캐시를 고려한 T-트리 인덱스 구조
(Cache Sensitive T-tree Index Structure)

이 익 훈 † 김 현 철 † 허 재 녕 ††
(Ig-hoon Lee) (Hyun Chul Kim) (Jae Yung Hur)
이 상 구 ††† 심 준 호 †††† 장 준 호 †††††
(Snag-goo Lee) (JunHo Shim) (Juno Chang)

요 약 지난 10년간 CPU의 속도는 메모리의 속도에 비해 급속한 속도로 발전하였다. 그 결과 데이타베이스 시스템을 포함한 다른 컴퓨터 응용분야에서 메모리의 접근이 병목현상을 일으키게 되었다. 메모리의 접근 속도를 줄이기 위해 캐시 메모리가 도입되었다. 하지만 캐시 메모리는 원하는 데이타가 캐시에 옮겨져 있어야 메모리 접근 속도를 줄일 수 있다. 때문에 응용프로그램에서 데이타를 어떤 순서로 액세스 하느냐에 따라 캐시의 활용도가 달라지고 응용프로그램의 성능이 달라지게 된다. 이 시점에서 현재 컴퓨터에서 B+-트리가 T-트리보다 더 빠르다는 사실이 알려졌다. B+-트리가 T-트리보다 캐시를 더 효율적으로 사용하기 때문이다. 또한 B+-트리를 개선하여 캐시를 더욱 효율적으로 사용하는 CSB+-트리(Cache Sensitive B+-tree)가 제안되기도 하였다. 본 논문의 목표는 T-트리가 캐시를 효율적으로 사용하도록 새로운 T-트리 구조를 개발 하는 것이다. CSB+-트리와 같이 시스템의 L2 캐시를 최대한 활용하며 기존 T-트리가 가지는 장점을 가지는 새로운 CST-트리(Cache Sensitive T-트리)를 설계 개발하고, 실험을 통해 기타 다른 인덱스 구조에 비교하여 CST-트리의 우수성을 보인다.

키워드 : CST-트리, Cache-Sensitive, 인덱스, 메인메모리, 데이타베이스

Abstract *In the past decade, advances in speed of commodity CPUs have far out-paced advances in memory latency. Main-memory access is therefore increasingly a performance bottleneck for many computer applications, including database systems. To reduce memory access latency, cache memory incorporated in the memory subsystem. but cache memories can reduce the memory latency only when the requested data is found in the cache. This mainly depends on the memory access pattern of the application. At this point, previous research has shown that B+ trees perform much faster than T-trees because B+ trees are more cache conscious than T-trees, and also proposed "Cache Sensitive B+ trees" (CSB+ trees) that are more cache conscious than B+ trees. The goal of this paper is to make T-trees be cache conscious as CSB+ trees. We propose a new index structure called a "Cache Sensitive T-trees (CST-trees)". We implemented CST-trees and compared performance of CST-trees with performance of other index structures.*

Key words : CST-tree, Cache-Sensitive, Index, Main-memory, Database

· 본 연구는 정보통신부의 대학 IT연구센터(ITRC) 지원을 받아 수행되었음
† 비 회 원 : 서울대학교 컴퓨터공학부
ihlee@europa.snu.ac.kr
jjanggu@europa.snu.ac.kr
†† 비 회 원 : ㈜유엔젤 연구원
gura@uangel.com
††† 종신회원 : 서울대학교 컴퓨터공학부 교수
sglee@europa.snu.ac.kr
†††† 정 회 원 : 숙명여자대학교 정보과학부 교수
jshim@sookmyung.ac.kr
††††† 종신회원 : 상명대학교 미디어학부 교수
jchang@smu.ac.kr
논문접수 : 2003년 7월 24일
심사완료 : 2004년 10월 20일

1. 서 론

인덱스 구조란 데이타를 빠르게 찾기 위한 자료 구조이다. 메인 메모리에서 사용 가능한 간단한 구조의 해쉬(hash), 이진 트리, AVL 트리 등과 디스크 기반의 대량의 데이타를 위한 B-트리, B+-트리 등이 만들어 졌다. 80년대에 들어 메모리의 용량이 점점 커지면서 데이타를 디스크에 저장하지 않고 메인 메모리에 저장하는 메인 메모리 데이타 베이스(Main Memory DataBase: MMDB)에 대한 다양한 연구가 이루어 졌고 그 연구들 중 하나로 메인 메모리 데이타베이스를 위한 인덱스 구

조인 T-트리가 고안되었다[1]. 실험을 통해 T-트리가 메인 메모리 데이타베이스에서 가장 우수한 성능을 보이며 가장 적합한 인덱스 구조인 것이 알려졌으며 지금까지 T-트리가 MMDB에서 널리 쓰였다.

그러나 최근 논문[2]을 보면 B-트리의 성능이 T-트리보다 뛰어나다는 사실을 확인할 수 있다. T-트리보다 느리던 B-트리가 갑자기 T-트리보다 빨라진 원인은 급격하게 발전한 CPU 기술에 있다. CPU의 클럭 스피드는 급격하게 빨라졌으나 메인 메모리의 속도는 CPU만큼 빨라지지 않은 것이다. 때문에 기존 CPU - 메모리 - 디스크 사이에서 발생했던 디스크 입출력 병목 현상이 CPU - 캐시 - 메모리 사이에서 메모리 액세스 병목현상으로 바뀌어 나타났다. 임의 접근(random access)과 포인터를 가정하고 디자인된 T-트리보다 디스크 I/O를 최대한 줄이기 위해 고안된 B-트리가 메인 메모리에서도 메모리 액세스 수가 적은 것은 당연하며 CPU와 메모리 사이의 성능격차가 커짐에 따라 두 개의 인덱스 구조 사이에 성능 역전현상이 일어난 것이다.

과거에 우리가 느린 디스크 I/O 속도를 고려하여 자료구조와 알고리즘을 설계했듯이 최신의 컴퓨터에서 최상의 성능을 내기 위해서는 느린 메모리 액세스 속도를 고려하여 자료구조와 알고리즘을 설계하여야 한다. 느린 메모리 액세스 속도를 극복하기 위해서는 CPU 와 메모리 사이의 속도 격차를 극복하기 위해 만들어진 캐시 메모리를 적절하게 이용하여야 한다. 단, 디스크 기반의 데이타베이스(Disk Resident DataBase : DRDB)와 MMDB의 차이점이 있다면 메모리의 데이타들은 CPU 명령어집합(Instruction Set)을 통해 우리가 자유자재로 제어(control) 할 수 있기 때문에 DRDB의 최대 성능을 끌어 낼 수 있었던 것에 비해 캐시 메모리의 데이타들은 우리가 컨트롤 할 수 있는 방법이 없기 때문에 MMDB의 성능을 높이기가 어렵다는 것이다. 하지만 캐시의 특성이나 캐시의 대체(replacement) 정책을 고려하여 조금만 주의 깊게 프로그래밍을 하여도 캐시의 영향으로 성능이 높아진다는 사실이 알려졌고 T-트리보다 B-트리가 캐시의 이점을 잘 살리고 있다는 것도 알려졌으며 캐시를 고려한 새로운 B-트리인 CSB+-트리도 고안되었다[3].

본 논문은 CSB+-트리의 접근방법처럼 캐시를 최대한 잘 이용할 수 있는 T-트리에 대해 연구를 하였고, 그 결과 CST-트리(Cache Sensitive T-트리)를 개발하였으며 CST-트리가 CSB+-트리보다 더 성능이 뛰어남을 보인다.

본문은 다음과 같이 구성 되어 있다. 2장에서 캐시와 인덱스 구조에 관련된 연구를 소개하며 3장에서 T-트리가 B-트리보다 성능이 낮은 이유를 분석하고 CST-트리의 설계와 검색, 삽입, 삭제, 밸런싱(balancing) 알고리즘에 대해 설명한다. 4장에서는 CST-트리와 기타 다른 트리와의 성능을 비교 분석하며, 5장에서 본 논문의 결론을 내리고 향후 과제에 대하여 기술한다.

2. 관련연구

2.1 캐시

캐시메모리는 적은 용량이지만 빠른 속도를 가지는 메모리로 최근에 사용되었던 데이타를 저장해서 프로그램의 성능을 높이는데 사용된다[4]. 캐시는 크게 용량(capacity), 연관성(associativity), 블록크기(block size) 세가지 요소를 가지고 있다. 용량은 캐시메모리의 크기이며 블록크기는 메모리에서 캐시메모리로 전송하는 기본 블록 크기이다.

CPU가 필요로 하는 데이타가 캐시메모리에 있을 경우 캐시 히트(cache hit)라 하며 캐시메모리에 데이타가 없을 경우 캐시 미스(cache miss)라 한다. 캐시 미스가 발생할 경우 메모리로부터 데이타를 캐시로 전송해야 하기 때문에 약간의 시간이 소요된다. 최근의 컴퓨터는 대부분 L1, L2라 불리는 두 개의 캐시메모리를 가지고 있다. L1 캐시는 CPU의 클럭과 비슷한 속도로 동작하지만 캐시메모리의 용량과 블록 크기가 매우 적다. L2 캐시는 메모리보다 빠르나 L1캐시보다 약간 느린 속도로 동작한다.

캐시 히트가 빈번하게 발생해야 성능이 좋은 건 당연하다. 이 캐시 히트의 발생빈도는 캐시의 대체 정책, 프로그램의 자료구조, 프로그램이 데이타를 액세스하는 패턴에 밀접한 관련이 있고, 주의해서 프로그램을 작성하지 않으면 캐시메모리를 효율적으로 사용하지 못한다.

[5]는 현대 컴퓨터에서 CPU와 메모리의 성능차이에 의해 메모리가 데이타베이스의 새로운 병목현상의 원인이 됨을 보이고 캐시를 고려한 프로그램 설계 구현으로 메모리의 병목 현상을 완화시킬 수 있음을 보였다.

현시점에서 널리 사용되고 있는 CPU의 경우 L1의 데이타를 읽는데 2 cycle 정도, L2의 데이타를 L1으로 읽어오는데 6~10 cycle 정도 필요한데 비해 메모리의 데이타를 L2로 옮기는데 30~100 cycle이나 소요된다. CPU의 클럭이 높아질수록 그 격차가 늘어나는 만큼 캐시의 중요성은 아무리 강조해도 지나침이 없을 것이다.

2.1 인덱스 구조

많이 사용되는 인덱스 구조로 해쉬, AVL 트리, B-트리, T-트리 등이 있다. AVL 트리[6]는 메인 메모리에서 사용하기 위해 설계된 인덱스 구조이다. AVL 트리는 이진 트리의 일종으로 AVL 트리의 각 노드는 데이타와 왼쪽 자식 노드의 포인터, 오른쪽 자식 노드의 포인터를 가지고 있으며 트리의 균형 유지를 위한 컨트롤

(control) 값을 가지고 있다. AVL 트리의 왼쪽 자식 노드의 데이타는 부모 노드의 데이타보다 작으며 오른쪽 자식 노드의 데이타는 부도 노드의 데이타보다 커야 한다. 또 왼쪽 자식 노드와 오른쪽 자식 노드의 높이 차는 1 이하여야 한다. 노드 삽입이나 노드 삭제로 인해 왼쪽 자식 노드와 오른쪽 자식 노드의 높이 차가 2 이상이 될 경우에는 회전 알고리즘(rotation algorithm)을 적용해 높이 차가 1 이하가 되도록 트리 밸런스를 재조정한다. AVL 트리의 회전 알고리즘은 총 네 가지가 있으나 서로 좌우 대칭이므로 LL, LR 두 개의 알고리즘만 소개한다. LL은 왼쪽 자식 노드의 왼쪽 자식노드 높이가 높을 때 사용되는 것이고 LR은 왼쪽 자식 노드의 오른쪽 자식노드 높이가 높을 때 사용되는 알고리즘으로 그림 1의 b)와 c)에 설명되어 있다.

AVL 트리의 단점은 노드가 하나의 데이타만을 가지기 때문에 공간을 비효율적으로 사용하며 밸런스를 맞추기 위해 회전 연산이 자주 수행된다는 점이다.

B-트리[7]는 디스크에서 사용하기 위해 설계된 인덱스 구조로 대부분의 DBMS는 B-트리나 B-트리를 개량한 B+-트리를 사용하고 있다. B-트리는 입출력이 느린 디스크를 고려하여 만들어 졌기 때문에 검색하는데 AVL 트리 보다 적은 액세스 횟수가 필요하다. 또한 삽입이나 삭제 시에 트리 재조정의 오버헤드가 AVL트리보다 적다.

T-트리[1]는 AVL 트리의 공간 낭비와 잦은 회전연산을 개선하기 위해 만들어진 인덱스 구조이다. AVL 트리가 하나의 노드에 데이타 한개만을 가지는 대신 T-트리는 하나의 노드가 n개의 데이타를 가질 수 있도록 자료구조를 변경하였다(그림 2). AVL 트리와 유사하게 T-트리 노드의 왼쪽 부분 트리에 있는 모든 데이타는 노드의 최소값보다 작고, T-트리 노드의 오른쪽 부분 트리에 있는 모든 데이타는 노드의 최대값보다 커야 한다. 또 T-트리는 노드가 추가되거나 삭제되었을 때 AVL 트리와 동일한 알고리즘을 사용하여 트리의 밸런

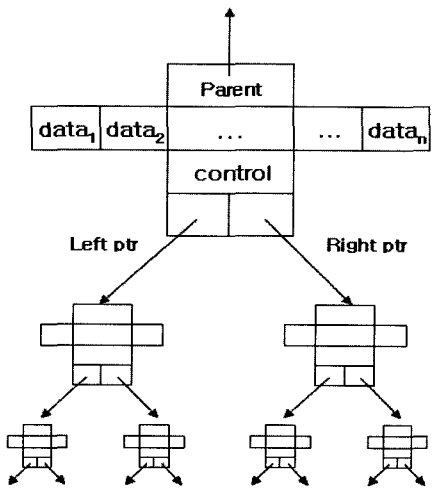


그림 2 T-트리 노드 구조

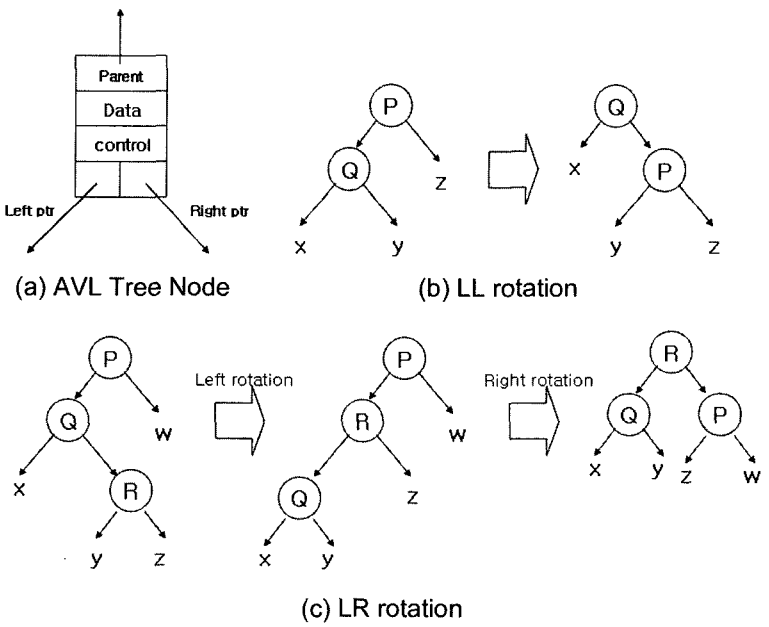


그림 1 AVL 트리

스를 유지한다. [1]에서는 AVL 트리, Hash, B-트리, B+-트리, T-트리와의 성능비교를 통해 메인 메모리 데이타베이스에서 T-트리가 최적임을 보였다.

[2]는 T-트리보다 B-트리가 캐시를 효율적으로 사용하고 있음을 분석하고 B+-트리의 노드 사이즈가 캐시 블록 사이즈와 동일할 때 최적의 성능이 나옴을 실험을 통해 보였다. 그리고, 캐시의 이점을 최대한 살리는 CSS(Cache Sensitive Search)-트리를 제안하였다. CSS-트리는 포인터를 완전히 제거한 배열 형태의 B-트리다. 하지만 CSS-트리는 캐시 미스 빈도가 매우 낮지만 삽입이나 삭제가 안 되는 트리라는 점에서 한계를 가지고 있다. 또한 키로 4바이트의 정수만을 사용해야 하는 것도 CSS-트리의 한계점 중의 하나이다.

[3]은 CSS-트리를 개선하여 삽입이나 삭제가 가능한 일반적인 CSB+-트리를 제안했다. CSS-트리가 포인터가 하나도 없었던 반면, CSB+-트리는 B+-트리의 노드에서 자식 노드들로 나가는 포인터들을 한 개의 포인터만을 사용하여 표현하였다. 이러한 포인터 제거 방법이 본 논문에서도 사용되었다. CSB+-트리는 이렇게 포인터를 사용하게 됨으로써 삽입, 삭제가 가능한 일반적인 트리가 되었다. 하지만 여전히 4byte의 정수 만을 키로 사용해야 하는 한계가 있었다.

[8]은 부분 키(partial key)와 부분 키 간의 비교연산을 정의하고 B+-트리와 T-트리에 이 부분 키를 적용하였다. 부분 키를 이용하면 CSB+-트리도 일반적인 트리로 바꿀 수 있으며 캐시 미스 없이 부분 키 간의 크기 비교를 할 수 있다.

3. CST-트리

3.1 T-트리가 캐시를 효율적으로 사용하지 못하는 이유

T-트리가 캐시를 효율적으로 사용하지 못하는 이유는 T-트리 검색 알고리즘의 데이타 액세스 패턴에 있으며[2], 그 원인은 다음과 같다.

첫번째, T-트리는 B-트리보다 캐시 미스가 많이 발생한다. T-트리의 높이는 B-트리의 높이보다 높기 때문에 트리의 단말 노드까지 도달하는데 포인터에 의한 메모리 액세스가 B-트리보다 많다. 포인터에 의한 메모리 액세스는 대부분 캐시 미스를 유발하므로 T-트리가 B-트리보다 캐시 미스가 많다.

두번째로 T-트리는 사용하지 않는 데이타를 캐시로 옮기는 경우가 많아 캐시를 비효율적으로 사용한다. 메모리의 특정 주소를 액세스할 경우 캐시 블록 크기 만큼의 데이타가 메모리에서 L2 캐시로 옮겨지게 된다. T-트리는 메모리 액세스에 의해 T-트리의 노드가 캐시에 옮겨진다 하더라도 노드의 최소값과 노드의 최대값만을 찾고자 하는 키 값과 비교한 후에 포인터를 따라 다음 노드를 액세스 하게 되므로 캐시에 옮겨진 데이타 중 최소, 최대를 제외한 값은 사용되지 않는다. 반면, B-트리는 노드가 캐시에 옮겨지면 이진 탐색(binary search)을 통해 절반 정도의 키 값들을 비교에 이용한다.

[2,3]의 실험과 위의 분석으로, 성능 향상을 위해 다음 사항들을 고려해야 함을 알 수 있다.

- 캐시에 옮겨지는 데이타의 가능한 많은 부분을 사용한다.
- 포인터의 사용을 최대한 없앤다.
- 트리의 노드 사이즈가 캐시 블록 크기일 때 최고의 성능을 보인다.

3.2 CST-트리

위에서 얻은 세 가지의 결론으로부터 CST-트리(cache sensitive T-트리)를 설계하였다.

• 캐시에 옮겨지는 데이타의 가능한 많은 부분을 사용

결국 T-트리 검색에 제일 중요한 것은 노드의 최소값과 최대값이다. [1]을 참조하면 최대값만을 사용해서 검색을 하는 알고리즘도 제시되어 있다. 따라서, 이 최대값들만을 취해 트리를 구성하며 캐시 미스가 일어 났을 경우 이 최대값들이 최대한 캐시 메모리에 복사 되도록 한다.

• 포인터의 사용을 최대한 제거

보통 힙(heap) 트리을 구현할 때 사용하는 방법으로 포인터를 사용하지 않고 이진 트리를 구성하는 방법으로 배열을 사용하는 방법이 있다[9]. 배열에서 루트노드의 인덱스를 1이라 하면 부모노드와 자식노드 사이에는 다음 규칙이 성립한다.

- 부모노드의 인덱스 = 인덱스 / 2
- 왼쪽 자식노드의 인덱스 = 인덱스 * 2
- 오른쪽 자식노드의 인덱스 = 인덱스 * 2 + 1

이 규칙을 이용하면 포인터를 사용하지 않고 자식노드를 액세스하는 것이 가능하므로 T-트리의 최대값들을 모아 배열로 이진트리를 구성한다.

• 노드블록 사이즈가 캐시 블록 사이즈 일때 가장 효율적이다.

위의 배열 트리는 '인덱스 * 2 + 1 < 캐시 블록 크기' 인 경우에는 자식 노드 역시 캐시블록에 카피되어 있기 때문에 캐시 미스를 일으키지 않고 자식 노드를 액세스할 수 있다. 반대의 경우 자식 노드를 액세스 하는데 항상 캐시 미스가 발생하게 되므로 포인터를 사용하는 것과 비교하여 큰 이점이 없다. 그렇기 때문에 트리의 높이가 높아지는 경우 부득이 포인터를 사용하여 트리를 분할한다. 캐시 블록 크기가 64바이트인 경우 4바이트인 정수 키 값 15개가 높이 4의 완전이진트리(complete binary tree)를 구성 할 수 있고 캐시 블록 사이즈가 128바이트인 경우 키 값 31개를 모아 높이 5

의 완전 이진트리를 만들 수 있다. 이렇게 T-트리 노드의 최대값을 캐시 블록 크기만큼 모아 만든 이진트리 블록을 '노드 블록'이라고 부르기로 한다. 노드 블록 내에서는 캐시 미스 없이 자식 노드의 최대값을 읽을 수 있다.

노드 블록과 노드 블록 사이의 연결은 포인터를 이용한다. 또한 노드 블록 내의 키 값들은 실제 노드의 최대값을 모은 것이므로 최대값마다 실제 노드로 가는 포인터가 필요하다. 이 CST-트리의 구조를 도식화 한 것이 그림 3이다(노드 블록 포함되는 노드의 수가 간략히 표시되었다. 캐시 블록 크기가 16 바이트인 경우이다).

그림 3의 CST-트리는 총 7개의 포인터를 가지고 있으며, 노드 블록내에서 28바이트를 차지한다. 일반적인 캐시메모리의 블록 크기가 64바이트인 것을 감안하면 포인터가 차지하는 공간이 매우 크므로, 캐시 블록에 노드 전체를 복사할 수 없게 된다. [3]에서 제안한 포인터 제거 방법을 사용하면, 그림 4에서처럼 단 2개의 포인터만이 필요하게 되며 상당한 공간을 절약할 수 있다. 그림 4를 보면 하나의 노드 블록에 달리는 자식 노드의 개수는 4개이다. 이 자식 노드의 공간을 할당 받을 때 각각을 할당 받게 되면 4개의 포인터가 필요하지만, 인덱스의 개수가 4인 노드 블록의 배열로 공간을 할당 받으면 포인터가 하나만 있어도 하나의 포인터와 배열의 인덱스로 각각의 자식 노드 블록을 액세스할 수 있다.

그림 5는 CST-트리의 노드 블록을 정의하는 코드이

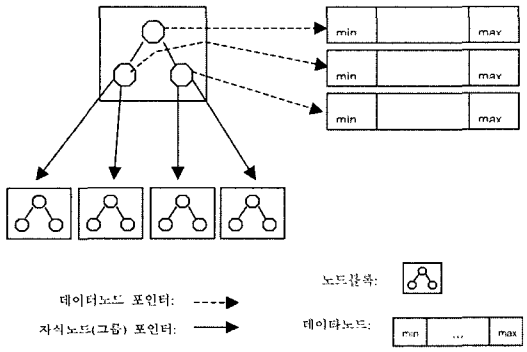


그림 3 CST-트리의 논리적 구조

```
struct TNodeBlock
{
    int mkeys[m];
    struct TNodePtrAncCtrl *ptr;
}
struct TNodePtrAndCtrl
{
    struct TNodeData *nodeData[m];
    int height;
    struct TNodeBlock *parent
    struct TNodeBlock *child;
}
```

그림 5 노드 블록 자료 구조

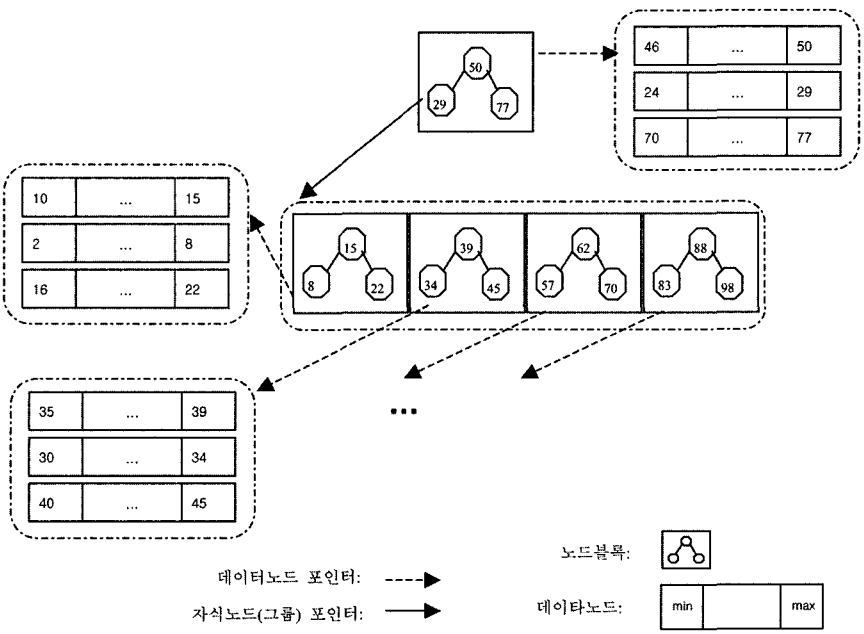


그림 4 CST-트리의 구조

다. m은 캐시 블록 크기에 의해 결정되는 되는 것으로 다음 방법으로 계산된다.

- 노드 블록 이진트리의 높이

$$d = \left\lfloor \log_2 \frac{CacheBlockSize}{sizeof(\text{int})} \right\rfloor$$

- 노드블록에 포함 되는 키의 개수 $m = 2^d - 1$
- child의 크기 : sizeof(TNodeBlock) * 2^d
- nodeData의 크기 : sizeof(TNode) * m

CST-트리의 기본 연산은 다음과 같이 정의된다.

- mkeys[i]의 왼쪽 자식 노드

```
if i * 2 +1 < m then return mkeys[i*2+1]
else return child[i*2+1-m].mkeys[0]
// 다음 노드블록의 root로 넘어간다.
```

- mkeys[i]의 오른쪽 자식 노드

```
if i * 2 +2 < m then return mkeys[i*2+2]
else return child[i*2+2-m].mkeys[0]
// 다음 노드블록의 root로 넘어간다.
```

- mkeys[i]의 실제 노드

```
return nodeData[i]
```

3.3 검색 알고리즘

검색 알고리즘은 최대값만을 사용하기 때문에 일반적인 T-트리와 약간의 차이점이 있으나. 최대값만을 사용하여 검색하는 알고리즘은 [1]에 기술되어 있다. 검색 알고리즘을 그림 6에 간단히 기술하였다.

검색을 진행하는 동안 노드 블록 내에서 트리를 탐색할 때는 캐시 미스가 발생하지 않고, 다음 노드 블록을 액세스(access)할 때만 캐시 미스가 발생한다. 검색 알고리즘의 시간복잡도는 4.6.1에 제시한다. 또한, 검색알고리즘의 성능은 4.2절에 제시한다.

3.4 삽입 알고리즘

트리 밸런싱 알고리즘을 제외하면 T-트리와 유사하며 다음에 알고리즘을 간략히 기술하였다.

- 검색을 통해 삽입할 노드를 찾음
- 삽입할 노드를 찾았고 삽입 가능하면 삽입
- 삽입할 노드를 찾았으나 삽입할 공간이 없으면 최소값을 빼고 그 자리에 삽입한다. 최소값을 현재 노드의 왼쪽 서브 트리에 삽입한다
- 삽입할 노드가 없으면 노드를 추가
- 노드가 추가된 단말 노드 블록에서 밸런스 검사
- 노드 추가로 새 노드 블록이 생성되었을 경우 트리 밸런스를 검사

※ 추가되는 노드는 항상 단말노드

트리 밸런싱 알고리즘은 3.6절에 제시하며 트리 밸런싱 알고리즘의 분석은 4.6.2절 에서 하였다. 더불어, 삽입 알고리즘의 성능은 4.3절에 제시한다.

3.5 삭제 알고리즘

트리 밸런싱 알고리즘을 제외하면 역시 T-트리와 유사하며. 다음에 그 알고리즘을 간략히 기술하였다.

- 삭제할 값을 찾아 삭제
  - 삭제로 인해 노드에서 언더플로우(underflow)가 발생할 경우 왼쪽 서브 트리에서 가장 큰 값을 삭제하고 현재 노드에 삽입
  - 왼쪽 서브 트리가 없을 경우 그냥 삭제
  - 삭제가 일어난 노드가 반단말(half-leaf)이거나 단말일 경우 근접 노드와 합병 가능
- 삭제로 인해 노드가 없어진 경우 단말 노드 블록내에서 밸런스 검사
- 삭제로 인해 노드가 없어지고 그로 인해 노드 블록이 없어지게 되면 트리 밸런스를 검사

※ 삭제되는 노드는 항상 단말 노드

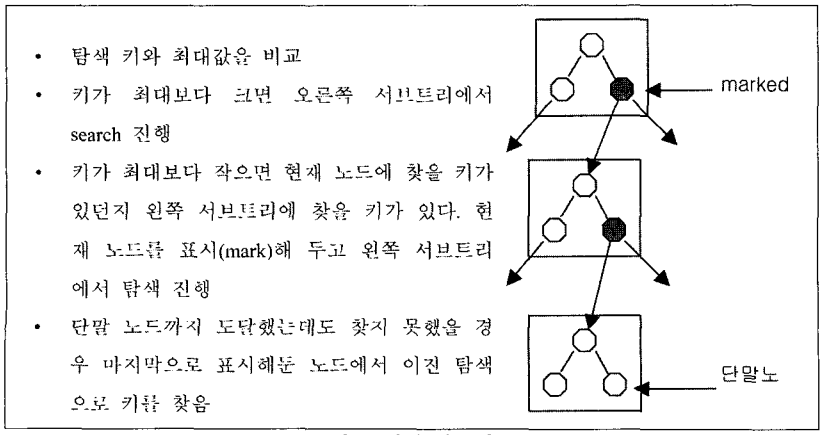


그림 6 검색 알고리즘

3.6 트리 밸런싱 알고리즘

CST-트리는 크게 보면 두 개의 트리 형태가 존재한다. 노드 블록 내에 포함되는 배열로 된 이진 트리와 노드 블록들로 이루어지는 m-ary 형태의 트리가 존재한다. 따라서, AVL이나 T-트리에서 사용하는 트리 밸런싱(Tree Balancing)알고리즘을 무작정 적용하는 것은 불가능하다.

검색 과정에서 성능에 가장 크게 영향을 미치는 것은 캐시 미스이다. 따라서 캐시 미스를 발생하게 하는 노드 블록 간의 밸런싱에 더욱 신경을 쓸 필요가 있다. 이 노드 블록 간의 트리 밸런싱은 AVL 트리의 알고리즘을 변형하여 고안하였으며 3.6.2절에 제시하였다.

노드 블록 간의 밸런싱 만큼이나 노드 블록 내의 이진 트리 밸런싱 알고리즘도 중요하다. 한 노드 블록 안에 최대한 많은 수의 노드가 포함되어 있어야 검색이 빨라지기 때문이다. 노드 블록 내의 이진 트리 밸런싱 알고리즘 역시 AVL 트리 알고리즘을 이용하며 3.6.1에 제시하였다.

3.6.1 노드 블록 내의 이진 트리 밸런싱 알고리즘

배열 이진 트리이기 때문에 포인터를 이용한 일반적인 LL, LR 회전 알고리즘을 사용할 수 없다. 하지만 노드 블록 내의 이진 트리는 캐시 블록 크기에 의해 그 크기가 제한 되어 있으므로 밸런스가 깨질 때 마다 배열의 인덱스를 재배열하는 방법으로 밸런싱을 조절한다. 노드 블록 내의 이진 트리는 한 번에 캐시 블록으로 복사가 되므로 밸런싱 작업을 하는 중에 추가적인 캐시 미스는 발생하지 않는다.

3.6.2 노드 블록 간의 밸런싱 알고리즘

1) 기본 회전 알고리즘 (Basic I to J rotation)

I to J 회전은 노드 블록 P의 i번째 노드 블록을 노드 블록 P의 j번째로 옮기는 회전 알고리즘(rotation - algorithm)이다. 그림 7은 회전이 일어나기 전과 회전이 일어난 후의 노드 변화를 보여 주고 있다. i가 j보다 클 경우 알고리즘은 다음과 같다(i가 j보다 작을 경우는 좌우 대칭이므로 생략한다).

- 노드 블록 P와 노드 블록 Q의 노드들을 이용해 P와 Q의 트리를 재구성하여 그 결과로 노드 블록 P'과 노드 블록 Q'이 만들어 진다.
- Q(Q')의 제일 오른쪽 노드 블록 w는 P'의 i번째 노드로 옮겨진다.
- Q(Q')의 제일 오른쪽을 제외한 다른 노드 블록(x,y,w)의 인덱스를 1씩 늘린다.
- 노드 블록 P(P')의 j번째 노드 블록인 b를 Q'의 가장 왼쪽으로 옮긴다.
- Q'을 P'의 j번째로 옮긴다.

위의 I to J 회전(if j<i)의 결과로 노드 블록 Q의 가장 오른쪽 노드 블록이 노드 블록 P로 옮겨지고 P의 j번째 노드 블록이 Q의 자식 노드 블록으로 옮겨지게 된다.

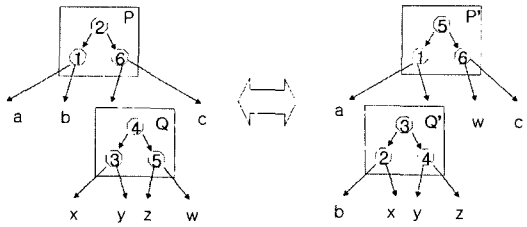


그림 7 기본 회전 알고리즘

2) 용어의 정의

위 회전 알고리즘의 응용에 들어가기 전에 몇 가지 용어에 대해 정의한다.

minH(p) : 노드 블록 p의 자식 노드 블록들 중 높이가 가장 낮은 노드 블록의 높이

maxH(p) : 노드 블록 p의 자식 노드 블록들 중 높이가 가장 높은 노드 블록의 높이

minI(p) : 노드 블록 p의 자식 노드 블록들 중 높이가 가장 낮은 노드 블록의 index 혹은 노드블록, minI(p)가 여러 개일 경우 maxI(p)에 가장 근접해 있는 것을 minI(p)로 선택한다.

maxI(p) : 노드 블록 p의 자식 노드 블록들 중 높이가 가장 높은 노드 블록의 index 혹은 노드블록

3) 회전 알고리즘의 응용 1

그림 8의 왼쪽 트리가 노드 삽입으로 인해 Q의 높이가 1증가되어 maxH(P) - minH(P) = 2인 상태로 밸런스가 깨진 트리라고 가정하자. 또한 maxI(P)는 Q이며 minI(P)는 c이며 maxI(Q)는 N이라고 가정하자. 이때, 4 to 3 회전(P)을 적용하면 그림 8의 오른편과 같이 밸런스가 맞추어 진다. 이것은 AVL 트리의 RR회전 (혹은 LL회전)을 연상케 하는 회전이다.

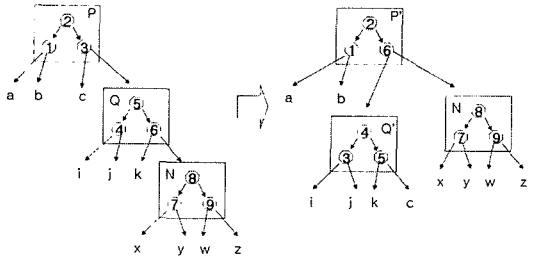


그림 8 회전 응용 1

4) 회전 알고리즘의 응용 2

회전 알고리즘의 응용 1에서 만약 minI(P)가 c(index는 3)이 아니라면 4 to 3 회전(P) 후에 Q'에서 밸런스

가 깨지는 경우가 발생한다. 그림 9는 minI(P)가 b (index는 2)인 경우이다. 이때는 우선 3 to 2 회전(P)를 이용하여 minI(P)를 o(index는 3)으로 만들면 회전 알고리즘의 응용 1과 같은 상황이 되어 4 to 3 회전(P)를 적용하여 밸런스를 맞출 수 있다.

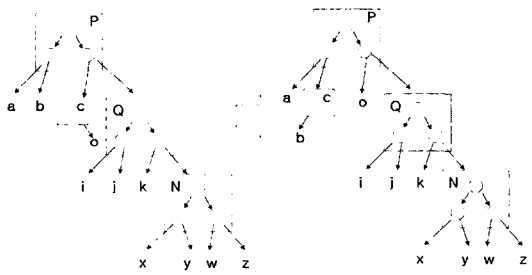


그림 9 Rotation 응용 2

만약 minI(P)가 a인 경우는 2 to 1 회전(P) , 3 to 2 회전(P)를 차례대로 적용하면 회전 알고리즘의 응용 1과 같은 상황이 된다.

5) 회전 알고리즘의 응용 3

그림 10은 회전 알고리즘의 응용 1과 유사한 상황이나 maxI(Q)의 index가 4가 아니라 3인 경우이다. 이 경우 4 to 3 회전(P)을 적용하면 밸런스가 맞지 않게 된다. 따라서, 우선 3 to 4 회전(Q)을 적용하여 그림 10의 오른편과 같이 만들면 회전 알고리즘의 응용 1과 같은 상황이 되고 4 to 3 회전(P)를 적용함으로써 밸런스를 맞출 수 있다.

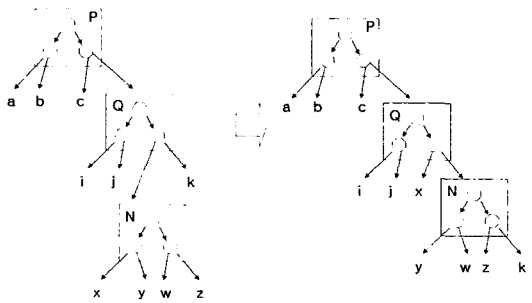


그림 10 회전 응용 3

만약 maxI(Q)의 index가 3이 아니라 2인 경우에는 2 to 3 회전(Q), 3 to 4 회전(Q)을 차례로 적용한 후에 4 to 3 회전(P)을 적용하면 된다. maxI(Q)의 index가 1인 경우에도 유사하게 1 to 2, 2 to 3, 3 to 4를 차례로 적용시키면 된다. 회전 알고리즘의 응용 3에서 회전 알고리즘의 응용 1로 이어지는 과정은 AVL 트리의 RL 회전(혹은 LR회전)을 연상케 한다.

6) 노드 블록 밸런싱 알고리즘

다음은 위에서 설명한 밸런싱 알고리즘을 정리한 것이다.

```
function balanceCheck(P)
  if maxH(P) - minH(P) > 1 then  // 밸런스가 깨져 있음
    if maxI(P) > minI(P) then  // 왼쪽으로 회전 해야함
    Q = maxI(P)
    if maxI(P) - minI(P) != 1 then  // minI(P)가 maxI(P)의
    바로 왼쪽에 있지 않으면
      to = maxI(P) -1
      from  = minI(P)
      for ( i = from ; i < to; i++) i_to_j_Rotation( P, i+1, i)
      end for
    end if
    if maxI(Q) != nchild then // maxI(Q)가 Q의 가장 오른쪽
    에 있지 않으면
      for ( i = maxI(Q) ; i < nchild ; i++ ) i_to_j_Rotation
      (Q , i , i+1) end for
    end if
      i_to_j_rotation(P , maxI(P) , minI(P) )

    else // 오른쪽으로 rotation 해야 함 왼쪽 rotation과 유사하
    므로 생략
    end if
  end if
balanceCheck( P->parent )
```

4. 실 험

4.1 실험환경

CST-트리는 C 프로그래밍 언어로 구현되었고 gcc로 컴파일했다. 실험을 한 컴퓨터는 Sun사의 UltraSparc IIi 332Mhz의 컴퓨터로 512k의 L2 캐시 메모리를 가지고 있고 L2 캐시 블록 사이즈는 64바이트이다. L2 Cache 미스 횟수를 구하기 위해서는 미국 썬(SUN) 사에서 개발한 시뮬레이션 툴인 Shade[10]를 이용하였다.

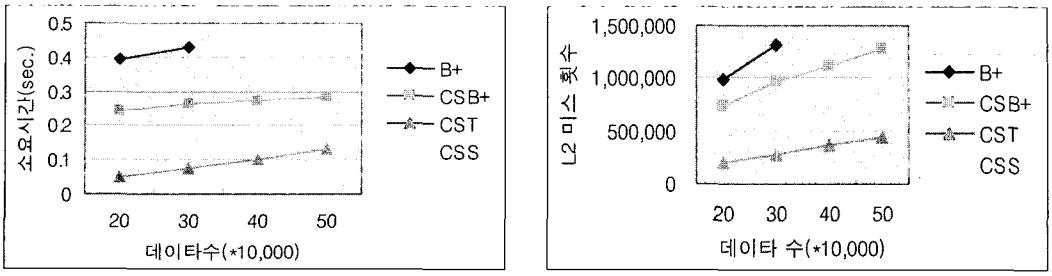
CST-트리와의 성능비교에 사용된 인덱스 구조는 CSS, CSB+, B+ 트리며 이중 CSS, CSB+, B+-트리는 [2,3]의 저자가 구현한 것을 다운로드 받아 실험하였다.

4.2 검색성능

그림 11은 트리가 각각 200,000, 300,000, 400,000, 500,000개의 키를 가지고 있을 때 200,000건의 임의 검색(Random Search)를 수행했을 때 걸린 시간을 그래프로 표시한 것이다. CST, CSS, CSB+, B+ 순으로 좋은 결과를 보이고 있다. 이 결과에 대해 자세한 분석은 4.6.1에 제시한다.

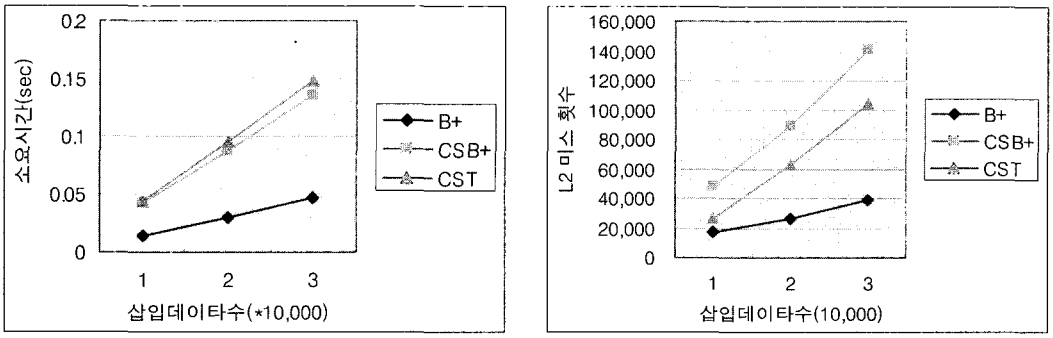
4.3 삽입(Insertion) 성능

그림 12는 10,000, 20,000, 30,000 개의 키를 임의로 생성하여 삽입했을 때 걸린 시간을 그래프로 표시한 것이다. B+ tree가 제일 좋은 성능을 보이고 있으며 CST



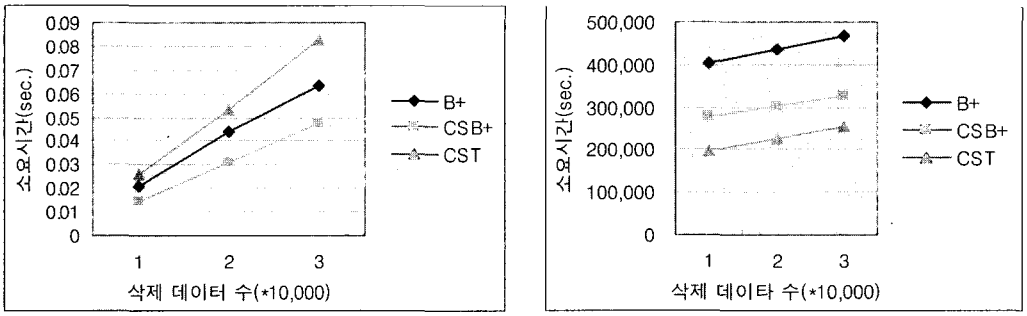
(a) 실행시간 (b) L2 캐시 미스 횟수

그림 11 검색 실험 결과



(a) 실행시간 (b) L2 캐시 미스 횟수

그림 12 삽입 실험 결과



(a) 실행시간 (b) L2 캐시 미스 횟수

그림 13 삭제 실험 결과

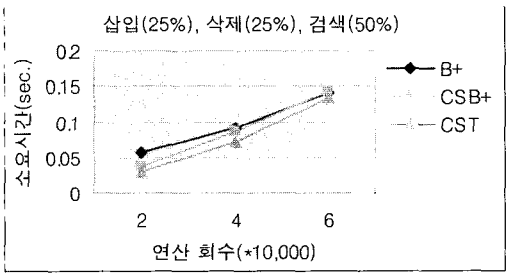
와 CSB+을 비교했을 때는 비슷한 성능을 보이고 있다. 이 결과에 대한 자세한 분석은 4.6.2절에 제시한다.

4.4 삭제(Deletion) 성능

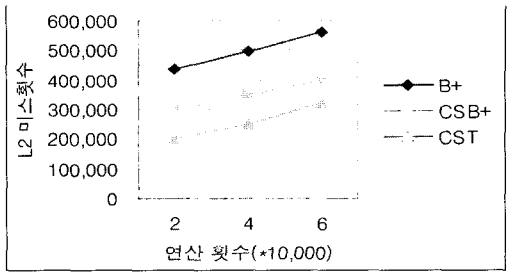
그림 13은 200,000개의 키를 임의 생성하여 트리를 구축한 후, 10,000, 20,000, 30,000 개의 키를 임의로 삭제했을 때 걸린 시간을 그래프로 표시한 것이다. CSB+ 트리가 제일 좋은 성능을 보이고 있으며 CST가 안 좋은 성능을 보이고 있다. 사실, 삭제 연산의 성능은 공정한 비교가 아니다. 왜냐하면, CSB+, B+-트리는 삭제 지연(lazy deletion)을 적용하고 있어서, 삭제 연산이 삭제할 키 값을 검색하는 비용만 필요하고, CST-트리는 키 값을 검색하고 실제로 키 값 삭제를 하고 있다. 따라서, CST-트리 삭제 실험에서는 키 값 삭제에 따른 트리 밸런싱 검사와 회전 비용이 추가로 소요되고 있다. 따라서, 성능 결과로 나온 차이는 무시할 수 있는 정도라고 판단할 수 있다.

4.5 검색/삽입/삭제 혼합 연산 성능

그림 14는 200,000개의 키를 임의 생성하여 트리를 구축한 후, 20,000, 40,000, 60,000 개의 키를 임의로 검색, 삽입, 삭제했을 때 걸린 시간을 그래프로 표시한 것이다. 그림 14는 검색 50%, 삽입 25%, 삭제 25% 비율로 연산을 수행했을 때의 결과이고, 그림 15는 검색

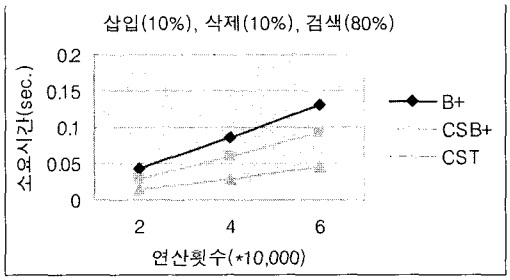


(a) 실행시간(검색 50%)

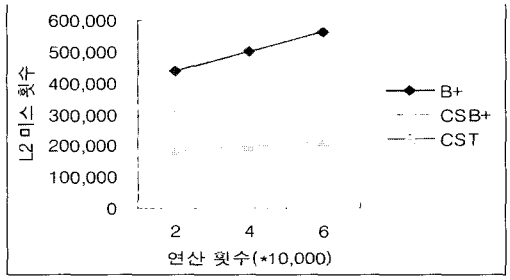


(b) L2 캐시 미스 횟수(검색 50%)

그림 14 혼합연산 실험 결과 1(검색 50%)



(a) 실행시간(검색 80%)



(b) L2 캐시 미스 횟수(검색 80%)

그림 15 혼합연산 실험 결과 2(검색 80%)

80%, 삽입 10%, 삭제 10% 비율로 연산을 수행한 결과이다. 검색비율이 높아질수록 CST 트리가 좋은 성능을 보이고 있으며, B+가 안 좋은 성능을 보이고 있다. 이 결과는 앞의 검색, 삽입, 삭제 성능 결과를 보면 유추할 수 있는 결과이며, 삭제 연산에 삭제지연(lazy deletion)을 적용하지 않았음을 고려한다면, CST-트리의 결과는 보여지는 결과보다 더 우수하다고 분석할 수 있다.

4.6 결과 분석

4.6.1 검색 시간복잡도

표 1은 각 트리의 높이와 검색했을 때 발생하는 캐시 미스의 수 그리고 비교연산의 횟수를 표로 정리한 것이다. n은 키의 개수이며 s는 T-트리의 한 노드가 가지고 있는 키의 개수, m은 CST-트리의 한 노드블록이 가지고 있는 키의 개수이다(B-트리의 경우는 한 노드가 가지고 있는 키의 개수).

CST-트리의 경우 하나의 노드에 s(=8)개의 키가 저장될 경우 n/s 개로 이진 트리를 만면 트리의 높이는 $\log_2 \frac{n}{s}$이 된다. CST-트리의 경우 하나의 노드 블록에 $\log_2(m+1)$(캐시블록 크기가 64바이트일 때 m=15)높이 만큼의 트리를 가지고 있으므로 노드 블록의 높이는 $\log_2 \frac{n}{s} / \log_2 m+1 = \log_{m+1} \frac{n}{s}$가 된다. 루트 노드에서 단말 노드까지 도달하는데 노드 블록의 높이만큼 캐시 미스가 발생하게 된다. CST-트리의 캐시 미스 발생 횟수는 다른 트리와 비교했을 때 CSS-트리와 거의 비슷하고 B+-트리나 T-트리 CSB+-트리에 비해 적다. 총 비교횟수는 다른 트리와 동일하지만 캐시 미스 횟수가 적어 그림 11과 같은 결과가 나타났다고 결론내릴 수 있다.

표 1 검색 시간복잡도

	Tree height	Cache Miss	Total Comparisons	실험에 사용된 값
B+	$\log_{m-1} n$	$\log_{m-1} n$	$\log_2 n$	m=8
CSS	$\log_{m+1} n$	$\log_{m+1} n$	$\log_2 n$	m=15
CSB+	$\log_{m-1} n$	$\log_{m-1} n$	$\log_2 n$	m=15
T	$\log_2 \frac{n}{s}$	$\log_2 \frac{n}{s}$	$2\log_2 \frac{n}{s} + \log_2$	s=8
CST	$\log_2 \frac{n}{s}$	$\log_{m+1} \frac{n}{s}$	$\log_2 n$	m=15 s=8

4.6.2 삽입 결과 분석

B+ tree나 CSB+-트리의 경우 키 삽입 시, 성능에 가장 영향을 미치는 요소는 노드 분할(Node Split)이다. CSB+의 경우 검색 속도가 빠른 대신 노드 분할 알고리즘의 속도가 매우 느리다. T-트리나 CST-트리에서 B+-트리의 노드 분할에 해당하는 것은 회전이다. 마찬

가지로 이 회전 알고리즘이 삽입 성능에 가장 큰 영향을 미친다. 특히, CST-트리는 복잡한 회전 알고리즘을 가지고 있기 때문에 더욱 신경을 쓸 필요가 있다. 왜 그림 12와 같은 성능 결과가 나왔는지 노드 분할 혹은 회전 발생 빈도와 그 오버헤드를 CST-트리와 CSB+-트리를 비교하며 분석한다.

• 회전 발생 빈도

표 2는 CST-트리에 키를 삽입하는 동안 발생하는 i to j 기본 회전의 수를 나타낸 것이다.

표 2 삽입시 i to j 회전 발생 횟수

키 시퀀스 \ 키의 개수	10000개	20000개	30000개
임의적(random)	79번	696번	1402번
1부터 차례대로	502번	1074번	1673번

키의 시퀀스(sequence)에 따라 발생 빈도가 차이가 나긴 하지만 약 20번의 키 삽입마다 기본 회전이 발생한다고 볼 수 있다. 이것은 CSB+-트리에서 평균 4번에 한번 꼴로 노드 분할이 일어나는 것[3]과 비교하면 매우 적은 발생 빈도이다.

• 회전 알고리즘의 오버헤드

CST-트리에서 노드 블록들은 배열로 이루어져 있기 때문에 노드 블록의 이동은 포인터만 바뀌어서 이루어지는 것이 아니라 노드 블록 크기만큼의 데이타가 복사되어야 한다.

그림 7의 회전 알고리즘의 경우 옮겨지는 노드 블록의 개수는 "노드 블록 배열의 크기+2"이다. 그림 7의 경우 노드 블록 배열의 크기는 4이고 노드 블록 한개의 크기는 16바이트이기 때문에 총 6*16=96 바이트만큼의 데이타가 복사되어야 한다. 만약 캐시 블록 사이즈가 64바이트인 경우에는 노드 블록 배열의 크기가 16이 되기 때문에 총 18*64=1152 바이트의 데이타 복사가 필요하다. 이렇게 회전 한번에 만만치 않은 오버헤드가 필요하다. 캐시 블록 사이즈가 64바이트인 CSB+의 경우 노드 분할이 발생할 때마다 평균 8*64= 512바이트의 데이타 복사가 필요하게 된다[3]. CST-트리의 절반 정도의 오버헤드이다.

• 삽입/삭제 성능 결론

CST-트리와 CSB+을 비교했을 때 회전 혹은 노드 분할시의 데이타 복사 오버헤드는 CSB+가 절반 정도이지만, CSB+에서의 회전 혹은 노드 분할의 발생 빈도가 4-5배 정도 더 많기 때문에 그림 12와 같은 성능 결과가 나왔다고 추정할 수 있다.

5. 결론 및 향후 과제

CST-트리는 캐시를 고려한 인덱스 설계로 성능을 높이고자 하는 아이디어를 T-트리에 적용하여 만들어졌다. 시스템 L2 캐시 활용도를 최적화할 수 있도록 T-트리의 구조를 새로이 설계하고 그에 따른 검색 알고리즘을 고안하였으며, 삽입/삭제 연산에서의 트리 밸런싱을 위해 제안한 트리에 맞게 새로운 회전 알고리즘을 제안하였다.

제안한 CST-트리의 우수성을 보이기 위해 수행한 실험에서 보인 바와 같이 검색에서 CSS, CSB+-트리보다 CST-트리가 우수한 성능을 보이며, 삽입, 삭제 시에 CSB+-트리와 비슷하거나 약간 안 좋은 성능을 보인다. 검색 성능을 높이기 위해 트리를 설계하다 보니 삽입/삭제 시에는 B-트리나 T-트리보다 낮은 성능을 보인다. 하지만, 데이타의 삽입보다 데이타의 검색이 더욱 빈번하게 일어나거나 검색성능이 제일 중요한 응용분야가 매우 많다. 이러한 응용분야에 본 연구에서 제안한 CST-트리가 많은 도움이 될 것이다.

본 논문에서 제안한 새로운 CST-트리에 대해 다음의 추가 연구가 필요하다. 첫째, CST-트리의 변형을 테스트 해 볼 수 있다. 현재 각 노드의 최대값만을 취해 트리를 구성했지만 최소, 최대값으로 트리를 구성해서 서로 비교하는 것, 노드 블록의 크기를 달리하면서 성능 변화를 분석하는 것, 노드 블록 배열 대신 포인터를 사용한 경우의 성능비교 등의 많은 아이디어를 적용할 수 있다. 둘째, 부분 키(partial key)를 적용하여 실제 메인 메모리 데이타베이스 시스템에 적용해 볼 수 있다.

참 고 문 헌

[1] Tobin J. Lehman, A Study of Index Structures for Main Memory Database Management System, VLDB 1986.

[2] Jun Rao, et al, Cache Conscious Indexing for Decision-Support in Main Memory, VLDB 1999.

[3] Jun Rao, et al, Making B+ Trees Cache Conscious in Main Memory, 2000 SIGMOD.

[4] Alan J. Smith. Cache memories. ACM Computing Surverys, 14(3):473{530, 1982.

[5] Peter Boncz, et al, Database Architecture Optimized for the new Bottleneck : Memory Access, VLDB 1999.

[6] A. Aho, J. Hopcroft and J. D. Ullman, The Design and Analysis of Computer Algorithm, Addison-Wesley Publising Companay, 1974.

[7] D. Comer, The Ubiquitous B Tree, Computing Surveys 11,2 (June 1979).

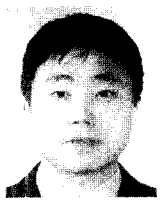
[8] Phlip Bohannon, et al, Main-Memory Index Structures with Fixed-Size Partial Keys, ACM SIGMOD 2001.

[9] Thomas H. Cormen, et al, Introduction to Algorithms, The MIT Press, 1990.

[10] Sun MicroSystems, SHADE 1.7.3 BETA, available via "http://wwws.sun. com/software/download/products-/3ff9c026.html"

이 익 훈

1996년 서울시립대학교 전산통계학과 졸업(학사). 1998년 서울시립대학교 전산통계과 졸업(석사). 1998년 3월~현재. 서울대학교 컴퓨터공학부 박사과정. 관심분야는 메인메모리 데이타베이스, eCatalog, 데이타베이스



김 현 철

2003년 한성대학교 컴퓨터공학부 졸업(학사). 2003년 3월~현재 서울대학교 컴퓨터공학부 석사과정. 관심분야는 메인메모리 데이타베이스, e-Catalog, 데이타베이스

허 재 녕

2001년 서울대학교 컴퓨터공학부 졸업(학사). 2003년 서울대학교 컴퓨터공학부 졸업(석사) 2003년 3월~현재. ㈜유엔젤 연구원 근무. 관심분야는 메인메모리 데이타베이스, 데이타 웨어하우스, 데이타베이스

이 상 구

서울대학교 학사, 미 Northwestern University 석사/박사. University of Minnesota 전임강사, EDS 연구원 등 역임. 현 서울대학교 컴퓨터공학부 교수, 서울대학교 e-비즈니스 기술연구센터장. 관심분야는 e-비즈니스 기술, e-Catalog,데이타베이스

심 준 호

서울대학교 계산통계학과 학사 /석사 1998년 미 Northwestern University, Electrical & Computer Engineering, 공학박사. 1999.2~1999.11 : Computer Associates Int'l, USA, R&D Staff, 1999.11~2001.2 : Drexel Univ., USA, Assistant Prof. ,현 숙명여자대학교 정보과학부 부교수. 관심분야는 데이타베이스, 데이타웨어하우스, 전자상거래, 웹

장 준 호

1990년 2월 서울대학교 계산통계학과 학사. 1992년 2월 서울대학교 전산학 석사 1998년 8월 서울대학교 전산학 박사. 1998년~2003년 2월 아이투 테크놀로지스(i2 Technologies). 2003년 2월~현재 상명대학교 미디어학부 교수. 관심분야는 E-Business, SCM, Enterprise SW