http://dx.doi.org/10.5392/JKCA.2016.16.08.706

플래시 메모리 B-트리를 위한 저비용 노드 갱신 기법

An Efficient Flash Memory B-Tree Supporting Very Cheap Node Updates

임성채
동덕여자대학교 컴퓨터학과

Seong-Chae Lim(sclim@dongduk.ac.kr)

요약

B-트리는 공간 효율성과 빠른 키 검색 시간으로 인해 하드 디스크 기반 DBMS의 색인 기법으로 널리 쓰이고 있다. 하지만 B-트리를 플래시 메모리에 저장해 사용한다면, 높은 노드 갱신 비용으로 인해 DBMS 성능을 크게 저하시킬 수 있다. 이는 B-트리 단말노드에 발생하는 임의(random) 갱신 연산이 플래시 저장 장치의 과도한 가비지 수집 비용을 낳을 수 있기 때문이다. 논문에서는 이런 문제를 막기 위해 단말노드의 부모 계층 노드들을 물리적으로 저장하지 않고 가상(virtual) 노드로 둔다. 키 검색을 위해 가상 노드가 필요할 때는 자식 노드들을 참조하여 가상 노드를 동적으로 생성한 후 버퍼에 두고 사용한다. 제안된 플래시 B-트리 알고리즘은 노드 갱신과 트리 재구성 동작이 단일 플래시 블록 안에서 수행되기 때문에 가비지 수집 비용과 노드 갱신 비용을 낮게 할 수 있다. 또한 기존에 제안된 플래시 기반 B-트리와 비교하여 매우 빠른 키 검색 시간을 보장한다. 논문에서는 수학적 성능 모델을 통해 제안된 플래시 B-트리의 성능을 검증한다.

■ 중심어 : | B+-트리 | 플래시 메모리 | 색인기법 | 데이터베이스 | 저장장치 |

Abstract

Because of efficient space utilization and fast key search times, B-trees have been widely accepted for the use of indexes in HDD-based DBMSs. However, when the B-ree is stored in flash memory, its costly operations of node updates may impair the performance of a DBMS. This is because the random updates in B-tree's leaf nodes could tremendously enlarge I/O costs for the garbage collecting actions of flash storage. To solve the problem, we make all the parents of leaf nodes the virtual nodes, which are not stored physically. Rather than, those nodes are dynamically generated and buffered by referring to their child nodes, at their access times during key searching. By performing node updates and tree reconstruction within a single flash block, our proposed B-tree can reduce the I/O costs for garbage collection and update operations in flash. Moreover, our scheme provides the better performance of key searches, compared with earlier flash-based B-trees. Through a mathematical performance model, we verify the performance advantages of the proposed flash B-tree.

■ keyword : | B+-trees | Flash Memory | Indexing Schemes | Databases | Storage |

* 이 논문은 2013년도 동덕여자대학교 "연구년제도" 지원에 의하여 수행된 것임.
접수일자 : 2016년 06월 23일
수정일자 : 2016년 07월 13일
심사완료일 : 2016년 07월 13일
교신저자 : 임성채, e-mail : sclim@dongduk.ac.kr

I. 서 론

NAND 플래시 메모리 기반 저장장치는 빠른 읽기속도, 낮은 전력소모, 경량화, 높은 충격 내구성 등의 장점으로, 향후 하드디스크를 대체할 수 있을 것으로 예상된다[1][2][16]. 하지만 B-트리와 같이 임의(random) 갱신 빈도가 높을 수 있는 데이터를 저장할 경우, 과도한 가비지 수집(Garbage Collection: GC) 비용으로 인해 저장 시스템 성능이 크게 저하될 위험이 있다[2-5][8]. 플래시 메모리의 임의 갱신 연산이 높은 GC 비용을 초래하는 것은 이 매체의 기록전소거(*erase-before-write*) 특성에 기인한다. 플래시 메모리에 한번 기록된 페이지를 제자리 갱신하려면 해당 페이지를 미리 소거해야 하는데, 소거 연산은 블록 단위로만 수행되며 그 비용이 매우 크다[4][5][12][16][19].

이런 높은 비용의 제자리 갱신을 피하기 위해 FTL(Flash Translation Layer)이라는 펌웨어가 호스트와 저장장치 하드웨어 사이에 존재한다[4][5]. FTL은 호스트가 보는 플래시 메모리의 논리적 주소와 내부의 물리적 주소 사이를 사상(mapping)하는 역할을 통해, 페이지의 비제자리(out-of-place) 갱신을 지원한다. 하지만 이런 비제자리 갱신은 GC 동작을 필요로 하고, GC 수행 시에 full-merge 빈도가 높다면 높은 I/O 비용이 발생한다. 따라서 full-merge 빈도를 최소화 시켜야 하는데 B-트리와 같이 노드 갱신이 쉽게 임의 갱신 연산을 유발하는 경우는 full-merge 발생을 피할 수가 없다. 따라서 FTL과는 다른 별도의 플래시 B-트리 알고리즘이 필요하다[6][8-10][12][13][21-23].

본 연구에서는 새로운 플래시 B-트리를 고안하기 위해 두 가지 점에 주목한다. 첫째는 B-트리의 특성상 대부분의 노드 갱신은 단말노드와 바로 위의 부모노드에 집중된다는 것이다. 그 위쪽 트리 계층에 갱신이 발생하는 경우는 연쇄적 트리 재구성 동작이 수행될 때인데 이는 낮은 확률을 가진다. 두 번째는 플래시 메모리의 읽기 비용이 갱신 비용에 비해 매우 낮다는 점이다. 이점을 고려하여, 읽기 연산을 추가로 사용하더라도 노드 갱신 횟수를 줄일 수만 있다면 성능 향상에 도움을 줄 수 있다는 점이다[9][12][14]. 이는 기존 하드 디스크와는 매우 다른 특성이다. 이런 두 가지 특성을 최대한 활용할 수 있도록 플래시 B-트리를 고안한다.

제안하는 플래시 기반 B-트리는 단말노드의 부모 노드 계층, 즉 단말 상위 계층(ULL: Upper-Level of Leaves) 노드들은 물리적으로 저장하지 않는다. 이런 ULL 노드가 키 검색 과정에서 필요한 경우, 이 노드의 자식 노드들을 읽어 들여 동적으로 생성한다. 이때 단말노드 읽기 시간을 단축하기 위해 동일 부모를 가지는 단말노드들은 하나의 블록에 저장된다. 가상(virtual) ULL 노드 개념을 사용함으로 해서 부모노드 갱신 없이도 단말노드의 저장 위치를 변경할 수 있고, 단말노드 계층의 트리 재구성이 수행될 때 노드 갱신 횟수를 줄일 수 있다. ULL 노드를 생성한 후에는 메모리 버퍼에 적재하기 때문에 반복적으로 자식노드 블록을 읽을 필요는 없다. 또한 읽기 속도가 빠른 플래시 메모리의 특성상 ULL 노드 생성 비용이 크지 않다. 이로 인해 제안된 플래시 B-트리는 기존의 플래시 B-트리에 비해 우수한 키 검색 성능을 보장하면서도 낮은 트리 갱신 비용을 가진다.

본 논문의 구성은 다음과 같다. 2장에서는 기존 플래시 B-트리 알고리즘과 단말노드를 사용한 부모노드 생성 기법에 대해 알아본다. 3장에서는 제안하는 플래시 기반 B-트리 구조와 관련 알고리즘을 기술한다. 4장에서는 제안된 B-트리의 장점을 보이기 위해 기존 B-트리 알고리즘과의 성능 비교를 수행한다. 그리고 5장에서 결론을 맺는다.

II. 관련 연구

1. 플래시 기반 B-트리

FTL 기능을 확장하여 B-트리의 노드 갱신 비용을 낮추고자 하는 연구가 있다[13][17][18]. 이 방식은 B-트리 색인 파일이 다른 종류의 데이터와 같이 저장될 때는 효과를 보기 어렵고, 임의(random) 노드 갱신이 많이 발생하는 경우는 full-merge를 피할 수가 없다[4][5][8]. 이런 이유로 FTL과는 독립적으로 동작하는 플래시 B-트리에 대한 연구가 일반적이라 할 수 있다.

FTL을 사용하지 않는 기존 플래시 B-트리 연구에서는 노드의 제자리 갱신이 피하기 위해 로깅 기법을 채용한다[6][8-10][13][16][17][19][21]. 즉, 노드 갱신이 필요하다면, 발생한 갱신 연산을 표현하는 ***갱신 로그***(update log)를 빈 페이지에 저장한다. 그리고 갱신된 노드가 이후 재접근될 때, 기록된 갱신 로그를 해당 노드에 적용(즉, redo 연산)함으로써 최신 노드 상태를 동적으로 생성한다.

이런 로깅 기반 플래시 B-트리 알고리즘은 갱신 로그가 트리의 어떤 계층에 기록되는 지에 따라 나뉠 수 있다. 갱신 로그를 B-트리 상위 계층에 두고 사용하는 방식은 키 검색자가 단말노드까지 트리 탐색을 하지 않고도 검색 연산을 종료할 수 있고, 로그 데이터 관리가 쉽다는 장점을 가진다. 이 중 [19]에서는 트리를 상하부 두 계층으로 분리하여, 트리 상태 초기에는 상부 계층을 구성하는 페이지를 빈 페이지로 둔다. 그리고 이후 발생하는 갱신 로그를 비어 있는 상부 계층 페이지에 저장한다. 연속적인 갱신 발생으로 갱신 로그를 저장할 빈 페이지가 없어지면 일괄처리 방식으로 전체 트리를 재구성한다. 한편 다른 연구에서는 갱신 로그 저장 공간을 동적으로 관리한다[17][21]. 즉, 동적으로 공간을 늘이기 위해 여러 개의 빈 페이지를 연결하여 사용하거나, 버퍼 내의 메모리를 할당하는 방식을 취한다. 여기에 NOR 형 플래시 메모리를 갱신 로그 저장에 사용하여 시스템 고장에 대비하기도 한다[13].

이렇게 갱신 로그를 트리 상부에 두는 방식은 두 가지 큰 문제를 가진다. 갱신 로그와 단말노드가 떨어져 위치하기 때문에, 단말노드를 이동하면서 수행되는 B+-트리의 범위검색을 적용하기 불가능하다[12][15][17]. 이는 상용 DBMS가 일반적으로 제공하는 색인 기법이 지원될 수 없음을 의미한다. 다음으로는 동시성 제어를 지원하기 어렵다는 점이다. 잠금(lock) 커플링[7][20]에 기반한 동시성 제어 기법을 적용할 경우, 잠금을 걸 트리 영역이 너무 크거나 불확실해 지기 때문에 기존 기법을 적용할 수 없다.

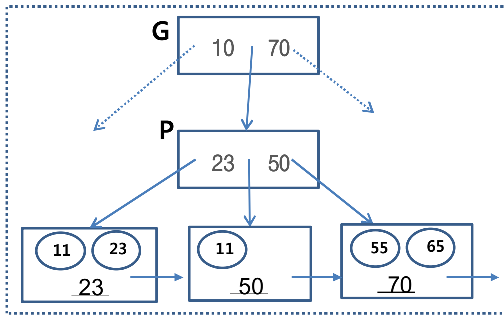
이런 이유로 갱신 로그를 단말노드 쪽에 위치시키는 방법이 실효성을 가진다고 할 수 있다. [9][10]에서는 데이터를 플래시 블록에 저장함에 있어, 각 블록의 뒤쪽에 갱신 로그를 위한 로그 영역을 미리 확보해 둔다. 그리고 해당 블록 안에 갱신이 발생한다면, 이를 표현하는 갱신 로그를 로그 영역에 순차적으로 기록한다. 이런 기법을 B-트리 저장에 적용한다면 임의의 노드 N을 버퍼로 읽어 들일 때, 노드 N이 저장된 블록의 로그 영역을 함께 읽어 들어야 한다. 그리고, 노드 N과 관련된 갱신 로그를 찾아 적용시켜야 한다. 이런 방식은 플래시 메모리의 읽기 속도가 매우 빠른 점을 이용한 방식이라 할 수 있다. [12]에서도 유사한 아이디어를 통해 효과적인 범위 검색을 지원하였다. 하지만 트리 재구성 동작을 수행하기 어려운 문제가 있다.

2. 단말노드를 이용한 부모노드 생성

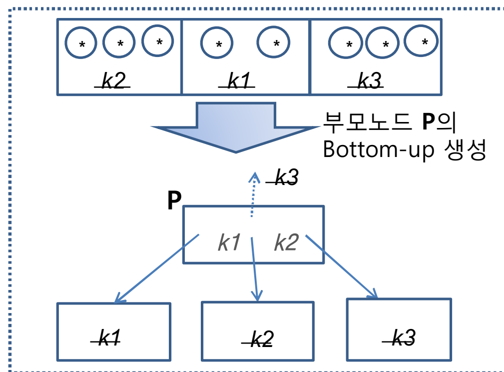
B-트리 단말노드의 색인 엔트리 만을 보고 B-트리를 bottom-up 방향으로 생성할 수 있고, 이를 B-트리 벌크(bulk) 로딩이라 한다[7][20]. 만약 단말노드가 B+-트리 단말 노드와 같이 키 범위를 표시하는 필드를 가지고 있다면, 간단한 방법을 통해 부모 계층의 노드를 생성할 수 있다. 이를 보이기 위해 [그림 1](a)의 예를 사용한다. 그림에서 단말노드 안에 밑줄 처진 숫자는 해당 노드의 *max-key* 필드 값을 나타낸다. 노드 N의 *max-key* 값이 k라고 한다면, N에는 k 보다 작거나 같은 값의 키 엔트리 만이 존재한다. 그림의 노드 P는 ULL 계층에 속한 노드이다.

[그림 1](a)에서 알 수 있듯이 단말노드에 존재하는 *max-key* 값과 부모노드 P의 색인 엔트리 키 값은 서로 연관성을 갖는다. 예를 들어, 노드 P에 있는 키 값 23의 왼쪽 화살표가 가리키는 단말노드가 가지는 *max-key* 필드 값 역시 23이다. 이런 점에 착안하여 bottom-up 방식으로 노드 P를 생성하는 예는 [그림 1](b)에서 보인다. 그림에는 세 개의 단말노드가 저장되어 있고 이들의 *max-key* 값들은 $k1 < k2 < k3$의 관계가 있다. 이 세 개 단말노드를 읽어 *max-key* 필드로 정렬한다면 [그림 1](b) 아래쪽과 같이 노드 P를 생성할 수 있다. [그림 1](b)의 방식은 기존 B-트리 벌크 적재에 사용된 아이디어와 다르지 않다.

제안하는 플래시 B-트리에서는 단말노드에 *max-key* 필드를 두고 이를 ULL 계층의 부모노드 생성에 이용한



(a) B+-트리 예.



(b) Bottom-up 노드 생성의 예.

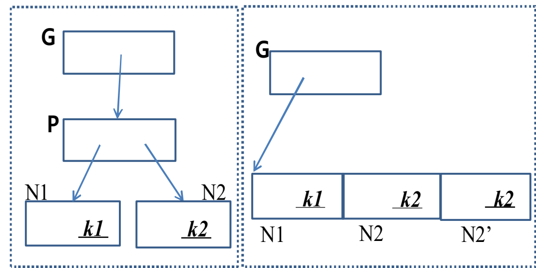
그림 1. B+-트리 단말노드를 이용한 부모 노드의 Bottom-up 생성 예.

다. 이런 방식을 취함으로써 키 검색 성능을 해치지 않고도 갱신 연산 비용을 크게 낮출 수 있다. 상세 내용은 이어지는 장에 기술된다.

III. 제안하는 플래시 B-트리 알고리즘

1. 제안하는 B-트리 구조

제안하는 플래시 B-트리는 기존 연구[6][9][10][12]와 같이 FTL 사용 없이 트리 갱신 작업을 직접 수행하며, 이는 단말노드와 단말노드의 부모노드 계층(ULL: Upper Layer of Leaves) 갱신에 적용된다. FTL 없이 직접 B-트리 노드를 갱신하는 경우 플래시 메모리의 특성으로 연쇄적 갱신이 발생할 수 있다. 예를 들어 [그림 2](a)에서와 같이 ULL 노드 *P*, 그리고 P의 두 자식 노드와 부모노드 G가 있다고 하자. 키 삽입으로 단말노드 N2를 갱신시켜야 한다면, 제자리 갱신이 되지 않기 때문에 다른 빈 페이지에 갱신된 N2를 저장해야 한다. 이에 따라 N2의 새로 바뀐 주소를 기록하기 위해 노드 P를 갱신해야 하고, 이는 다시 노드 G를 포함한 트리 상위 계층으로의 연쇄적 갱신을 유발한다. 이런 문제가 있기 때문에 노드의 위치 변경이 필요 없는 로깅 기반의 플래시 B-트리 알고리즘이 제안되었다[6][9][10][19][21].



(a) 일반적 B-트리 저장구조. (b) ULL 노드가 없는 B-트리 구조.

그림 2. 사용되는 B-트리 저장 방식의 예

제안하는 플래시 B-트리는 별도의 로그 데이터를 사용하는 대신 ULL 노드를 저장하지 않음으로써 문제가 되는 연쇄적 노드 갱신을 피한다. 이에 대한 아이디어는 [그림 2](b)에서 보인다. [그림 2](b)는 [그림 2](a)의 N2가 N2'로 갱신된 후의 저장 상태를 보인다. [그림 2](b)의 노드 G는 P의 자식노드인 N1과 N2가 위치한 블록 주소를 저장한다. 그리고 2.2에서 소개된 방식으로 가상노드 P를 생성한다. 제안된 방식에서는 노드 P가 저장되지 않기 때문에, N2의 저장 위치가 변경되어도 노드 P를 플래시 상에 갱신할 필요가 없다. 단지, 새로 저장된 노드 N2'의 *max-key* 값을 N2의 *max-key* 값과 같게 함으로써, 노드 P가 동적으로 생성될 때 N2가 N2'로 대체되도록 한다.

논문에서는 동일 부모에 속한 단말노드들을 동일 블록에 저장하며, 이런 블록을 SLB(Sibling Leaf Block)라고 한다. SLB 안에는 노드 갱신 시 새로운 노드를 저장할 빈 페이지 공간이 존재하며, 이 공간은 순차적으로 사용된다. 빈 페이지가 모두 사용된 후에는 SLB 블록을 초기화(erasing)하고 최신 노드 내용으로 재 기록해 줌으로써 다시 빈 페이지 공간을 얻는다. 이는 일종의 GC 동작이며, 이런 GC 동작이 하나의 블록 영역 안

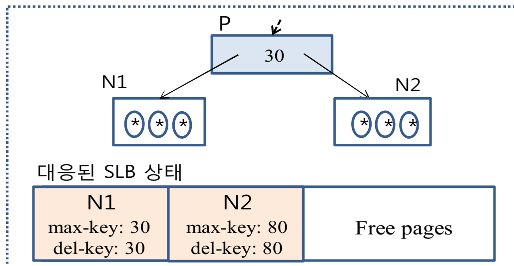
에서 수행되기 때문에 그 비용은 매우 낮다고 할 수 있다. 또한 가상노드에 오버플로나 언더플로가 발생하는 상황이 아니라면 [그림 2](b)의 G를 갱신할 필요가 없다. 노드 G를 갱신할 때는 FTL을 통해 갱신한다. 즉, ULL 위의 계층에 속한 노드의 갱신은 FTL를 통한다. 이런 FTL 갱신 빈도는 매우 낮기 때문에 이로 인한 B-트리의 성능 저하는 거의 없다고 할 수 있다[15][20].

2. 트리 재구성 기법

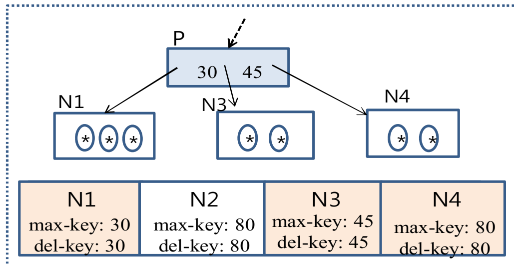
본 절에서는 단말노드를 위한 트리 재구성 동작에 대해 기술한다. 트리 재구성 동작은 노드 분할 혹은 노드 병합(merge)으로 수행되며, 먼저 노드 분할 방식에 대해 설명한다. [그림 3]은 단말노드 N2에 오버플로가 발생하여 노드 분할이 수행된 상황을 보인다. 편의상 B-트리의 단말노드는 최대 3개의 색인 엔트리를 저장한다고 가정한다.

[그림 3](a)의 N2에 오버플로가 발생하여 새로운 두 노드 N3와 N4가 [그림 3](b)와 같이 SLB에 저장된다. [그림 3](b)에서 보이듯이 N4의 *max-key* 값이 N2의 *max-key* 값과 같기 때문에, 이후 노드 P가 다시 버퍼로 적재될 때 N2는 트리에서 삭제된다. N3와 N4가 SLB에 저장될 때, 노드 P는 이 두 노드의 주소를 저장하도록 수정된다. 노드 P는 가상노드이므로 버퍼 내에서만 갱신된다. 제안하는 플래시 B-트리의 단말 노드에는 *max-key* 필드와 함께 *del-key* 필드가 만들어 진다. 보통 이 두 필드의 값은 같으며, 다를 때는 *del-key* 필드가 노드 합병에 사용된다.

이어서 키 언더플로에 대한 노드 합병 방법에 대해 설명한다. [그림 4]는 노드 N3에 발생한 키 삭제로 인해 N2와 노드 합병이 수행된 상황을 보이고 있다. [그림 4](a)의 N3에 언더플로가 발생하여 이웃 노드인 N2와 노드 합병을 한다면 이 두 노드는 트리에서 삭제된다. 대신 이 두 노드의 색인 엔트리를 저장하는 새로운 노드 N4가 [그림 4](b)의 SLB에 저장된다. [그림 4](b)의 N4는 *max-key* 필드 값으로 80을 가지며, 이를 통해 N3를 대체한다. 또한 N4가 가지고 있는 del-key 값은 동일한 *max-key* 값을 가진 이전 노드를 삭제한다. 즉, 노드 N2를 삭제한다. 이처럼 어떤 노드 N의 *del-key* 값과 *max-key* 값이 다를 경우, 이를 통해 N 보다 앞서 저장되었던 두 노드를 트리에서 삭제할 수 있다.

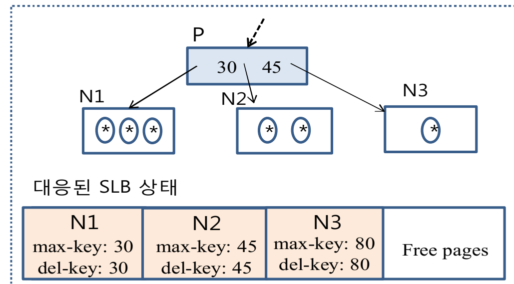


(a) N2의 키 삽입 전 B-트리와 SLB 상태.

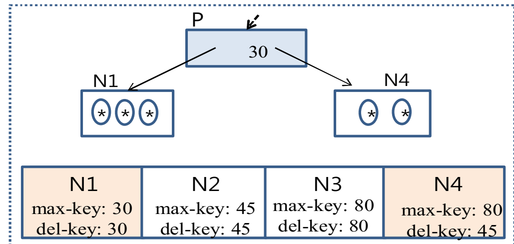


(b) N2에 노드 분할이 수행된 상태.

그림 3. 노드 분할을 통한 트리 재구성



(a) N3의 키 삭제 전 B-트리와 SLB 상태.



(b) N4로 노드 합병이 수행된 후의 상태.

그림 4. 노드 합병을 통한 트리 재구성

```
CreateVirtualParent (Address slb)
Output: 가상 부모노드를 생성하여 Parent로 반환
(1) Keys.Init() // 색인엔트리를 저장할 벡터 초기화
(2) B ← Read_Block(slb); // SLB 블록 로딩
(3) N ← B.FirstNode(); // 가장 왼쪽 노드를 복사
(4) while (N is not empty node)
(5) Keys.Remove(N.max-key, N.del-key); // SLB에서 삭제되었던 노드를 Keys에서 삭제
(6) addr ← the flash address of node N;
(7) Keys.Input(N.max_key, addr); // 색인엔트리 저장
(8) N ← B.NextNode(); // 다음 오른쪽 노드 복사
(9) while_end
(10) Keys.SortWithKey(); // 오름차순 정렬
(11) Parent ← Keys.MakeNode(); // 정렬된 색인 엔트를 사용하여 가상노드 생성
(12) return Parent; // 부모노드 반환
```

그림 5. SLB를 사용한 ULL 가상노드 생성 알고리즘

3. 키 검색 알고리즘

앞에서 SLB의 개념과 SLB에 저장된 단말노드의 *max-key*, *del-key* 필드를 사용한 트리 갱신 방법에 대해 설명하였다. 본 절에서는 이에 대응되어 SLB로부터 가상노드인 ULL 노드를 생성하는 알고리즘과 이를 이용한 키 검색 알고리즘을 제안한다.

[그림 5]는 ULL 노드 생성 알고리즘이다. 알고리즘은 단말노드들이 위치한 SLB 주소를 입력 받아 가상노드 Parent를 생성한다. 해당 SLB를 읽어 들인 후, 라인 5에서 노드 N의 *max-key*나 *del-key* 필드와 같은 값의 *max-key* 필드를 가진 이전 단말노드(왼쪽 위치)들을 모두 삭제하고 있다. 그리고 라인 7에서는 노드 N의 *max-key* 값과 주소를 가상노드의 색인 엔트리로 저장한다. 라인 4 - 9 의 반복 연산을 통해 모여진 색인 엔트리를 키 값으로 정렬한 후, 라인 11에서 가상노드의 색인 엔트리로 사용한다. 예를 들어 색인 엔트리가 3개이고, [(max-key1, addr1), (max-key2, addr2), (max-key3, addr3)]와 같이 정렬되었다면 [addr1, max-key1, addr2, max-key2, addr3]이 가상노드의 색인 엔트리가 된다.

```
KeySearchToLeaf (Key key, Address root)
Output: 결과 단말 노드 주소 leaf 반환
(1) Root ← Buffer.Copy(root) // 버퍼 내 루트노드 복사
(2) addr ← SearchToSLB(Root, key); // Top-down 검색을 통해 이동할 SLB 블록 주소를 반환
(3) if (Buffer.Exist(addr) != true)
(4) Parent ← Buffer.CreateVirtualParent(addr);
(5) if_end
(6) leaf ← SearchToNext(Parent);
(7) return leaf; // target 단말노드 주소 반환
```

그림 6. 단일 키 검색 알고리즘

[그림 6]은 검색 키 값과 루트노드 주소를 받아 해당 키를 저장한 단말노드를 찾는 검색 알고리즘을 보인다. 검색 알고리즘은 기존 디스크 기반 B-트리의 검색 알고리즘과 차이가 거의 없다. 차이가 발생하는 경우는 이동할 ULL 노드가 버퍼에 없을 때이며, 이 경우에는 라인 3-5를 통해 가상 노드를 생성한다. 가상노드를 버퍼에 생성한 후에는 라인 6에서 다음 이동할 단말노드를 결정한다.

범위 검색이 필요한 경우 제안된 B-트리는 SLB 노드를 연결하는 링크를 만들기만 하면 효과적인 범위 검색이 가능하다. 즉, 단말 노드 각각에 링크 연결을 생성해두는 대신, SLB를 키 순서대로 연결해 둠으로써 범위 검색을 수행할 수 있다. 예를 들어, 키 범위 [k1 k2]에 속하는 레코드를 순차 검색하고 싶다면 먼저 k1에 대한 키 검색을 수행하여 k1을 저장한 SLB를 찾고, SLB 안에서 k2 보다 작거나 같은 키들을 가진 단말노드를 검색한다. 현재 읽은 SLB의 키 범위를 넘어 검색이 수행되어야 한다면, SLB 간의 링크를 따라 오른쪽 SLB로 이동함으로써 이후의 키 범위를 검색할 수 있다. 제안한 B-트리 구조에서는 여러 개의 단말노드를 한 번의 I/O로 읽을 수 있기 때문에, 검색 키 범위가 클 경우 범위 검색 효율성을 크게 향상 시킬 수 있다.

표 1. 성능평가 비용 요소 및 의미

symbols	descriptions
H	B-트리의 높이
P_i	레벨 i의 노드가 버퍼에 존재할 확률
M_r	버퍼 내의 노드 접근 시간
N_r	플래시 I/O를 사용한 노드 읽기 시간
N_w	플래시 I/O를 사용한 노드 쓰기 시간
C_u	FTL을 이용한 노드 갱신 비용
B_r	블록의 읽기 시간
L	블록 내 로그 페이지 갯수
K_m	노드 하나의 최대 색인 엔트리 수

$$T_{search} = \sum_{i=1}^{H}(P_i \times M_r + N_r \times (1-P_i^r))$$

(a) Original

$$T_{search} = \sum_{i=1}^{H}(P_i \times M_r + (L+1) \times N_r \times (1-P_i))$$

(b) LB-Tree

$$T_{search} = \sum_{i=1}^{H-2}(P_i \times M_r + N_r \times (1-P_i)) + (1-P_{H-1}) \times (B_r + 2M_r) + P_{H-1} \times (M_r + (P_H \times M_r + N_r \times (1-P_H)))$$

(c) Proposed

그림 7. 단일 키 검색 수행의 평균 시간

IV. 성능 평가

1. 수학적 평가모델

본 절에서는 기존 디스크 기반 B-트리(이하 Original), 로깅 기반 플래시 B-트리(이하 LB-Tree)에 대해서, 제안된 기법인 Proposed의 성능을 비교한다. LB-Tree는 공정한 성능비교가 가능하도록, 동시성 제어 및 범위 검색이 가능한 형태의 로깅 기반의 플래시 B-트리 기법을 의미한다. 즉, 갱신 로그와 갱신 전 노드 데이터가 같은 블록에 저장된다[6][10]. 세 기법 모두 노드 접근은 메모리 버퍼를 통하며, 적재된 노드는 버퍼 교체 알고리즘에 따라 버퍼 방출 전까지 재사용된다. 이 때 내부노드를 단말노드에 우선하여 버퍼링한다.

[표 1]은 성능평가 모델에서 고려된 비용 요소들과 그 의미를 보인다. 기호 P_i은 키 검색 과정에서 트리 레벨이 i인 노드를 접근하려 할 때, 해당 노드가 버퍼에 존재할 확률을 나타낸다. 즉, 버퍼 적중률(hit rate)을 나타낸다. 루트노드의 트리 레벨 값은 1이며, 아래 계층으로 갈수록 값이 커진다. 이미 버퍼링된 노드에 접근하기 위한 시간은 M_r이며, 그렇지 않고 저장 장치에서 버퍼로 읽어 접근할 때의 시간은 N_r로 표시된다. 버퍼 내 노드를 빈 페이지에 씀으로써 노드 갱신할 때의 시간은 N_w로, FTL을 통해 갱신할 때의 시간은 C_u로 표시한다. Original 방식은 C_u의 시간으로 노드를 갱신하게 되며, LB-Tree 방식은 N_w의 노드 갱신 시간을 가진다. Proposed 방식의 경우 ULL 위쪽 노드의 갱신은 C_u의 시간을, 그 외는 N_w의 갱신 시간을 가진다.

기호 L은 LB-Tree에만 적용되는 기호로서 블록 안에 있는 로그 영역의 크기를 표시한다. 즉, 블록 안의 로그 영역은 L 개의 페이지로 구성되며, 블록 안의 노드를 처음 버퍼링할 때마다 이 로그 영역 전체를 읽어야 한다[6][10]. 노드 하나에 저장 가능한 최대 색인 엔트리 개수는 K_m으로 표시된다. B-트리 정의에 따라 노드 안에는 $K_m/2$에서 K_m개의 색인 엔트리가 존재하며, 이 범위 내에서 색인 엔트리 개수가 일정한(uniform) 확률로 분포한다고 가정한다[20] 이런 가정에 따라 특정 노드에 발생한 키 삽입/삭제 연산이 트리 재구성 동작을 발생시킬 확률은 $2/K_m$로 계산된다. 이 값은 이어지는 성능평가 모델에서 사용된다.

[그림 7]은 비교되는 세 가지 B-트리 알고리즘이 갖는 단일 키 검색 시간을 보인다. Original 기법은 루트 노드에서 단말노드로 이동하는 과정에서 노드가 버퍼에 있는 경우는 M_r의 시간으로, 버퍼에 있지 않다면 N_r의 시간으로 노드에 접근한다. LB-Tree는 노드가 버퍼에 없다면 로그 데이터를 읽고 관련된 redo 연산을 수행해야 한다. 로그 데이터는 L개의 페이지에 저장되며, 페이지 크기를 노드 크기로 두었기 때문에 $(L+1) \times N_r$의 시간이 소요된다. 편의 상 평가 모델에서는 redo 시간은 고려하지 않았다.

$$T_{update} = T_{search} + C_u + \sum_{i=1}^{H}((\frac{2}{K_m})^i \times 3C_u)$$

(a) Original

$$T_{update} = T_{search} + N_w + \sum_{i=1}^{H}((\frac{2}{K_m})^i \times 3N_w)$$

(b) LB-트리

$$T_{update} = T_{search} + N_w + \frac{2}{K_m} \times 1.5N_w + \sum_{i=2}^{H}((\frac{2}{K_m})^i \times 3C_u)$$

(c) Proposed

그림 8. 트리 갱신 연산의 평균 처리 시간

Proposed의 경우 ULL 계층 노드가 버퍼에 없다면 해당 블록 전체를 읽어 가상노드를 생성 한 후 트리 검색을 수행한다. 읽은 블록 안에 이동할 단말노드가 포함되어 있기 때문에, 단말노드를 위한 추가적인 노드 읽기 없이 2번의 메모리 접근으로 검색이 수행된다. 반면 ULL 노드가 버퍼에 있는 경우는, M_r의 시간을 거쳐 이동할 단말노드를 알아낸다. 이 후, 이동할 단말노드가 버퍼에 있을 경우와 없을 경우에 대해 [그림 7](c)와 같이 계산된다.

[그림 8]은 키 삽입/삭제를 위한 B-트리 갱신 시간을 보인다. Original은 FTL을 사용하여 노드 갱신을 수행하며, LB-Tree는 빈 로그 영역에 갱신 로그를 저장함으로써 노드를 갱신한다. 갱신 로그가 기록되는 페이지 크기와 노드 크기가 일치하기 때문에, 로그 기록 시간은 N_w와 같다. Proposed 기법은 ULL 계층 위에서는 C_u, 그 아래 계층에서는 N_w의 시간으로 노드를 갱신한다. 언더플로나 오버플로로 인해 트리 재구성 동작이 수행되면, 일반적으로 3개의 노드가 갱신된다. 하지만 Proposed에서는 단말노드를 트리 재구성할 때, 부모노드인 가상노드는 저장 장치를 통해 갱신할 필요가 없다. 또한 노드 합병 시에 노드 하나만 새로 기록하면 되기 때문에 평균 1.5 번의 노드 쓰기가 발생한다. 이에 대한 내용은 [그림 8](c)에서 보인다.

표 2. Micron NAND 플래시 메모리 사양

하드웨어 요소	사양
자장 공간 크기	128GB
셀 구성방식	MLC (Multi-Level Cell)
페이지/블록 크기	2KB/128KB
페이지 읽기 속도	40us
페이지 쓰기 속도	320us
블록 소거 속도	3.5ms
블록 읽기 속도	365us

2. 성능 비교

본 절에서는 앞에서 제안된 성능평가 모델에 따라 성능 비교를 수행한다. 성능 비교되는 B-트리는 2KB 크기의 노드를 가지며, (키, 주소) 쌍으로 구성된 키 엔트리 크기는 40 바이트로 한다. 이에 따라 K_m은 50으로 주어지며, 노드의 평균 키 엔트리 수는 이의 75%로 가정한다[15][20]. 최소 3 이상의 높이를 가진 B-트리에 대해서 성능평가를 수행한다. LB-Tree의 L 값은 로그 영역이 블록의 20% 정도를 점유하도록 설정된다.

성능평가에 적용될 플래시 메모리 사양은 [표 2]에 보인다. 이는 범용 SSD(Solid State Disk) 저장 장치에 널리 사용되는 형태이다[1][3]. 플래시 블록의 읽기 속도는 블록에 속한 개별 페이지를 각각 개별적으로 읽을 때에 비해 5-10배 정도의 속도를 가진다[3][19][12]. 본 논문에서는 이런 속도 범위 중 7배의 중간 속도를 취한다. 또한 보다 작은 크기의 LB-Tree의 로그 영역을 읽는 속도는 5배로 둔다.

DBMS에서 B-트리 사용 시 재사용 빈도가 매우 높은 루트노드와 그 자식노드는 버퍼에 캐시하고 사용하는 것이 일반적이다. 본 논문에서도 이 두 계층의 노드들은 항상 메모리에 캐시됨을 가정한다. 이런 가정에서 B-트리 높이가 3인 경우, 비교되는 기법들의 평균 키 검색 시간을 평가했다. [그림 9]는 이에 대한 결과이다.

실험에서 단말노드의 버퍼 적중률(P_3)을 변화시키며 검색 성능을 비교했다. 그림에서 보이듯이 트리의 상위 2개 계층이 모두 캐시된 상태이기 때문에 Proposed와 Original의 성능 차이는 없다. 이는 제안한 기법의 가상노드가 모두 생성되어 버퍼에 올라와 있는 상태에서는

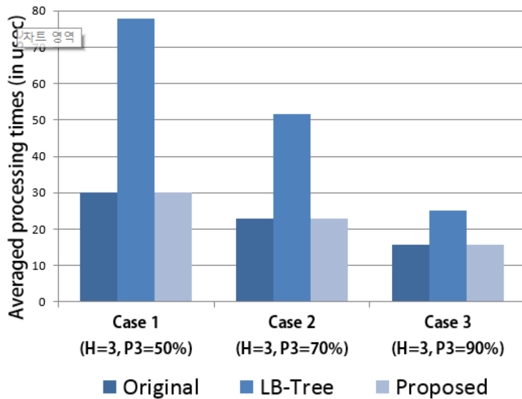


그림 9. H=3인 경우의 평균 키 탐색시간 비교

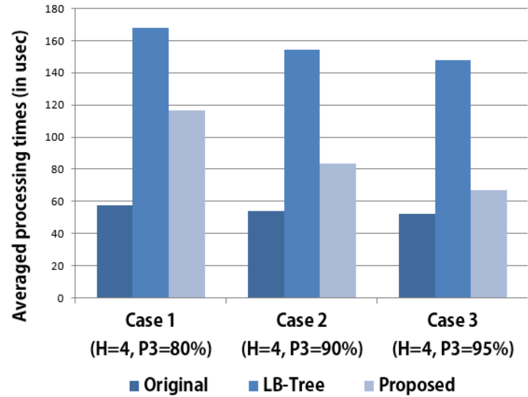


그림 10. H=4인 경우의 평균 키 탐색시간 비교

기존 B-트리 구조와 차이가 없기 때문이다. 반면 LB-Tree는 단말노드에 접근할 때, 로그 영역을 읽는 추가 비용으로 인해 다른 두 기법에 비해 좋지 않은 검색 성능을 보인다. 일반적으로 단말노드의 버퍼 적중률은 매우 작기 때문에 LB-Tree는 성능 저하를 피할 수 없다.

[그림 9]의 트리에 비해 보다 많은 수의 레코드를 색인하는 H=4인 경우에 대해서도 성능평가를 수행한다. 이에 대한 결과는 [그림 10]에서 보이며, [그림 10]은 단말노드의 버퍼 적중률이 5%임을 가정한다. Original 기법은 노드 접근 비용이 상대적으로 낮기 때문에 트리가 커지는 경우 검색 성능이 우수함을 알 수 있다. 하지만, 가상노드의 버퍼 적중률이 커질수록 제안된 기법은 빠르게 Original 기법의 성능에 접근하며, 모든 경우 LB-Tree 보다 우수한 성능을 보인다. 노드에 평균 40개 색인 엔트리가 존재할 때, 루트노드에서 레벨 3까지의 노드들을 128MB 크기의 메모리 버퍼에 모두 캐시할 수 있다. 따라서 제한한 방법은 대부분의 색인 환경에서 좋은 검색 성능을 제공한다고 할 수 있다.

[그림 11]은 키 삽입/삭제 연산이 혼합되어 수행될 때의 평균 질의 처리 시간을 보인다. 평가에 사용되는 B-트리는 [그림 10]의 높이 4인 트리(Case 2)이다. FTL을 통해 B-트리 노드를 갱신할 때는 full-merge가 발생하여 GC 비용이 크게 증가한다. Full-merge 발생 시에 최대 64개의 연관된 블록이 초기화 되어야 하며, 이에 관련된 페이지 복사가 수행되어야 한다. 따라서 하나의 블록 안에서 GC가 수행될 때에 비해 수십 배 이상의 비용이 발생한다[5][6][18]. 논문에서는 C_u의 크기를 보수적으로 잡아, N_w 의 3배 정도로 산정한다. [그림 11]에서 보이듯이 C_u 를 작게 두었음에도 Original의 경우 갱신 연산 비율의 증가로 전체 성능이 크게 저하됨을 알 수 있다. 반면 제안한 방식은 타 기법에 비해 좋은 성능을 보인다. 갱신 비율이 10%의 상황에서도 Original에 비해 40%의 성능 향상이 있으며, 갱신 비율이 30%의 상황에서는 2배의 성능을 보이고 있다.

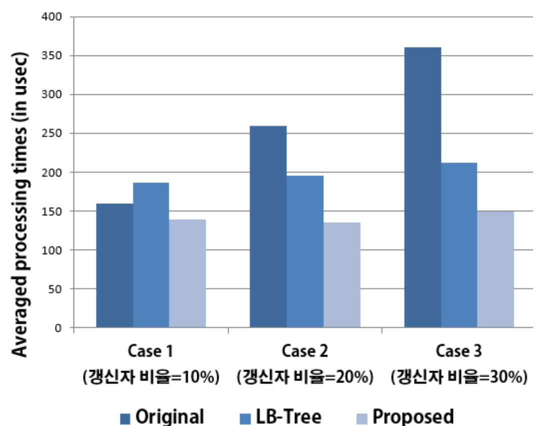


그림 11. H=4인 경우의 평균 키 탐색시간 비교

V. 결론

플래시 메모리를 사용한 저장 장치에 B-트리 색인 파일을 저장하여 사용할 때 발생할 수 있는 문제점은 노드에 발생하는 갱신 연산의 빈도가 높을 때 색인 성능이 크게 저하될 수 있다는 점이다. 이는 플래시 메모리가 제자리 갱신을 수행할 수 없기 때문에 기인한다. B-트리는 본래 제자리 갱신이 수행되는 저장 장치인 하드디스크에 맞게 고안되었기에, 플래시 저장장치에서 사용할 때는 이를 고려한 갱신 알고리즘과 저장 방식이 요구된다.

논문에서는 B-트리에서 발생하는 대부분의 갱신이 단말노드와 그 부모노드 계층에서 발생한다는 점에 착안하여, 이 두 계층에 대한 노드 갱신을 효과적으로 수행할 수 있는 플래시 B-트리를 제안하였다. 제안된 방식은 가상 노드 개념과 bottom-up 노드 생성 방법을 적용함으로써 플래시 메모리 B-트리에서 발생하는 연쇄적 갱신 문제를 피했다. 또한 SLB 블록을 사용하여 여러 개의 단말노드에 대해서 갱신 정보를 효과적으로 관리할 수 있게 했다. 이를 통해 낮은 트리 갱신 비용을 보장하면서도 범위 검색을 효과적으로 할 수 있도록 하였다.

제안된 방식은 플래시 메모리 기반의 상용 DBMS 사용이 늘어나는 상황에서 중요한 의미를 가진다. 상용 DBMS의 테이블 생성을 위해서는 B-트리를 저장해야 하며 만약 B-트리에 발생하는 갱신 연산으로 인해 플래시 저장장치의 성능 저하 현상이 빈번히 발생한다면, 이는 큰 문제가 된다. 특히, 기비지 수집에 의한 성능 저하 현상은 발생 시점의 예측이 어렵고, 발생 시 DBMS 시스템에서 동작 중인 모든 갱신 연산의 수행 속도에 영향을 끼치기 때문에, 이를 미리 막는 것이 매우 중요하다. 이런 관점에서 제안된 B-트리는 가비지 수집 비용이 매우 작게 유지할 수 있기 때문에 상용 DBMS에 적용될 경우, 안정적인 시스템 운영에 크게 기여할 수 있을 것이다.

참 고 문 헌

[1] S. W. Lee, B. Moon, and C. Park, “Advances in Flash Memory SSD Technology for Enterprise Database Applications,” In Proceedings of SIGMOD, 2009.

[2] Stepahan Baumann, Giel de Nijs, Michael Strobel, and Kai-Uwe Sattler, “Flashing Databases: Expectations and Limitations,” In Proceedings of DaMon (Data Management on New Hardware), 2010.

[3] Adam Leventhal, “Flash Storage Memory,” Communications of the ACM, Vol.51, No.7, pp.47-51, 2008.

[4] S. J. Lee, D. K. Shin, Y. J. Kim, and J. H. Kim, “LAST: Locality-Aware Sector Translation for NAND Flash Memory-based Storage Systems,” ACM SIGOPS Review, pp.36-42, 2008.

[5] Asyush Gupta, Y. J. Kim, and Bhuvan Urgaonkar, “DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings,” In Proceedings of ASPLOS, 2009.

[6] G. J. Na, S. W. Lee, and B. K. Moon, “Dynamic In-Page Logging for B+-tree Index,” IEEE Transactions on Knowledge and Data Engineering, Vol.24, No.7, pp.1231-1243, 2012.

[7] C. Mohan, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz, “ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging,” ACM Trans. on Database Systems, Vol.17, No.1, pp.94-162, 1992.

[8] Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Singh, "Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices," In Proceedings of VLDB, pp.361-372, Aug. 2009.

[9] S. W. Lee and B. K. Moon, "Design of Flash-based DBMS: An In-page Logging Approach," In Proceedings, of ACM SIGMOD, pp.55-66, 2007.

[10] S. W. Lee and B. K. Moon, "Transactional In-Page Logging for Multiversion Read Consistency and Recovery," In Proceedings of ICDE, pp.876-887, 2011.

[11] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil, "The Log-Structured Merge-Tree (LSM-Tree)," Acta Informatica, Vol.33, No, pp.351-385, 1996.

[12] 임성채, 박창섭, "효율적 범위 검색을 위한 플래시 기반 B+-트리," 한국콘텐츠학회논문지, 제13권, 제9호, pp.28-38, 2013.

[13] Xiaoyan Xiang, Lihua Yue, Zhanzhan Liu, and Peng Wei, "A Reliable B-Tree Implementation over Flash Memory," In Proceedings of ACM SAC, pp.1487-1491, 2008.

[14] 김학철, 박용훈, 윤종현, 서동민, 송석일, 유재수, "Hot 데이터 블록 병합 지연을 이용한 효율적인 메모리 로그 버퍼 관리 기법," 한국콘텐츠학회논문지, 제10권, 제1호, pp.68-77, 2010.

[15] Marcel Kornacker, C. Mohan, and Joseph Hellerstein, "Concurrency and Recovery in Generalized Search Trees," In Proc. Of SIGMOD, 1997.

[16] Mustafa Ganim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lan, "SSD Bufferpool Extensions for Database Systems," In Proceedings of VLDB, pp.1435-1446, 2010.

[17] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang, "An Efficient B-tree Layer Implementation for Flash-memory Storage Systems," ACM Transactions on Embedded Computing Systems, Vol.6, No.3, 2007.

[18] S. W. Park, H. J. Song, and D. H. Lee, "An Efficient Buffer Management Scheme for Implementing a B-Tree on NAND Flash Memory," In Proceedings of ICESS '07, 2007.

[19] Yinan Li, Bingsheng He, Robin J. Yang, Qiong Luo, and Ke Yi, "Tree Indexing on Solid State Derives," In Proceedings of VLDB, pp.1195-1206, 2010.

[20] S. C. Lim and M. H. Kim, "Restructuring the Concurrent B+-tree with Non-blocked Search Operations," Informaction Sciences, Vol.147, pp.123-142, 2002.

[21] Chang Xu, Lidan Show, Gang Chen, Cheng Yan, and Tianlei Hu, "Update Migration: An Efficient B+-tree for Flash Storage," In Proc. of DASFAA, pp.276-290, 2010.

[22] Hua-Wei Fang, Mi-Yen Yeh, Pei-Lun Suei, and Tei-Wei Kuo, "An Adaptive Endurance-aware B+-tree for Flash Memory Storage Systems," IEEE Transactions on Computers, 2013.

[23] H. C. Roh, S. H. Park, S. H. Kim, M. C. Shin, and S. W. Lee, "B+-tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives," In Proc. of VLDB, pp.286-297, 2011.

저 자 소 개

임 성 채(Seong-Chae Lim) 정회원

- 2003년 3월 : KAIST 전산학과 (이학박사)
- 2000년 ~ 2005년 : 코리아 와이즈넛 책임연구원/이사
- 2005년 3월 ~ 현재 : 동덕여자대학교 컴퓨터학과 부교수

<관심분야> : 고성능 색인기법, 빅 데이터