http://dx.doi.org/10.5392/JKCA.2013.13.09.028

효율적 범위 검색을 위한 플래시 메모리 기반 B+-트리

A Flash Memory B+-Tree for Efficient Range Searches

임성채, 박창섭
동덕여자대학교 컴퓨터학과

Sung-Chae Lim(sclim@dongduk.ac.kr), Chang-Sup Park(cspark@dongduk.ac.kr)

요약

지난 수십 년간 B+-트리는 디스크 기반 데이터베이스를 위한 색인 구조로 가장 널리 사용되고 있다. 디스크 기반 B+-트리에서의 노드 갱신은 해당 노드가 저장된 디스크 페이지를 제자리 갱신함으로써 간단히 수행되며, 이런 제자리 갱신 비용은 크지 않다. 반면에 B+-트리를 플래시 메모리에 저장하여 사용할 때는 플래시 메모리의 과도한 제자리 갱신 비용 문제로 인해 기존 디스크 기반 B+-트리 알고리즘을 그대로 사용하기 어렵다. 이런 이유로 기존 플래시 메모리 기반 B+-트리 연구에서는 실시간으로 발생하는 갱신 연산 정보를 추가적인 임시 공간에 저장하는 방식을 사용하였다. 이런 방식은 B+-트리의 제자리 갱신 횟수를 쉽게 줄일 수 있다는 장점이 있지만 저장 공간의 추가 사용과 키 검색 시간을 지연시킬 수 있다는 문제가 있다. 특히 단말노드 계층의 링크 연결을 사용한 범위 검색을 효과적으로 수행할 수 없다는 문제를 가지고 있다. 이런 문제점을 해결하기 위해 본 논문에서는 단말노드들과 이들의 부모노드를 p-node 블록이라는 하나의 플래시 메모리 블록에 저장할 수 있는 알고리즘을 제안한다.

■ 중심어 : | B+-트리 | 플래시 메모리 | 범위 검색 | 공간 재사용 |

Abstract

During the past decades, the B+-tree has been most widely used as an index file structure for disk-resident databases. For the disk based B+-tree, a node update can be cheaply performed just by modifying its associated disk page in place. However, in case that the B+-tree is stored on flash memory, the traditional algorithms of the B+-tree come to be useless due to the prohibitive cost of in-place updates on flash memory. For this reason, the earlier schemes for flash memory B+-trees usually take an approach that saves B+-tree changes from real-time updates into extra temporary storage. Although that approach can easily prevent frequent in-place updates in the B+-tree, it can suffer from a waste of storage space and prolonged search times. Particularly, it is not allowable to process range searches on the leaf node level. To resolve such problems, we devise a new scheme in which the leaf nodes and their parent node are stored together in a single flash block, called the p-node block.

■ keyword : | B+-tree | Flash Memory | Range Searches | Garbage Collection |

* 이 논문은 2010년도 동덕여자대학교 학술연구비 지원에 의하여 수행된 것임.

접수일자 : 2013년 07월 08일
수정일자 : 2013년 07월 22일
심사완료일 : 2013년 07월 25일
교신저자 : 임성채, e-mail : sclime@dongduk.ac.kr

I. 서 론

플래시 메모리(Flash Memory)는 우수한 충격 흡수력, 작은 전력 사용량, 균일하고 빠른 데이터 접근 속도 등의 장점으로 인해 기존 하드 디스크를 대체할 수 있는 저장 장치로 인식되고 있다[1][2]. 이런 변화에 따라 디스크 기반 DBMS(Database Management System)의 색인 기법인 B+-트리 구조를 플래시 메모리에서도 사용하고자 하는 요구가 있어왔다[3-11]. 하지만 대용량 데이터 저장에 널리 사용되는 NAND 타입 플래시 메모리의 경우 제자리 갱신(in-place update) 비용이 매우 크다는 특징이 있다. 즉, 특정 노드를 저장한 플래시 메모리 페이지(page)를 제자리 갱신하려면 해당 페이지를 포함하고 있는 블록 전체를 초기화하고 이 때 삭제된 다른 페이지들은 재 기록해야 하기 때문에 제자리 갱신 비용이 매우 크다. 이런 이유로 디스크 기반 B+-트리에서 사용하던 갱신 알고리즘을 그대로 사용할 수 없는 문제가 있다[3][6].

기존 연구에서는 플래시 메모리의 과도한 제자리 갱신 비용 문제를 피하기 위해 임의의 B+-트리 단말노드 N에 키 삽입/삭제 연산이 발생할 경우, 노드 N과 떨어진 별도의 갱신 연산 공간에 관련 갱신 정보를 저장하는 방식을 사용한다. 이와 맞춰 B+-트리에 접근하는 다른 키 탐색자들은 하향식(top-down) 검색 과정 중에 이런 갱신 연산 정보를 읽어 자신의 검색 연산에 반영한다[6-11].

이처럼 실제 갱신이 수행되어야 할 단말노드 대신 별도의 공간에 갱신 연산 정보를 두기 때문에 B+-트리에 존재하는 전체 키 값을 단말노드에 저장된 색인 엔트리만으로 알아 낼 수 없다는 단점이 있다. 이로 인해 단말노드를 서로 연결한 단말노드 링크를 사용한 범위 검색(range search)을 수행할 없는 문제가 발생한다. 또한 단일 키 검색에 있어서도 추가 공간을 참조해기 때문에 시간 지연이 생기고, 추가 공간 할당에 따른 플래시 메모리 낭비가 있을 수 있다. 이를 해결하기 위해 논문에서는 단말노드 링크를 사용한 범위 검색을 지원하면서도 B+-트리의 키 검색 시간을 줄이고 공간 사용을 효과적으로 할 수 있는 플래시 메모리 기반 B+-트리를 제안한다.

본 논문의 구성은 다음과 같다. 2장 연구 배경에서는 기존의 연구 및 새로운 아이디어에 대해 기술한다. 3장에서는 제안하는 트리 구조 및 알고리즘을 설명한다. 마지막 4장에서는 제안한 방식의 성능을 분석하고 5장에서 결론을 맺는다.

II. 연구 배경

1. FTL 기반 기법

대용량 데이터 저장에 널리 쓰이는 NAND 타입 플래시 메모리는 데이터 읽기/쓰기의 최소 단위로 페이지를 사용한다. 이런 페이지들의 연속된 모음을 블록(block)이라 하고 블록은 데이터 삭제의 최소 단위로 사용된다. 따라서 임의의 페이지에 저장된 B+-트리 노드를 제자리 갱신하려면 해당 페이지를 포함한 블록을 초기화시킨 후 이 때 삭제된 다른 페이지 내용과 갱신하려고 하는 노드 내용을 모두 재 기록해야 한다. 이처럼 노드의 제자리 갱신을 위해서는 블록 초기화와 여러 번의 페이지 쓰기가 필요하기 때문에 비용이 매우 크다[1][5][6]. 또한 블록 초기화 횟수에 제한이 있기에 플래시 메모리 기반 B+-트리 연구에서는 노드 제자리 갱신을 최소화시키려 하였다.

플래시 메모리 기반 B+-트리를 위한 연구로는 FTL(Flash Translation Layer)에 B+-트리 갱신에 적합한 알고리즘을 내제시키는 방식이 있을 수 있으며, BFTL과 IBTL 방식이 대표적이라 할 수 있다[5]. 하지만 이런 방식은 하나의 플래시 메모리 장치에 B+-트리와 다른 종류의 데이터 파일이 함께 저장되는 경우 성능을 보장하기 어렵다. 더욱이 DBMS에서는 트랜잭션의 복구나 취소를 위해 로깅(logging) 기법을 사용하는데[13][14], DBMS에서 기록되는 로그 데이터와 FTL에 저장되는 B+-트리와 서로 불일치되는 문제가 있을 수 있다. 즉, FTL에 B+-트리 갱신 알고리즘을 내제시킬 경우 DBMS의 로그 기록과 FTL에 반영된 데이터 간에 불일치 문제가 있어 복구 및 취소 기능 구현에 어려움이 있을 수 있다[7-9].

2. 별도 공간 사용 기법

앞서 언급된 이유로 인해 FTL과는 독립적인 플래시 메모리 B+-트리 방식을 연구하는 것이 보다 일반적이라 할 수 있다. 이런 연구들[6-11]에서는 키 삽입/삭제가 발생한 단말노드를 수정하기 위해 제자리 갱신을 수행하는 것이 아니라, 발생한 갱신 연산 내용을 별도 공간에 저장한 후, 키 검색자가 이를 반영한 검색 연산을 하도록 한다. 예를 들어 단말노드 N에 키 k 를 삽입하려 할 때, 실제 노드 N에 키를 삽입하는 것이 아니라 이에 해당되는 연산 정보인 (+, k, N)를 별도의 저장 공간에 기록한다. 여기서 "+"는 키 삽입 연산임을 표시한다. 이런 방식에서는 (+, k, N)와 같은 갱신 연산 정보를 저장시킬 저장 공간 관리의 효율성과 이런 정보에 접근할 때 발생하는 키 탐색자의 검색 시간 지연을 작게 하는 것이 매우 중요하다[3][6][7].

앞서 언급한 갱신 연산 정보는 갱신 노드 N의 조상노드 쪽에 저장되는 것이 일반적이다. 이는 다른 키 검색자가 탐색 과정 중에 쉽게 발견할 수 있다는 장점과 함께, 단말노드 인접한(단말노드에서 접근 가능한) 곳에 저장할 경우 이런 정보들이 플래시 메모리 공간에 매우 흩어져 존재하게 되어 공간 관리에 어려움이 있기 때문이다. 이런 이유로 기존 연구들은 갱신 연산 정보를 단말노드의 상부 계층에 기록해 둠으로써 상대적으로 집중적인 공간 내에 갱신 연산 정보를 관리할 수 있다.

기존 연구 [7]에서는 B+-트리를 상부와 하부 두 계층으로 나누고, 상부 계층에 해당하는 부분 트리에 갱신 연산 정보를 기록한다. 이런 갱신 연산 저장에 사용되는 부분 트리는 생성 시에 모든 노드의 색인 엔트리가 비어 있으며, 갱신 연산이 발생할 때 마다 적절한 탐색 경로상에 위치한 노드로 관련 정보가 삽입된다. 지속적으로 노드 갱신이 발생하여 갱신 연산 정보를 삽입할 노드가 없어지면 일괄 처리 방식으로 아래 계층 부분 트리에 반영된다. 즉, 전체 트리가 재구성되며 이때 상부의 갱신 연산 트리는 다시 초기화된다. 다시 말해 [7]에서는 갱신 연산이 기록될 별도의 부분 트리를 사용하고, 일괄 작업을 통해 갱신 연산을 반영한 B+-트리를 새로 구성한다. 이 방식에서는 키 갱신 연산을 위한 노드의 제자리 갱신이 실시간으로 수행되지 않는다.

관련 연구 [8][9]에서는 DBMS에서 일반적으로 사용하고 있는 노드 버퍼링 공간에 갱신 연산 정보를 저장할 수 있게 한다. DBMS에서는 I/O 횟수를 줄이기 위해 한 번 읽혀진 노드를 버퍼 공간에 두고 이를 재사용 하게 된다. 이를 플래시 메모리 저장 B+-트리에도 적용할 수 있으며, 버퍼 공간에 있는 노드 저장 공간을 플래시 메모리에 저장된 공간보다 좀 더 크게 잡아 둔다. 그리고 이 공간에 갱신 연산 정보를 저장함으로써 플래시 메모리 갱신 횟수를 줄일 수 있게 한다. 이 때 버퍼링 된 노드가 버퍼 공간 부족으로 버퍼에서 삭제될 상황이 되면, 해당 노드에 기록되어 있던 갱신 연산 정보를 부모노드 측으로 올려 저장한다. 즉, 갱신 연산 정보가 상위 계층으로 점차 전이된다고 할 수 있다. 이런 방식을 통해 갱신 연산을 트리의 내부 공간에 유지하며, 상위 계층에 더 이상 갱신 연산 정보를 기록할 공간이 없을 때는 단말노드를 갱신함으로써 버퍼 내 공간에 기록되었던 갱신 연산을 삭제한다.

이와 비슷하게 [10]에서는 갱신 연산 저장을 위한 별도의 공간을 상위 트리 계층에 생성하여 사용한다. 하지만 이런 저장 공간은 [8][9]에서와는 다르게 플래시 메모리 내에서 동적으로 관리된다. 이 공간이 일정 이상 커져 탐색 연산 시간을 크게 지연시킬 상황이 되면 해당 공간에 저장된 갱신 연산 내용을 단말노드로 반영시키고 관련된 플래시 메모리 공간도 함께 삭제된다. [10]에서도 키 탐색자의 검색 속도를 높이기 위해 갱신 연산을 플래시 메모리 공간에 버퍼링 시킬 수 있다. 물론 이런 버퍼링 기법은 갑작스런 정전이나 시스템 고장이 발생하였을 때 트리 일관성이 깨지는 문제를 가진다. 이런 이유로 [11]과 같은 연구에서는 갱신 연산 정보를 별도의 NOR 형 플래시 메모리에 저장하기도 한다. 지금까지 언급한 기존 방식들이 공통으로 취하는 정책은 갱신 연산 정보를 기록하는 공간을 트리의 단말노드 계층이 아닌 그 상위 내부노드 측에 둔다는 것이다.

3. 새로운 아이디어

기존 연구에서 갱신 연산 정보를 실제 갱신이 발생하는 단말노드 인접한 공간에 두지 않고 상위 계층 쪽에 두는 것은 크게 두 가지 이유에서이다. 첫째는 데이터

저장에 필요한 추가 공간의 관리 편의성이다. 만약 개수가 많은 단말노드 측에 갱신 정보를 저장한다면 갱신 정보 저장 장소가 플래시 메모리 상에 매우 흩어져 존재하게 된다. 이로 인해 추가되는 저장 공간을 작게 하기 어렵고, 플래시 메모리 장치가 안고 있는 저장 공간 재사용(garbage collection) 비용이 증가하기 쉽다. 둘째는 갱신 정보를 트리 상부 계층에 둠으로써 키 검색 속도의 저하를 줄일 수 있다는 점이다. 예를 들어 키 k를 찾는 검색자가 단말노드까지 내려오기 전에 내부노드 계층에서 키 k가 삭제되었음을 안다면 중도에 검색을 종료할 수 있을 것이다. 물론 갱신되지 않은 키를 검색 중이라면 갱신 정보를 참조해야 하는 추가 비용이 필요한 것이지만 단말노드 측에 기록하는 경우에 비교한다면 상대적으로 추가 비용이 적다.

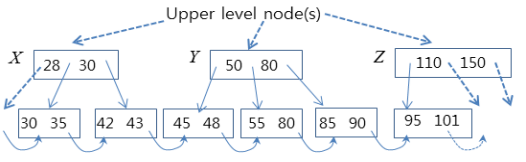
하지만 이런 방식은 갱신 연산 정보가 단말노드 쪽에 없기 때문에 단말노드를 따라 수행하는 범위 검색이 불가능하다는 문제가 있다. 만약 내부 노드를 함께 검색한다면 범위 검색 시에 B+-트리의 동시성(concurrency) 정도가 크게 저하된다. 검색 중에 내부 노드에 잠금을 사용해야 하기 때문에 동시성을 가지고 트리에 접근하는 다른 프로세스를 블록 시키기 때문이다[14][15]. 또한 갱신 연산이 일정 이상 쌓이면 실제 단말 노드 측에 반영시켜야 하는데, 이런 일괄 작업 진행 중에는 트리 검색이 모두 중지되는 단점도 가진다.

논문에서는 이런 문제를 없애기 위해 갱신 연산 정보를 단말노드 인접 공간에 저장하면서도 공간 사용의 효율성을 높일 수 있는 방식을 제안한다. 제안하는 방식은 버퍼링 공간을 최대한 효과적으로 이용할 수 있으며 단말노드의 부모노드를 갱신하지 않고도 B+-트리 일관성을 지킬 수 있다.

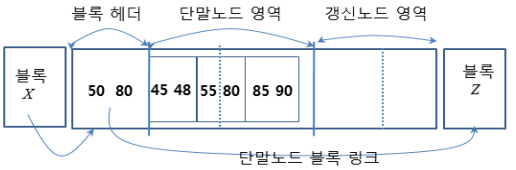
III. 제안하는 플래시 메모리 B+-트리

1. 기본 아이디어

B+-트리가 DBMS에서 사용될 때 버퍼링 공간을 통해 트리 노드가 접근되며, 한번 메모리에 버퍼링 된 노드는 버퍼 공간에서 삭제될 때까지 반복적으로 사용됨



(a) 저장할 B+-트리 구조.



(b) 노드 Y에 대한 단말노드 블록의 저장 구조.

그림 1. B+-트리를 저장하는 P-node 블록 구조 예

으로써 I/O 횟수를 줄일 수 있다[12][14][15]. 본 연구에서도 이런 버퍼링 공간을 통해 플래시 메모리 저장 B+-트리가 접근됨을 가정한다. 또한 트랜잭션의 복구를 위해 기록되는 로깅은 WAL(Write-Ahead Logging) 규약에 따라 수행됨을 가정한다. WAL 규약에 따르면 노드 갱신이 발생할 때 마다 갱신 후의 상태를 항상 플래시 메모리 장치에 저장할 필요는 없다. 갱신된 노드가 버퍼 공간에서 삭제되는 경우에 플래시 메모리 장치에 반영하면 충분하다[14]. 이런 규칙을 고려하여 플래시 메모리 B+-트리 구조를 설계한다.

[그림 1]은 제안하는 플래시 메모리 기반 B+-트리의 저장 구조를 보인다. [그림 1](a)는 저장하려는 B+-트리 일부이며 편의상 단말노드와 그 부모노드 계층만을 표시했다. 제안하는 방법은 동일한 부모노드를 갖는 단말노드들과 이들의 부모노드를 동일한 플래시 메모리 블록에 함께 저장하고, 단말노드의 갱신 내용도 같은 블록 안에 저장하는 방식이다. 이런 블록의 구조를 보이는 것이 [그림 1](b)이며 블록은 블록 헤더, 단말노드 영역, 갱신 노드 영역으로 나뉜다. 논문에서는 이런 플래시 메모리 블록을 p-node 블록이라 부른다. 그리고 설명의 편의상 단말노드의 부모노드 계층을 **p-node 계층**이라 부르겠다.

블록 헤더에는 p-node 계층인 노드 Y의 색인 엔트리가 기록되며 이곳에 Y의 자식노드 주소와 이웃한 형제노드인 노드 Z의 블록 주소가 저장된다. [그림 1](b)의 아래쪽 화살표 링크는 블록 X, Y, Z를 연결된다. 이런

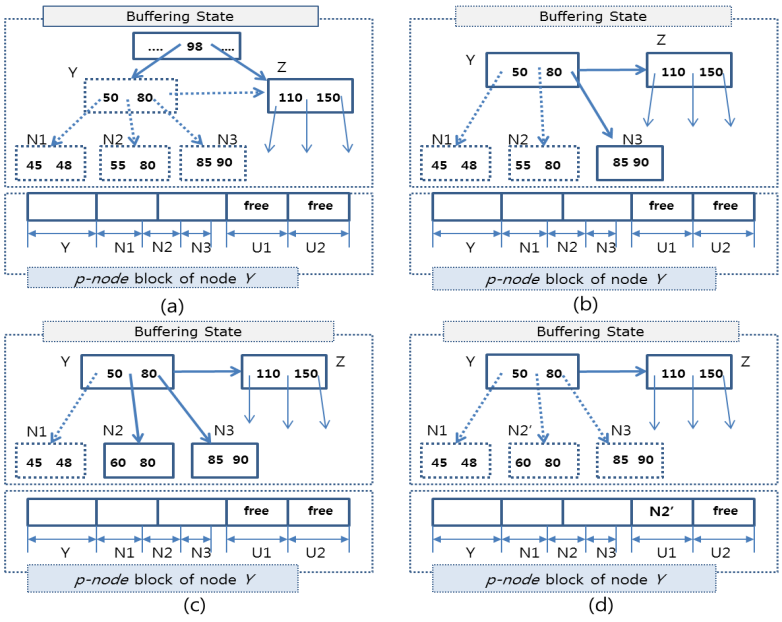


그림 2. 키 탐색 시의 버퍼링 상태와 p-node 블록 저장 데이터 예

링크 정보를 사용하여 범위 검색 시에 이웃한 단말노드들로 이동할 수 있다. 그림의 단말노드 영역을 보면 알 수 있듯이 이곳의 단말노드들은 플래시 메모리 페이지를 경계로 저장되는 것이 아니고 서로 인접하여 저장될 수 있다. 이를 통해 B+-트리 단말노드에 존재하는 내부 단편화를 없앨 수 있다.

[그림 2]는 p-node 블록이 키 검색 시에 어떻게 메모리에 버퍼링 되는지 보인다. [그림 2](a)의 상단부는 버퍼링 상태를 보이는데 점선 표시된 노드는 버퍼링 되지 않았음을, 실선 표시된 노드는 버퍼링 되었음을 표시한다. [그림 2](a)에서는 p-node 계층 노드 중 하나인 Z와 그 부모노드 만이 버퍼링 되었다. 하단부는 노드 Y를 저장한 p-node 블록 내용을 표현한다. 편의상 전체 5개 페이지가 하나의 플래시 메모리 블록을 구성함을 가정했다. 현재 단말노드 영역에 N1, N2, N3 세 개 노드가 저장되어 있고, 갱신 노드 영역에 존재하는 U1, U2는 초기화 상태의 두 페이지를 나타낸다.

이제 키 90을 찾는 탐색 프로세스를 가정한다. 이 탐색자는 트리 탐색 경로를 따라 노드 N3에 접근하게 되는데, 이 과정에서 [그림 2](b)와 같이 Y와 N3가 버퍼링 된다. 키 탐색 후에 다른 두 갱신 프로세스가 트리에 접근하여 노드 N2에 키 60을 삽입하고, 이어서 키 55를 삭제했다고 가정해 보자. 이 경우 두 개의 갱신 연산이 버퍼링 되어 있는 노드 N2에 적용되며 이런 상태는 [그림 2](c)에서 보인다. 앞서 언급했듯이 WAL 규약에 따라 실제 p-node 블록으로 기록되는 시점은 N2가 버퍼링 공간에서 삭제될 때이며, 아직 N2는 버퍼링 된 상태이므로 대응되는 플래시 메모리에는 아무런 갱신이 발생하지 않았다.

마지막으로 임의의 시간이 흐른 후 버퍼 공간 부족으로 노드 N2와 N3가 삭제되는 상황을 가정해 본다. 노드 N3는 갱신 노드가 아니므로 버퍼 공간에서 단순 삭제되면 되지만 갱신 노드 N2는 p-node 블록에 기록된다. 이 두 노드가 버퍼링 공간에서 삭제된 후의 상태가 [그림 2](d)이며 갱신 노드 영역에 속하는 U1 페이지에 N2'가 저장되고, 부모노드 Y도 이를 가리키도록 수정된다. 플래시 메모리의 쓰기 단위는 페이지이므로 N2'는 전체 페이지를 점유하고, 노드 Y에 대응되는 플래시 메모리 영역은 수정되지 않는다. 이에 대한 내용은 이어지는 장에서 설명한다.

2. 제안하는 알고리즘

본 절에서는 앞서 기술한 p-node 블록과 버퍼링 공간을 사용하는 키 검색 알고리즘을 설명한다. 제안하는 방법은 트리의 p-node 계층의 부모노드와 단말노드 계층 자식노드들을 p-node 블록에 저장하고, 나머지 상위 계층의 노드들은 기존 디스크 기반 B+-트리와 같은 방식으로 저장한다. 즉, 상위 계층의 각 노드는 플래시 메모리의 페이지에 대응되고 노드 갱신 시 해당 페이지를 제자리 갱신한다. 이런 방법을 사용하는 것은 대부분의 갱신이 최하위 2개 계층 노드에 한정되며, 그 위쪽 조상 노드들로의 갱신 전이는 드물기 때문이다[3][14][15]. 본 논문은 이런 점을 고려하고 있다.

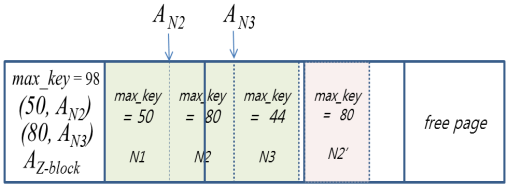


그림 3. P-node 블록의 세부 구조

알고리즘 설명에 앞서 p-node 블록의 저장 내용에 대해 좀 더 자세히 알아보기로 한다. [그림 3]은 앞의 [그림 2](d) 상태에서의 p-node 블록 내용이다. 가장 앞에 존재하는 블록 헤더에는 *max_key*란 필드가 존재하며 자신의 자식노드 들에서 찾을 수 있는 최대 키 값을 나타낸다. 예를 들어 현재 *max_key* 값이 98이므로 단말노드들에 저장되는 키 값은 98을 넘을 수 없다. 이 필드는 단말노드들에도 존재하며 노드에 저장되는 키들의 최대값을 나타낸다. 단말노드들에 저장하는 *max_key* 필드는 기존 논문들에서 범위 검색을 위해 사용되었다[14][15]. 즉, 단말노드 링크를 따라 이동할 때 이 값과 범위 검색 최대값을 비교함으로써 이웃한 단말노드로 계속 이동할지 여부를 결정한다.

범위 검색에 이용된 *max_key* 필드를 본 논문에는 트리 일관성 유지에도 사용한다. [그림 2](d)에서 알 수 있듯이 노드 N2는 갱신되어 N2'로 바뀌었음에도 부모노드 Y에 해당하는 블록 헤더는 수정되지 않는다. [그림 2](c)에서와 같이 Y가 버퍼에 계속 존재한다면 트리 일관성을 지킬 수 있겠지만 버퍼에서 삭제된다면 트리 일관성을 지킬 수 없을 것이다.

이런 문제를 해결하기 위해 논문에서는 p-node 계층의 노드가 버퍼링 되는 상황에서는 대응되는 p-node 블록 전체를 읽어 갱신된 단말노드가 있는지를 체크한다. 이 때 사용되는 필드가 단말노드에 기록된 *max_key* 값이다. 즉, 동일한 *max-key* 값을 갖는 단말노드가 갱신 노드 영역에 존재한다면 가장 오른쪽에 페이지에 저장된 단말노드를 최신 단말노드로 사용한다. 예를 들어 [그림 3]에서 노드 N2와 N2'의 *max_key* 값이 일치하고 N2'가 오른쪽에 위치한 노드이므로 N2'를 최신 노드로 정한다. 따라서 노드 Y를 버퍼링 시킬 때 N2'를 가리키도록 함으로써 트리 일관성을 유지할 수 있다. 이처럼 *max_key* 값을 사용하여 부모노드를 수정하지 않고도 트리 일관성을 유지할 수 있다. 이 밖에도 블록 헤더 부분에는 자식노드에 대한 색인 엔트리가 존재하며, 이웃한 p-node 블록에 대한 주소인 $A_{Z-block}$ 값도 기록된다.

[그림 4]는 p-node 블록을 사용한 키 검색 프로세스의 탐색 알고리즘이다. 알고리즘은 찾고자 하는 단일 키(즉, *kv*)와 B+-트리 루트에 대한 주소(즉, *Root*)가 주어졌을 때, 해당 키를 저장하고 있는 단말노드를 찾아 버퍼에 읽고 이에 대한 메모리 주소 *Leaf*를 반환한다. 설명의 편의 상 찾고 있는 키가 트리에 없는 경우는 고려하지 않았다.

알고리즘에서 라인 8까지는 기존 디스크 기반 B+-트리에서와 같이 하향식 트리 검색을 통해 탐색 경로를 정한다. 이 때 *read_node()* 함수를 통해 자식노드로 이동해 가며, 각 노드는 하나의 플래시 메모리 페이지에 저장되어 있다. 공간 제약 상 상술하진 않지만 버퍼 관리자가 존재하여 일정 크기의 버퍼링 공간 안에 자주 사용될 노드가 관리되며, 버퍼 관리자는 B+-트리의 색인 엔트리에 저장된 노드 주소(혹은 레코드 주소)에 대해 버퍼링 공간 주소의 대응(mapping) 정보를 관리한다[12][15].

이런 하향식 검색이 끝나고 p-node 계층 노드에 접근할 때는 라인 9에서와 같이 해당 노드가 버퍼링 되어 있는지 아닌지를 체크한다. 만약 이미 버퍼링 되어 있다면 *read_node()* 함수를 통해 해당 노드에 대한 버퍼 주소

```
search_key ( kv, Root, Leaf )
{ // function for loading a leaf node containing the key value of kv into a buffering area.
1.  height ← the height of the tree pointed by Root;
2.  Current ← read_node(Root); // Root가 가리키는 노드를 읽고 그 메모리 주소를 Current에 저장
3.  while ( height > 2 )
4.        Check the index entries in Current and find the next child to be moved toward.
5.        Let Next be the node address of the next child found above.
6.        Current ← read_node (Next). // child 노드로 이동
7.        height ← height - 1. // Root 다음 트리 레벨로 이동
8.  end_while. // 단말노드와 그 부모노드를 제외한 트리 검색
9.  Check if the node of Current has been already loaded in the buffering area. // 버퍼링 체크
10. if ( Current has been already buffered ) then
11.       Parent ← read_node(Current ). // 단말노드의 부모노드를 읽음
12. else
13.       Parent ← load_block(Current ). // 플래시 메모리에서 읽어 들임
14. end_if.
15. Check the index entries in Current and find the next child to be moved toward.
16. Let Next be the node address of the next child found above.
17. Leaf ← read_node (Next). // leaf 노드를 읽어 들임.
}
```

그림 4. 키 검색 알고리즘의 의사 코드

를 얻어 탐색을 계속한다. 반면 버퍼링 상태가 아니라면 *load_block()* 함수를 통해 전체 p-node 블록을 읽어 들이고 갱신된 단말노드가 존재한다면 이를 반영하여 p-node 계층 노드의 색인 엔트리를 조정한다. 사용된 함수 *read_node()*와 *load_block()*는 [그림 5]의 알고리즘에서 소개된다.

제안하는 방법은 p-node 블록 헤더에 대해서는 갱신이 발생하지 않기 때문에 p-node 계층 노드를 처음 버퍼로 읽어 들일 때는 p-node 블록 전체를 읽어야 한다. 이런 전체 블록 읽기가 자주 발생한다면 키 검색 시간이 지연될 수 있다. 하지만 단말노드를 제외한 대부분의 노드들은 버퍼링 되어 있는 것이 일반적이기에 실제 이와 같은 전체 블록 읽기는 자주 발생하지 않는다. 예를 들어 B+-트리가 30 정도의 평균 차수(fan-out)를 갖는다면 내부노드 전체가 차지하는 비율은 단말노드의 4% 수준이다. 즉, 전체 노드에서 내부노드는 차지하는 그 비율이 매우 작다. 또한, 버퍼링 공간이 부족할 때에도 단말노드를 버퍼링 공간에서 우선적으로 삭제할 수 있기 때문에, B+-트리의 내부노드가 버퍼링 공간에 존재하는 것이 예외적인 상황은 아니다.

[그림 5]는 앞서의 키 검색 알고리즘에서 사용한 두 개 함수의 알고리즘이다. 두 함수는 모두 버퍼링 공간을 관리하는 버퍼 관리자의 존재를 가정한다. 버퍼 관리자는 버퍼링 공간의 부족 시 특정 노드를 버퍼링 공간에서 삭제할 수 있으며, 삭제할 노드가 갱신된 단말노드라면 해당 p-node 블록의 갱신 노드 영역에 있는 미사용 페

```
read_node ( NodePtr )
{ // function to read the node of NotePtr and return its memory address.
1.  Check if the node of NodePtr exists in the buffering memory.
    // 버퍼 매니저를 통해 특정 노드의 버퍼링 상태를 체크함
2.  if ( NodePtr exists in the buffering memory ) then
3.     Get the memory address that corresponds to NodePrt from
       the buffer manager; and return it.
4.  else
5.     Request that the buffer manager load the node pointed to by
       NodePtr into a buffer area.
6.     Let BufNodePtr be the buffer area address of the loaded node.
7.     Return the address saved in BufNodePtr .
8.  end_if.
}
load_block ( NodePtr )
{ // function to load a p-node block and return its memory address.
9.  From the buffer manager, get the flash memory address
    corresponding to NodePtr; let BlckPtr its flash memory address.
9.  Load the whole flash memory block pointed to by BlckPtr;
    let Blck be the memory area where the block is loaded.
11. Copy the part of the block header in Blck into Parent.
    // Parent가 가리키는 메모리에 p-node 계층노드 엔트리 복사
12. Modify Parent's index entries using the area of updated node pages.
13. Request that the buffer manager save the node of Parent; get the
    buffer address of the saved Parent and let ParentPtr be the address.
14. Return the buffer address of ParentPtr.
}
```

그림 5. 노드 읽기 함수들의 의사 코드

이지에 갱신 노드를 기록한다. 이 중 *read_node()* 함수는 기존의 디스크 기반 B+-트리에서 사용되는 알고리즘과 차이가 없다고 할 수 있다. 이 함수는 *NodePtr*에 저장된 노드 주소를 사용하여 버퍼링 여부를 체크한다. 만약 버퍼링 상태가 아니라면 라인 5에서와 같이 버퍼 매니저를 통해 해당 노드를 버퍼 공간으로 읽어 들이고, 노드가 읽혀진 버퍼 주소를 반환한다.

함수 *load_block()*은 *NodePtr*이 가리키는 노드 주소를 가장 앞쪽 페이지로 하는 p-node 블록을 읽어 들인 후, 갱신 노드 영역에 저장되어 있는 최신의 단말노드 데이터를 반영하는 작업을 수행한다. 이를 통해 갱신된 단말노드가 있다면 버퍼링 될 부모노드의 색인 엔트리는 관련된 갱신 노드 영역의 페이지를 가리키도록 설정된다. 이를 위해 라인 11-12에서는 블록 헤더 내용을 복사하고, 갱신 노드 영역의 노드 정보를 이용하여 최신 단말노드를 결정한다. 이에 대해서는 [그림 3]을 통해 설명되었다. 이렇게 최신 상태의 단말노드를 가리키도록 수정된 p-node 계층 노드는 버퍼 매니저를 통해 버퍼 공간에 저장된다. 버퍼 공간에 저장된 후에는 해당 노드가 위치한 메모리 주소를 반환함으로써 함수가 수행 종료된다. 앞서 언급하였듯이 p-node 계층의 노드는 버퍼 공간상에서 그 내용이 갱신될 수는 있지만, 버퍼 공간에서 삭제되는 상황에서도 관련 내용이 플래시 메모리에는 반영되지 않는다.

3. 기타 알고리즘

기존 B+-트리에서와 큰 차이가 없어 따로 기술하지는 않았지만 범위 검색의 경우는 하향식 검색을 수행한 후 블록 헤더에 저장된 이웃 블록 주소를 사용하여 수행될 수 있다. 즉 [그림 1]의 p-node 블록 간 링크 연결을 따라 이웃한 블록으로 이동함으로써 범위 검색을 손쉽게 수행할 수 있다. 편의상 링크 연결을 한쪽 방향으로만 했지만 양방향 링크도 가능하다.

P-node 블록의 초기화 및 갱신은 두 가지 경우 발생한다. 첫째는 갱신노드 영역의 빈 공간 부족으로 더 이상 갱신 노드를 기록하지 못할 때이다. 노드 갱신이 자주 발생하여 갱신 노드를 추가할 공간이 없을 경우, 해당 블록 전체를 초기화 한 후 갱신 노드 영역에 있는 노드 정보를 단말노드 영역에 반영한다. 이를 통해 갱신 노드 영역을 새로 마련한다. 둘째는 단말노드에 오버플로우나 언더플로우가 발생하는 경우이다. 단말노드에 오버플로우나 언더플로우 발생으로 재구성이 필요할 때는 기존 B+-트리와 같이 키 이동이나 노드 합병을 통해 이를 해결해야 한다. 이 때 트리 재구성을 위해 p-node 블록이 수정된다. 두 경우 모두 p-node 블록이 초기화된 후 최신 상태를 반영하도록 블록이 갱신된다. 블록 합병 및 분할은 기존 디스크 기반 B+-트리 알고리즘의 노드 합병 및 분할 알고리즘을 사용할 수 있다.

IV. 성능 평가

전술한 바와 같이 B+-트리를 플래시 메모리에 두고 사용할 때 고려해야 할 점은 노드 갱신이 필요할 때 제자리 갱신을 수행할 수 없다는 것이다. 제자리 갱신 문제를 피하기 위해 기존 연구에서는 내부노드 영역에 갱신 연산 정보를 저장해 두고 이를 키 탐색자가 참조하여 트리 일관성을 유지하는 방식을 취했다.

이런 방식이 갖는 단점은 추가 공간 사용이 매우 커질 수 있다는 것과 단말노드를 사용한 범위 검색이 가능하지 않다는 것이다. 또한 이런 임시적 공간에 저장된 갱신 정보는 일정 시간 후에는 B+-트리에 반영되어야 하는데 이런 일괄 작업이 수행되는 동안 키 탐색 질의가 처리 될 수 없다는 문제가 있다. 이런 문제들을 피하기 위해서는 갱신이 발생한 단말노드 측에 갱신 연산 정보나 갱신 노드가 저장되는 것이 바람직하다.

본 논문에서는 이런 점을 고려하여 갱신된 단말노드를 갱신 전 상태의 단말노드와 인접한 공간에 함께 저장하고 부모노드에 해당하는 p-node 계층 노드를 갱신 하지 않더라도 트리 일관성을 유지할 수 있는 방법을 취했다. 즉 p-node 계층 노드를 처음 버퍼링 할 때, 대응되는 p-node 블록 전체를 읽어 들여 최신 상태의 단말노드를 결정한다. 이런 특성과 함께 p-node 블록이 제공하는 하나의 장점은 단말노드 영역에 노드를 연속적으로 저장할 수 있다는 점이다. 단말노드의 저장 단위를 페이지가 아닌 바이트 단위로 함으로써 단말노드에 색인 엔트리가

완전히 채워져 있지 않을 경우 발생하는 페이지 내부 단편화를 없앨 수 있다. 이런 장점은 여러 개의 단말노드를 동일 블록에 저장하기 때문에 얻을 수 있는 이점이다. 일반적으로 단말노드 공간에는 최대 50%, 평균적으로 25% 정도의 페이지 단편화가 발생할 수 있고 이를 제거함으로써 플래시 메모리 공간 사용 효율을 높일 수 있다.

키 검색 질의의 처리 시간만을 고려한다면 기존 디스크 기반 B+-트리 구조 그대로 플래시 메모리에 저장하는 것이 가장 유리하다. 이런 점을 고려해서 제안한 p-node 블록 사용 방식이 기존 구조의 B+트리 구조와 비교해 어느 정도의 검색 질의 처리 성능을 보이는지 평가하여 성능을 간접 평가해 본다. 다른 기존 방법[6-11]의 경우는 단말노드를 사용한 범위 검색을 수행할 수 없고, 공간 사용이 상대적으로 큰 관계로 직접적 비교는 하지 않았다.

성능 비교에 있어 [표 1]에 주어진 플래시 메모리 의 물리적 특성을 사용한다[10]. 표에서 읽기 속도의 경우 한 개 페이지를 읽을 때의 속도와 전체 블록을 읽을 때의 속도를 나눠 적었다. NAND 플래시 메모리의 경우 전체 블록 읽기 속도가 개별 페이지 읽기 속도보다 5-7배의 대역폭을 얻을 수 있으며[10], 본 논문에서는 이중에서 보수적인 수치인 5배 정도 속도를 기준 속도로 하였다. 이 경우 표에서 알 수 있듯이 블록 읽기 시간은 페이지 읽기 시간에 비해 13배 정도의 시간 지연을 갖는다.

표 1. 플래시 메모리 성능.

Read		Write	Erase
25ms (2KB, a page)	320ms (128KB, a block)	200ms (2KB, a page)	1,500ms (128KB, a block)

[그림 6]은 기존 디스크 기반 B+-트리와 같이 노드를 하나의 페이지에 저장해서 사용하는 ORG-k와 제안한 방식인 Proposed-k의 키 탐색 질의의 처리 시간을 비교한다. 여기서 정수 값 k는 탐색이 수행되는 B+-트리 높이를 표시한다. [그림 6]의 세로축은 키 탐색 질의의 평균 처리 시간을 나타내며, 가로축은 키 탐색 과정에서 읽어 들일 p-node 계층 노드가 버퍼에서 발견되지 않을 확률을 나타낸다. 일반적으로 단말노드를 제외한 노드들은 버퍼링 되지만 성능 평가에서는 p-node 계층 노드가 버퍼에 존재하지 않을 확률에 따른 키 탐색 시간의 변화를 계산했다.

[그림 6]은 제안된 방식은 p-node 계층의 노드가 캐시되어 있지 않을수록 전체 블록 읽기로 인하여 질의 처리 시간이 증가함을 보인다. 하지만 버퍼 miss 비율이 3% 정도만 되어도 10% 미만의 시간 증가만이 있고, 1% 이하의 miss 비율에서는 4% 이하의 작은 시간 증가가 있다. 앞서 언급하였듯이 p-node 계층의 버퍼링 비율이 매우 높기 때문에 제안한 방식은 ORG 방식과 비교해 거의 차이가 없다고 할 수 있다.

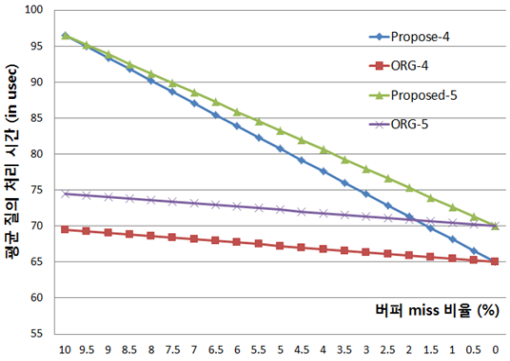


그림 6. 검색 속도 비교 그래프.

[그림 6]의 성능 비교는 B+-트리에 키 삽입/삭제와 같은 노드 갱신 연산이 발생하지 않는 상태에서 질의 처리 시간을 평가한 것이다. 하지만 실제 환경에서는 갱신 연산이 존재하게 되므로 이를 고려한 성능 평가 결과는 그림 7에서 보인다. 그림에서 세로축은 노드 갱신 연산을 포함했을 때의 평균 질의 처리 시간을 나타내고, 가로축은 처리된 질의 중 갱신 관련 질의가 차지하는 비율을 나타낸다. 갱신 관련 질의 비중이 커질수록 쓰기 연산 비용이 상대적으로 큰 플래시 메모리의 특성으로 인해 두 경우 모두 평균 질의 처리 시간이 증가한다.

[그림 7]에서 비교되는 방법의 이름 옆 괄호에 있는 % 비율은 p-node 계층 노드가 버퍼에서 miss될 확률을 나타낸다. [그림 7]에서 5%일 때와 20%일 때를 비교 값으로 사용했으며, 비교되는 두 방법 모두 이 값이 클수록 처리 시간은 증가한다. 특히 제안된 방법의 경우 5%일

때와 20%일 때의 처리 시간 차이가 ORG에 비해 상대적으로 크다. 이는 블록 전체 읽기 횟수가 증가하기 때문이다. 하지만 가로축의 갱신 질의 비율이 커질수록 기존 ORG 방식은 노드의 제자리 갱신 비율이 빠르게 늘면서 평균 질의 처리 시간이 상대적으로 크게 증가한다.

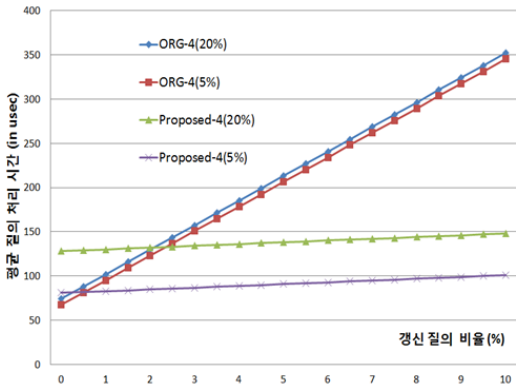


그림 7. 갱신질의가 포함된 성능 비교 그래프

[그림 7]에서 알 수 있듯이 버퍼 miss 비율이 5% 정도인 상황에서는 갱신 연산 질의 비중이 1% 정도일 때도 제안한 방식이 상대적으로 우수한 성능을 보인다. 또한 버퍼 miss 비율이 20%로 매우 높은 상황에서도 갱신 연산 질의가 2%로 넘으면 평균 처리 시간이 서로 동일함을 알 수 있다. 일반적인 환경에서는 버퍼 miss 비율이 이 보다 낮기 때문에 갱신 질의가 처리될 상황이라면 대부분의 환경에서 제안한 방식인 Proposed가 좋은 성능을 가진다고 할 수 있다.

V. 결 론

DBMS의 색인 구조로 널리 사용되고 있는 B+-트리를 하드 디스크가 아닌 플래시 메모리에 저장할 경우 발생하는 문제는 노드 갱신을 제자리에서 할 수 없다는 점이다. 이로 인해 기존 연구들에서는 단말노드를 갱신해야 할 상황에서 해당 노드를 제자리 갱신하는 대신 갱신 연산 정보를 B+-트리의 상위 계층에 저장해 두고, 키 검색 시에 이를 반영하여 질의를 처리하도록 했다. 이런 방식은 기존 B+-트리에서 가능했었던 링크 연결을 사용한 범위 검색을 불가능하게 할 뿐 아니라 추가적인 공간 사용이 크고 검색 시간 지연이 생긴다는 문제가 있었다.

본 논문에서는 이런 문제를 해결하기 위해 p-node라는 플래시 메모리 블록을 사용한다. 이 블록에 부모노드와 이에 딸린 자식노드를 저장하고, 자식노드에 갱신이 발생할 때 그 내용을 갱신 영역 페이지들에 저장한다. 그리고 부모노드를 처음 버퍼로 읽어 들일 때 이런 갱신 노드 정보를 함께 읽어 들임으로써 부모노드의 갱신 없이도 트리 일관성을 유지할 수 있다. 이처럼 부모노드를 갱신할 필요가 없기 때문에 bottom-up 방향의 노드 갱신 전이를 차단할 수 있다. 또한 p-node 블록간의 링크를 사용하여 효과적인 범위 검색이 가능하고 단말노드의 페이지 단편화 문제를 없앨 수 있음으로 해서 저장 공간의 낭비도 줄일 수 있다. 수행된 성능평가를 통해 제안된 방식은 일정 수준의 버퍼링 공간이 있는 환경에서 우수한 성능을 보임을 알 수 있었다.

본 연구에서는 성능 비교를 위해 단순한 확률 모델을 취하고 있다. 이를 확장하여 보다 실제 환경에 근접한 상황에서의 성능 평가가 요구되며, 이를 위해서는 동시성 제어 부분이 추가된 성능 시뮬레이션이 필요하다. 현재는 다른 논문들에서도 이에 대한 고려는 거의 없다고 할 수 있으며, 플래시 메모리 B+-트리가 단순히 stand-alone PC가 아닌 다중 사용자 서버 환경에서 사용되는 것을 고려한다면 반드시 필요한 추가 연구라 할 것이다.

참 고 문 헌

[1] Stepahan Baumann, "Flashing Databases: Expectations and Limitations," In Proc. of DaMon '10, 2010.

[2] Y. K. Wang, Kazuo Goda, and Masaru Kitsuregawa, "Evaluating Non-In-Place Update Techniques for Flash-Based Transaction Processing Systems," In Proc. of DEXA, pp.777-791, 2009.

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

[4] S. G. Moon, S. P. Lim, D. J. Park, and S. W. Lee, "Crash Recovery in FAST FTL," LNCS Vol.6399, pp.13-22, 2011.

[5] 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.3, 2007.

[6] 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 Proc. of ICESS '07, 2007.

[7] D. Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi, "Tree Indexing on Solid State Drives," In Proc. of the VLDB, 2010.

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

[9] 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.

[10] Sai Tung On, Haibo Hu, Yu Li, and Jianliang Xu, "Flash-Optimized B+-Tree," Journal of Computer Science and Technology, Vol.25, No.3, 2010.

[11] Xiaoyan Xiang, Lihua Yue, Zhanzhan Liu, and Peng Wei, "A Reliable B-Tree Implementation over Flash Memory," SAC 08, 2008.

[12] 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.1, pp.351-385, 1996.

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

[14] C. Mohan and F. Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method using Write-ahead Logging," In Proc. of SIGMOD, 1992.

[15] S. C. Lim and M. H. Kim, "Restructuring the concurrent B+-tree with non-blocked search operations," Inf. Sci., Vol.147, No.1-4, 2002.

저 자 소 개

임 성 채(Sung-Chae Lim) 정회원

- 1994년 2월 : KAIST 전산학과 (공학석사)
- 2003년 2월 : KAIST 전산학과 (공학박사)
- 2005년 3월 ~ 현재 : 동덕여자대학교 컴퓨터학과 부교수

<관심분야> : 플래시 메모리 기반 데이터 처리, 고성능 색인기법, 웹 데이터 마이닝

박 창 섭(Chang-Sup Park) 정회원

- 1997년 2월 : KAIST 전산학과 (공학석사)
- 2002년 2월 : KAIST 전산학과 (공학박사)
- 2002년 2월 ~ 2005년 2월 : (주)KT 책임연구원
- 2005년 3월 ~ 2009년 2월 : 수원대학교 인터넷정보공학과 전임강사, 조교수
- 2009년 3월 ~ 현재 : 동덕여자대학교 컴퓨터학과 조교수, 부교수

<관심분야> : 데이터베이스 시스템, 정보 검색, 그래프 데이터베이스, 모바일 데이터베이스