DOI: 10.3745/KIPSTA.2008.15-A.6.325

# 플래시 메모리 상에서 $B^+$-트리 노드 크기 증가에 따른 성능 평가

박 동 주† · 최 해 기††


## 요 약

플래시 메모리는 크기가 작고 적은 전력을 사용하며 충격에 강하기 때문에 휴대폰, MP3 플레이어, PDA와 같은 이동 기기에 널리 사용되고 있다. 또한, 노트북과 개인용 컴퓨터에서 사용하던 하드디스크를 플래시 메모리로 교체하려는 시도도 진행되고 있다. 최근에는 플래시 메모리 저장 시스템에서 대용량의 데이터를 효율적으로 검색하기 위한 플래시 메모리용 $B^+$-트리 인덱스를 개발하려는 연구가 이루어지고 있다. 이러한 연구는 $B^+$-트리에 키의 삽입 또는 삭제 시 발생하는 "덮어쓰기"를 최소화하는데 초점을 두고 있다. 그러나 이것뿐만 아니라 하나의 $B^+$-트리 노드에 할당되는 물리적 페이지의 크기도 $B^+$-트리 성능에 영향을 줄 수 있다. 본 논문에서는 다양한 실험을 통해 노드 크기에 따른 $B^+$-트리의 구축 성능, 검색 성능, 그리고 저장 공간 사용량을 비교 및 분석한다. 노드에 키 삽입 시 정렬 및 비정렬 알고리즘을 제시하며, 또한 효율적인 노드 검색을 위한 적절한 인덱스 노드 헤드 구조를 제안한다.


**키워드 : 플래시 메모리, $B^+$-트리, 인덱스**

# Effect of Node Size on the Performance of the $B^+$-tree on Flash Memory

Dong-Joo Park† · Haegi Choi††


## ABSTRACT

Flash memory is widely used as a storage medium for mobile devices such as cell phones, MP3 players, PDA's due to its tiny size, low power consumption and shock resistant characteristics. Additionally, some computer manufacturers try to replace hard-disk drives used in Laptops or personal computers with flash memory. More recently, there are some literatures on developing a flash memory-aware $B^+$-tree index for an efficient key-based search in the flash memory storage system. They focus on minimizing the number of "overwrites" resulting from inserting or deleting a sequence of key values to/from the $B^+$-tree. However, in addition to this factor, the size of a physical page allocated to a node can affect the maintenance cost of the $B^+$-tree. In this paper, with diverse experiments, we compare and analyze the costs of construction and search of the $B^+$-tree and the space requirement on flash memory as the node size increases. We also provide sorting-based or non-sorting-based algorithms to be used when inserting a key value into the node and suggest an header structure of the index node for searching a given key inside it efficiently.


**Keywords : Flash Memory, $B^+$-tree, Index**

## 1. 서 론

플래시 메모리는 소비전력이 적고 하드디스크에 비해 빠른 접근 속도를 가지며 물리적 충격에 강하다. 또한 크기가 작고 무게가 가벼우며 전원이 꺼지더라도 저장된 정보는 사라지지 않는 비휘발성의 성질 때문에 휴대 전화기, MP3플레이어, 디지털 카메라와 같은 휴대용 정보기기들의 보조기억장치로 널리 사용되고 있다. 최근에는 플래시 메모리 집적도의 증가로 인해 대용량화 되어 감에 따라, 휴대용 정보기기들의 보조기억장치뿐만 아니라 개인용 컴퓨터(PC)나 노트북(LAPTOP)에서도 보조기억장치로 활용되고 있다. 또한 대용량의 데이터를 효율적으로 접근하기 위해 $B^+$-트리와 같은 자료구조를 플래시 메모리상에서 저비용으로 구현하려는 연구들이 이루어지고 있다[1][2].

그런데 플래시 메모리는 하드 디스크와 달리, 데이터가 존재하고 있는 섹터(sector)에 또 다른 데이터를 쓰고자 하는 경우 기존의 섹터에 있는 데이터를 지우고 난 후에 새로운 데이터를 쓸 수가 있다. 다시 말해서 섹터에 대해 덮어쓰기(overwrite)가 허용되지 않는다. 그리고 섹터의 데이터를 지우기 위해서는 섹터를 포함하고 있는 블록 전체를 소거(erase)해야 하며, 한 블록이 안정적으로 소거될 수 있는 횟수는 제한적이다. 이와 함께 플래시 메모리에서 데이터를 읽거나 쓰는 단위는 하드디스크와 달라서, 기존의 하드 디

※ 본 연구는 숭실대학교 교내연구비 지원으로 이루어졌음.
† 정 회 원 : 숭실대학교 컴퓨터학부 조교수
†† 준 회 원 : 메디슨 연구원
논문접수 : 2008년 5월 6일
수 정 일 : 1차 2008년 9월 8일
심사완료 : 2008년 9월 8일

스크에서 사용되던 알고리즘들을 플래시 메모리에 적용시키면 비효율적인 입출력(I/O) 비용이 발생된다.

일반적으로 하드 디스크에서 $B^+$-트리를 구현할 때 트리의 노드 크기를 디스크의 블록 단위로 할당해서 사용한다. 또한 플래시 메모리에서의 입출력(I/O) 단위는 섹터(512B)이기 때문에, 노드 크기를 섹터 단위로 할당하여 $B^+$-트리를 구축하는 연구가 진행되어 왔다[1][2]. 그러나 아직까지 여러 개의 섹터를 하나의 노드에 할당하여 $B^+$-트리를 구현할 때 플래시 메모리에서 일어나는 블록소거 횟수나 쓰기연산, 읽기연산의 증감 변화에 대한 연구가 이루어지지 않았다. 노드에 할당된 섹터의 개수가 $B^+$-트리 성능에 미치는 영향을 측정함으로써, 플래시 메모리 상에서 $B^+$-트리 구현 시 보다 적합한 노드 크기를 측정하고 플래시 메모리에서 $B^+$-트리를 저비용으로 구현할 수 있는 연구가 필요하다. 따라서 본 논문에서는 노드에 할당되는 섹터의 개수를 1개(512B)에서부터 32개(16KB)까지 증가시키면서 노드 크기가 $B^+$-트리의 성능에 미치는 영향을 비교·분석한다.

본 논문의 구성은 다음과 같다. 2장에서는 플래시 메모리의 특성과 FTL 그리고 플래시 메모리 상에서 $B^+$-트리를 구현한 연구를 간단히 설명한다. 3장에서는 노드 크기 증가 방식과 검색성능 향상을 위한 노드 인덱스 헤드 구조를 설명하고 4장에서는 실험 결과를 비교·분석한다. 마지막으로 5장에서 결론을 맺는다.

## 2. 관련 연구

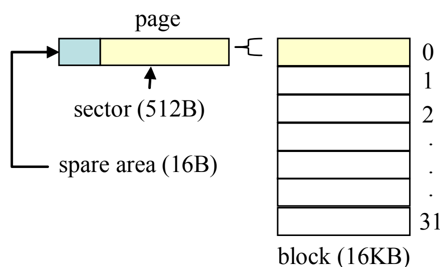
### 2.1 플래시 메모리의 특성과 FTL

데이터 저장 용도로 많이 사용되는 NAND 형 플래시 메모리는 다수 개의 블록(block)으로 구성된다. (그림 1)의 (a)와 같이 하나의 블록은 32개의 페이지로 이루어지며, 하나의 페이지는 512 바이트의 데이터를 저장하는 영역(sector area)과 16 바이트의 부가 정보(ECC, LSN)를 저장하는 영역(spare area)으로 구성된다. 플래시 메모리는 데이터 입출력 단위와 데이터 소거 단위가 각각 다르다. 플래시 메모리의 데이터 입출력 단위는 페이지(page)이며 데이터 소거 단위는 블록이다. 이미 데이터가 존재하는 섹터에 대해서 덮어쓰기(overwrite)를 할 수 없는 플래시 메모리의 특성으로 인해서 기존의 데이터를 변경하려는 쓰기연산이(즉, 덮어쓰기) 발생하면 그 섹터를 포함하는 블록을 소거한 후에 새로운 데이터를 쓸 수 있다. 다시 말해서, 하나의 섹터를 지우기 위해서는 그 섹터를 포함하는 블록 전체를 소거해야 한다. 이러한 단점을 보완하기 위해서 플래시 메모리와 기존의 파일 시스템 사이에는 FTL(Flash Translation Layer)이 존재한다. FTL은 플래시 메모리가 처리할 수 없는 덮어쓰기를 효율적으로 처리하기 위한 기법인데, 현재 알려진 FTL중 가장 성능이 우수한 기법은 FAST(Fully Associative Sector Translation)이다[4]. 이 기법은 별도의 소수 블록을 사용하여 효율적인 덮어쓰기 연산을 수행함으로써 덮어쓰기가 발생했을 때 플래시 메모리에서 발생하는 비용을 최대한 낮출 수 있다[4].

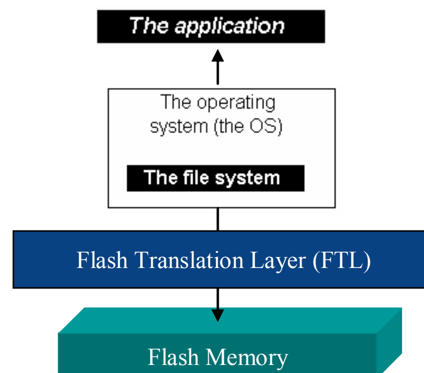
### 2.2 플래시 메모리상에서의 $B^+$-트리 연구

BOF[1]와 BFTL[2]은 플래시 메모리의 물리적 특성을 고려하여 $B^+$-트리를 저비용으로 구축하기 위한 기법이다. BFTL은 플래시 메모리에서 비용소모가 가장 큰 블록 소거를 줄이기 위해서 유보 버퍼(Reservation buffer)와 노드 변환 테이블(Node Translation Table)을 두고 있으며 BOF 또한 구축비용을 줄이기 위해서 유보 버퍼를 사용하고 있다. 그리고 버퍼의 저장 공간을 활용하기 위해서 색인 유닛(Index Unit)을 사용하고 있다. 색인 유닛이란 $B^+$-트리의 기본적인 노드 정보(노드 식별자, 부모 노드 포인터, 단말 플래그 등)와 함께 해당되는 키(Primary key) 값과 포인터(pointer) 값으로 구성되며 새로운 키 값이 삽입 되었을 경우 유보 버퍼 안에는 색인 유닛 단위로 새로운 키 값과 포인터 값들이 저장된다.

버퍼에 있는 내용을 플래시 메모리로 저장하는 알고리즘을 수용정책(commit policy)라 부른다. BFTL은 $B^+$-트리를 저비용으로 구축하기 위해 플래시 메모리의 특성을 고려한 수용정책을 사용한다. 플래시 메모리에서 이미 데이터가 저장되어 있는 섹터를 대상으로 하는 쓰기 연산은 블록 소거




(a) 플래시 메모리 구성




(b) FTL

(그림 1) 플래시 메모리 구성과 FTL

횟수를 높이는데 결정적인 영향을 미치기 때문에 이러한 쓰기 연산을 지연시키는 것이 플래시 메모리 상에서 데이터를 보다 저비용으로 구축하는데 유리하다. 따라서 BFTL에서의 수용정책은 버퍼에서 선정된 희생자(victim) 색인 유닛들을 데이터가 존재하지 않는 섹터에 저장함으로써 블록 소거 횟수를 감소시킨다. 또한 쓰기 연산의 횟수를 낮추기 위하여 하나의 섹터 안에 다른 노드의 키 값들이 함께 저장되는 방법을 사용하였다. 그리고 섹터에 저장되어 있는 색인 유닛의 정보들을 노드 변환 테이블에 유지함으로써 섹터 안에 색인 유닛을 검색할 수 있게 설계되었다.

BFTL에서는 쓰기연산을 감소시켜 $B^+$-트리의 구축비용을 감소시킨 장점이 있지만 부가적으로 존재하는 노드 변환 테이블 비용과 하나의 노드를 읽을 때 여러 개의 섹터에 접근하게 되어 검색성능이 저하되는 단점이 발생한다. 그래서 BOF는 이러한 단점을 보완하여 $B^+$-트리의 검색성능을 향상시키는 기법을 제안하였다. BOF는 각 노드에 하나의 섹터를 할당함으로써 검색 시 한 번의 섹터접근으로 하나의 노드를 읽을 수 있다. 이로 인해서 BFTL에서 관리하던 노드 변환 테이블을 유지하지 않아도 노드를 검색할 수 있다. 따라서 본 논문에서는 효율적인 삽입과 검색이 가능한 BOF의 버퍼관리와 유사한 기법으로 버퍼를 관리한다.

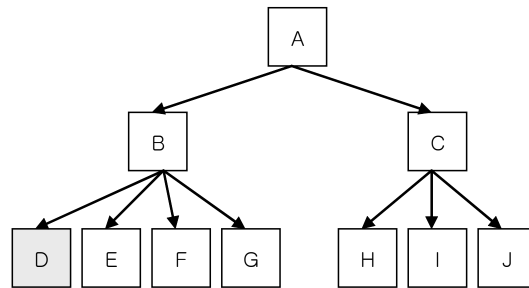
〈표 1〉 섹터 개수에 따른 노드 정보

| 섹터 개수 | 1 | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|---|
| 노드 크기 | 512B | 1KB | 2KB | 4KB | 8KB | 16KB |
| 키 개수 | 62 | 126 | 254 | 510 | 1022 | 2046 |

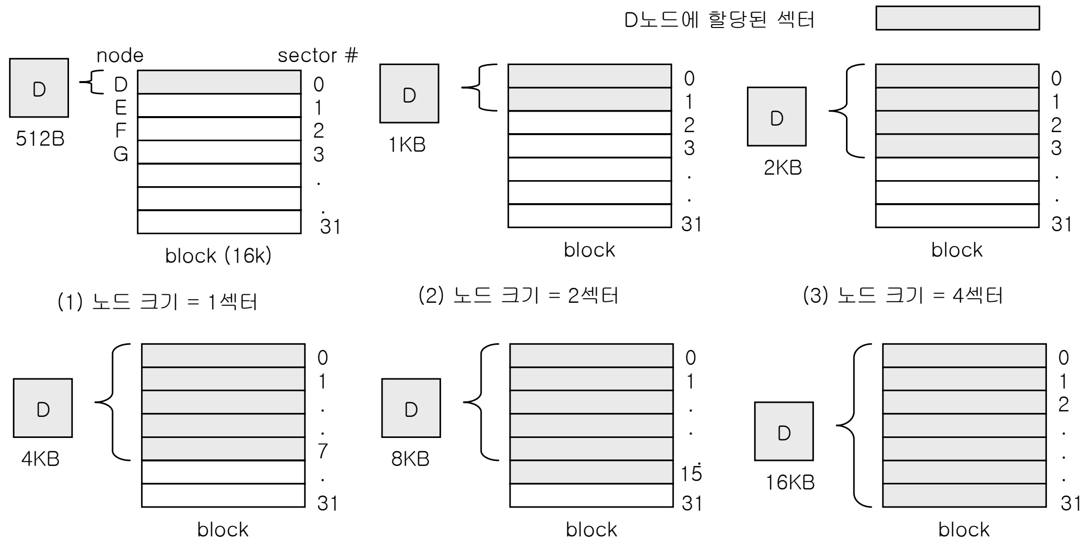
## 3. 플래시 메모리 상에서 $B^+$-트리 노드 확장 설계 및 구현

### 3.1 $B^+$-트리 노드 확장 설계

노드의 크기를 섹터 크기보다 크게 할당하였을 경우에 $B^+$-트리의 성능에 미치는 영향을 분석하기 위해서 $B^+$-트리의 노드크기를 섹터 1개에서부터 32개(블록크기)까지 증가시켜 $B^+$-트리에 미치는 성능을 비교·분석한다. (그림 2)와 같이 한 노드를 2개 이상의 섹터에 할당하는 경우 물리적으로 연속되는 섹터를 하나의 노드로 사용한다. <표 1>은 하나의 노드에 할당되는 섹터의 개수를 증가시킬 때 영향을 받는 노드 크기(node size)와 하나의 노드 안에 저장될 수




(a) B+트리




(b) 노드 크기에 따른 섹터 할당

(그림 2) 노드 확장 방법

있는 키의 개수를 나타낸 것이다.

(그림 2)에서 (b)의 (1)번은 노드 크기를 하나의 섹터로 할당 하였을 때 (a)에서의 노드에 할당된 섹터를 나타낸다. (b)의 (2)번은 노드 크기를 섹터 2개로 할당하였을 때 물리적으로 연속되는 2개의 섹터에 할당하는 것을 보여준다. 그리고 (b)의 (6)번은 노드 크기를 하나의 블록으로 할당하였을 때 D 노드가 사용하는 32개의 섹터를 나타내고 있다. 이와 같이 노드가 할당하는 섹터의 개수가 1, 2, 4, 8, 16, 32개일 때 플래시 메모리에 구축되는 $B^{+}$-트리 성능을 각각 비교 분석한다.

### 3.2 삽입 알고리즘

본 논문에서는 키 삽입 시 정렬, 비정렬 알고리즘을 각각 사용하여 노드 크기 증가 시 $B^{+}$-트리 성능을 비교·분석한다. 정렬 알고리즘은 일반적인 $B^{+}$-트리에서 사용하는 키 삽입 방식이다. 이 알고리즘은 노드 안의 키 값들을 항상 정렬된 상태로 유지한다. 이와 달리 비정렬 알고리즘은 노드 안의 키 값들을 정렬시키지 않고 키 값을 삽입하는 방식이다. 정렬 알고리즘은 $B^{+}$-트리 구축 시 노드가 할당하는 섹터의 수가 증가할수록 부가적인 쓰기 연산을 빈번히 발생시키는 문제점을 가지고 있다. 그러나 비정렬 알고리즘은 정렬 알고리즘에서 발생하는 부가적인 쓰기연산을 대부분 방지하며 노드가 할당하는 섹터의 수가 증가할수록 구축 성능을 크게 향상시킨다.

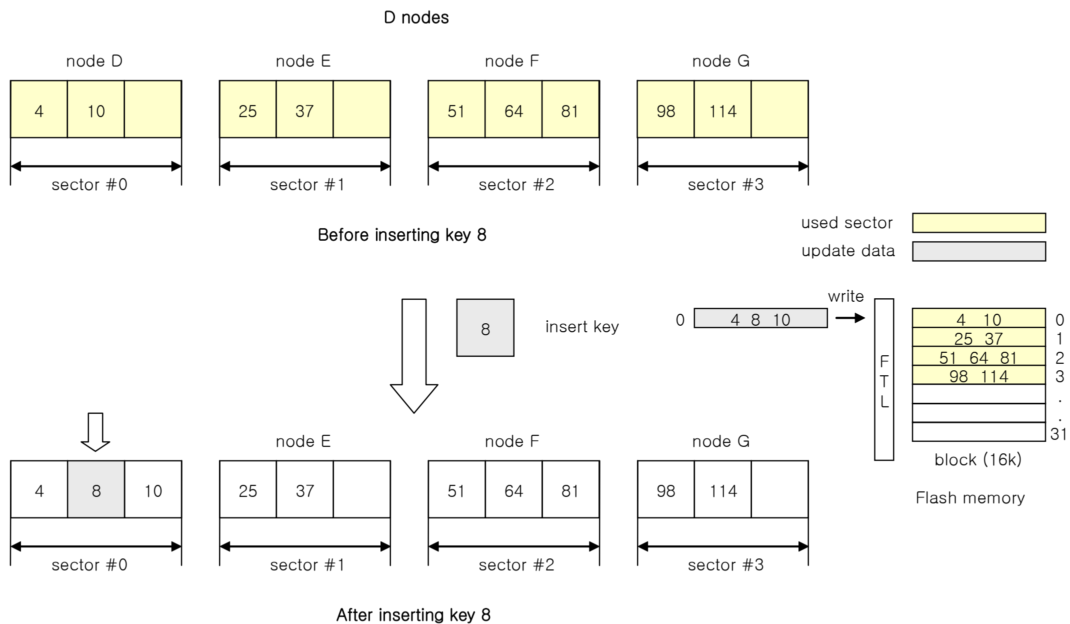
#### 3.2.1 정렬 알고리즘

기본적인 $B^{+}$-트리 개념에서 노드 안의 키 값들은 항상 정렬된 상태를 유지한다. 이렇게 키 값들을 정렬된 상태로 유지하면 노드 안의 키 값들을 이분 검색(binary search) 할 수 있게 되어 검색 성능이 향상된다. (그림 3)은 노드가 할당하는 섹터의 개수가 1개일 때 정렬 알고리즘을 사용하는 예시이다. D노드는 플래시 메모리의 0번 섹터에 저장이 되며, E노드는 1번 섹터, F와 G는 2번과 3번 섹터에 각각 저장된다. 이때 키 값 '8'이 삽입되면 이 키 값이 들어갈 D노드가 사용하고 있는 섹터 0번에 쓰기연산이 발생한다. 즉 노드가 1개의 섹터를 사용할 때 노드 안의 키 값들을 정렬시키더라도 1개의 키 값 삽입에 대해서는 키 값이 삽입되는 노드에 할당된 1개의 섹터에 대하여 쓰기연산이 발생된다.

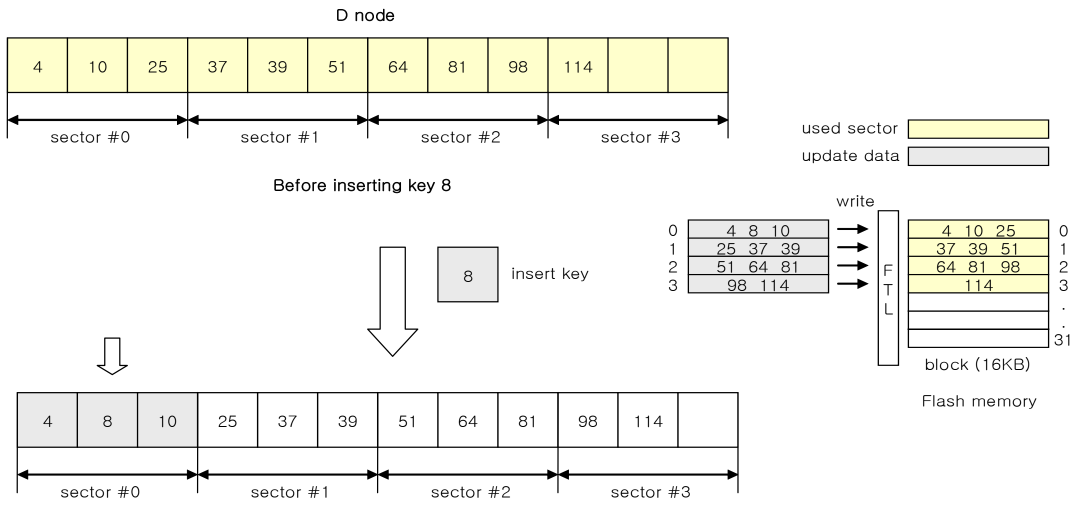
노드 크기가 커져 노드가 할당하는 섹터의 수가 증가하면 정렬 알고리즘으로 인해서 비효율적인 쓰기연산이 발생된다. 예를 들어 (그림 4)에서와 같이 노드에 할당된 섹터 개수가 4개이고 따라서 D노드는 플래시 메모리의 0번~3번 섹터를 사용한다. 0번 섹터에는 D노드의 키 값 4, 10, 25가 저장되어 있고 섹터 1번에는 키 값 37, 39, 51 이 저장되어 있다. 이렇게 노드 안의 키 값은 정렬된 순서로 섹터에 저장된다. 이때 D 노드에 키 값 '8'이 삽입되면 노드 안의 키 값들을 정렬시키기 위해서 키가 삽입된 노드가 할당하고 있는 4개의 섹터 정보들을 모두 수정해야 한다. 즉 1개의 키 값 삽입에 대해서 최악의 경우 4개의 섹터에 대하여 쓰기연산이 발생하므로 노드가 1개의 섹터를 할당할 때보다 쓰기연산 횟수가 많아진다. 결국 쓰기연산이 많아지면 $B^{+}$-트리에 키 삽입 시 플래시 메모리의 쓰기연산 발생을 증가시켜 구축 성능에 악영향을 미친다. 이러한 비효율적인 쓰기 연산은 노드에 할당된 섹터의 개수가 증가할수록 더 빈번히 발생한다.

#### 3.2.2 비정렬 알고리즘

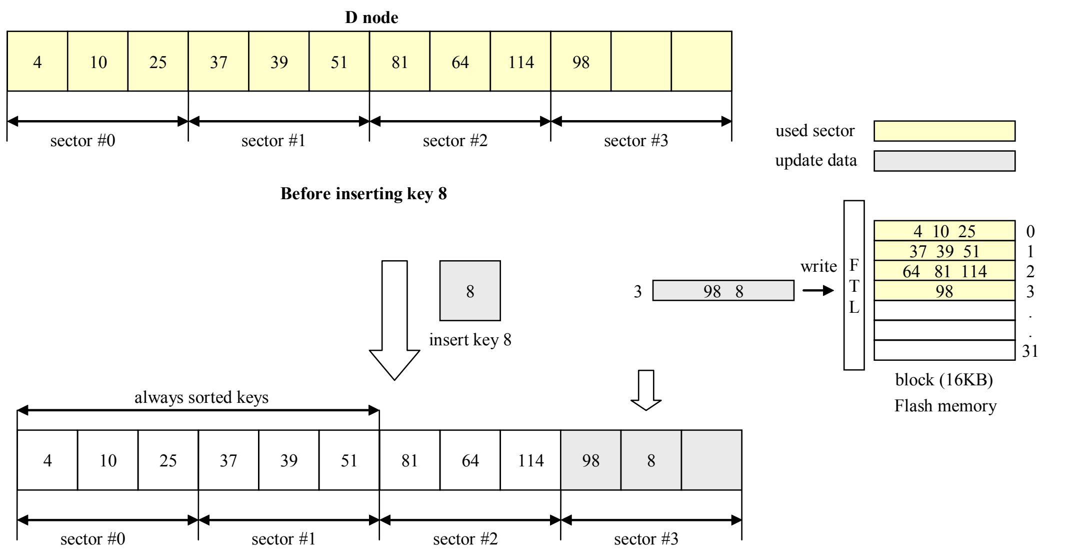
노드에 할당된 섹터의 수가 많아질 때 정렬 알고리즘을 사용하면 부가적인 쓰기 연산이 발생한다. 이러한 비효율적인 쓰기 연산을 방지하기 위하여 비정렬 알고리즘을 사용한다. 비정렬 알고리즘은 키 값이 삽입되면 단말노드에 한해서 노드 안의 키 값들을 정렬 시키지 않고 삽입한다. (그림




(그림 3) 정렬 알고리즘 (노드 크기 = 섹터 1개)




(그림 4) 정렬 알고리즘 (노드 크기 = 섹터 4개)




(그림 5) 비정렬 알고리즘 (노드 크기 = 섹터 4개)

5)는 노드가 할당하는 섹터의 개수가 4개일 때 D노드가 플래시 메모리의 0번~3번 섹터를 사용하고 있으며 키 값 8이 삽입될 때 정렬 알고리즘에서 발생되는 부가적인 쓰기연산을 발생시키지 않고 키 값을 삽입하는 방식을 보여준다. 비정렬 알고리즘은 노드가 할당하는 섹터의 개수가 증가하더라도 1개의 키 삽입에 대해서는 항상 키 값이 저장되는 섹터에만 쓰기 연산을 발생시키므로 정렬 알고리즘일 때 발생되는 부가적인 쓰기 연산을 방지한다.

키 삽입 시 해당 단말노드에 키 값을 더 이상 삽입할 공간이 없을 때 노드분리(node split)가 발생한다. 이때 해당 단말노드의 키 값은 정렬되어 있지 않으므로 우선 정렬을 한 후 앞쪽의 반은 새로 할당된 단말노드에 저장하고, 나머지 반은 기존의 단말노드에 저장한다. 따라서 (그림 5)와 같이, 비정렬 알고리즘을 사용할 경우 모든 단말노드의 최대 키 값의 반 정도는 항상 정렬 상태를 유지하게 된다.

### 3.3 검색 알고리즘

노드가 할당하는 섹터의 개수가 증가하면 한 노드 안에 있는 키 값을 검색할 때 그 노드가 할당하고 있는 여러 개의 섹터를 읽어야 하므로 검색 성능을 저하시킬 수 있다. 노드의 크기를 섹터 1개로 할당하였을 때와 4개를 할당하였을 때에 한 노드 안에 있는 어떤 키 값을 검색하기 위해서는 섹터 1개를 할당한 노드는 1번의 섹터 읽기 연산을 수행하여 원하는 키 값을 검색할 수 있지만, 4개를 할당한 노드는 일반적인 B-트리 노드 상의 키 검색 기법인 이분 검색 기법을 사용하여 검색 비용을 줄일 수 있다. 하지만 이와

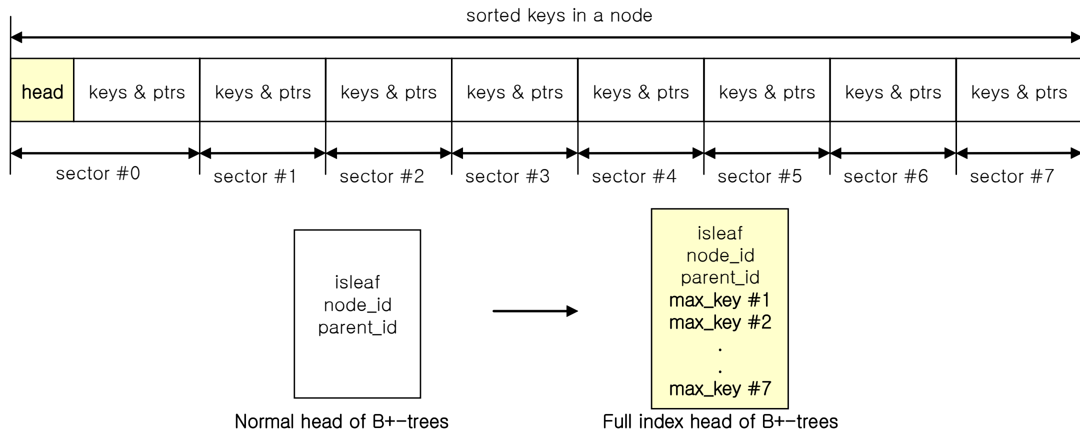
같은 이분 검색 기법은 노드가 할당하는 섹터의 개수를 증가시키면 검색성능이 저하되는 단점이 있다. 노드가 할당하는 섹터의 개수가 증가할 때 이와 같은 단점을 해결하기 위하여 삽입 알고리즘(정렬, 비정렬)에 따라 노드 헤드를 전인덱스(full index), 반인덱스(half index) 구조로 각각 설계한다. 그리고 키 검색 시 노드 헤드 구조에 따라 전인덱스, 반인덱스 검색 알고리즘을 각각 사용하여 노드가 할당하는 섹터의 개수가 증가할 때 나타나는 취약한 검색 성능을 보완한다.

### 3.3.1 전인덱스 헤드

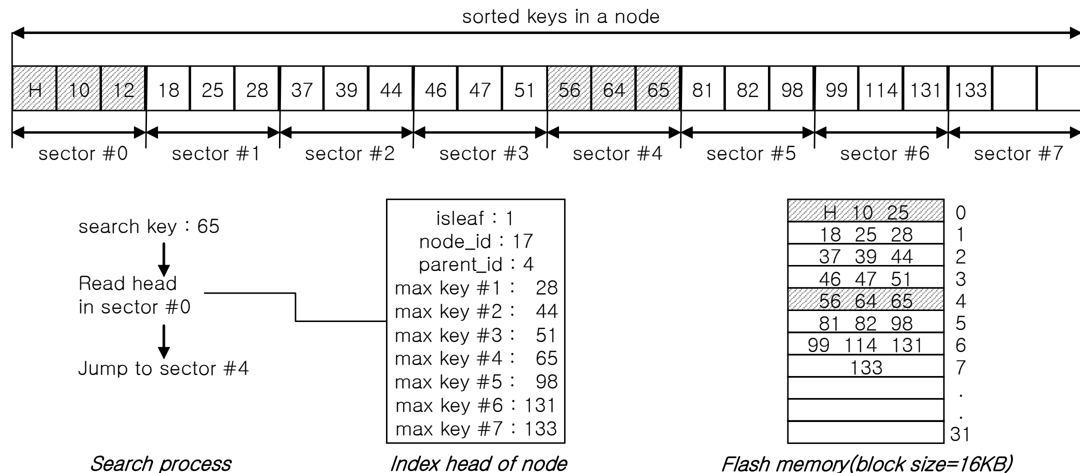
키 삽입을 정렬 알고리즘으로 사용하면 노드 안의 키 값들은 항상 정렬된 상태를 유지하고 있다. 노드가 할당하고 있는 섹터의 개수가 많아질 때 정렬된 노드 안에 있는 키 값을 효율적으로 검색하기 위해서 이분 검색(binary search) 방법을 사용할 수 있다. 그러나 이분 검색을 사용하더라도 노드가 할당하는 섹터의 수가 많아질 때 크게 저하되는 검색 성능을 보완하는 데에는 한계가 있다. 일반적으로 $B^+$-트리 헤드 정보에는 단말 플래그(leaf flag), 노드 식별자(node id), 부모 노드 포인터(parent node pointer)와 같은 기본적인 노드정보가 저장된다. 본 논문에서는 노드가 할당하는 섹터의 개수가 많아질 때 저하되는 검색 성능을 향상시키기 위해 노드 헤드에서 노드가 할당한 각 섹터에 저장되어있는 최대 키 값 정보를 관리한다.

(그림 6)은 노드가 할당하는 섹터의 개수가 8개이고, 정렬 알고리즘으로 삽입된 키 값이 모두 정렬되어 있을 때에 각 섹터의 최대 키 값을 노드 헤드에 저장하는 구조를 나타내고 있다. 즉 하나의 노드가 섹터 0번~7번까지의 총 8개의 섹터를 사용하고 있고 섹터 1번부터 7번까지 저장하고 있는 키 값 중 각각 최대 키 값을 섹터 0번에 있는 헤드에 유지하고 있다. 이러한 헤드 구조를 유지함으로써 검색 시에 향상된 검색 성능을 발휘할 수 있다.

(그림 7)은 이러한 전인덱스 헤드(full index head) 구조를 사용하여 키 값 65를 검색할 때에 요구되는 읽기 연산 순서를 나타낸다. 8개의 섹터를 할당하고 있는 노드 안의 키 값 65를 검색하기 위해서는 먼저 섹터 0번에 저장되어 있는 헤더 정보를 참조한다. 헤더 정보에는 노드가 할당한 각 섹터마다 저장된 최대 키 값이 있으며 이 정보를 통해서 키 값 65가 있는 섹터가 섹터 4번이라는 것을 유추할 수 있다. 따



(그림 6) 전인덱스 헤드 구조



(그림 7) 전인덱스 검색 방법

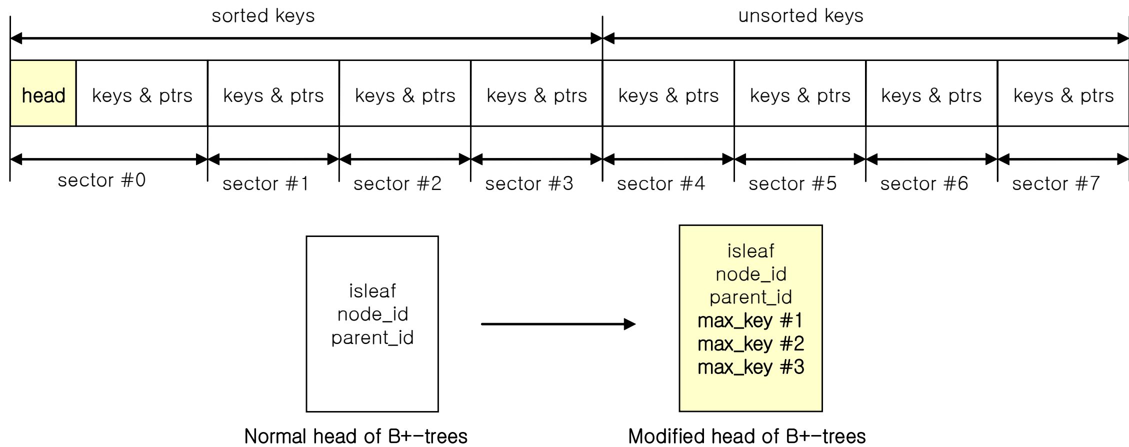
라서 총 두 번의 섹터 읽기 연산으로 키 값이 저장된 섹터를 읽어올 수 있다. 이러한 전인덱스 검색 방법은 노드가 할당하는 섹터의 개수가 32개가 되더라도 최대 두 번의 읽기 연산만으로 노드 안의 원하는 키 값이 들어있는 섹터를 읽을 수 있다.

3.3.2 반인덱스 헤드

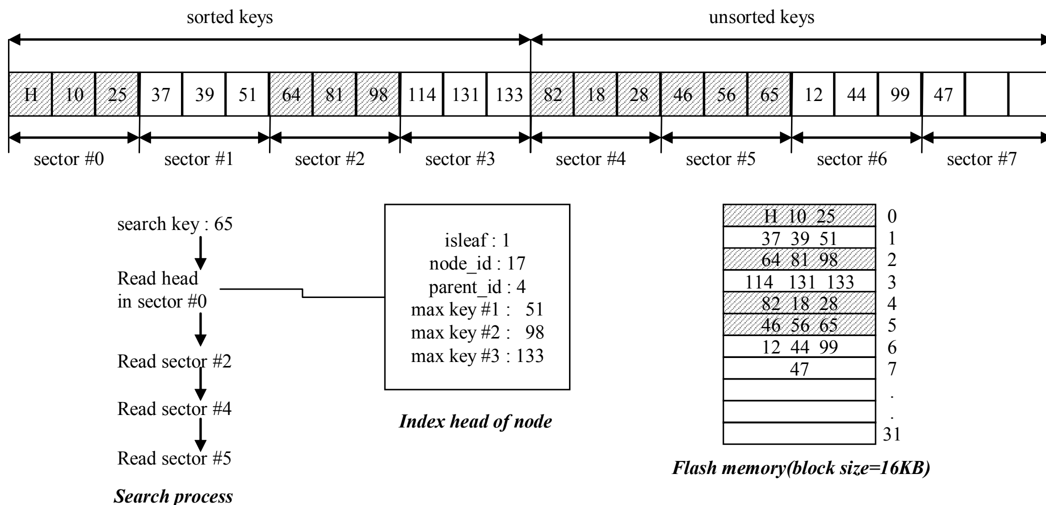
키 삽입을 비정렬 알고리즘으로 사용하면 노드 안의 키 값들은 정렬된 상태와 비정렬된 상태로 나뉜다. 이러한 노드 구조로 저장되어 있는 키 값을 효율적으로 검색하기 위해서 정렬된 부분에 대해서 최대 키 값 정보를 유지하여 인덱스 검색방법을 사용하고 비정렬된 부분의 키 값은 순차적 검색을 적용할 수 있다. (그림 8)은 노드가 할당하는 섹터의 개수가 8개이고, 비정렬 알고리즘으로 키를 삽입했을 때 단말노드에 키가 저장된 상태를 보여준다. 노드의 처음부터 전체 노드의 반 정도 크기의 키 값은 항상 정렬된 상태를 유지하고 나머지 반은 비정렬 상태로 키가 삽입되는 구조이다. 섹터 0번에는 노드의 기본 정보(단말 플래그, 노드 식별자, 부모 노드 포인터)와 함께 섹터 1번~3번이 가지고 있는 최대 키 값을 저장한다. (그림 9)는 반인덱스 헤드 구조를 이용해 노드 안의 키 값 65를 검색하는 순서를 보여준다. 먼저 섹터 0번에 저장되어 있는 헤더 정보를 참조한다. 헤더 정보 안에 반인덱스에서 키 값 65가 있는 섹터를 유추해 섹터 2번을 읽는다. 그러나 섹터 2번 안에는 해당되는 키 값이 존재하지 않으므로 비정렬 상태를 유지하는 섹터 부분에 대해서 순차적으로 검색한다. 따라서 비정렬 부분의 첫 번째 섹터인 섹터 4번을 읽고 섹터 4번에는 키 값이 존재하지 않으므로 그 다음 섹터인 섹터 5번에 대해서 읽기 연산을 수행하며 섹터 5번에서 키 값 65를 검색한다. 제 4절의 (그림 11)과 같이, 반인덱스 검색은 전인덱스 검색 보다 성능이 낮지만 이분 검색 보다는 더 향상된 검색성능을 보여준다.

## 4. 성능 측정 및 비교 평가

본 논문의 실험은 Red Hat Linux release 9(Shrike) 운영체제에서 ANSI C로 구현하였으며 FAST 기법의 FTL을 사




(그림 8) 반인덱스 헤드 구조




(그림 9) 반인덱스 검색 방법

용하였다. 본 장에서는 노드에 할당된 섹터 개수를 증가시킬 때 플래시 메모리에서 발생하는 비용을 측정한다. 플래시 메모리 비용은 읽기, 쓰기, 블록 소거 연산으로 나누며 1:3:200 비율로 처리 시간이 각각 다르다. 또한 한 블록 소거 연산은 십만 ~ 백만 번으로 제한되며 제한된 연산 횟수를 초과할 경우 불량 블록을 생기게 하는 원인이 된다. 따라서 성능향상을 위해서는 읽기연산 감소보다는 쓰기연산, 블록소거연산의 감소가 중요하며, 특히 블록소거연산의 감소는 플래시 메모리에서 불량 블록의 발생을 감소시키는 중요한 역할을 한다.

### 4.1 구축 성능

(그림 10)은 $B^{+}$-트리에 오십만 개의 임의(random) 키 삽입 시 노드 크기에 따라 플래시 메모리에서 발생하는 쓰기연산과 블록소거연산 횟수를 나타낸다. 정렬 알고리즘 사용 시 노드가 섹터 2개를 할당할 때 섹터 1개 일 때 보다 쓰기연산 횟수가 많아지는 것은 섹터 안에 키 값을 정렬시키기 위해서 부가적인 쓰기연산이 발생했기 때문이다. 그러나 노드가 섹터 4개를 할당할 때부터 쓰기연산 횟수가 적어지는 것은 키 값을 정렬시키기 위해 발생하는 부가적인 쓰기연산 패턴이 FTL에서 보완되기 때문이다. 쓰기연산과 블록소거연산은 노드가 섹터 4개를 사용할 때부터 노드가 섹터 32개(블록)를 사용할 때까지 계속 감소하여 결국 섹터 1개를 사용할 때 보다 더 적은 쓰기연산과 블록소거연산 비용을 발생시킨다.

비정렬 알고리즘으로 키를 삽입할 경우 노드가 할당하는 섹터의 개수가 증가하더라도 정렬 알고리즘에서 발생하는 부가적인 쓰기연산이 발생하지 않는다. 또한 노드의 크기가 커질수록 FTL 알고리즘에 더 효율적인 섹터 쓰기연산을 발생시켜 플래시 메모리에서 발생되는 쓰기연산과 블록소거연산을 감소시킨다. 노드가 섹터 1개와 32개를 사용할 때 쓰기연산 감소 비율을 비교해보면 약 50%가 감소하는 것을 알 수 있다. 이러한 쓰기연산 횟수의 감소는 $B^{+}$-트리 구축 시간과 플래시 메모리의 발생 비용을 감소시켜준다. 쓰기연산의 감소는 블록 소거 연산에 영향을 미친다.

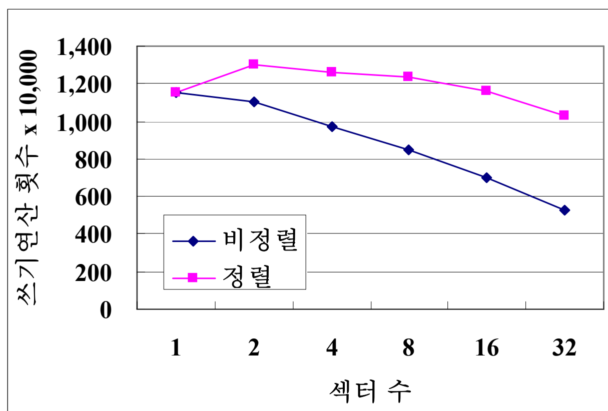
비정렬 알고리즘에서 블록 소거 연산 감소 비율은 약 50%이며 이러한 블록 소거 연산 감소는 불량 블록 발생률을 감소시켜 플래시 메모리의 내구성을 높이게 하는 효과를 기대할 수 있다. 따라서 비정렬 알고리즘에서 노드가 섹터 1개를 사용할 때 보다 블록(섹터 32개) 1개를 사용할 때에 $B^{+}$-트리 키 삽입이 저비용으로 이루어지며 정렬 알고리즘에 비해서도 키 삽입 시 더 우수한 성능을 나타낸다.

### 4.2 검색 성능

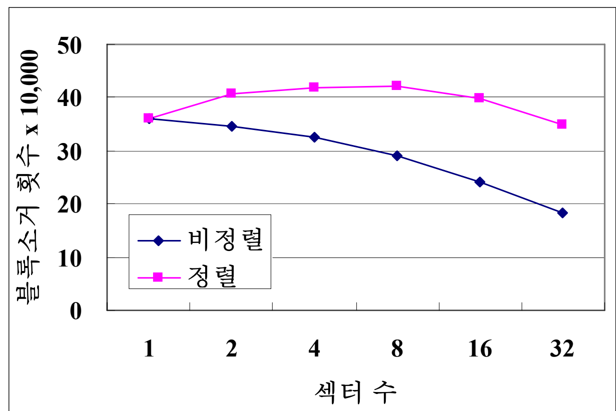
(그림 11)은 노드가 할당하는 섹터의 개수가 증가하면서 $B^{+}$-트리에서 키를 검색할 때 발생하는 플래시 메모리의 읽기 연산 비용을 나타낸다. $B^{+}$-트리의 루트 노드는 키를 삽입할 때나 검색할 때 항상 참조된다. 따라서 루트 노드는 읽기 연산 없이 버퍼에서 참조가 이루어지도록 구현하였다. (그림 11)은 오십만, 백만 개의 키가 삽입된 $B^{+}$-트리에서 하나의 키를 검색하기 위해서 평균적으로 발생되는 플래시 메모리 읽기 연산 횟수를 비교한다. 예를 들어 (그림 11)에서 노드가 섹터 1개를 사용할 때 이분 검색(binary search)으로 하나의 키를 검색하기 위해서는 평균적으로 플래시 메모리에 3번의 읽기 연산을 발생시키는 것을 알 수 있다.

노드가 할당하는 섹터의 개수가 증가할 때 검색 성능을 향상시키기 위해서 정렬 알고리즘으로 키를 삽입한 경우 전인덱스 헤드(full index head), 비정렬 알고리즘으로 키를 삽입한 경우 반인덱스 헤드(half index head) 구조를 사용했으며 검색 방법도 인덱스 헤드 구조에 따라 달라진다.

$B^{+}$-트리의 일반적인 헤드 구조를 사용하여 검색하는 이분 검색 시에는 노드가 할당하는 섹터의 개수가 커질수록 플래시 메모리에서 발생하는 읽기 연산의 횟수가 증가하여 검색 성능이 저하된다. 비록 트리의 레벨이 낮아질 때에는 검색 성능이 약간 향상되지만 레벨이 낮아지지 않으면 다시 성능이 감소하여 결국 검색성능은 노드 크기가 커질수록 전반적으로 감소된다. 이러한 검색 성능을 보완하기 위하여 사용한 반인덱스와 전인덱스에서는 트리의 레벨이 낮아질 때에 검색 성

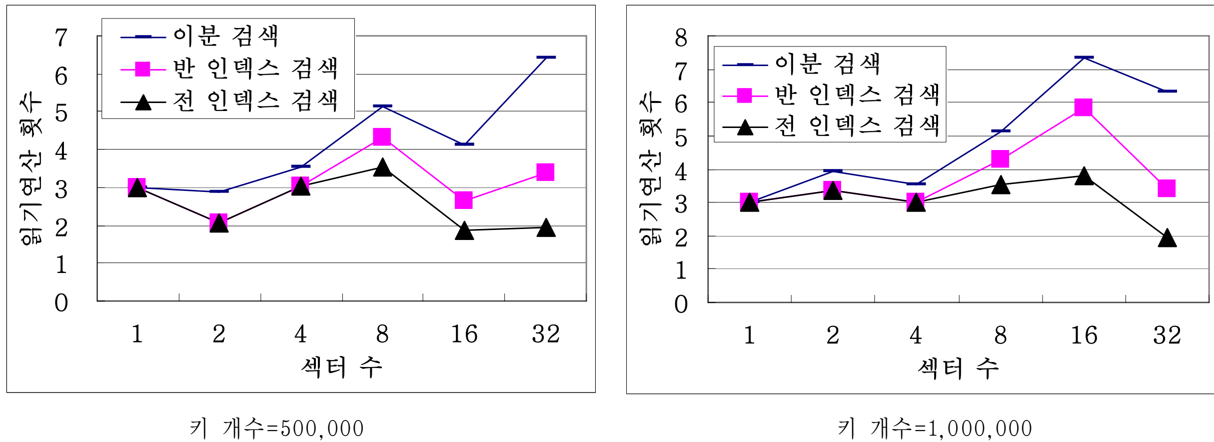



쓰기연산

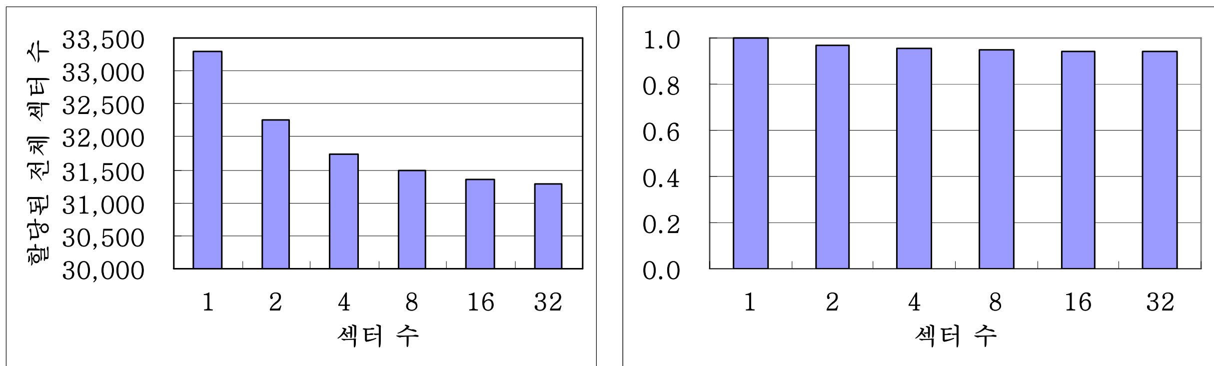



블록소거 연산

(그림 10) $B^{+}$-트리 구축 성능 (키 개수 = 500,000)




(그림 11) $B^+$-트리 검색 성능




(그림 12) 플래시 메모리 사용량

능이 크게 향상되는 것을 알 수 있다. 반인덱스 검색에서 노드가 섹터 1개를 할당할 때와 블록(섹터 32) 1개를 할당할 때에 검색 성능은 비슷한 수준인 것을 알 수 있다. 또한 전인덱스 검색에서는 노드가 블록 1개를 할당할 때가 섹터 1개를 할당할 때보다 검색에서 더 우수한 성능을 나타낸다.

### 4.3 플래시 메모리 사용량

(그림 12)는 노드가 할당하는 섹터의 개수가 증가할 때 플래시 메모리 사용량을 나타낸다. 왼쪽 그래프는 백만 개의 키 값을 $B^+$-트리에 삽입했을 때 플래시 메모리에서 사용하는 전체 섹터 개수를 나타낸다. 그리고 오른쪽 그래프는 왼쪽 그래프를 정규화 시킨 것이다. $B^+$-트리에서 하나의 노드는 일정한 개수 이상의 키를 가지고 있어야 하기 때문에 노드에서 발생되는 공간낭비가 제한된다. 따라서 노드의 크기가 커지더라도 플래시 메모리의 사용량에는 큰 차이가 없는 것을 알 수 있다.

## 5. 결론 및 향후 연구

본 논문에서는 플래시 메모리에 구현되는 $B^+$-트리에서 노드 크기가 $B^+$-트리 성능에 미치는 영향을 비교·분석하였다. 플래시 메모리에 $B^+$-트리를 구축할 때에는 정렬 알고리즘과 비정렬 알고리즘을 사용하였으며 검색성능의 보완을 위하여 전, 반인덱스 헤드 구조를 사용하였다. 이러한 실험은 플래시 메모리에 노드 크기를 섹터 크기로 사용하기 보다는 블록 크기로 사용하는 것이 더 효율적임을 보여주고 있다. 또한 정렬 알고리즘과, 비정렬 알고리즘은 구축하려는 $B^+$-트리의 성향에 따라 선택하여 적용할 수 있다. 검색보다 구축에 보다 더 많은 비중을 두고 있는 $B^+$-트리를 구현하려면 구축비용을 크게 감소시키는 비정렬 알고리즘을 사용하고, 구축보다 검색에 보다 더 많은 비중을 두고 있는 $B^+$-트리를 구현하려면 검색성능을 향상시키는 정렬 알고리즘을 사용할 수 있다.

향후 연구로는 플래시 메모리에 $B^+$-트리를 구축할 때 노드 크기를 블록으로 설계하였을 경우 노드 크기를 고려한 보다 적합한 버퍼 관리, 수용 정책에 관한 연구가 필요하다. 또한 최근의 플래시 메모리는 대용량화 되면서 페이지의 크기가 2KB로 확장되는 추세이다. 이러한 플래시 메모리에서의 적합한 $B^+$-트리 노드 크기에 대한 연구도 필요하다.

## 참 고 문 헌

[1] J.-H. Nam and D.-J. Park, "Design and Implementation

of the B-Tree on Flash Memory," Korea Information Science Society, Vol.34, No.2, pp.109-118, 2007.
[2] C.-H. Wu, T.-W. Kuo and L.-P. Chang, "An efficient B-tree Layer Implementation for Flash Memory Storage Systems," ACM Transactions on Embedded Computing Systems, Vol.6, No.19, July, 2007.
[3] J. Kim, J. M. Kim, S. H. Noh, S. L. Min and Y. Cho, "A Space-Efficient Flash Translation Layer for Compact Flash System," IEEE Transactions on Consumer Electronics, Vol.48, No.2, pp.366-375, May, 2002.
[4] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park and H.-J. Song, "A Log Buffer based Flash Translation Layer using Fully Associative Sector Translation," ACM Transactions on Embedded Computing Systems, Vol.6, No.18, July, 2007.
[5] A. Silberschatz, H.F Korth, S. Sudarshan, "Database System Concepts", 4th Ed., McGraw Hill, New York, 2002.
[6] R. A. Hankins, J. M. Patel, "Effect of Node Size on the performance of Cache-Conscious $B^+$-trees," ACM SIGMETRICS Performance Evaluation Review, Vol.31, No.1, pp.283-294, June, 2003.
[7] E. Gal, S. Toledo, "Algorithms and Data Structures for Flash Memories," ACM Computing Surveys, Vol.37, No.2, pp.138-163, June, 2005.

박 동 주

e-mail : djpark@ssu.ac.kr
1995년 서울대학교 컴퓨터공학과(학사)
1997년 서울대학교 컴퓨터공학과(석사)
2001년 서울대학교 전기전자컴퓨터공학부 (박사)
2001년~2003년 삼성전자 책임연구원
2004년~2005년 숭실대학교 컴퓨터학부 전임강사
2006년~현 재 숭실대학교 컴퓨터학부 조교수
관심분야 : 플래시 메모리, 내장형 소프트웨어, 멀티미디어 데이터베이스 등

최 해 기

e-mail : seastand@gmail.com
2005년 숭실대학교 컴퓨터학부(학사)
2007년 숭실대학교 대학원 컴퓨터학과 (석사)
2007년~현 재 메디슨 연구원
관심분야 : 데이터베이스, 플래시 메모리 기반 소프트웨어, DICOM 등