다고 알려져 있다[4]. 이에 따라 플래시 메모리 상에서의 데이터베이스 시스템의 성능을 극대화하기 위해, 데이터베이스 시스템이 FTL을 사용하지 않고 플래시 메모리에 직접 접근하는 방법들이 제안되었다[3,4]. 이러한 방법들은 어떤 데이터의 변경에 따라 로그가 발생했을 때, 원래 위치의 데이터를 덮어쓰는 대신 해당 로그만을 별도의 위치에 기록하는 방법을 사용한다. 어떤 데이터의 최신 내용을 얻기 위해서는 원래 위치의 데이터를 읽어 이에 해당 데이터에 대한 로그를 반영하면 된다. 따라서 이 방법들은 읽기 성능이 빠르면서도 제자리 갱신 불가와 같은 제약을 가진 플래시 메모리에 적합하다. 로그를 어떤 위치에 어떤 방식으로 기록하느냐에 따라 여러 가지 방법이 존재할 수 있다. 이들 중 대표적인 방식인 로그-구조(log-structured) 방식[3,6]은 어떤 데이터에 변경이 발생했을 때, 해당 변경 사항에 대한 로그를 항상 데이터베이스의 맨 끝에

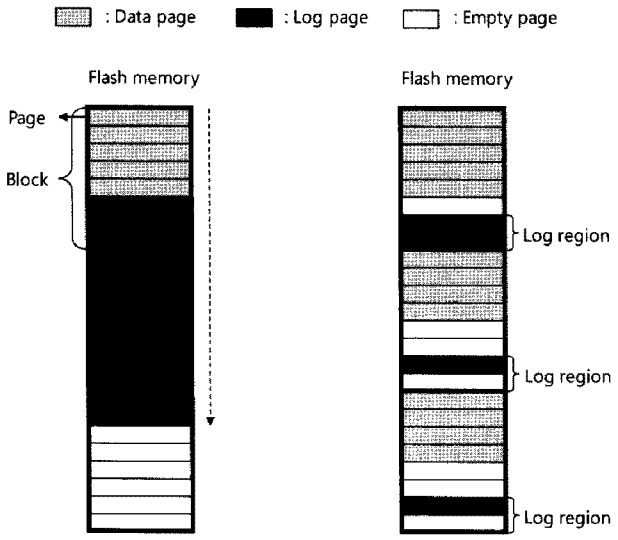


그림 2 로그-구조 방식과 페이지내 로깅 방식의 비교

쓴다. 그림 2(a)는 로그-구조 방식에 따라 플래시 메모리가 쓰여지는 예를 보여주는 그림이다. 그림 2에서 데이터 페이지(Data page)는 데이터를 저장하고 있는 플래시 메모리 페이지를 나타내며, 로그 페이지(Log page)는 로그를 저장하고 있는 플래시 메모리 페이지를 나타낸다. 이 방식에서는 변경된 데이터의 위치에 관계없이 쓰기 연산은 항상 데이터베이스의 맨 끝에서만 순차적으로 일어나므로 임의 쓰기 요청에 대해서도 좋은 성능을 보일 수 있다. 그러나 이 방식은 어떤 데이터의 최신 내용을 얻기 위해서 로그 전체를 검색해야 하므로 읽기 성능이 매우 떨어질 수 있다는 단점이 있다. 페이지내 로깅(In-Page Logging, IPL) 방식[4]은 이러한 문제를 극복하기 위해 제안된 방식으로, 그림 2(b)와 같이 플래시 메모리의 각 블록에 고정된 크기의 영역을 로그 영역(log region)으로 할당한다. 이 방법은 어떤 데이터에 변경이 발생하면, 해당 데이터의 변경 사항에 대한 로그를 해당 데이터와 같은 블록 내의 로그 영역에 기록한다. 따라서 이 방식은 어떤 데이터의 최신 내용을 얻기 위해서 해당 데이터가 담긴 블록 내의 로그 영역에 저장된 로그만 찾아보면 되므로 읽기 성능이 향상된다. 만약 어떤 블록 내의 로그 영역이 로그로 가득 차게 되면, 해당 로그 영역 내의 로그들을 해당 블록 내의 데이터에 반영하여 최신 내용으로 갱신된 데이터를 새로 할당 받은 빈 블록에 쓰고, 기존의 블록을 지우는 합병(merge) 연산을 수행한다. 일반적으로 데이터에는 80-20 법칙과 같이 자주 참조되고 갱신되는 데이터와 그렇지 않은 데이터가 있다. 또한 최근에 갱신된 데이터는 추후에 또 갱신될 확률이 높다. 그러나 IPL은 모든 플래시 메모리의 블록에 대해 동일한 고정된 크기의 로그 영역을 할당한다. 따라서, 빈번히 갱신되는 데이터를 담고 있는 블록은 그의 로그 영역이 로그들로 금방 차게 되어 빈번한 합병 연산이 발생하게 된다. 반면에 갱신이 거의 일어나지 않는 데이터들만을 담고 있는 블록은 로그가 거의 발생하지 않기 때문에 그의 로그 영역은 거의 빈 상태로 유지되게 된다. 합병 연산은 다수의 쓰기 및 지우기 연산을 유발하므로, 이렇게 특정 블록의 데이터가 다른 블록의 데이터보다 빈번히 갱신되는 경우 IPL 방식은 많은 수의 쓰기 및 지우기 연산을 발생시킬 수 있다. 또한 IPL에서는 블록의 데이터 페이지 영역이 거의 비어 있는 상태라도, 로그 영역만 로그로 가득 차게 되면 빈 영역을 충분히 사용하지 못하고 합병 연산이 발생하게 된다.

3. 동적 블록 할당 기반 로깅 방법

본 논문에서는 플래시 메모리 기반의 데이터베이스를 위한 효율적인 로깅 방법을 제안한다. 제안하는 방법은 기존 방법과 달리 로그만을 저장하는 로그 블록들을 별도로 두고, 데이터베이스의 데이터를 저장하는 데이터 블록과 로그 블록들 간의 N:1 연관 관계를 유지한다. 어떤 데이터 블록의 데이터에 대해 로그가 발생하면, 이를 해당 데이터 블록과 연관된 로그 블록에 기록한다. 제안하는 방법은 각 데이터 블록의 로그 발생 빈도와 각 로그 블록의 여유 공간을 고려하여 데이터 블록과 로그 블록 간의 연관 관계를 유지함으로써, 특정 로그 블록에만 로그가 몰리는 것을 방지한다. 이로 인해 제안하는 방법은 로그 블록이 로그로 가득 찼을 때 발생하는 합병 연산의 수를 크게 줄일 수 있다.

3.1 제안 방법의 개요

그림 3은 제안하는 방법에 따른 플래시 메모리의 블록 구성 및 데이터베이스 버퍼의 구조를 나타내는 그림이다. 플래시 메모리의 블록은 다음의 세 종류로 나뉜다.

- 데이터 블록: 데이터베이스의 데이터를 저장하는 블록
- 로그 블록: 데이터 블록에 저장된 데이터의 변경 사항에 대한 로그를 저장하는 블록
- 여유 블록: 빈 페이지만을 가지고 있는 블록

로그 블록의 개수는 동적으로 변할 수 있으며, 로그 블록의 최대 개수 MAXlog는 시스템 변수로 설정된다. 로그 블록의 최대 개수는 데이터베이스에 저장되는 데이터의 특성에 따라 다르게 설정될 수 있다. 예를 들어, 쓰기가 빈번하게 발생하는 환경에서는 합병 연산의 발생을 줄이기 위해 MAXlog의 값을 크게 하여 더 많은 수의 로그 블록을 할당할 수 있으며, 반대로 쓰기가 거의 발생하지 않는 환경에서는 MAXlog의 값을 작게 하여 적은 수의 로그 블록만을 할당할 수 있다. 각 데이터 블록은 고정된 크기의 데이터베이스 페이지로 구성된다. 데이터베이스 페이지의 크기가 8KB이고 플래시 메모리

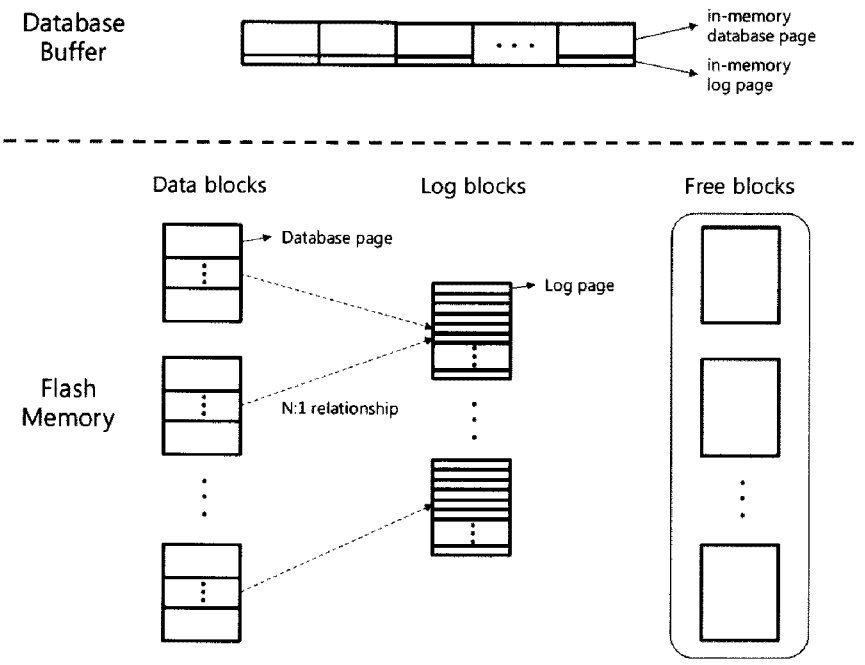


그림 3 제안하는 방법에 따른 플래시 메모리 구성 및 데이터베이스 버퍼 구조

의 블록의 크기가 256KB이라면, 하나의 데이터 블록에는 32개의 데이터베이스 페이지가 저장된다. 각 로그 블록은 고정된 크기의 로그 페이지로 구성된다. 본 논문에서는 로그 페이지의 크기를 플래시 메모리의 최소 쓰기 단위와 동일한 크기로 설정하여, 로그를 플래시 메모리에 쓸 때 불필요하게 플래시 메모리에 쓰여지는 데이터의 양을 최소화한다. 예를 들어, 최소 쓰기 단위가 페이지인 MLC(Multi-Level Cell) NAND 플래시 메모리의 경우, 페이지의 크기가 2KB라면 로그 페이지의 크기를 플래시 메모리 페이지의 크기와 동일한 2KB로 설정한다. 만약 블록의 크기가 256KB라면 하나의 로그 블록에는 총 128개의 로그 페이지가 저장된다. 여유 블록은 모든 페이지가 지워진 빈 블록으로서, 추후 데이터 블록이나 로그 블록으로 활용된다. 데이터베이스 버퍼에는 플래시 메모리에서 읽어 들인 데이터베이스 페이지들이 저장되며, 데이터베이스 버퍼 내의 각 데이터베이스 페이지에 대해서는 IPL과 유사하게 메모리 내 로그 페이지(in-memory log page)가 하나씩 할당된다. 메모리 내 로그 페이지의 크기는 로그 블록의 로그 페이지의 크기와 동일하다. 제안하는 방법은 메모리 상에 테이블의 형태로 데이터 블록과 로그 블록 간의 N:1 연관 관계 정보를 유지한다. 이 테이블을 연관 관계 테이블이라 부른다. 연관 관계 테이블은 어떤 데이터 블록에 대해 로그가 발생했을 때, 해당 로그를 어느 로그 블록에 기록할지를 알려준다. 이 연관 관계 테이블에 대해서는 3.2절에서 자세히 설명한다.

데이터 블록 내의 어떤 데이터베이스 페이지에 대해 변경이 발생하면, 해당 데이터베이스 페이지를 데이터베이스 버퍼로 읽어 들인 후 데이터베이스 버퍼 내에서 해당 데이터베이스 페이지를 갱신한다. 이 때, 변경 사항에 대한 로그를 해당 데이터베이스 페이지에 할당된 메모리 내 로그 페이지에도 기록한다. 만약 해당 데이터베이스 페이지가 버퍼 교체 정책 등에 의해 데이터베이스 버퍼로부터 방출되거나 그에 할당된 메모리 내 로그 페이지가 로그로 가득 차게 되면, 해당 데이터베이스 페이지 대신 그에 할당된 메모리 내 로그 페이지만을 해당 데이터베이스 페이지가 속한 데이터 블록과 연관된 로그 블록 내에 쓴다. 만약 어떤 로그 블록이 로그로 가득 차게 되면 합병 연산이 발생한다. 이 합병 연산은 해당 로그 블록 내의 로그들을 해당 로그 블록과 연관된 각각의 데이터 블록에 반영하여 각각 새로 할당 받은 빈 데이터 블록에 쓴 뒤, 기존의 로그 블록과 데이터 블록들을 지운다. 그리고 연관 관계 테이블에서 해당 로그 블록과 데이터 블록들 간의 연관 관계를 제거한다. 이러한 합병 연산은 3.3절에서 자세히 설명한다.

데이터베이스 버퍼에 올라와 있지 않은 데이터베이스 페이지를 플래시 메모리로부터 데이터베이스 버퍼로 읽어 들일 때는, 먼저 데이터 블록에서 해당 데이터베이스 페이지를 찾는다. 그 후, 해당 데이터베이스 페이지가 속한 데이터 블록과 연관된 로그 블록을 검색하여 해당 데이터베이스 페이지에 대한 로그를 찾아 해당 데이터베이스 페이지를 최신의 내용으로 갱신 한 뒤, 이를 데이터베이스 버퍼에 올린다. 이러한 읽기 연산은 데이터베이스 페이지를 플래시 메모리로부터 읽을 때마다 연

관된 로그 블록 전체를 검색해야 하는 비용이 발생하므로, 본 논문에서는 이러한 비용을 줄이기 위한 최적화 방법을 제안한다. 이러한 읽기 연산을 위한 최적화 방안은 3.4절에서 자세히 설명한다.

3.2 데이터 블록과 로그 블록 간의 연관 관계 유지

제안하는 방법은 데이터 블록과 로그 블록 간의 N:1 연관 관계를 유지한다. 이러한 연관 관계는 그림 4과 같이 테이블의 형태로 메인 메모리에 유지된다. 이 테이블을 연관 관계 테이블이라 부른다. 그림 4에서 Data block ID는 데이터 블록의 번호를 나타내며, Log block ID는 해당 데이터 블록과 연관된 로그 블록의 번호를 나타낸다. Last log page ID는 해당 데이터 블록에 대해 발생된 로그가 마지막으로 쓰여진 해당 로그 블록 내의 로그 페이지 번호를 나타낸다. Last log page ID는 데이터 페이지를 플래시 메모리로부터 데이터베이스 버퍼로 읽어 들일 때 발생하는 비용을 줄이기 위해 사용되는 정보로서, 추후 3.4절에서 다시 설명하도록 한다. 연관 관계 테이블은 어떤 데이터 블록에 대해 발생한 로그를 어떤 로그 블록에 기록할지를 알려준다. 어떤 데이터 블록에 대해 발생한 로그를 플래시 메모리에 쓰기 위해서는, 연관 관계 테이블로부터 해당 데이터 블록과 연관된 로그 블록을 찾은 뒤, 찾아진 로그 블록 내에 해당 로그를 순차적으로 쓴다.

Data block ID	Log block ID	Last log page ID
1	23	8
2	23	6
3	19	10
...	...	...

그림 4 연관 관계 테이블의 예

어떤 데이터 블록에 대해 발생한 로그가 최초로 플래시 메모리로 쓰여지기 전까지는 해당 데이터 블록은 어떠한 로그 블록과도 연관되어 있지 않다. 제안하는 방법은 어떤 데이터 블록에 대해 발생한 로그가 최초로 플래시 메모리로 쓰여지는 시점에 해당 데이터 블록을 어떤 로그 블록과 연관시킨다. 이제 주어진 데이터 블록을 로그 블록과 연관시키는 방법에 대해 설명한다. 어떤 로그 블록이 로그로 가득 차면 추후 설명할 합병 연산이 발생한다. 빈번한 합병 연산은 다수의 쓰기와 지우기 연산을 발생시켜 시스템의 성능을 크게 저하시키므로, 제안하는 방법은 합병 연산의 발생이 최소가 되도록 데이터 블록과 로그 블록을 연관시킨다. 합병 연산의 발생을 최소화하기 위해서는 어떤 특정한 로그 블록이 다른 로그 블록보다 로그로 빨리 차는 것을 막아서 모든 로그 블록들 간에 로그가 비슷한 정도로 차도록 해야 한다. 따라서 제안하는 방법은 어떤 데이터 블록에 로그 블록을 연관시킬 때, 로그로 가득 차게 되는 시점이 가장 늦다고 예상되는 로그 블록을 해당 데이터 블록에 연관시킨다. 이를 위해 메인 메모리에 현재 데이터 블록과 연관되어 있는 각 로그 블록에 대한 정보를 그림 5와 같이 유지한다.

Log block ID	Log page count	First log time

그림 5 로그 블록 정보 테이블

그림 5에서 Log block ID는 로그 블록의 번호를 나타내고, Log page count는 해당 로그 블록에 현재 몇 개의 로그 페이지가 저장되어 있는지를 나타낸다. First log time은 해당 로그 블록에 첫 번째 로그 페이지가 쓰여진 시각을 나타낸다. 데이터 블록과 로그 블록을 연관시키는 방법을 설명하기 전에 각 로그 블록에 대해 다음과 같은 용어를 정의한다.

- *LogPageCount(L)* = 로그 블록 *L*에 현재 저장된 로그 페이지의 수
- *FreeLogPageCount(L)* = 로그 블록 *L*에 현재 여유 로그 페이지의 수
- *FirstLogTime(L)* = 로그 블록 *L*에 첫 번째 로그 페이지가 쓰여진 시각
- *LogFrequency(L)* = *LogPageCount(L)* / (현재 시각 - *FirstLogTime(L)*)
- *EstimatedTimeToFull(L)* = *FreeLogPageCount(L)* / *LogFrequency(L)*

*LogFrequency(L)*은 로그 블록 *L*에 현재 저장된 로그 페이지의 수를 첫 번째 로그 페이지가 쓰여진 시각부터 지금까지 경과한 시간으로 나눈 값으로서, 로그 블록 *L*이 얼마나 빠른 속도로 로그로 채워지고 있는지를 나타내는 수치이다. 예를 들어, 빈번히 갱신되는 데이터를 가지고 있는 데이터 블록과 연관되어 있는 로그 블록일수록 큰 *LogFrequency(L)* 값을 가진다. *EstimatedTimeToFull(L)*은 로그 블록 *L*에 현재 남아있는 여유 로그 페이지의 수를 LogFrequency(L)로 나눈 값으로서, 로그 블록 L이 어느 정도의 시간 후에 로그로 가득 차게 될 것인지를 예상하는 수치이다. 어떤 데이터 블록에 대한 로그가 처음으로 플래시 메모리로 쓰여진다고 하자. 제안하는 방법은 현재 데이터 블록들과 연관되어 있는 로그 블록의 수 N을 확인한다. N이 로그 블록의 최대 허용 개수인 MAX_{log}보다 작으면, 제안하는 방법은 여유 블록에서 한 블록을 선택하여 이를 로그 블록으로

만들고, 이를 해당 데이터 블록과 연관시킨다. 만약, N이 MAX_{log}와 같으면, 제안하는 방법은 기존의 로그 블록들 중 *EstimatedTimeToFull(L)*이 가장 큰 로그 블록을 해당 데이터 블록과 연관시킨다. 이러한 방법을 통해 어떤 특정한 로그 블록에 로그가 몰리는 것을 방지할 수 있으며, 결과적으로 합병 연산이 불필요하게 빈번히 발생하는 것을 막을 수 있다. 그림 6은 지금까지 설명한 데이터 블록과 로그 블록을 연관시키는 방법을 나타내는 알고리즘이다.

```
Algorithm LogBlockAllocation

D: 로그 블록을 할당할 데이터 블록
{L1, L2, ..., LN}: 현재 데이터 블록들과 연관되어 있는 로그 블록들의 집합
MAXlog: 로그 블록의 최대 허용 개수

1: If (N < MAXlog) {
2:     여유 블록들 중 하나를 선택하여 로그 블록으로 만든다.
3:     해당 로그 블록을 D와 연관 시킨다.
4: }
5: else {
6:     L1, L2, ..., LN에 대해 각각 EstimatedTimeToFull(Li)을 구한다. (1 ≤ i ≤ N)
7:     EstimatedTimeToFull(Li)이 가장 큰 Li을 D와 연관시킨다.
8: }
9: 연관 관계 테이블을 갱신한다.
```

그림 6 로그 블록을 데이터 블록에 할당하는 알고리즘

연관 관계 테이블은 시스템의 전원이 나가면 메인 메모리에서 사라지게 된다. 따라서, 이 테이블을 시스템 구동 시 다시 복구할 수 있도록 하기 위해, 플래시 메모리의 각 블록의 첫 번째 페이지의 여유 영역에는 해당 블록이 로그 블록인지 아닌지를 나타내는 표시를 기록한다. 이 표시는 각 블록의 첫 번째 페이지가 기록될 때 같이 기록된다. 이와 함께, 로그 페이지가 로그 블록에 쓰일 때는 그의 여유 영역에 해당 로그 페이지에 담긴 로그를 발생시킨 데이터 블록의 번호를 기록한다. 시스템이 다시 구동되면 먼저 플래시 메모리의 각 블록의 첫 번째 페이지의 여유 영역을 검색하여 로그 블록들을 찾는다. 그 후, 각 로그 블록에 저장된 로그 페이지들의 여유 영역에 기록된 데이터 블록 번호를 탐색하면 연관 관계 테이블을 다시 구축할 수 있다.

3.3 합병 연산

어떤 로그 블록이 로그로 가득 차게 되면 합병 연산이 발생한다. 합병 연산은 해당 로그 블록 내의 로그들을 해당 로그 블록과 연관된 데이터 블록들에 반영하여 각각 최신의 내용으로 갱신하고, 이를 각각 여유 블록으로부터 새로 할당 받은 빈 블록에 쓴다. 이 블록들은 새로운 데이터 블록들이 된다. 그리고 기존의 로그 블록과 데이터 블록들을 지워 여유 블록으로 만들고, 연관 관계 테이블에서 해당 로그 블록과 데이터 블록 간의 연관 관계를 제거함으로써 합병 연산이 완료된다. 따라서 어떤 데이터 블록과 로그 블록의 연관 관계는 해당 로그 블록이 데이터 블록에 할당된 시점으로부터 로그 블록이 로그로 가득 차서 합병 연산이 발생할 때까지 유지된다.

3.4 읽기 연산의 최적화

제안하는 방법에서는 데이터베이스 버퍼에 존재하지 않는 어떤 데이터베이스 페이지를 플래시 메모리로부터 데이터베이스 버퍼로 읽어 들일 때, 해당 데이터베이스 페이지의 최신 내용을 얻기 위해 해당 데이터베이스 페이지가 속한 데이터 블록과 연관된 로그 블록 전체를 검색해야 하는 비용이 발생한다. 이는 각 블록 내에 할당된 로그 영역만을 검색하면 되는 IPL 방식보다 더 많은 비용을 요구하므로, 이를 감소시키고자 다음과 같은 방법을 사용한다. 먼저, 로그 블록에 저장된 각 로그 페이지의 여유 영역에 3.2절에서 언급한 해당 로그 페이지의 로그를 발생시킨 데이터 블록의 번호 외에 그림 7과 같은 정보를 추가로 기록한다.

Data block ID (2 bytes)	Database page ID (4 bytes)	Previous log page ID (1 bytes)	First log time (4 bytes)

그림 7 로그 블록 내 로그 페이지의 여유 영역에 기록되는 정보

그림 7에서 Data block ID와 Database page ID는 각각 해당 로그 페이지의 로그가 발생된 데이터 블록의 번호와 데이터베이스 페이지의 번호를 나타낸다. Previous log page ID는 동일한 데이터베이스 페이지에 대해 발생된 로그를 저장하고 있는 로그 블록 내에서의 바로 이전 로그 페이지 번호를 나타낸다. 이 정보는 로그 블록 내에서 주어진 데이터베이스 페이지에 대한 로그 페이지들만을 빠르게 찾기 위한 포인터의 용도로서 사용된다. First log time은 그림 5의 First log time과 같은 의미로서, 각 로그 블록의 첫 번째 로그 페이지의 여유 영역에만 기록된다. 이 정보는 시스템의 전원이 나갔다가 다시 구동 될 때 그림 5의 로그 블록 정보 테이블을 메인 메모리에 재구축하기 위해 사용된다. 이러한 정보는 각 로그 페이지 당 11 bytes에 불과하므로, 플래시 메모리 페이지의 여유 영역에 충분히 저장될 수 있다. 또한 여유 영역은 플래시 메모리 페이지의 주 데이터 영역과 동시에 기록이 되므로, 위와 같이 여유 영역에 추가적인 정보를 기록하는 것은 별도의 오버헤드를 발생시키지 않는다.

어떤 데이터베이스 페이지에 대한 로그를 로그 블록에서 찾기 위해서는, 먼저 연관 관계 테이블에서 해당

데이터베이스 페이지가 속한 데이터 블록과 연관된 로그 블록(Log block ID)과, 해당 데이터 블록에 대한 로그가 마지막으로 기록된 로그 페이지(Last log page ID)를 찾는다. 이후 해당 로그 블록에서 Last log page ID로부터 0번 로그 페이지 방향으로 진행하면서, 각 로그 페이지의 여유 영역에 저장된 Database page ID를 확인하여 해당 데이터베이스 페이지에 대해 마지막으로 저장된 로그 페이지를 찾는다. 이렇게 해당 데이터베이스 페이지에 대해 마지막으로 저장된 로그 페이지를 찾고 나면, 이후 해당 데이터베이스 페이지에 대한 나머지 로그 페이지들은 로그 페이지의 여유 영역에 저장된 Previous log page ID를 따라서 쉽게 찾을 수 있다. 이렇게 Previous log page ID를 사용하면, 로그 블록 내에서 해당 데이터베이스 페이지에 대한 로그 페이지들만 찾아 읽을 수 있으므로 검색해야 할 로그 페이지의 수를 줄일 수 있다.

4. 성능 평가

본 장에서는 Zipf 분포에 기반한 합성(synthetic) 데이터와 TPC-C 벤치마크[11] 데이터를 사용한 시뮬레이션을 통해 제안하는 방법과 IPL 방법의 성능을 비교한 결과를 보인다. 제안하는 방법을 Log block logging (LBL)이라 부르기로 하자. 3장에서 언급하였듯이 제안하는 방법은 데이터페이지를 데이터베이스 버퍼로 읽어들일 때 연관 로그 블록 전체를 검색해야 하므로, 로그 영역만을 검색하면 되는 IPL에 비해 더 큰 읽기 비용을 요구한다. 그러나 플래시 메모리는 쓰기 또는 지우기 연산에 비해 읽기 연산이 매우 빠르므로, 데이터베이스 페이지의 읽기 비용이 증가되더라도 쓰기 및 지우기 연산을 크게 감소시킴으로써 전체적인 시스템의 성능 향상을 가져올 수 있다.

4.1 실험 환경

본 실험에서는 제안하는 방법과 IPL 방법의 성능을 비교하기 위해 각 방법에 대한 시뮬레이터를 사용하였다. 제안하는 방법을 위한 시뮬레이터는 직접 구현하였으며, IPL을 위한 시뮬레이터는 [4]의 저자로부터 제공받을 수 있었다. 두 시뮬레이터 모두 C로 구현되었으며, 데이터베이스 페이지에 대한 읽기 및 쓰기 요청이 담긴 트레이스를 입력으로 받아, 각 방법에 따라 발생하는 플래시 메모리 페이지의 쓰기 및 읽기 연산의 횟수와 플래시 메모리 블록의 지우기 연산의 횟수를 출력한다. 본 실험에서는 최근 대용량 플래시 메모리의 대부분을 차지하고 있는 MLC NAND 플래시 메모리 환경을 가정하였다. 현재 일반적인 MLC NAND 플래시 메모리의 구조와 동일하게 플래시 메모리 페이지의 블록의 크기를 각각 2KB와 128KB로 설정하였으며, 하나의 데이터베이스 페이지의 크기는 일반적으로 주로 쓰이는 크기인 8KB로 설정하였다. 로그 페이지의 크기는 3.1에서 설명한 바와 같이 MLC NAND 플래시 메모리에서의 최소 쓰기 단위인 페이지와 동일한 크기인 2KB로 설정하였다. IPL 방법을 위해서는 [4]에서 저자가 제시한 바와 같이 128KB의 각 블록 당 8KB 크기의 로그 영역을 할당하였다. 따라서 데이터 영역과 로그 영역의 비율이 (128-8)KB:8KB = 15:1이 된다. 제안하는 방법을 위해서는 이와 동일한 비율로 전체 블록 중에서의 로그 블록의 최대 개수를 제한하였다. 실험에서는 데이터베이스의 크기를 1GB로 가정하였으며, 이는 IPL 방법에서 총 8739개의 블록에 저장된다. 이에 따라 제안하는 방법에서 로그 블록의 최대 개수는 8739 * 1/16 = 547개가 된다. 시뮬레이터 내에 구현된 데이터베이스 버퍼는 LRU 캐시 교체 알고리즘을 사용하며, [4]에서 가정한 데이터베이스 버퍼 크기를 따라 20MB의 크기로 설정되었다. 마지막으로, 로그 레코드 하나의 크기는 [4]에서 TPC-C 벤치마크를 상용 데이터베이스 시스템에서 수행하여 얻은 평균 로그 레코드의 크기인 50 Bytes로 가정하였다. 따라서 하나의 로그 페이지는 최대 40개까지의 로그 레코드를 저장할 수 있다.

본 실험에서는 제안하는 방법의 성능을 측정하기 위해 합성 데이터와 TPC-C 벤치마크에 기반한 데이터를 사용하였다. 합성 데이터는 읽기 및 쓰기 요청이 자주 일어나는 데이터베이스 페이지와 그렇지 않은 페이지가 있는 환경에서, 읽기 및 쓰기 요청이 특정 데이터베이스 페이지들에 집중되는 정도를 직접 변경시켜가며 제안하는 방법의 성능을 측정하기 위해 사용되었다. 이 때, 각 데이터베이스 페이지에 대한 읽기 및 쓰기 요청의 빈도수를 표현하기 위해 인기도에 따른 데이터의 접근 빈도를 모델링하는데 흔히 사용되는 Zipf 분포[12]를 사용하였다. TPC-C 벤치마크는 온라인 트랜잭션 환경(OLTP)을 나타내는 대표적인 벤치마크이며, 실생활의 작업부하(workload)를 나타내기 위해 여러 논문에서 표준적인 벤치마크로 사용되고 있다.

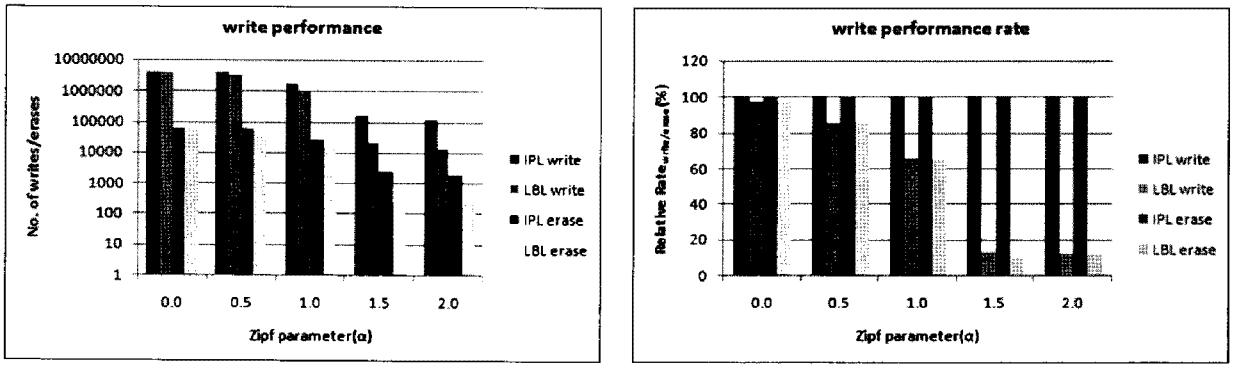
4.2 합성 데이터를 사용한 성능 평가

본 절에서는 Zipf 분포[12]에 기반한 합성 데이터를 사용하여 제안하는 방법과 IPL 방법의 쓰기, 읽기 및 전체 성능 비교를 하였다. 2.2절에서 언급한 바와 같이 일반적으로 데이터에는 많이 접근되고 갱신되는 데이터와 그렇지 않은 데이터가 있다. 특정 데이터에 대한 읽기 및 쓰기 요청의 집중 정도(skewness)에 따른 두 가지 방법의 성능을 비교하고자 Zipf 분포를 사용하여 데이터베이스 페이지에 대한 읽기 및 쓰기 요청을 생성하였다. Zipf 분포는 일부 데이터에 대한 집중 정도를 나타내기 위해 α라는 매개변수를 사용한다. α 값이 클수

록 적은 수의 일부 데이터베이스 페이지에 대해 요청이 집중되며, α 값이 0이면 모든 데이터베이스 페이지에 대해 동일한 빈도로 요청이 생성된다. 총 데이터베이스 페이지의 개수를 N이라 하고, i($1 \le i \le N$)를 데이터베이스 페이지의 번호라고 하면 i번째 페이지에 요청이 생성될 확률은 $f(i;\alpha,N)=\frac{1/i^{\alpha}}{\sum_{n=1}^{N}(1/n^{\alpha})}$로 표현된다. 본 실험에서는 균등 분포를 나타내는 $\alpha=0$부터 요청이 특정 데이터베이스 페이지에 심하게 집중되는 $\alpha=2.0$까지 α 값을 0.5씩 변화시켜가며 성능을 측정하였다. 데이터베이스의 크기는 1GB로 가정하였으며, 이를 저장하기 위해 필요한 데이터베이스 페이지 개수의 2배만큼인 262,144번의 쓰기와 읽기 요청을 생성하였다.

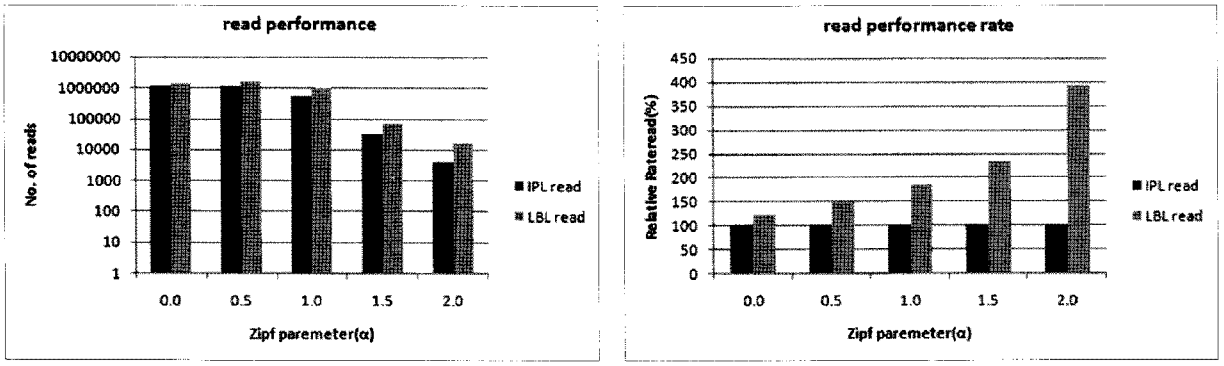
그림 8은 데이터베이스 페이지에 대한 262,144번의 쓰기 요청을 수행했을 때, IPL과 LBL의 쓰기 성능을 보여준다. 그림 8(a)는 각각의 방법에서 발생한 플래시 메모리 페이지 쓰기 연산 및 블록 지우기 연산의 횟수를 나타내며, 그림 8(b)는 IPL에서 발생한 연산의 횟수를 100%로 했을 때, LBL에서 발생한 연산 횟수의 상대적인 비율을 보여준다. 262,144번은 논리적인 쓰기 요청의 수인 한편, 그림 8(a)의 쓰기 횟수는 물리적인 쓰기 연산의 횟수임에 유의하라. 따라서 그림 8(a)에는 요청된 데이터를 쓰기 위한 쓰기 연산의 횟수 외에도, 합병 연산으로 인해 발생하는 쓰기 연산의 횟수까지 포함되어 있다. 예를 들어, IPL의 경우 합병 연산이 한 번 발생하면 60회의 플래시 메모리 쓰기 연산이 발생한다. (120KB/2KB = 60). 그림에서 볼 수 있듯이 LBL은 IPL에 비해 페이지 쓰기 연산 및 블록 지우기 연산 횟수에서 더 좋은 성능을 보이고 있다. 또한 α 값이 클수록, 즉 쓰기 요청이 일부의 데이터베이스 페이지에 집중될수록 LBL은 IPL에 비해 현저히 좋은 성능을 보이고 있다. 이것은 쓰기 요청이 일부의 데이터베이스 페이지에 집중될수록 IPL에서는 합병 연산이 빈번하게 발생하기 때문이다. 이에 비해 LBL은 로그가 일부 로그 블록에 몰리는 것을 방지하여 합병 연산의 발생을 감소시킨다. 대부분의 환경에서는 쓰기 요청이 모든 데이터에 대해 고르게 발생하지 않고 일부 데이터에 집중되는 것이 일반적이므로, 이러한 경우 LBL은 IPL에 비해 쓰기 및 지우기 연산의 발생을 크게 줄일 수 있다.

그림 9는 Zipf 분포의 α 값을 현실에서 주로 나타나는 값인 1.5로 하여[13] 262,144번의 쓰기 요청을 수행시킨 후, 다시 262,144번의 읽기 요청을 수행했을 때 IPL과 LBL의 읽기 성능을 보여준다. 이 때, 쓰기 요청이 집중되는 데이터베이스 페이지와 읽기 요청이 집중되는 데이터베이스 페이지는 동일하다고 가정하였다. 즉, 쓰기 요청이 많이 일어나는 데이터에 읽기 요청도 많이 일어나는 환경을 가정하였다. 그림 9(a)는 읽기 요청이 집중된 정도에 따라 각각의 방법에서 발생한 플래시 메모리 읽기 연산 횟수를 나타내며, 그림 9(b)는 IPL에서 발생한 연산의 횟수를 100%로 했을 때, LBL에서 발생한 연산 횟수의 상대적인 비율을 보여준다. 예상했던 바와 같이, 그림에서 보여주는 결과에서 LBL은 IPL에 비해 좋지 않은 읽기 성능을 보임을 알 수 있다. 이것은 데이터베이스 페이지를 플래시 메모리로부터 데이터베이스 버퍼로 읽어 들일 때, LBL에서는 IPL 보다 더 많은 로그 페이지를 검색해야 하기 때문이다. 읽기 요청이 쓰기 요청이 집중된 데이터베이스 페이지에 집중될수록 LBL에서의 페이지 읽기 연산 횟수가 증가한다. 쓰기 요청이 집중된 데이터베이스 페이지일 수록 로그 블록에 더 많은 수의 로그 페이지들이 존재하는데, 이러한 데이터베이스 페이지들에 읽기 요청이 집중되었기 때문이다. 만약, 읽기 요청이 집중된 곳과 쓰기 요청이 집중된 곳이 다르다면 LBL과 IPL의 읽기 성능 간의 격차는 줄어들 것이다. 그러나 LBL이 IPL에 근접한 읽기 성능을 낼 수 있다고 하더라도, LBL이 IPL 보다 좋은 읽기 성능을 내기는 어려울 것이다.



(a) (b)

그림 8 IPL과 LBL의 쓰기 성능 비교



(a) (b)

그림 9 IPL과 LBL의 읽기 성능 비교

그림 10은 데이터베이스 페이지에 대한 읽기 요청과 쓰기 요청의 상대 비율을 변화시켜 가면서 LBL과 IPL에서 발생하는 연산의 총 예상 수행 시간을 측정한 결과이다. 읽기 요청과 쓰기 요청을 합한 총 요청 횟수를 524,288번으로 고정하고, 해당 횟수 내에서 읽기 요청과 쓰기 요청의 상대적인 비율을 변화시켰다. 이 실험에서 읽기 요청과 쓰기 요청에 대한 Zipf 분포의 α 값은 1.5로 고정하였으며, 앞 실험과 마찬가지로 쓰기 요청과 읽기 요청이 집중되는 데이터베이스 페이지는 동일하다고 가정하였다. 발생하는 연산의 총 예상 수행 시간은 다음의 식으로 계산하였다.

총 예상 수행 시간 = (페이지 읽기 연산 발생 횟수) × 80μs
+ (페이지 쓰기 연산 발생 횟수) × 200μs
+ (블록 지우기 연산 발생 횟수) × 1.5ms

위 식에서 페이지 읽기 연산에 걸리는 시간인 80μs, 페이지 쓰기 연산에 걸리는 시간인 200μs, 블록 지우기 연산에 걸리는 시간인 1.5ms는 실제 플래시 메모리(Samsung K9WAG08U1A)에 대한 사양으로부터 가져온 것이다. 그림 10의 결과는 모든 경우에 대해 LBL이 IPL 보다 좋은 성능을 보이고 있음을 보여준다. 또한, 쓰기 요청의 비율이 높아질수록 LBL과 IPL 간의 성능 격차가 커짐을 알 수 있다. 쓰기 요청과 읽기 요청의 비율이 1:9인 경우에도, LBL은 IPL 보다 좋은 성능을 보임에 주목하라. 이것은 플래시 메모리에서 읽기 연산이 쓰기 연산과 지우기 연산에 비해 매우 빠르기 때문이다. 제안하는 방법은 IPL에 비해 더 많은 로그 페이지를 읽어야 하지만, 쓰기 연산과 지우기 연산의 수를 크게 줄임으로써 전체 성능을 크게 향상시킬 수 있다. 그림 10에서 LBL은 IPL보다도 적은 수의 플래시 메모리 페이지 읽기 횟수를 보임에 주목하라. 이것은 LBL이 IPL에 비해 합병 연산의 발생을 크게 줄임으로써, 합병 연산에 따라 발생하는 쓰기 연산 외에 읽기 연산의 횟수도 크게 줄이기 때문이다. 그러나 합병 연산이 발생하지 않는 순수한 읽기 환경에서는 LBL이 IPL에 비해 더 많은 읽기 횟수를 보일 것이다.

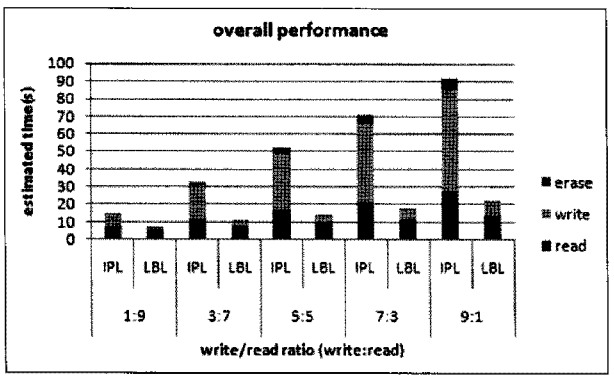
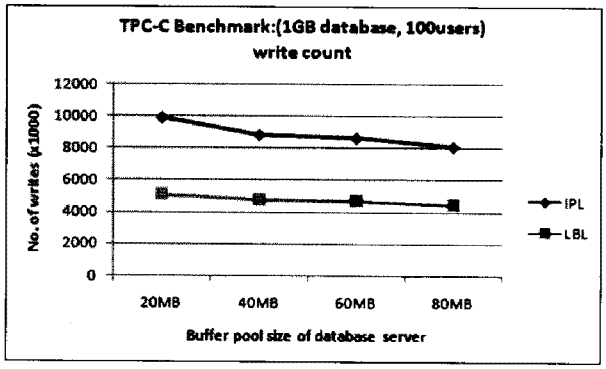


그림 10 IPL과 LBL의 전체 성능 비교

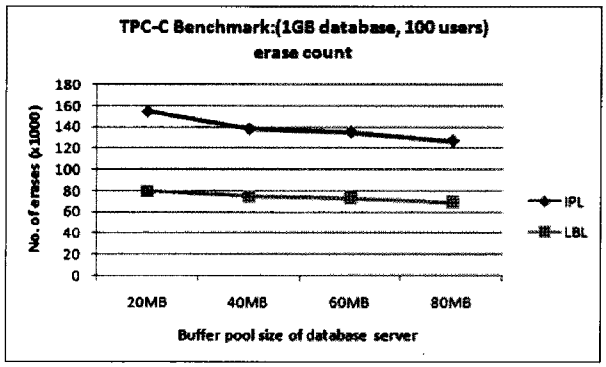
4.3 TPC-C 벤치마크를 사용한 성능 평가

본 절에서는 TPC-C 벤치마크에 기반한 데이터를 사용하여 제안하는 방법과 IPL 방법의 쓰기 성능을 비교하였다. 이 실험에서 사용한 트레이스는 TPC-C 벤치마크를 Oracle 데이터베이스에서 수행했을 때 생성되는 Redo 로그로부터 얻은 정보이다. 해당 트레이스에는 삽입, 삭제, 갱신 세가지 종류의 연산에 대한 물리-논리적 로그(physiological log)와 어떤 데이터베이스 페이지가 데이터베이스 버퍼로부터 방출되어 플래시 메모리에 기록될 때 발생하는 물리적 페이지 쓰기에 대한 로그가 기록되어 있다. 표 1은 실험에 사용된 각 트레이스에 포함된 물리-논리적 로그와 물리적 페이지 쓰기의 횟수를 나타낸다. 이들 트레이스는 [14]의 저자로부터 얻은 것이며, 각각 [DB 크기].[데이터베이스 버퍼 크기].[사용자 수]의 형태로 표기된다. 예를 들어, 1G.20M.100U는 데이터베이스의 크기가 1GB일 때, 데이터베이스 버퍼의 크기를 20MB로 설정한 상태에서, 100명의 사용자가 접속하는 환경에서 추출된 트레이스를 나타낸다.

그림 11은 데이터베이스 버퍼의 크기를 변화시킨 위 4개의 트레이스를 수행하였을 때, IPL과 LBL에서 발생한 페이지 쓰기 횟수와 블록 지우기 횟수를 비교한 그



(a)



(b)

그림 11 TPC-C 벤치마크에서의 IPL과 LBL의 쓰기 성능 비교

표 1 TPC-C 트레이스 정보

트레이스	물리-논리적 기록	물리적 페이지 쓰기
1G.20M.100U	1,853,429	56,170
1G.40M.100U	1,846,834	718,395
1G.60M.100U	1,847,148	703,882
1G.80M.100U	1,829,274	641,963

래프이다. 그래프에서 LBL은 쓰기 및 지우기 연산의 횟수를 IPL에 비해 약 50% 정도 감소시켰음을 알 수 있다. TPC-C 트레이스에 포함된 쓰기 요청은 일부 데이터에 집중되는 현상을 보인다. 이렇게 쓰기 요청이 일부 데이터에 집중되는 경우 LBL은 IPL에 비하여 합병 연산의 발생을 크게 줄일 수 있기 때문에, LBL은 IPL에 비하여 좋은 쓰기 성능을 보인다. 제안하는 방법은 모든 블록에 동일한 크기의 로그 영역을 할당하는 IPL과 달리, 갱신이 발생되지 않는 데이터에 대해서는 로그 영역을 할당해 주지 않고 이를 갱신이 빈번하게 발생하는 데이터 블록에 할당해 줌으로써 합병 연산의 발생을 줄인다. 플래시 메모리에서는 읽기 연산에 비해 쓰기와 지우기 연산의 비용이 매우 크므로, 이들의 발생 횟수를 줄임으로써 전체적인 시스템의 성능 향상을 가져올 수 있다.

5. 결 론

플래시 메모리는 하드 디스크에 비해 빠르고 가벼우며 충격에 강하다는 장점으로 인해 다양한 기기의 저장 매체로 사용되고 있다. 그러나 플래시 메모리는 하드 디스크와는 달리 제자리 갱신이 불가능하고, 읽기 연산에 비해 쓰기 및 지우기 연산이 매우 느리다는 단점을 가진다. 이에 따라 기존의 하드 디스크 기반의 데이터베이스 시스템은 플래시 메모리 상에서 최적화된 성능을 내기 어렵다. 본 논문은 이러한 단점을 극복하기 위해, 플래시 메모리 기반의 데이터베이스 시스템을 위한 효율적인 로깅 방법을 제안하였다. 제안하는 방법은 기존 방법과 달리 데이터의 변경에 대한 로그만을 저장하는 로그 블록들을 별도로 두고, 데이터의 변경 빈도를 고려하여 데이터 블록과 로그 블록간의 연관 관계를 유지함으로써 합병 연산의 발생 횟수를 줄인다. 이에 따라 쓰기 및 지우기 연산의 횟수가 크게 감소되어 전체적인 시스템의 성능이 향상된다. Zipf 분포에 기반한 합성 데이터와 TPC-C 벤치마크 데이터를 사용한 시뮬레이션 실험 결과는 제안하는 방법이 IPL에 비해 쓰기 및 지우기 연산을 크게 감소시킴을 보여주었다.

참 고 문 헌

[1] F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber, "Storage alternatives for mobile computers," In *Proceedings of the First Symposium on Operating Systems Design and Implementation(OSDI-1)*, pp.25-37, 1994.

[2] M. Slocombe. Samsung CEO, "NAND Flash Will Replace Hard Drives," http://digital-lifestyles.info/display-page.asp?section=platforms&id=2573, 2005.

[3] G. Kim, S. Baek, H. Lee, H. Lee, and M. Joe, "LGeDBMS: A Small DBMS for Embedded System with Flash Memory," In *Proceedings of VLDB*, 2006.

[4] Sang-Won Lee, and Bongki Moon, "Design of Flash-Based DBMS: An In-Page Logging Approach," ACM SIGMOD, 2007.

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

[6] Mendel RosenBlum and John K. Ousterhout., "The Design and Implementation of a Log-Structured File System," ACM Transactions on Computer Systems, vol.10, no.1, pp.26-52, 1992.

[7] 박상원, "플래시메모리와 데이터베이스", *정보과학회지*, 제25권, 제6호, pp.40-47, 2007.

[8] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," *ACM Computing Surveys*, 37(2):138-163, June 2005.

[9] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for compactflash systems," *IEEE Transactions on Consumer Electronics*, 48(2), pp.366-375, 2002.
[10] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee, "A Superblock-based Flash Translation Layer for NAND Flash Memory," *EMSOFT*, 2006.
[11] TPC-C benchmark, http://www.tpc.org/tpcc/, 2009.
[12] Ziff, G., "Human Behavior and The Principle of Least Effort," Addison-Wesley, Reading, MA., 1949.
[13] L. Breslau, P. Cao, L. Fan, G. Philips, and S. Shenker, "Web Caching and Zipf-like Distributions: Evidence and Implications," *IEEE INFOCOM '99*, pp.126-134, 1999.
[14] 김재명, 나갑주, 이상원, "IPL 기법의 인덱스 연산 분석", *인터넷 정보학회 2007년 춘계학술발표대회*, 2007년 5월.

하 지 훈
2007년 2월 고려대학교 컴퓨터학과 학사. 2009년 2월 한국과학기술원 전산학과 석사. 2009년 2월~현재 KT 네트워크연구소 전임연구원. 관심분야는 Embedded DB, 네트워크 보안

이 기 용
정보과학회논문지 : 데이터베이스
제 36 권 제 1 호 참조

김 명 호
정보과학회논문지 : 데이터베이스
제 36 권 제 1 호 참조