DOI: 10.3745/KIPSTD.2011.18D.3.157

AS B-트리: SSD를 사용한 B-트리에서 삽입 성능 향상에 관한 연구

김 성 호[†]·노 홍 찬[††]·이 대 욱[†††]·박 상 현[††††]

요 약

최근 플래시 메모리 및 SSD가 노트북이나 PC의 저장장치로 사용되는 것뿐 아니라, 기업용 서버의 차세대 저장장치로 주목 받고 있다. 대용량의 데이터를 처리하는 데이터베이스에서는 삽입, 삭제, 검색을 빠르게 하기 위해 다양한 색인 기법을 사용하는데 그 중B-트리 구조가 대표적인 기법이다. 하지만 플래시 메모리 상에서는 하드디스크와 달리 덮어쓰기(overwrite) 연산을 수행하기 위해서는 먼저 해당 블록(block)에 대하여 플래시 메모리의 연산 중 가장 비용이 많이 요구되는 삭제(erase) 연산을 수행 해야만 한다. 이러한 문제점을 극복하기 위해 플래시 메모리 사이에 위치하는 플래시 변환 계층(Flash memory Translation Layer)을 사용한다. 이 플래시 변환 계층은 수정한 데이터를 동일한 논리 주소에 덮어쓰기를 하더라도 실제로 임의의 다른 물리 주소에 저장하도록 하여 이 문제를 해결할 수 있다. NAND 플래시 메모리를 배열 형태로 포함하고 있는 SSD는 한 개 이상의 플래시 메모리 패키지를 병렬로 접근할 수 있다. 이러한 병렬 접근 방식을 사용하여 쓰기 연산 성능을 향상하기 위해서는 연속한 논리 주소에 쓰기 연산을 요청하는 것이 유리하다. 하지만 B-트리는 구성 노드에 대한 삽입 삭제 연산 시에 대부분 연속되지 않은 논리 주소 공간에 대한 갱신 연산이 일어나게 된다. 따라서 SSD의 병렬 접근 방식을 최대한 활용할 수 없게 된다. 본 논문에서는 수정한 노드를 연속한 논리 주소에 쓰도록 하는 AS B-트리 구조를 제안하여 SSD의 병렬 접근 방식을 최대한 활용할 수 있도록 하였다. 구현 및 실험한 결과 AS B-트리에서의 삽입 시간이 B-트리보다 21% 개선된 것을 확인하였다.

키워드 : 플래시 메모리, B-트리, 플래시 변환 계층, SSD, 병렬 접근 방식

AS B-tree: A study on the enhancement of the insertion performance of B-tree on SSD

Sungho Kim[†]·Hongchan Roh[††]·Daewook Lee[†††]·Sanghyun Park[††††]

ABSTRACT

Recently flash memory has been being utilized as a main storage device in mobile devices, and flashSSDs are getting popularity as a major storage device in laptop and desktop computers, and even in enterprise-level server machines. Unlike HDDs, on flash memory, the overwrite operation is not able to be performed unless it is preceded by the erase operation to the same block. To address this, FTL(Flash memory Translation Layer) is employed on flash memory. Even though the modified data block is overwritten to the same logical address, FTL writes the updated data block to the different physical address from the previous one, mapping the logical address to the new physical address. This enables flash memory to avoid the high block-erase cost. A flashSSD has an array of NAND flash memory packages so it can access one or more flash memory packages in parallel at once. To take advantage of the internal parallelism of flashSSDs, it is beneficial for DBMSs to request I/O operations on sequential logical addresses. However, the B-tree structure, which is a representative index scheme of current relational DBMSs, produces excessive I/O operations in random order when its node structures are updated. Therefore, the original b-tree is not favorable to SSD. In this paper, we propose AS(Always Sequential) B-tree that writes the updated node contiguously to the previously written node in the logical address for every update operation. In the experiments, AS B-tree enhanced 21% of B-tree's insertion performance.

Keywords : Lash Memory, B-Tree, FTL, SSD, Parallelism

※ 이 논문은 2011년도 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임(2011-0004382).
† 준 회 원 : 연세대학교 컴퓨터과학과 석사과정
†† 준 회 원 : 연세대학교 컴퓨터과학과 박사과정
††† 준 회 원 : 서강대학교 컴퓨터공학과 연구교수
†††† 종신회원 : 연세대학교 컴퓨터과학과 교수(교신저자)
논문접수 : 2011년 1월 18일
수 정 일 : 1차 2011년 2월 24일
심사완료 : 2011년 3월 14일

1. 서 론

최근 플래시 메모리 및 SSD가 노트북이나 PC의 저장장치로 사용되는 것뿐 아니라, 기업용 서버의 차세대 저장장치로 주목 받고 있다. SSD는 여러 NAND 플래시 메모리가 결합한 형태로 페이지 단위로 읽기와 쓰기가 이루어지는 NAND 플래시 메모리와 다르게 하드디스크처럼 연속된 읽기와 쓰기가 가능하도록 병렬 접근 방식(parallelism)과 인터리빙(interleaving) 등의 기능을 지원한다. 하지만 SSD에서도 병렬 접근 방식을 사용하기 위해서는 연속적인 논리주소를 가지는 페이지들을 한꺼번에 요청하는 것이 유리하다[1].

플래시 메모리 상에서는 하드디스크와 달리 덮어쓰기(overwrite) 연산을 수행하기 위해서는 먼저 해당 블록(block)에 대하여 플래시 메모리의 연산 중 가장 비용이 많이 요구되는 삭제(erase) 연산을 수행 해야만 한다. 이러한 문제점을 극복하기 위해 플래시 메모리 사이에 위치하는 플래시 변환 계층(Flash memory Translation Layer)이란 시스템 소프트웨어를 사용한다. 플래시 변환 계층은 다양한 블록교체기법을 적용하여 사용자에게 비교적 수행시간이 오래 걸리는 삭제연산에 대하여 이미 삭제연산을 수행한 빈 블록으로 재사상(re-mapping) 함으로써 플래시 메모리를 기존의 하드디스크와 같이 사용할 수 있도록 해 준다[2].

일반적으로 데이터베이스 시스템은 성능향상을 위해 B-트리, 해시 테이블(hash table) 등에 기반한 색인 구조를 구성하게 된다. B+트리를 포함한 B-트리 인덱스의 경우 레코드 삽입 시 한 개 또는 그 이상의 노드 수정이 이루어지는데 이는 바로 파일시스템의 연속되지 않은 논리 주소 상에 덮어쓰기 연산를 수행하도록 요청하게 된다. 연속되지 않은 논리주소에 대한 덮어쓰기 요청은 SSD에서 병렬 접근 방식의 이점을 사용할 수 없도록 하여 상대적으로 높은 I/O 비용을 초래하게 된다.

이러한 기존 B-트리의 문제점을 해결하기 위해, 본 논문에서는 SSD의 병렬 접근 방식의 이점을 사용할 수 있도록, 수정된 노드에 대해 항상 연속된 논리 주소에 쓰기 연산을 수행할 수 있는 새로운 B-트리 구조인 AS B-트리를 제안한다. 우선 기존 B-트리에서 사용하는 노드의 논리 주소를 노드 식별 번호로 대체하고 실제로 수정된 노드를 저장할 때는 항상 파일의 끝부분에 저장을 하여 수정된 노드들이 항상 연속한 논리 주소에 저장되도록 한다. 그리고 저장한 논리 주소와 노드 식별 번호를 연결하는 주소 변환 테이블을 RAM에 저장한다. SSD의 병렬 접근 방식의 이점을 최대한 활용하기 위해 쓰기 버퍼에 수정된 노드들을 저장하고 한번에 여러 노드들을 연속된 주소 공간에 저장한다.

이후 2 장에서는 플래시 메모리와 SSD의 기본적인 속성과 최근 연구들에 대해서 소개를 하고 B-트리에 대해서도 간단히 설명한다. 3 장에서는 본 논문에서 제안하는 AS B-트리의 구조와 동작에 대해 설명을 한다. 그리고 4 장에서는 AS B-트리의 구현과 실험을 기존 B-트리와 BFTL과 함께 진행하여 개선 효과를 분석한다.

2. 관련 기술

2.1 NAND 플래시 메모리 속성

하나의 NAND 플래시 메모리 칩은 여러 개의 블록(block)으로 구성되어 있고, 그 블록은 여러 개의 페이지로 구성되어 있다. 각 페이지는 main data area와 spare area로 구성되어 있으며 spare area는 보통 오류 정정 코드(ECC) 및 그 외 부가 정보들이 저장되어 있다.

NAND 플래시 메모리는 페이지 단위로 읽기와 쓰기가 가능하며 블록 단위로 삭제가 가능하다. 그리고 새로운 데이터가 같은 페이지에 저장되기 위해서는 먼저 해당 블록이 삭제되어야 한다.

NAND 플래시 메모리를 저장 매체로 사용하기 위해서는 플래시 변환 계층(Flash Translation Layer)라는 소프트웨어 계층을 사용한다. 플래시 변환 계층은 디스크 기반으로 설계된 파일 시스템이나 DBMS가 별도의 수정 없이NAND 플래시를 사용할 수 있도록 도와준다[3].

2.2 B-tree

B-트리는 대용량의 데이터를 효율적으로 관리하는데 널리 사용되는 색인 구조 중 하나로 (그림 1)과 같다.

B-트리에서 한 노드 당 최대 d 개의 키 값(K_1, K_2, …, K_d)을 가질 수 있고 d+1 개의 포인터(P_1, P_2, …,P_{d+1})를 가질 수 있으며 이 키 값들은 모두 정렬되어 있다.

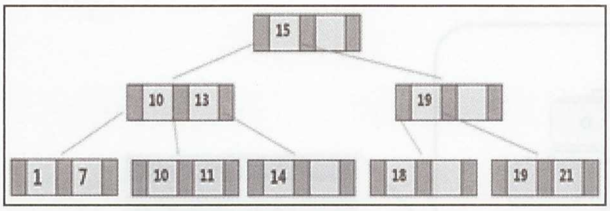
B-트리는 삽입과 삭제가 발생할 때마다 항상 모든 노드들이 골고루 분포되도록 균형을 유지하고 있다. 따라서 B-트리에서는 모든 단말노드에 대한 접근에서 균일한 깊이를 유지하고 있다.

B-트리에서 K 라는 키 값을 가지고 있는 레코드를 찾으려고 할 때, 루트노드부터 단말노드까지의 경로에 있는 여러 노드를 접근한다.

K 값을 가진 키를 삽입하려고 할 때, 루트노드부터 단말노드까지 키 값을 비교하여 삽입할 위치를 찾는다. 그 다음에 그 위치에 키를 삽입한다. 이 때 삽입할 노드가 키 값들로 꽉 찬 상태라면 분할(split)이 발생한다. 일반적으로 d+1 개의 키들의 경우, 앞에 있는 개의 키들은 현재 노드에 그대로 두고 나머지 키들은 새 노드에 저장한다. 분할 후에는 그 노드의 부모노드에 새 노드의 가장 작은 키 값을 삽입한다.

K값을 가지 키를 삭제하려고 할 때, 마찬가지로 해당 키 값을 가지고 있는 키를 찾은 후에 그 키를 가지고 있는 노드에서 그 키를 제거한다. 이 때 노드의 키의 개수가 d/2 보다 작다면 노드의 균형을 맞추기 위해 재분배(redistribution) 및 연결(concatenation)이 일어난다[3].

d 개의 키를 가지고 있고 전체 레코드 수가 n 인 B-트리의 경우 검색, 삽입 또는 삭제에 드는 비용이 logd/2n에 비례한다. 따라서 d 가 클수록, 즉 노드 의 크기가 클수록 비용이 줄어들게 된다.

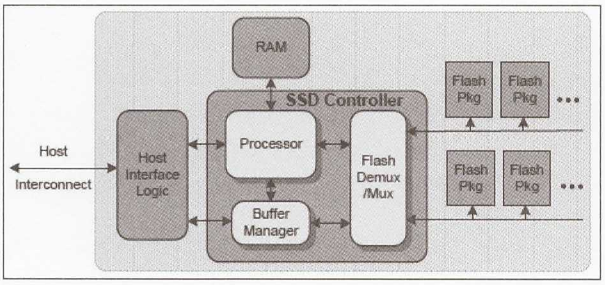


(그림 1) B-트리 구조

2.3 SSD의 속성

(그림 2)는 SSD의 논리 구조를 나타내고 있다. NAND 플래시 메모리 패키지는 각각 제한된 대역폭(약 40MB/sec)을 갖고 있는데, SSD는 배열 형태로 연결된 여러 플래시 메모리 칩들에 연속된 논리 페이지를 구성하도록 하고 있으며 이들에 대해 병렬 접근(parallel access)을 하여 더 넓은 대역폭을 갖도록 한 것이다[1]. 시리얼 I/O 버스는 플래시 메모리 패키지를 SSD 컨트롤러에 연결시켜 준다. SSD 컨트롤러는 프로세서로부터 요청을 받으면 연결된 인터페이스를 통하여 플래시 패키지로부터 데이터를 읽어 오거나 플래시 패키지에 데이터를 쓰게 된다.

SSD에서 데이터를 읽어오는 과정은, 먼저 플래시 메모리의 페이지로부터 데이터를 읽어와서 그 플레인(plane)의 레지스터에 저장하고 그 후 시리얼 버스를 통하여 데이터가 컨트롤러로 이동한다. 데이터를 쓰는 과정은 읽어오는 과정과 반대로 진행된다. 이런 방식으로SSD는 여러 플래시 메모리 칩들이 배열형태로 구조를 이루고 있으면서 SSD 컨트롤러로 하여금 병렬 접근 방식을 통하여 한 개 이상의 연속된 논리 페이지를 동시에 각각 접근할 수 있다[4].



(그림 2) SSD 논리 구조[4]

<표 1>은 Linux OS, 8 Core CPU, 16GB RAM, IOMeter 워크로드 환경에서 자체 실험을 통하여 얻은 결과로SSD의 일반적인 성능을 나타내고 있다. 표에서 SR(Sequential Read)는 연속된 페이지를 읽는 경우를, SW(Sequential Write)는 연속된 페이지를 쓰는 경우, RR(Random Read)는 연속되지 않은 페이지를 읽는 경우, RW(Random Write)는 연속되지 않은 페이지를 쓰는 경우를 나타낸다. <표 2>에서 I/O단위가 8KB인 경우를 보면 SW가 188Mbps이고 RW가 22Mbps로 대략 9 배 빠른 성능을 보이고 있음을 알 수 있다.

〈표 1〉 SSD 읽기/쓰기 성능(Linux OS, Intel 8 Core CPU, 16GB RAM, IOMeter 워크로드, SSD 사용)

(단위: Mbps)

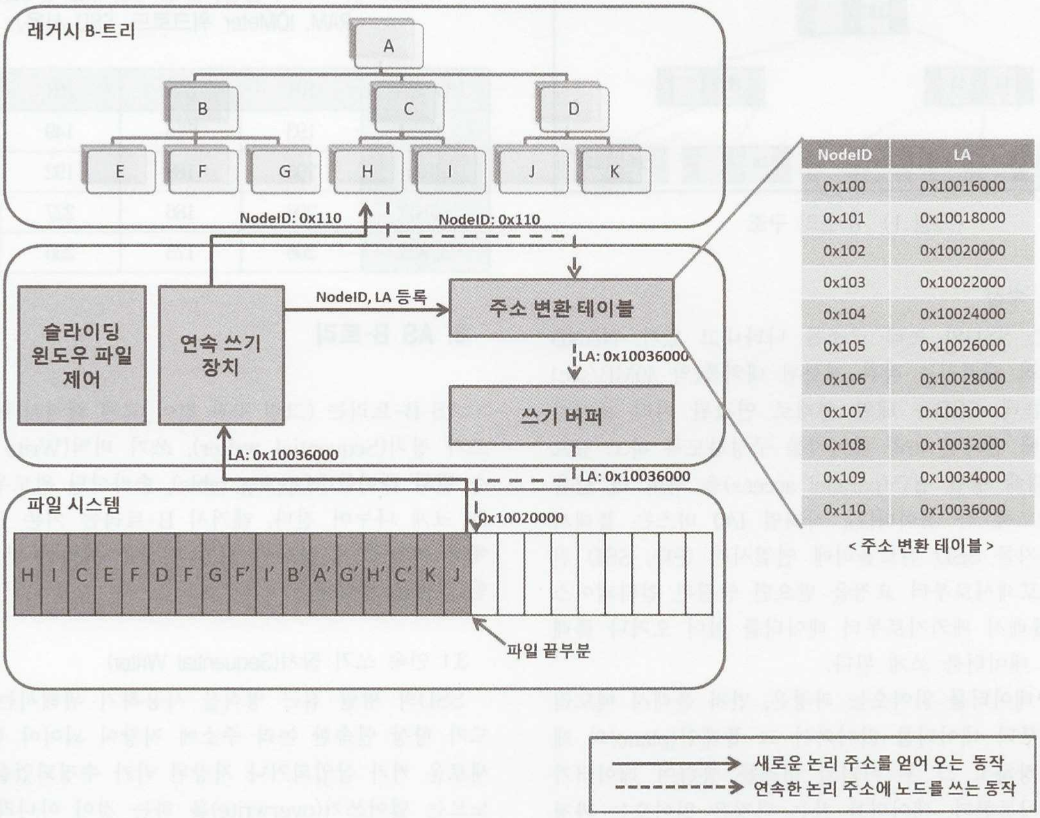
I/O 단위	SR	SW	RR	RW
4KB	193	158	149	21
8KB	206	188	192	22
16KB	206	186	227	32
32KB	206	175	256	40

3. AS B-트리

AS B-트리는 (그림 3)과 같이 크게 레거시 B-트리, 연속 쓰기 장치(Sequential writer), 쓰기 버퍼(Write Buffer), 주소 변환 테이블(Mapping table), 슬라이딩 윈도우 파일 제어로 크게 나누어 진다. 레거시 B-트리는 기존 B-트리 구조에서 파일 쓰기 연산과 읽기 연산을 제외한 부분으로 자세한 설명은 생략한다.

3.1 연속 쓰기 장치(Sequential Writer)

SSD의 병렬 접근 방식을 사용하기 위해서는 수정된 노드가 항상 연속한 논리 주소에 저장이 되어야 한다. 따라서 새로운 키가 삽입되거나 저장된 키가 수정되었을 때 수정된 노드는 덮어쓰기(overwrite)를 하는 것이 아니라 항상 파일의 끝부분에 저장을 한다. 이렇게 함으로써 수정된 노드들은 파일의 끝부분에 노드의 크기만큼 이동하면서 연속한 논리 주소에 저장이 된다. 자세한 동작 방식은 아래 (알고리즘 1)과 같다. 해당 노드가 수정이 되었을 때 수정된 노드는 원래 저장이 되었던 논리 주소에 다시 저장되는 것이 아니라 새로운 논리 주소를 얻어 온다(라인 12). 즉 파일의 끝부분의 주소를 얻어 온다. AS B-트리에서는 각 노드에 대해 각 노드가 저장된 논리 주소를 가지고 접근하는 것이 아니라 그 노드의 고유한 노드 식별 번호를 가지고 접근을 한다. 각 노드의 노드 식별 번호를 NodeID(예, 0x00, 0x01, 0x02, …)라고 하고 실제로 파일 시스템에 접근하기 위해 사용하는 주소를 LA(Logical Address)(예, 0x10000000, 0x10016000, 0x10036000, …)라고 할 때, 이 둘을 서로 연결시켜주는 테이블이 필요한데 이를 주소 변환 테이블(Mapping Table)이라고 한다. 따라서 레거시 B-트리에서는 모든 노드를 가리키는 주소로 NodeID를 사용하고 실제로 파일을 쓰거나 읽을 때에는 주소 변환 테이블을 사용하여 해당 NodeID와 연결된LA를 사용한다(라인 13,14). 이렇게 함으로써 수정된 노드는 레거시 B-트리에서 사용하는 NodeID를 동일한 값으로 유지한 채 항상 연속된 논리 주소 공간(LA)에 저장이 되어 SSD의 병렬 접근 방식을 활용할 수 있게 된다. 아울러 NodeID, LA 그리고 주소 변환 테이블을 사용하여 수정된 노드를 저장할 때 항상 새로운 주소로 LA를 변경함에도 불구하고 NodeID를 일정하게 유지하므로 그 노드의 NodeID를 가지고 있는 부모노드는 전혀 수정이 되지 않는다.



(그림 3) AS B-트리 구조 및 동작 방법

```
Procedure: Insert(K, NodeID)
Input: inserted key K, NodeID
//NodeID: node identifier used for B-tree;
// LA: logical address used for file system;
// MapTbl[]: table that maps NodeID to LA;

LA ← MapTbl[NodeID];
Node ← read the node from LA of the disk;

InsertKey( K , Node);

LA ← get the end position of the file; // allocate a new logical address.
MapTbl[NodeID] ← LA;
write Node to LA of the disk;

Function InsertKey(K, Node)
    //Node.KEYS: the array of the inserted keys in Node
    i ← search a location that should be inserted in Node.KEYS;
Node.KEYS[i] ← K;

```

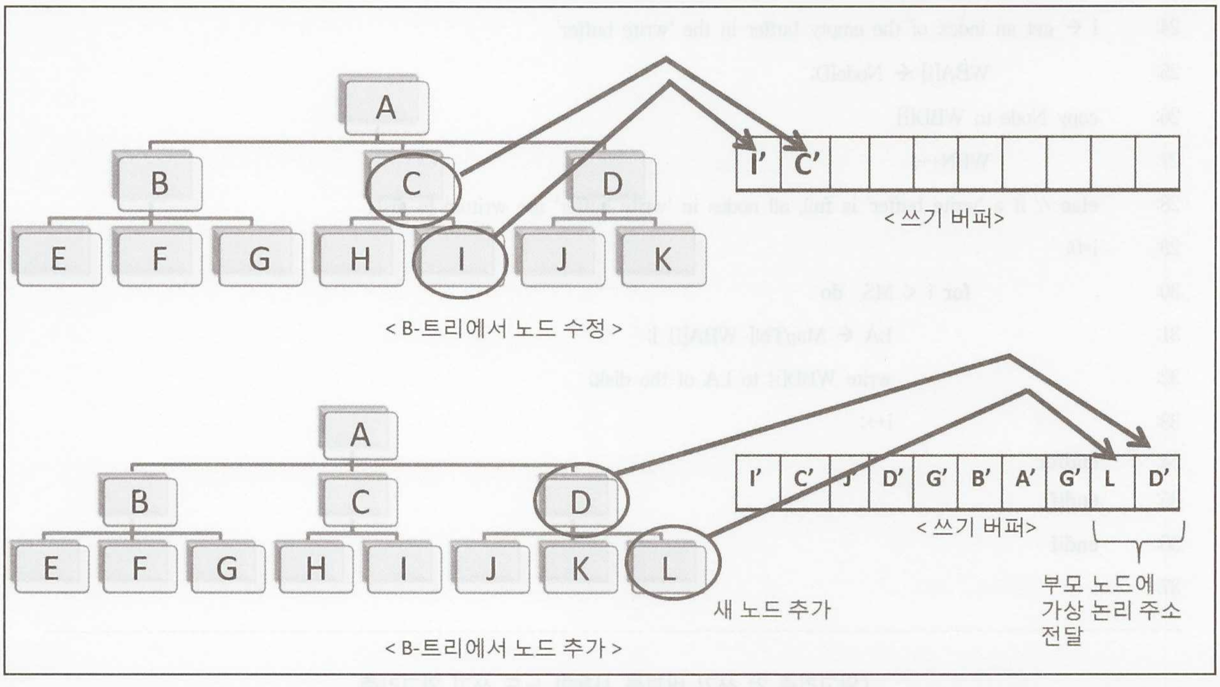
(알고리즘 1) 연속한 논리 주소 할당 알고리즘

3.2 쓰기 버퍼(Write Buffer)

B-트리 구조에서 삽입이 일어날 때 삽입할 노드를 찾기 위해 그 B-트리의 높이(height)만큼 노드를 읽어야 한다. 따라서 하나의 키가 삽입될 경우 높이가 3인 B-트리에서는 '루트노드 읽기 → 중간노드 읽기 → 단말노드 읽기 → 단말노드 쓰기'의 연산이 이루어 진다. 즉 쓰기 연산들 사이에 세 번의 읽기 연산이 수행된다. 데이터베이스가 버퍼 캐시를 사용할 때, 쓰거나 읽어 들일 노드가 버퍼에 있는 경우(hit)에는 실제로 디스크에 대해 읽기 및 쓰기 연산이 수행되지 않는다. 하지만 쓰거나 읽어 들일 노드가 버퍼에 없는 경우(miss)에는 캐시 교체 정책에 의해 읽기 또는 쓰기 연산을 요청한 노드를 디스크로부터 읽어서 캐시 버퍼에 저장한다. 이처럼 새로운 노드를 캐시 버퍼에 저장하기 위해서는 캐시 버퍼에서 교체할 한 노드를 먼저 디스크에 써야 한다. 따라서 버퍼 캐시를 사용하는 경우에도 마찬가지로 쓰기와 읽기 연산들이 교차되어 서로 간섭이 발생한다. SSD 상에서 실험을 해 본 결과 쓰기 연산과 읽기 연산이 반복될 경우 SSD의 성능을 악화시키는 것을 확인할 수 있었다. 관련 논문에서도 SSD에서의 이러한 문제점을 지적하고 있다 [5]. 따라서 수정된 노드를 항상 연속한 논리 주소에 저장을 하더라도 그 사이에 읽기 연산이 반복된다면 성능을 충분이 향상시킬 수가 없다. 이러한 문제점을 해결하기 위해 쓰기 버퍼를 추가하였다. (그림 4)과 같이 수정한 노드를 바로 파일에 저장하는 것이 아니라 메모리(RAM)에 있는 쓰기 버퍼에 임시로 저장하였다가 쓰기 버퍼가 꽉 차게 되면 저장된 여러 노드들을 한번에 각각 가지고 있는 LA에 저장을 하는 방식이다. 이렇게 함으로써 각각의 수정된 노드들은 읽기 연산의 방해 없이 연속한 논리 주소에 저장이 되어 성능 향상을 얻을 수 있다. 파일 시스템이 BUFFERED_IO를 사용하는 경우라면 쓰기 버퍼를 통한 성능 개선을 정확히 확인할 수가 없기 때문에 본 논문의 실험에서는 모두 DIRECT_IO를 사용하였다.

(알고리즘 2)는 수정된 노드를 새로 할당 받은 LA와 함께 쓰기 버퍼에 저장하는 방식과 구조에 대해 설명하고 있다. 쓰기 버퍼는 수정된 노드를 버퍼 크기만큼 저장하고 있는데 저장되는 노드는 WBA(Write Buffer Address)와 WBD(Write Buffer Data)로 이루어진 구조체 형식으로 저장된다. WBA는 저장할 노드의 NodeID를 의미하고 WBD는 저장할 노드를 의미한다. 쓰기 연산을 요청한 노드가 쓰기 버퍼에 이미 존재하는 경우는 메모리 상에서 WBD를 새로 수정한 노드로 변경하기만 하면 된다(라인 11~19). 따라서 이 경우 쓰기 버퍼는 캐시와 같은 역할을 하게 된다. 쓰기 버퍼에 존재하지 않는 경우라면 쓰기 버퍼에 저장할 공간을 할당 받고 WBA와 WBD의 형식으로 저장한다(라인 21~36). 만약 쓰기 버퍼가 버퍼 크기만큼 차게 되면 쓰기 버퍼에 있는 모든 노드들의 WBD를 각각의 WBA에 저장한다(라인 28~35).

(알고리즘 3)과 같이 읽기 연산의 경우도 마찬가지로 해당 노드가 먼저 쓰기 버퍼에 있는지 여부를 확인한 다음 해당 노드가 쓰기 버퍼에 있는 경우 바로 메모리에 있는 쓰기 버퍼에서 해당 노드를 읽어 들인다(라인 12~20). 쓰기 버퍼에 없는 경우는 주소 변환 테이블을 사용하여 NodeID를 LA로 변경하고 해당 LA로 파일 시스템에서 노드를 읽어 들인다(라인 22~26).



(그림 4) B-트리에서 수정된 노드를 쓰기 버퍼에 저장한다.

```
Procedure: WriteNode(NodeID, Node)
Input: NodeID, address of modified node Node

// MS = the maximum size of 'write buffer'
// WBN = the number of keys in the 'write buffer'
// WBA[i] = the NodeID of the node stored in the 'write buffer'
// WBD[i] = the data of the node stored in the 'write buffer'

bFound = FALSE;
i=0;
// check if there is a previous version of the modified node in 'write buffer'.
for i < WBN  do
     if(WBA[i] = NodeID )  then
          // the node in the 'write buffer' is replaced with Node.
               copy Node to WBD[i];
bFound = TRUE;
endif
i++;
endfor

if(bFound = FALSE)  then
     if (WBN < MS) then
// if a 'write buffer' is not full, the modified node is written to 'write buffer'.
i ← get an index of the empty buffer in the 'write buffer'
               WBA[i] ← NodeID;
copy Node to WBD[i]
               WBN++;
else // if a 'write buffer' is full, all nodes in 'write buffer' are written to disk.
i=0;
                    for i < MS  do
                              LA ← MapTbl[ WBA[i] ];
                              write WBD[i] to LA of the disk;
                              i++;
endfor
endif
endif
```

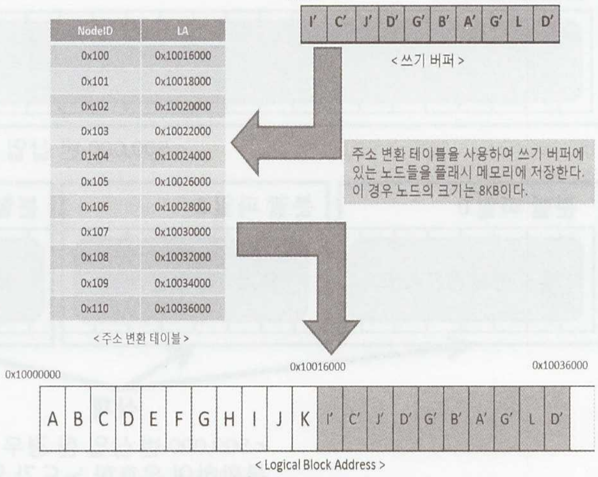
(알고리즘 2) 쓰기 버퍼를 사용한 노드 쓰기 알고리즘

```
Procedure: ReadNode(NodeID)
Input: NodeID
Output: address of selected node Node

// MS = the maximum size of 'write buffer'
// WBN = the number of keys in the 'write buffer'
// WBA[i] = the NodeID of the node stored in the 'write buffer'
// WBD[i] = the data of the node stored in the 'write buffer'

bFound = FALSE;
i=0;
// check if there is a node to be read in 'write buffer'.
for    i < WBN  do
    if(WBA[i] = NodeID)  then
    // the node in the 'write buffer' is read.
        copy WBD[i] to Node;
bFound = TRUE;
endif
i++;
endfor

// if the node  is not found in 'write buffer', the node is read from the disk.
if(bFound = FALSE)  then
    LA ← MapTbl[NodeID];
        Node ← read the node from LA of the disk;
endif
return (Node);
```

(알고리즘 3) 쓰기 버퍼를 사용한 노드 읽기 알고리즘

3.3 주소 변환 테이블(Mapping Table)

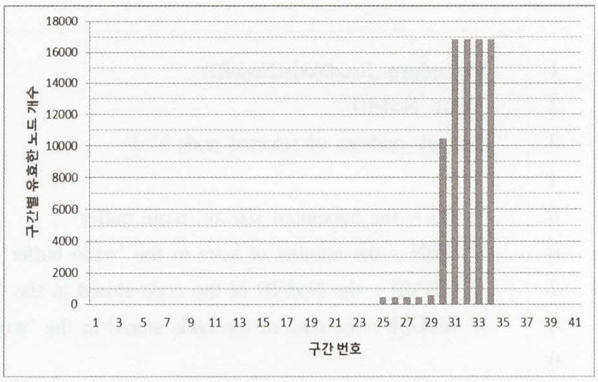
항상 연속된 논리 주소에 수정된 노드를 저장하기 위해서 NodeID를 생성하고 이를 레거시 B-트리에서 사용하는 LA와 연결하기 위해서 (그림 5)와 같이 주소 변환 테이블을 사용한다. (그림 5)에서와 같이 쓰기 버퍼가 꽉 차게 되면 쓰기 버퍼에 있는 노드들을 각각의 NodeID에 한번에 저장하는데 이 모든 노드들이 연속한 논리 주소로 되어 있기 때문에 쓰기 버퍼에 저장된 노드의 수만큼 연속한 논리 주소에 저장하게 된다. 주소 변환 테이블은 B-트리를 디스크로부터 RAM으로 읽어 들일 때 디스크에 저장된 전체노드를 읽어서 새로 구성한다. 순서대로 노드를 읽으면서 각 노드에 저장된 NodeID와 해당 노드가 저장된 물리 주소를 주소 변환 테이블에 등록한다. 노드를 읽어 들이는 방향은 파일의 앞부분부터 시작해서 끝부분으로 진행하여 동일한 NodeID에 대해서 이전에 저장된 물리 주소를 최근에 저장한 물리 주소로 변경한다.



(그림 5) 주소 변환 테이블을 사용하여 NodeID를 LA로 변경한다.

3.4 슬라이딩 윈도우 파일 제어(Sliding Window File Control)

AS B-트리의 경우 수정된 노드를 기존에 저장된 논리 주소에 덮어쓰기 연산을 하지 않고 항상 파일의 끝부분에 저장하므로 노드가 수정될 때마다 파일 사이즈가 지속적으로 증가하는 문제가 있다. 하지만 (그림 6)과 같이 대부분의 노드들은 파일의 끝부분으로부터 1/6 지점 내에 저장이 되어 있다. 따라서 파일에서 이 부분을 제외한 영역에는 유효한 노드들이 거의 존재하지 않는다. 이는 실험을 통해서 확인을 하였다. 이 부분은 실제적으로 유효하지 않으면서도 많은 논리 주소 공간을 차지하고 있어서 정리를 할 필요가 있다. 따라서 (그림 7)과 같이 파일을 일정한 크기로 분할해서 저장을 하고 각 파일마다 유효한 노드 수가 많지 않는 경우는 그 파일의 유효한 노드를 가장 마지막으로 생성된 파일에 저장하고 그 파일을 삭제하도록 하였다. 여기서는 한 파일의 크기를 128MB로 고정하였다. 하지만 플래시 메모리 기반 저장장치에 있어서는 덮어쓰기 연산도 실제 플래시 메모리의 같은 물리 주소에 저장되는 것이 아니라 플래시 변환 계층에 의해 임의의 다른 빈 공간에 저장되므로, B-트리의 삽입 연산에 의해서 초래되는 플래시 메모리 공간의 소모는 AS B-트리와 큰 차이가 나지 않는다.

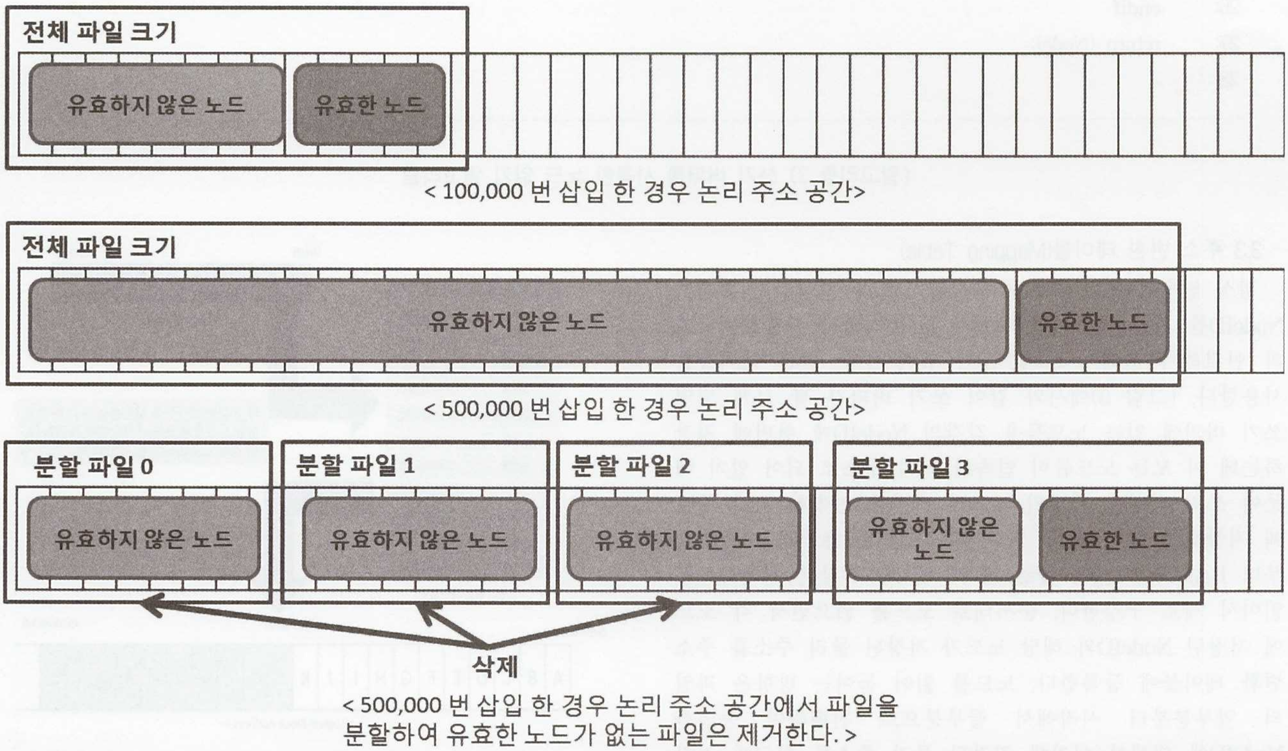


(그림 6) AS B-트리에서 구간별 유효한 노드의 수
(구간 크기: 128MB)

4. 성능 분석

AS B-트리의 성능 실험은 일반적인 B-트리와 BFTL과 함께 400,000 번 키를 삽입한 후 다시 100,000 번 키를 삽입하는 실험을 진행하였다. 성능 측정은 이후 삽입한 100,000 개의 키에 대한 삽입 시간과 검색 시간 측정하여 그 결과를 비교하였다. AS B-트리를 실험할 때 두 개의 변수를 변경하면서 진행하였는데, 한 노드에 저장할 수 있는 키의 수(F)와 쓰기 버퍼에 저장할 수 있는 노드의 개수(버퍼 크기)가 그 것들이다. F는 64, 128 그리고 256으로 이 세 가지의 경우로 나누어서 실험을 진행하였고 버퍼 크기는 1,8, 16, 32로 네 가지 경우로 나누어서 실험을 진행하였다.

BFTL의 실험 환경은 다음과 같다. 예약 버퍼(Reservation buffer)의 크기, 즉 예약 버퍼가 수용할 수 있는 최대 인덱스 유닛(index unit)의 수,는 F와 동일하게 유지하고 NTT(Node-Translation Table)에 있는 각 노드들의



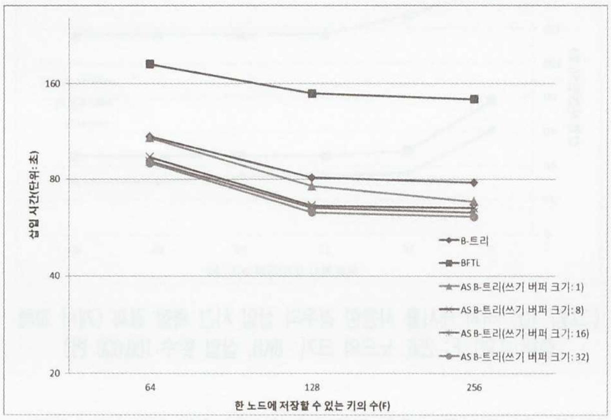
(그림 7) 슬라이딩 윈도우 파일 제어 방법

리스트 길이(C)는 BFTL 논문의 실험에서 사용되었던 값인 3으로 설정하여 더 길어질 경우 컴팩션(compaction)을 하도록 하였다[6].

<표 2>와 <표 3>은 B-트리, BFTL, AS B-트리의 F 값을 각각 64, 126, 256로 변경하여 실험한 삽입 및 검색 시 측정 시간이다. 그리고 (그림 8)과 (그림 9)은 <표 2>와 <표 3>의 값을 그래프로 나타낸 것이다.

〈표 2〉 삽입 시간 측정 결과
(노드의 크기: 8KB, 삽입 횟수: 100,000 번) (단위: 초)

	F: 64	F:128	F: 256
B-트리	109	81	78
BFTL	184	149	143
AS B-트리(쓰기 버퍼 크기:1)	108	76	68
AS B-트리(쓰기 버퍼 크기:8)	94	66	65
AS B-트리(쓰기 버퍼 크기:16)	91	65	63
AS B-트리(쓰기 버퍼 크기:32)	90	63	61



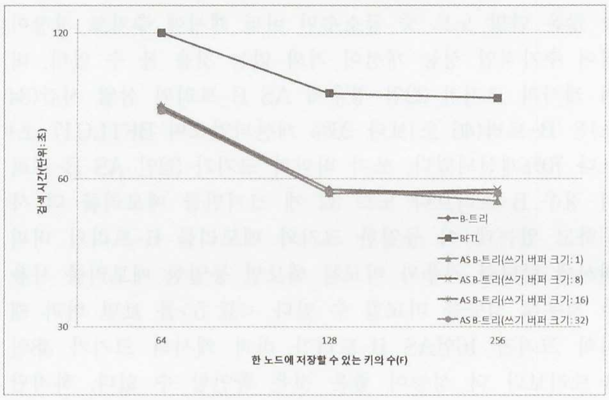
(그림 8) 삽입 시간 측정 결과를 로그 눈금으로 비교
(노드의 크기: 8KB, 삽입 횟수: 100,000 번)

삽입 시간은 AS B-트리가 가장 뛰어나고 AS B-트리 중에서도 F가 클수록 그리고 쓰기 버퍼의 크기가 클수록 더욱 성능이 뛰어난 것을 확인할 수 있었다. 첫 번째 경우는 F가 클수록 B-트리의 높이가 작아져 한 번 삽입 시 발생하는 읽기 연산의 수가 줄어들기 때문이다. 그리고 두 번째 경우는 쓰기 버퍼가 클수록 한 번에 더 큰 단위로 쓰기 연산을 요청할 수 있어서 SSD의 병렬 접근 방식으로 인한 성능 개선을 더욱 얻을 수 있었기 때문이다. 뿐만 아니라 읽기 연산과 쓰기 연산이 교차될 때 쓰기 연산만 수행될 때보다 쓰기 연산의 성능이 저하되는 SSD의 특성으로 인하여[1], 쓰기 버퍼에 저장된 노드들이 연속한 논리 주소에 저장될 때 그 사이에 읽기 연산이 발생하지 않아 성능이 개선된 부분도 있다. 이 실험 결과로 AS B-트리가 SSD상의 B-트리 구조에서 뛰어난 삽입 성능을 나타내는 것을 확인할 수 있었다. 본 실험에서는 F가 256인 경우, 버퍼의 크기가 32인 B-트리보다 21% 개선이 되었고 BFTL보다는 57%가 개선되었다. 실험 결과에 의하면 BFTL이 B-트리보다 성능이 더 나쁘게 나왔다. NTT의 길이(C)가 3인 경우, 쓰기 연산을 수행할 때 바로 파일에 접근하지 않고 수정한 정보를 RAM에 저장하므로 성능이 향상되지만 읽기 연산을 수행할 때에는 하나의 노드를 읽기 위해서 1~3 번(평균 1.5 번) 파일을 읽어야 한다. B-트리의 경우 높이가 3인 경우 루트 노드부터 단말 노드까지 파일로부터 단계적으로 읽어야 하는데 각 노드를 읽을 때마다 B-트리보다 평균적으로 1.5 배 이상 많이 노드를 읽어야 하기 때문에 성능이 더 나쁘게 나온 것으로 분석된다.

검색 시간의 경우에는 AS B-트리와 B-트리가 거의 비슷한 결과를 보이고 BFTL은 이들보다 더 소요된 것을 확인할 수 있었다. 이는 읽기 연산 요청 횟수가 AS B-트리와 B-트리의 경우는 거의 동일한 값을 가지고 있는 반면 BFTL은 이들보다 37% 많기 때문인 것으로 분석된다. BFTL의 읽기 연산 요청 횟수가 더 많은 이유는 한 노드를 구성하기 위해 NTT에 각 노드별로 등록되어 있는 1~3 개의 섹터를 읽어야 하기 때문이다[6]. 실험 결과 한 노드를 읽기 위해서 평균 1.49 개의 섹터를 읽은 것으로 확인되었다.

〈표 3〉 검색 시간 측정 결과
(노드의 크기: 8KB, 삽입 횟수: 100,000 번) (단위: 초)

	F: 64	F:128	F: 256
B-트리	84	57	56
BFTL	120	90	88
AS B-트리(쓰기 버퍼 크기:1)	85	57	54
AS B-트리(쓰기 버퍼 크기:8)	84	56	57
AS B-트리(쓰기 버퍼 크기:16)	83	56	56
AS B-트리(쓰기 버퍼 크기:32)	83	56	56



(그림 9) 검색 시간 측정 결과를 로그 눈금으로 비교
(노드의 크기: 8KB, 삽입 횟수:100,000 번)

<표 4>는 F가 256인 경우, 버퍼 크기가 32인 AS B-트리와 B-트리의 삽입 시간을 읽기 연산 수행 시간과 쓰기 연산 수행 시간으로 구분하여 측정한 것이다. AS B-트리의 삽입 시 쓰기 연산 수행 시간은 B-트리에 비해 50% 개선이 되었는데 이것은 쓰기 버퍼에 있는 노드를 메모리 상에서 수정하

는 데서 온 이득 외에 쓰기 버퍼에 있는 연속한 논리 주소를 갖는 노드들을 한번에 쓰기 요청을 해서 병렬 접근 방식에 의해 이득을 본 것으로 분석된다. 실험 결과 읽기 연산의 경우 해당 노드가 쓰기 버퍼에 존재하는 확률(hit ratio)은 1.1%로 나왔고 쓰기 연산의 경우는 0.6%로 나와 실제로 캐시로써의 쓰기 버퍼 효과는 크지 않은 것으로 확인되었다. 하지만 <표 4>에서와 같이 삽입 시 읽기 연산의 시간도 17% 개선이 된 것을 확인할 수 있는데, 이는 항상 수정한 노드를 저장할 곳을 찾는B-트리 구조와 다르게 수정된 노드를 항상 파일의 끝 부분에 쓰게 함으로써 쓰기 연산을 수행할 파일 위치를 찾는 연산이 줄어들었기 때문이다.

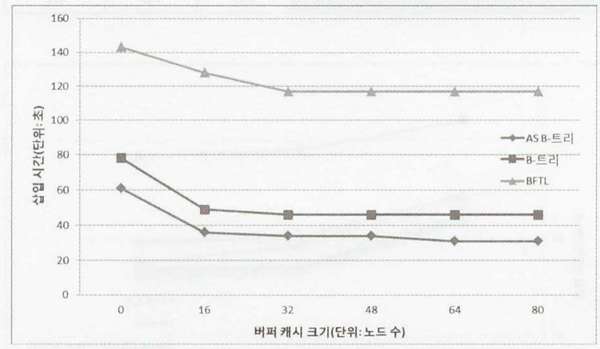
〈표 4〉 F:256 경우 AS B-트리(버퍼 크기: 32)와 B-트리의 삽입 시간 비교 (단위: 초)

	시간		
	읽기 연산	쓰기 연산	합계
AS B-트리	56	5	61
B-트리	68	10	78
개선 효과	17%	50%	21%

데이터베이스의 버퍼 캐시와 유사하게 동작하는 버퍼 캐시를 구현하여 삽입 및 검색 연산에 대해서 비교 실험을 하였다. 버퍼 캐시의 교체 정책은 LRU(Least Recently Used)를 사용하였으며 버퍼 캐시의 크기는 버퍼 캐시가 수용 가능한 노드의 수로 표시하였다. 삽입 시간을 측정한 결과 AS B-트리, B-트리 그리고 BFTL 세 경우 모두 버퍼 캐시의 크기에 따라 동일한 비율(캐시의 적중률(hit ratio)가 동일함)로 개선이 된 것을 확인할 수 있다. 루트 노드와 중간 노드를 모두 포함할 수 있는 정도인, 버퍼 캐시의 크기가 32일 때까지는 세 경우 모두 성능이 꾸준히 증가하지만 버퍼 캐시의 크기가 더 커질 경우(버퍼 캐시의 크기: 48 ~ 80)에는 많은 단말 노드 중 극소수만 버퍼 캐시에 추가로 저장이 되어 추가적인 성능 개선이 거의 없는 것을 볼 수 있다. 버퍼 캐시의 크기가 32인 경우에 AS B-트리의 삽입 시간(34 초)은 B-트리(46 초)보다 26% 개선되었으며 BFTL(117 초)보다 70%개선되었다. 쓰기 버퍼의 크기가 32인 AS B-트리의 경우 B-트리보다 노드 32 개 크기만큼 메모리를 더 사용하고 있는데 이 동일한 크기의 메모리를 B-트리의 버퍼 캐시에 할당한 경우와 비교를 해보면 동일한 메모리를 사용한 경우에 성능을 비교할 수 있다. <표 5>를 보면 버퍼 캐시의 크기가 16인AS B-트리가 버퍼 캐시의 크기가 48인 B-트리보다 더 성능이 좋은 것을 확인할 수 있다. 하지만 버퍼 캐시의 크기가 0인 AS B-트리는 버퍼 캐시의 크기가 32인 B-트리보다 삽입 시간이 32% 더 소요된 것을 볼 수 있다. 이는 버퍼 캐시가 루트 노드와 중간 노드를 모두 포함할 때까지는 버퍼 캐시가 동일한 메모리를 사용한 AS B-트리보다 더 효율적인 것을 알 수 있다. 하지만 동일한 성능을 유지하기 위해서는 B-트리의 크기가 커질수록 더 큰 버퍼 캐시가 필요하다. AS B-트리의 쓰기 버퍼는 전체 B-트리의 크기와 상관없이 쓰기 버퍼 크기만큼 일정한 성능을 향상시킬 수 있다. 따라서 동일한 크기(노드 48 개의 크기)의 메모리를 사용하더라도 AS B-트리가 효율성이 더 좋은 것을 알 수 있다. <표 6>과 (그림 11)은 버퍼 캐시를 사용한 경우에 검색 시간을 측정한 결과로 삽입 시간 측정 결과와 비슷한 비율로 성능이 개선된 것을 확인할 수 있다. 버퍼 캐시의 크기가 32인 경우 AS B-트리의 검색 시간은 B-트리의 검색 시간과 같이 28초로 나왔으며 44초로 나온 BFTL보다는 36% 개선되었다.

〈표 5〉 버퍼 캐시를 사용한 경우의 삽입 시간 측정 결과 (캐시 교체 정책: LRU, F: 256, 노드의 크기: 8KB, 삽입 횟수:100,000 번) (단위: 초)

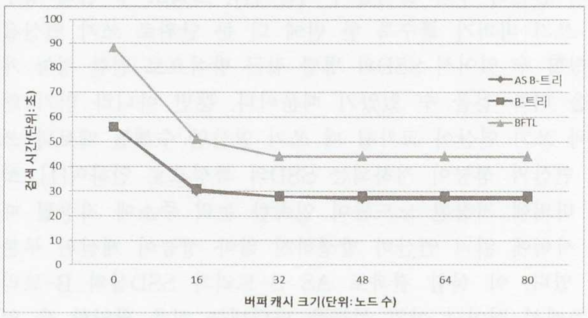
	버퍼 캐시 크기(단위: 노드 수)					
	0	16	32	48	64	80
AS B-트리	61	36	34	34	31	31
B-트리	78	49	46	46	46	46
BFTL	143	128	117	117	117	117



(그림 10) 버퍼 캐시를 사용한 경우의 삽입 시간 측정 결과 (캐시 교체 정책: LRU, F: 256, 노드의 크기: 8KB, 삽입 횟수:100,000 번)

〈표 6〉 버퍼 캐시를 사용한 경우의 검색 시간 측정 결과 (캐시 교체 정책: LRU, F: 256, 노드의 크기: 8KB, 삽입 횟수:100,000 번) (단위: 초)

	버퍼 캐시 크기(단위: 노드 수)					
	0	16	32	48	64	80
AS B-트리	56	30	28	27	27	27
B-트리	56	31	28	28	28	28
BFTL	88	51	44	44	44	44



(그림 11) 버퍼 캐시를 사용한 경우의 검색 시간 측정 결과 (캐시 교체 정책: LRU, F: 256, 노드의 크기: 8KB, 삽입 횟수:100,000 번)

사용한 파일의 크기는 <표 7>과 같이 B-트리가 가장 작았으며 BFTL이 가장 크게 나왔다. 플래시 메모리에서는 B-트리에서 주로 발생하는 덮어쓰기 연산도 플래시 변환 계층에 의해 실제로 다른 물리 주소 공간에 쓰기 연산이 이루어져서 물리 주소 공간에서의 사용량 차이는 크게 없다. 따라서 AS B-트리는 B-트리에 비해서 연속하지 않은 논리 주소 영역에의 쓰기 연산을 연속한 논리 주소 영역에 쓰기 연산을 하도록 위치를 바꿀 뿐 실제 물리 주소 공간에서 이루어지는 쓰기 연산의 수는 기존 B-트리와 크게 차이가 나지 않는다. AS B-트리의 경우 동일한 실험 조건에서 1,000,000 번 삽입을 시도했을 때도 파일 사이즈가 거의 비슷하게 나와 앞에서 언급한 슬라이딩 윈도우 파일 제어가 정상적으로 동작하면 일정한 파일 사이즈(600~700MB)를 유지하는 것을 확인할 수 있었다.

〈표 7〉 파일 크기 비교(F: 256) (단위: MB)

	B-트리	BFTL	AS B-트리
파일 크기	22.4	1615	642

5. 결 론

B+트리를 포함한 B-트리 인덱스의 경우 레코드 삽입 시 한 개 또는 그 이상의 노드 수정이 이루어지는데 이는 바로 파일시스템의 연속되지 않은 논리 주소 상에 덮어쓰기 연산를 수행하도록 요청하게 된다. 연속하지 않은 논리주소에 대한 덮어쓰기 요청은 SSD에서 병렬 접근 방식의 이점을 사용할 수 없도록 하여 상대적으로 높은 I/O 비용을 초래하게 된다.

이러한 기존 B-트리의 문제점을 해결하기 위해, 본 논문에서는 SSD의 병렬 접근 방식의 이점을 사용할 수 있도록, 수정된 노드에 대해 항상 연속된 논리 주소에 쓰기 연산을 수행할 수 있는 새로운 B-트리 구조인 AS B-트리를 제안하였다. 우선 기존 B-트리에서 사용하는 노드의 논리 주소를 노드 식별 번호로 대체하고 실제로 수정된 노드를 저장할 때는 항상 파일의 끝부분에 저장을 하여 수정된 노드들이 항상 연속한 논리 주소에 저장되도록 하였다. 그리고 저장한 논리 주소와 노드 식별 번호를 연결하는 주소 변환 테이블을 RAM에 저장하였다. SSD의 병렬 접근 방식의 이점을 최대한 활용하기 위해 쓰기 버퍼에 수정된 노드들을 저장하고 한번에 여러 노드들을 연속된 주소 공간에 저장하도록 하였다. 수정된 노드들에 대해 덮어쓰기 연산을 수행하는 것이 아니라 항상 파일의 끝부분에 저장하기 때문에 파일 사이즈가 계속 증가하는 문제가 있었는데, 슬라이딩 윈도우 파일 제어 방법을 사용하여 파일의 앞부분에 유효하지 않는 부분을 삭제하도록 하여 파일 사이즈가 계속 증가하는 문제를 해결하였다. 이렇게 함으로써 AS B-트리에서 삽입 시 B-트리보다 삽입 시간을 21% 단축하였다. 전체 삽입 시간 중 쓰기 연산에 소요된 시간은 B-트리보다 50% 만큼 개선이 되었는데 이러한 결과로부터 삽입 시간 개선은 대부분 쓰기 연산 시간을 단축시킴으로써 얻은 것이라고 할 수 있다. 이는 앞에서 언급한 연속 쓰기 장치와 쓰기 버퍼를 사용하여 수정한 노드들을 한번에 연속한 논리 주소 공간에 저장함으로써 SSD의 병렬 접근 방식의 이점을 최대한 활용한 결과라고 할 수 있다. 이와 같은 결과로 볼 때 AS B-트리는 비록 기존 B-트리 구조보다 사용하는 파일의 크기는 크지만 삽입 성능에서는 크게 개선이 되어 대용량의 데이터를 삽입하는 B-트리 구조에서 더욱 좋은 성능을 얻을 수 있음을 확인할 수 있었다.

참고문헌

[1] 남정현(Jung-hyun Nam), 박동주(Dong-joo Park), “플래시 메모리 상에서 B-트리 설계 및 구현”, 정보과학회논문지 : 데이타베이스 제34권 제2호, 2007. 4.

[2] Feng Chen, David A. Koufaty, and Xiaodong Zhang, “Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives”, Joint International Conference on Measurement and Modeling of Computer Systems, Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems, 2009.

[3] Dongwon Kang, Dawoon Jung, Jeong-Uk Kang, Jin-Soo Kim, Computer Science Division Korea Advanced Institute of Science and Technology (KAIST), “ μ-tree: an ordered index structure for NAND flash memory”, International Conference On Embedded Software, Proceedings of the 7th ACM & IEEE international conference on Embedded software, 2007.

[4] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, Rina Panigrahy Microsoft Research, Silicon Valley, University of Wisconsin-Madison, “Design tradeoffs for SSD performance”, USENIX 2008 Annual Technical Conference on Annual Technical Conference, 2008.

[5] Chen, Feng, Doctor of Philosophy, Ohio State University, Computer Science and Engineering, 2010, “ON PERFORMANCE OPTIMIZATION AND SYSTEM DESIGN OF FLASH MEMORY BASED SOLID STATE DRIVES IN THE STORAGE HIERARCHY”, 2010.

[6] CHIN-HSIEN WU, TEI-WEI KUO and Li Ping Chang, National Taiwan University, “An efficient B-tree layer implementation for flash-memory storage systems, ACM Transactions on Embedded Computing Systems (TECS), 2007. 7.

김 성 호

e-mail : runtodream@gmail.com
2003년 연세대학교 전기전자공학부(학사)
2010년~현 재 연세대학교 컴퓨터과학과 석사과정
관심분야 : 데이터베이스시스템, 플래쉬 메모리, SSD

노 홍 찬

e-mail : fallsmal@cs.yonsei.ac.kr
2006년 연세대학교 컴퓨터과학과(학사)
2008년 연세대학교 컴퓨터과학과(공학석사)
2008년~현 재 연세대학교 컴퓨터과학과 박사과정
관심분야 : 데이터베이스시스템, 플래쉬 메모리, SSD

이 대 욱

e-mail : daewook@gmail.com
1995년 경북대학교 컴퓨터공학과(공학사)
1997년 경북대학교 컴퓨터공학과(공학석사)
2009년 서울대학교 전기, 컴퓨터공학부(공학박사)
1997년~2002년 (주)퓨쳐인포넷 근무
2009년~2011년 연세대학교 컴퓨터과학과 BK21 박사후연구원
2011년~현 재 서강대학교 컴퓨터공학과 연구교수
관심분야 : 데이터베이스 시스템, XML, 웹 서비스, SSD

박 상 현

e-mail : sanghyun@cs.yonsei.ac.kr
1989년 서울대학교 컴퓨터공학과(학사)
1991년 서울대학교 컴퓨터공학과(공학석사)
2001년 UCLA 컴퓨터과학과(공학박사)
1991년~1996년 대우통신 연구원
2001년~2002년 IBM T. J. Watson Research Center Post-Doctoral Fellow
2002년~2003년 포항공과대학교 컴퓨터공학과 조교수
2003년~2006년 연세대학교 컴퓨터과학과 조교수
2006년~현 재 연세대학교 컴퓨터과학과 부교수
관심분야 : 데이터베이스, 데이터마이닝, 바이오인포매틱스, 적응적 저장장치 시스템, 플래쉬메모리 인덱스, SSD