# KDBcs-트리 : 캐시를 고려한 효율적인 KDB-트리

## (KDBcs-Tree : An Efficient Cache Conscious KDB-Tree for Multidimentional Data)

여 명 호 † 민 영 수 †† 유 재 수 †††
(Myung Ho Yeo) (Young Soo Min) (Jae Soo Yoo)

**요 약** 본 논문에서는 데이타의 갱신이 빈번한 상황에서 데이타의 갱신을 효율적으로 처리하기 위한 색인 기법을 제안한다. 제안하는 색인구조는 대표적인 공간 분할 색인 기법 중 하나인 KDB-트리를 기반으로 하고 있으며, 캐시의 활용도를 높이기 위한 데이타 압축 기법과 포인터 제거 기법을 제안한다. 제안하는 기법의 우수성을 보이기 위해서 기존의 대표적인 캐시를 고려한 색인 구조중 하나인 CR-트리와 실험을 통해 성능을 비교하였으며, 성능평가 결과, 제안하는 색인 구조는 삽입 성능과 갱신 성능, 캐시 활용도 면에서 기존 색인 기법에 비해 각각 85%, 97%, 86% 의 성능이 향상 되었다.

**키워드** : 캐시실패, 갱신성능, 색인 구조, KDB-트리

***Abstract*** We propose a new cache conscious indexing structure for processing frequently updated data efficiently. Our proposed index structure is based on a KDB-Tree, one of the representative index structures based on space partitioning techniques. In this paper, we propose a data compression technique and a pointer elimination technique to increase the utilization of a cache line. To show our proposed index structure's superiority, we compare our index structure with variants of the CR-tree(e.g. the FF CR-tree and the SE CR-tree) in a variety of environments. As a result, our experimental results show that the proposed index structure achieves about 85%, 97%, and 86% performance improvements over the existing index structures in terms of insertion, update and cache-utilization, respectively.

**Key words** : Cache-miss, Update Performance, Index Structure, KDB-tree

## 1. 서 론

전통적인 디스크 기반의 데이타베이스 시스템에서는 디스크 I/O가 시스템 성능 저하에 큰 영향을 끼친다. 그래서 디스크 기반 시스템에서는 디스크 I/O를 줄이기 위한 연구들이 많이 진행되고 있다. 최근 메인 메모리의 가격이 하락함에 따라 데이타베이스 시스템의 대부분(테이블과 색인)을 주기억 장치에서 실행하는 것이 가능해졌다. 주기억 장치 상주 테이블, 색인을 사용함으로써 과거 디스크 I/O에 대한 병목현상이 줄어들었으며 메모리와 프로세서의 성능이 매년 크게 향상되고 있어 성능 향상이 가속화되고 있다. 하지만 이러한 성능 향상에도 불구하고 상업용 데이타베이스 시스템은 프로세서의 성능을 충분히 활용하지 못하고 있다[1]. 그것은 프로세서의 접근 속도와 메모리의 접근 속도의 차이로 인한 병목현상에서 기인한 것으로 이를 보완하기 위한 방법으로 캐시의 역할과 동작이 더욱 중요해졌다[2]. [3]은 주기억 장치에서 상업용 데이타베이스 관리 시스템의 캐시 동작을 살펴보고 2차 레벨 데이타 캐시실패와 1차 명령 캐시실패가 실행시간에서 큰 비중을 차지하고 있다는 결론을 도출했다. 이러한 점에 착안하여 주기억 장치 색인의 성능을 향상시키기 위해서 캐시를 고려한 색인 구조들이 제안되었다.

· 이 논문은 2007년 정부(교육인적자원부)의 재원으로 한국학술진흥재단 지원(지방연구중심대학육성사업/충북BIT연구중심대학육성사업단)과 산업자원부 지역혁신인력양성사업의 연구결과물임
† 학생회원 : 충북대학교 정보통신공학과
mhyeo@netdb.chungbuk.ac.kr
†† 정 회 원 : 한국전자통신연구원 홈네트워크연구단 연수연구원
minys@etri.re.kr
††† 종신회원 : 충북대학교 전기전자컴퓨터공학부 교수
yjs@chungbuk.ac.kr
(Corresponding author임)
논문접수 : 2006년 6월 15일
심사완료 : 2007년 4월 11일

대표적인 캐시를 고려한 색인 알고리즘으로는 CSS-트리(Cache-Sensitive Search-트리)[4], $CSB^+$-트리(Cache-Sensitive $B^+$-트리)[5], CR-트리(Cache-conscious R-트리)[6]가 있다. 이러한 캐시를 고려한 색인 구조는 캐시실패를 줄이기 위해 노드의 크기를 캐시 블록의 배수로 정하고, 노드의 팬-아웃(fan-out)을 증가시켜 상대적으로 캐시 실패 확률을 줄이는 것을 목표로 하고 있다. 그 결과, 기존의 전통적인 색인 구조보다 2~2.5배의 성능이 향상되는 결과를 얻을 수 있었다.

최근 유비쿼터스 환경에서 이동성을 지닌 공간 객체를 관리하거나 센서 네트워크 환경에서 센서기반의 스트리밍 데이타를 처리하는 새로운 응용 분야들이 많이 연구되고 있다. 이러한 응용분야의 데이타들은 빈번한 갱신을 요구하는 다차원 데이타의 특성을 가지고 있는데, 이를 처리하기 위한 방법으로 기존의 캐시를 고려한 색인 구조를 사용하는 것을 고려할 수 있다. 하지만 기존의 $CSB^+$-트리는 다차원 색인구조에 적합하지 않고, R-트리 기반의 CR-트리는 빈번히 갱신되는 데이타를 색인하기 위해서 MBR(Minimum Bounding Rectangle)을 변경하고 유지하는 비용이 많이 든다. 또, MBR 정보가 압축되었다 하더라도 고차원 데이타일수록 MBR 정보가 커지기 때문에 팬-아웃이 감소하는 문제점을 가지고 있다.

본 논문에서는 갱신이 빈번한 데이타 처리에 적합한 새로운 캐시를 고려한 다차원 색인구조를 제안한다. 기존 R-트리 기반의 색인 구조가 가지고 있는 단점을 보완하기 위해서 공간 분할 기법을 사용하였으며, 캐시라인의 활용도를 높이기 위한 방법으로 공간 압축 기법과 포인터 제거 기법을 제안한다.

본 논문의 구성은 다음과 같다. 제2장에서는 기존의 캐시를 고려한 색인 구조와 공간 분할 기법을 사용하는 색인 구조, 포인터 제거 기법의 관련 연구에 대해서 논의 한다. 제3장에서는 제안하는 색인구조의 특징과 기본적인 연산 알고리즘을 기술한다. 제4장에서는 성능평가와 분석을 통해 제안하는 색인 구조의 우수성을 보이고, 제5장에서 결론과 향후 연구방향을 제시한다.

## 2. 관련연구

본 장에서는 제안하는 색인 구조의 관련연구로 먼저 공간 분할 기법과 대표적인 공간 분할 기법중 하나인 KDB-트리의 특징에 대해서 알아본다. 그 다음, 기존에 제안된 캐시를 고려한 색인 구조를 살펴보고, 분석을 통하여 문제점을 제시한다.

### 2.1 공간 분할 기법과 KDB-트리

기존의 색인 기법은 크게 공간 분할 색인 기법(SP-based index structure)과 데이타 분할 색인 기법(DP-based index structure)로 나눌 수 있다[7]. 데이타 분할 색인 기법은 데이타를 포함하는 영역을 BR(bounding region) 형식으로 나타내고, 여러 개의 BR을 포함하는 영역을 다시 상위 BR 영역으로 구성한다. 이때, BR은 서로 겹칠 수 있다. 이와 달리 공간 분할 색인 기법은 전체 데이타 영역을 기준으로 서로 겹칠 수 없는 하위 공간으로 분할하며 이 분할의 계층을 트리 형태로 나타낸다[7].

대표적인 공간 분할 색인 기법인 KDB-트리[8]는 삽입과 갱신이 비교적 간단하지만, 검색 시에 데이타가 존재하지 않는 공간도 검색이 이루어질 가능성이 있기 때문에 검색 부분에 있어서 데이타 분할 색인 기법에 비해 성능이 나쁘다는 단점이 있다[9]. 대표적인 데이타 분할 색인 기법인 R-트리는 검색 시 데이타가 존재하는 공간만 검색이 이루어지므로 공간 분할 색인 기법에 비해 검색 성능이 좋지만, 삽입과 갱신 연산에 있어서 MBR의 빈번한 갱신을 요구하므로 공간 분할 색인 기법에 비해 성능이 좋지 못하다[10].

KDB-트리는 이미지, 멀티미디어, 과학 정보 데이타베이스와 같은 응용 분야에서 객체를 고차원 공간의 한 점으로 표현한다. 차원의 수가 d-차원으로 증가함에 따라 객체는 길이 d인 벡터로 표현된다. 그 결과 한 페이지에서 수용할 수 있는 객체의 수가 감소하고, 트리의 높이가 증가하여 검색 성능이 나빠지게 된다. 이것이 대부분의 고차원상의 질의처리 방법에서 성능을 제한하는 문제점 중 하나이다. 고차원 공간 접근 방법의 문제점을 해결하기 위해서 제안된 알고리즘이 $KDB_{HD}$-트리[11]이다.

$KDB_{HD}$-트리는 FD(First Division)-분할 정책과 MBR Descriptor로부터 중복 정보를 제거한 KDB-트리의 변형이다. 본래 KDB-트리의 FS(Forced Splitting) 정책은 마지막으로 오버플로우된 단말 페이지를 두 개의 그룹으로 분할하기 위해서 공간 영역을 확장한다. 따라서 확장된 공간 영역과 겹치는 모든 색인 영역이 두 개의 부분공간으로 나누어져야 한다. 추가된 중간 노드의 색인 영역들을 나누는 과정에서 노드 분할 정보를 아래로 전파하게 된다. 반면 FD-분할 정책은 분할될 노드를 처음으로 분할하는 정보를 기준으로 두 개의 자식 페이지로 분할함으로써, 노드의 분할 정보를 아래로 전파하는 과정을 생략한다. 중간 노드는 <m, low1, high1, ..., lowm, highm, node pointer>와 같은 구조를 가지고 있다. 이때 m은 해당 MBR 노드의 유용한 차원의 수를 의미하고, low, high는 MBR의 low, high end-point이다. 고정된 길이의 KDB-트리의 엔트리와 비교하면 $KDB_{HD}$-트리는 중간 노드마다 m값을 유지하기 위한 비용을 발생시키지만 MBR Descriptor에서 중복된 부분을 제거할 수 있다. 단말 노드의 엔트리 형태는 KDB-트리와 동일하다. $KDB_{HD}$-트리는 고차원 공간에

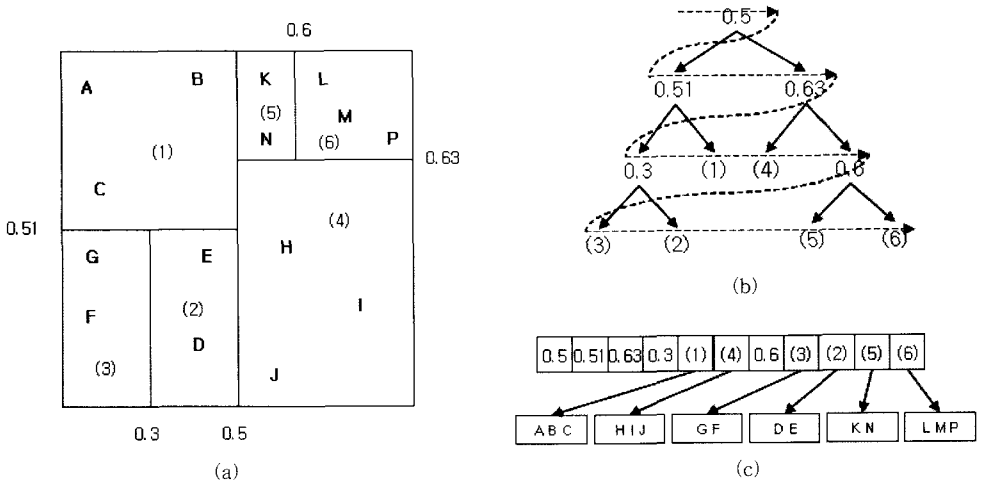



그림 1 $KDB_{KD}$-트리의 예

서의 KDB-트리의 문제점을 효율적으로 해결하고 있으나 중간 노드의 region descriptor에는 여전히 중복된 정보를 가지고 있다. 저차원에서 중복된 정보는 무시할 정도의 수준이지만 고차원으로 가면 이야기가 달라진다.

$KDB_{KD}$-트리[12]는 중간 노드의 색인 엔트리 구조를 변경함으로써 $KDB_{HD}$-트리에서 여전히 남아있는 중복된 정보를 제거하는 방법을 제안했다. 예를 들면, 그림 1과 같이 분할된 데이타 공간이 있다고 가정할 때, 그림 1(a)의 공간 분할은 그림 1(b)와 같은 KD-트리로 표현할 수 있다. 이 KD-트리는 그림 1(b)와 같이 점선 방향의 트리 순회를 통해 그림 1(c)의 $KDB_{KD}$-트리 노드로 표현이 가능하다. 그 결과, $KDB_{HD}$-트리의 중복을 완벽히 제거하는 효과를 가져왔다.

### 2.2 캐시를 고려한 색인 구조

2.2.1 CSS-트리와 $CSB^+$-트리

1990년대 말에 Rao와 Ross는 메인메모리 데이타베이스관리시스템의 성능을 향상시키기 위해서 캐시를 고려한 CSS-트리[4]와 $CSB^+$-트리[5] 색인 기법을 제안했다. 이 두 색인 기법이 캐시를 고려한 색인 기법의 발단이다. OLAP 환경을 위해 고려된 CSS-트리는 엔트리들의 밀집도가 매우 높은 공간 효율적인 트리이다. CSS-트리는 색인구조에서 자식노드를 가리키는 포인터를 제거하고 물리적으로 연속적인 공간에 키를 저장한다. 이로 인해 포인터가 차지하고 있던 공간에 더 많은 키를 저장할 수 있게 되어, 트리의 높이를 줄임으로써 캐시 접근 실패를 효과적으로 줄일 수 있다. 하지만, 트리 전체를 물리적으로 연속적인 공간에 저장하므로 변경이 일어날 경우 트리 전체를 재구성해야 하는 단점이 있다. 분명 CSS-트리는 이진 검색과 T-트리보다 뛰어난 검색 성능을 보이지만 고정적인 데이타를 처리하기 위해 설계되었기 때문에 상대적으로 갱신빈도가 높은 데이타 처리에는 적합하지 않다.

한편, $B^+$-트리는 이진 검색이나 T-트리에 비해 갱신 성능이 뛰어나지만 자식 포인터를 저장하기 위한 공간으로 인하여 캐시 라인의 활용도가 떨어진다. [5]은 $B^+$-트리를 이용하여 갱신 비용을 크게 증가시키지 않고, CSS-트리처럼 캐시 활용을 높이는 것을 목표로 자식 포인터의 대부분을 제거하여 캐시 라인의 활용도를 높인 형태인 $CSB^+$-트리를 제안한다.

$CSB^+$-트리는 팬-아웃을 증가시키기 위해 그림 2처럼 첫 번째 자식 포인터를 제외하고 나머지는 상대적인 값, 즉, i번째 자식 포인터를 첫 번째 자식 포인터의 상대값으로 나타낸다. 그 결과 기존의 $B^+$-트리에 비해 팬-아웃이 2배 이상 증가되어 트리의 높이가 감소하고 상대적으로 캐시 활용도가 높아지는 결과를 보인다.

2.2.2 CR-트리

CR-트리는 팬-아웃을 늘리기 위한 방법으로 MBR을 양자화하여 비트로 표현하는 압축 방법을 제안했다. 그림 3은 3개의 엔트리를 포함하는 양자화의 CR-트리의 노드 구조를 나타낸다. 노드 구조를 살펴보면 저장된 엔트리들이 단말노드와 중간노드를 구분하기 위한 플래그, 엔트리 수, 그리고 노드 내 모든 자식 노드의 모든 MBR을 포함하는 최소 크기의 참조 MBR로 구성되어 있다.

그림 4는 CR-트리의 MBR 압축 기법을 보여준다. 그림 4(a)는 R0~R3의 MBR들을 절대 좌표로 표현하고 있다. 그림 4(b)는 R0의 좌측-아래(150,180)를 기준으로 하는 R1~R3의 MBR들을 상대 좌표로 표현하고 있다. 그림 4(c)은 R1~R3의 상대적인 좌표들을 16레벨 또는 4비트로 양자화된 좌표 값으로 표현하고 있다. 이렇게 상대적인 좌표들을 고정된 비트로 양자화하여 만

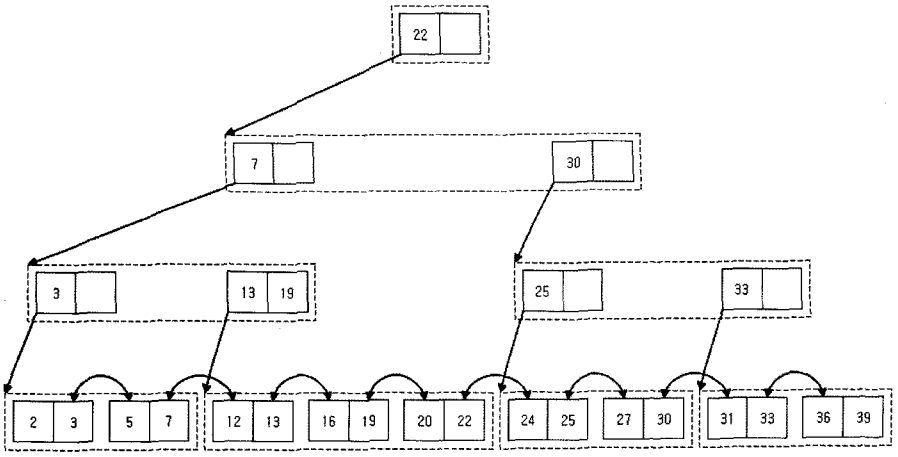



그림 2 CSB$^+$-트리의 구조

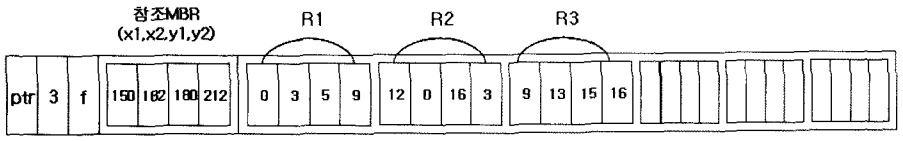



그림 3 CR-트리의 노드 구조

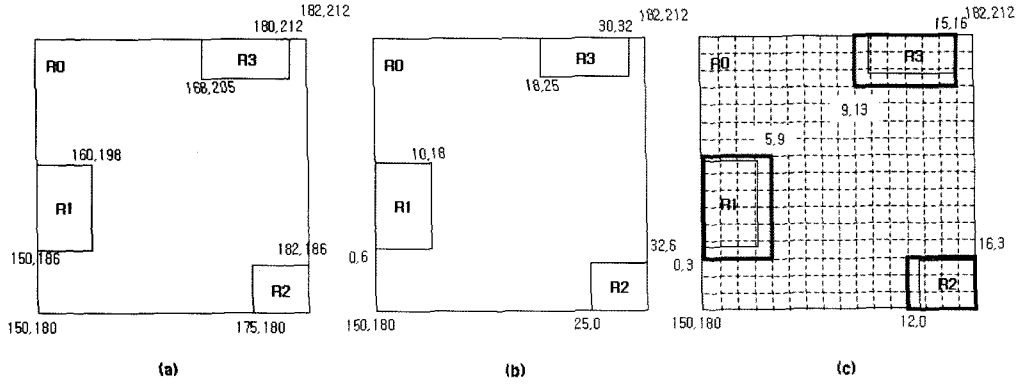



그림 4 MBR 압축 기법의 예

들어진 MBR을 QRMBR(quantized relative representation of MBR)이라고 부른다[6].

### 2.3 기존 연구 분석

KDB$_{KD}$-트리는 기존의 KDB$_{HD}$-트리에서 여전히 남아있는 중복된 정보를 제거하기 위해서 노드의 엔트리를 KD-트리 형식으로 구성하였다. 기존의 중복된 정보는 모두 제거 되었지만 포인터 정보가 남아 있고, KD-트리 자체가 이분트리이기 때문에 자식 노드에 대한 포인터 정보를 기록하기 위해서는 동일한 수의 분할 정보가 필요하다. 즉, n개의 자식 포인터를 표현하기 위해서는 n 또는 n-1개의 분할정보가 필요하다. 예를 들어 4개의 자식 포인터가 있다고 가정할 때, 3개의 분할값을 기준으로 분리되게 된다. 따라서 32-bit OS를 사용하는 컴퓨터에서 분할 정보의 크기를 4byte로 가정할 때, 전체 크기는 28byte가 된다. 노드의 크기를 S, 분할값을 표현하는 데이타의 크기를 I, 포인터의 크기를 p라고 할 때, 팬-아웃 F는 $F \leq \frac{(S-I)}{p+I}$와 같음을 알 수 있다. 64byte 노드 크기를 기준으로 7개의 팬-아웃을 계산할 수 있다. 또, 분할 정보의 크기는 4byte로 고정하고 64bit OS를 기준으로 계산을 해보면 팬-아웃이 3개로 급격히 감소한다. KDB$_{KD}$-트리를 캐시를 고려한 색인 구조로 바로 사용하기는 어렵고, 적절한 포인터 제거 기법이 요구된다.

$CSB^+$-트리는 포인터의 대부분을 제거했기 때문에 팬-아웃은 크게 늘었지만, 데이타들이 반드시 정렬되어 있어야 한다는 점과 $B^+$-트리 자체가 다차원 데이타를 처리하기에 적합하지 않다는 단점이 있다. 차원수를 d 이라고 할때, $CSB^+$-트리의 팬-아웃을 계산해 보면 $F \leq \frac{(S-p*2)}{d*I}$과 같다. 예를 들어, 데이타의 크기를 차원당 4byte로 고정하고, 32-bit OS, 1차원 데이타를 가정하면 64byte의 노드를 기준으로 14개의 팬-아웃을 가지지만, 2차원, 3차원으로 차원이 변함에 따라 7개, 4개로 팬-아웃이 감소한다. 차원이 높아짐에 따라 $CSB^+$-트리의 성능은 급격히 감소하는 문제점을 가지고 있다.

CR-트리는 포인터 제거 기법과 함께 MBR 정보를 압축하여 노드의 팬-아웃을 증가시키는 방법을 제안했다. 엔트리의 MBR 정보는 압축하여 QRMBR 형태로 저장하고, 정보의 압축과 실제 데이타 복원을 위해서 전체 엔트리의 MBR을 포함하는 RMBR 정보를 노드에 저장한다. 데이타의 양자화된 크기를 I′ 라고 할 때, 팬-아웃 F는 $F \leq \frac{S-2Id-p}{2d*I'}$과 같다. 예를 들어 캐시 라인의 크기가 64바이트이고 하나의 좌표가 4비트를 차지한다면, 2차원의 데이타는 2바이트의 QRMBR로 표현이 가능하고, QRMBR 이외의 정보들이 20바이트를 차지하므로 노드의 최대 엔트리 수는 22개로 크게 늘어났다. 하지만 CR-트리 자체가 R-트리에 기반하고 있기 때문에 빈번한 갱신이 이루어지는 데이타 상황에서 MBR 조정의 전파로 인한 성능 저하의 문제가 남아 있다.

또한 차원이 높아짐에 따라 QRMBR 의 크기가 *(압축 레벨 * 차원수)* 만큼 선형적으로 증가하게 된다. 저차원 데이타의 경우 큰 문제가 되지 않지만 차원이 높아짐에 따라 압축된 MBR 정보 자체가 커지기 때문에 차원수에 따른 확장성을 보장할 수 없다. 이러한 문제점을 떠나서도 R-트리는 데이타를 기반으로 MBR영역을 설정하기 때문에 MBR 영역의 겹침이 크게 발생하여 데이타 검색 성능이 저하되는 문제점을 가지고 있다.

기존 연구에 대한 분석의 결과, 압축기법과 포인터 제거기법을 이용하여 팬-아웃을 증가시킬 수 있지만, 차원수에 따른 확장성을 보장하는 것과 R-트리 기반 트리 알고리즘의 겹침과 MBR 조정 전파의 문제점이 남아 있는 것을 알 수 있다.

## 3. $KDB_{CS}$-트리 : 제안하는 캐시를 고려한 KDB-트리

본 장에서는 데이타 갱신이 빈번한 상황에서 효율적인 데이타 갱신을 처리하기 위한 캐시를 고려한 KDB-트리를 제안한다. 제안하는 색인기법은 공간 분할 기법의 대표적인 색인기법인 KDB-트리를 기반으로 하고 있으며, 팬-아웃을 증가시키고 캐시의 활용도를 높이기 위해서 압축기법과 포인터 제거 기법을 동시에 사용하고 있다. 본 장에서는 먼저 색인구조의 노드 구조에 대해서 기술하고, 캐시의 활용도를 높이기 위한 분할 정보 압축, 포인터 제거 기법을 기술한다. 그 다음, 제안하는 색인구조의 분할정책과 주요연산 과정을 기술한다.

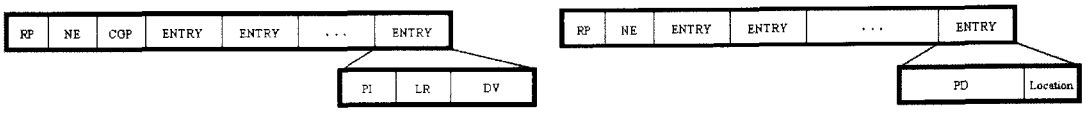
### 3.1 노드 구조

그림 5는 $KDB_{CS}$-트리의 노드 구조를 나타낸다. 공통적으로 RP(Region page or Point page flag)와 NE (the number of Entry)를 포함하고 있으며, 중간노드의 경우, 자식 노드 그룹의 주소를 저장하고 있는 CGP (Child Group Pointer)와 키 엔트리를 가지고 있다. RP는 단말노드와 중간노드를 구분하기 위한 플래그이고, NE는 노드가 포함하는 엔트리 수를 의미한다.

그림 5(a)의 중간노드의 엔트리는 분할정보를 저장하며 그 구조를 살펴보면 PI(Parent entry Index), LR(Left or Right), DV(Division Value)으로 구성되어 있다. PI는 노드 내에 상위 키 엔트리에 해당하는 키 엔트리의 노드 내 순서 번호을 의미하며, LR은 상위 엔트리로부터 왼쪽 자식 엔트리인지, 오른쪽 자식 엔트리인지 구분하는 플래그이다. DV는 현재 분할 값을 8bit로 압축한 값을 의미한다. 그림 5(b)의 단말노드의 엔트리 구조를 살펴보면 PD(Point Data)와 Location으로 구성되어 있다. PD는 대상 객체를 나타나는 정보이고, Location은 실제 데이타의 주소를 나타낸다.

### 3.2 분할 정보 압축

전통적인 KDB-트리는 분할축에 의해 나눠진 정보를 MBR형태로 저장한다. 하지만 고차원 데이타의 색인 환경에서 MBR정보의 중복이 크게 발생하여 성능이 저하되게 된다. 이러한 중복을 제거하기 위해 $KDB_{HD}$-트리가 제안되었고, 여전히 남아 있는 노드 중복을 제거하기 위해서 KD-트리 구조로 엔트리를 구성하는 $KDB_{KD}$-트리가 제안되었다. 중복이 충분히 제거되었지만 캐시 라




(a) 중간노드 (b) 단말 노드

그림 5 $KDB_{CS}$-트리의 구조

인을 이용하기에는 여전히 정보의 크기가 크다. 이에 KDBcs-트리는 KDBKD-트리에 적용가능한 양자화 기법을 제안한다. CR-트리와 같은 데이타 분할 색인 기법에서 사용된 양자화 기법은 양자화된 MBR값을 저장하는 것이 일반적이지만, 공간 분할 색인 기법을 사용하는 KDB-트리의 경우, 분할정보 즉, 분할축에 대한 정보를 양자화해야 한다.

그림 6은 KDBcs-트리의 양자화 기법을 설명한 예이다. 그림 6(a)와 같이 양자화 레벨이 2이고, 전체 영역이 (0,0)~(320,320)의 범위를 갖는 공간이 있다고 가정할 때, 공간의 분할이 필요한 경우, 해당 공간을 균등하게 분할하는 양자화 축(00~11) 중에서 데이타의 분할이 가능한 양자화 축을 선정한다. 이 양자화 축에 의해 해당 공간은 하위 공간으로 분할되며, 다시 하위 공간의 분할이 필요한 경우 하위 공간의 범위를 전체 범위로 하는 분할이 재귀적으로 이루어진다. 이때, 분할이 이루어지는 축은 Round-robin 방식에 의해 선정되며, 분할축의 정보는 그림 6(b)와 같이 KD-트리의 형태로 나타낼 수 있다. KD-트리의 정보를 KDBcs-트리의 한 노드로 표현하기 위해서 그림 6(b)의 화살표 방향을 따라 KD-트리의 각 노드 정보는 그림 6(c)와 같이 노드의 키 엔트리 형식으로 저장된다. 기존의 KDB-트리는 분할 정보의 실제 값을 저장하는데 비해 제안하는 방법은 분할 정보를 양자화하여 표현함으로써 분할 정보의 크기를 줄였고, 분할 정보의 크기를 줄임으로써 노드의 팬-아웃을 증가시켰다.

### 3.3 포인터 제거 기법

3.2절에서 제안한 양자화 기법을 통해 노드를 구성하는 분할 정보의 값을 압축할 수 있었다. 하지만 그림 6(c)와 같이 CP1~CP5에 해당하는 포인터 정보가 그대로 남아 있는 것을 알 수 있다. 캐시의 라인의 활용도를 높이기 위해서 이 포인터 정보를 제거하기 위한 적절한 포인터 제거 기법이 필요하다. 포인터 제거 기법은 캐시 동작을 최적화하는데 있어서 중요한 기술이다. 포인터 제거는 포인터를 포함하고 있는 관련키의 크기와 관련이 있다. 일반적으로 색인 구조에 키 엔트리는 포인터와 함께 영역 정보나 데이타 ID값 등을 가지고 있으며 포인터의 크기는 Integer의 크기와 같다. 예를 들어 32-bit OS에서 64-bit OS로 변경될 때, 포인터의 크기가 4바이트에서 8바이트로 증가하여 잠재적으로 포인터를 제외한 정보 데이타보다 더 많은 공간을 필요로 한다는 것을 의미한다. 따라서 색인 구조 설계에 있어 적절한 포인터 제거 기법은 반드시 고려되어야 할 필수적인 요소이다. KDBcs-트리의 기반이 되고 있는 KDBKD-트리를 기준으로 포인터를 제거하는 기법으로 몇 가지 방법을 고려할 수 있다.

먼저 그림 7(a)와 같은 KDBKD-트리의 노드가 있다고 가정할 때, 포인터를 제거하기 위한 방법으로 CSB+-

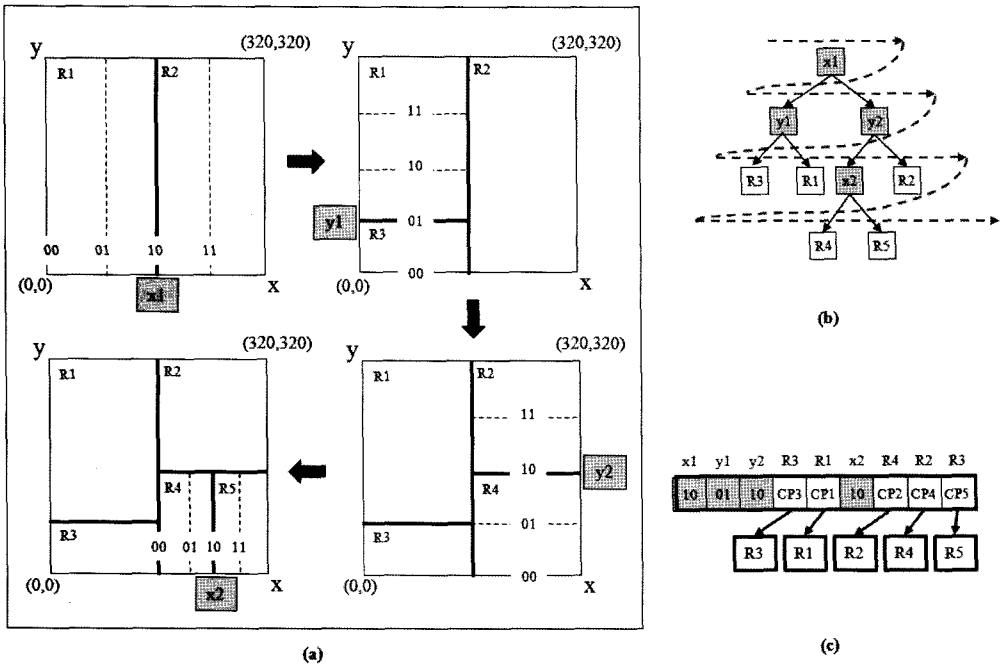



그림 6 KDBcs-트리의 분할 정보 압축의 예

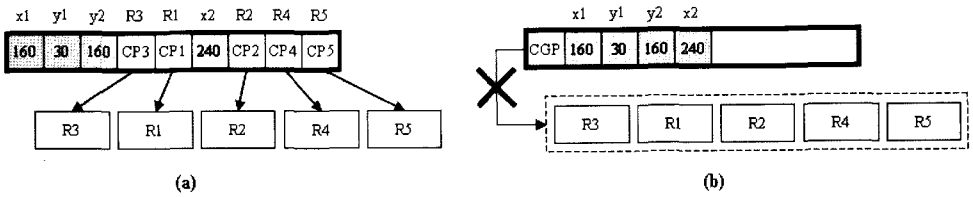



그림 7 CSB$^+$-트리의 포인터 제거기법을 바로 적용했을 경우의 문제점

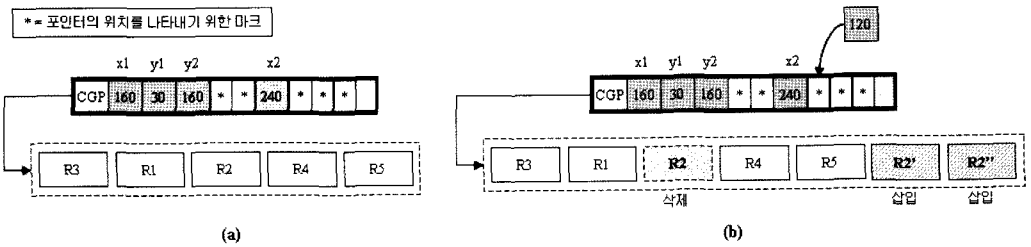



그림 8 포인터의 위치를 나태내기 위한 마크를 사용한 경우의 문제점

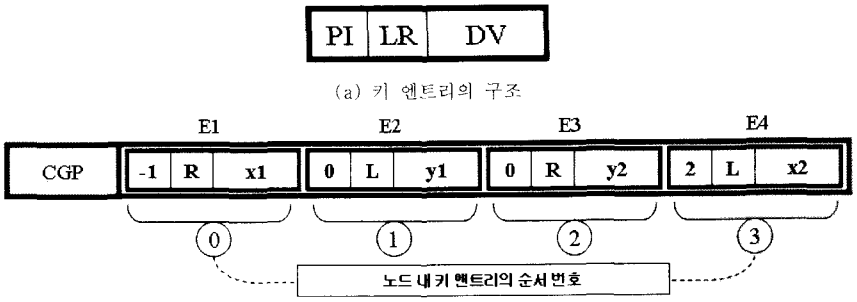



그림 9 계층 정보 보존을 위한 포인터 제거 기법

트리에서 제안된 포인터 제거 기법을 사용하는 것을 고려해 볼 수 있다. CSB$^+$-트리는 현재 노드는 연속된 공간에 자식 노드들을 할당하고 첫 번째 자식 노드에 대한 포인터만 저장함으로써 나머지 포인터를 제거하는 방법을 사용한다[4]. 그림 7(b)와 같이 KDB$_{KD}$-트리 역시 포인터를 제거하기 위해서 자식 노드들을 연속된 공간에 할당하고 첫 번째 자식 노드의 포인터만 저장하는 방법을 사용할 수 있다. 하지만, KDB$_{KD}$-트리의 경우, 노드 내부에 KD-트리 형식의 계층적인 구조를 가지고 있기 때문에 포인터를 제거하는 과정에서 계층 정보를 잃어버려 자식노드를 검색할 수 없게 된다.

이러한 문제점을 해결하기 위한 방법으로 그림 8과 같이 포인터를 제거하는 대신 포인터의 위치에 포인터 마크를 표시하는 방법을 고려해 볼 수 있다. 그림 8(b)와 같이 공간이 분할되어 새로운 분할 정보 "120"을 추가한다고 가정할 때, 기존의 포인터 위치에 해당하는 자식 노드 *R2*를 삭제하고, 자식 노드 그룹의 마지막에 2개의 자식 노드 *R2'*와 *R2"*를 추가해야한다. 이때, 자식 노드들은 연속적으로 구성되어 있기 때문에 삭제된 노드 *R2*를 기준으로 자식 노드간의 재정렬이 불가피한 상황이 발생한다.

KDB$_{CS}$-트리는 새로운 분할 정보가 추가되거나 삭제되더라도 계층정보가 유지되고, 노드간의 재정렬이 필요없는 포인터 제거 기법을 제안한다. 그림 9(a)는 노드를 구성하는 키 엔트리의 구조를 나타내며, 그림 9(b)는 포인터를 제거한 중간 노드를 나타낸다. 키 엔트리는 앞서 설명했듯이 *PI*, *LR*, *DV*로 구성된다. *PI*와 *LR*은 계층 정보를 보존하기 위한 값으로, PI는 부모 키 엔트리에 대한 순서 번호를 의미한다. *LR*은 부모 키 엔트리를 기준으로 왼쪽, 오른쪽 방향을 나타내는 플래그이다. 각 키 엔트리는 이 정보를 유지함으로써 계층 정보의 보존은 물론 새로운 분할 정보가 삽입될 경우 별도의 재정렬 과정 없이 분할정보를 노드의 마지막에 추가할 수 있게 된다. 다수의 키 엔트리로 구성된 노드의 경우, 자식노드에 해당하는 *R1*~*R5*는 연속된 공간으로 할당하여 노드 그룹을 생성하고, 첫 번째 노드의 포인터를

*CGP*로 저장하고 나머지 포인터 정보를 제거한다. 연속된 메모리 공간에 자식 노드를 할당하고, 첫 번째 자식 노드의 주소만 저장하는 것은 CSB+-트리의 포인터 제거 기법과 동일하지만, 노드 내부의 계층 정보를 보존하고, 새로운 분할 정보의 추가시 재정렬을 수행하지 않기 위해서 추가적인 정보를 기록한다는 것은 차이점이 있다. 추가된 PI, LR 정보를 이용하여 검색하고자 하는 자식노드의 주소를 계산하는 방법은 *ComputeOffset* 알고리즘과 같다.

*ComputeOffset* 알고리즘의 연산과정을 예를 들어 설명하면, (1) *R4*에 해당하는 영역을 질의 영역이라고 가정할 때, 먼저 0번째 엔트리 *E0*의 *DV*값(=*x1*)을 기준으로 질의 영역의 방향 즉, *LR* 값을 결정해야한다. 이때 *"LeftOffset=0"*이다. 질의 영역이 분할축을 중심으로 더 큰 좌표값을 가진 영역에 위치할 경우, *LR*은 *R*의 값을, 그렇지 않으면, *L*의 값으로 결정된다. *PI*와 *LR*은 *I*번째 엔트리의 하위 엔트리를 찾기 위한 조건이 된다. 따라서 *R4*에 해당하는 질의 영역은 분할축 *x1*보다 크기 때문에 노드를 순회하면서 *"PI=0, LR=R"* 인 값을 가지는 있는 엔트리가 있는지 검사한다. (2) 2번째 엔트리, *E3*가 이 조건을 만족하며, *E3*의 *DV*값(=*y2*)을 기준으로 질의 영역의 방향을 결정한다. 이때, *"LeftOffset=3"* 이다. 질의 영역은 분할축 *y2*보다 작기 때문에 노드를 순회하면서 *"PI=2, LR=L"* 인 값을 가지는 엔트리가 있는지 다시 검사한다. (3) 3번째 엔트리 *E4*가 이 조건을 만족하며, *E4* 역시 *DV*값(=*x2*)를 기준으로 질의 영역의 방향을 결정한다. 질의 영역이 분할축 *x2* 보다 작기 때문에 *"PI=3, LR=L"* 인 값을 가지는 엔트리가 있는지 다시 검사한다. (4) 하지만 해당 엔트리가 존재하지 않기 때문에 더 이상 분할축 정보가 없다고 판단하고, 오프셋을 결정한다. (5) 최종으로 *LR*의 값이 *L*이므로 질의 영역에 해당하는 오프셋은 *"LeftOffset=3"* 으로 결정된다. 결국 질의 영역 *R4*에 해당하는 자식 노드의 주소는 *"CGP + 3 * {노드의크기}"* 가 된다. *R5*에 해당하는 질의 영역의 경우, 최종 결과에서 *RightOffset*의 값으로 결정되며, 자식 노드의 주소는 *"CGP + 4 * {노드의 크기}"* 가 된다.

**Algorithm *ComputeOffset***
*현재 노드의 엔트리를 순회하여 찾고자 하는 값 E가 포함된 자식노드의 오프셋을 결정*

1. 엔트리의 순서 번호 *I =0*
2. *I* 번째 엔트리의 LR 값이 'R'인 경우, *LeftOffset = [PI 번째 키 엔트리 순서 번호] + 1* (단, *I*=0 인 경우, *LeftOffset = 0*)
3. 찾고자 하는 값 E 가 현재 엔트리의 DV값의 역변환 값보다 작은지 큰지를 비교하고, 다음 엔트리의 방향을 결정 (*LR = 'L' or 'R'*)
4. *I* 이후의 엔트리 중 *PI,LR*의 값이 3단계까지의 *PI,LR* 값과 일치하는 자식 엔트리 검색
5. 자식 엔트리가 있다면, *I = {자식 엔트리의 인덱스 번호}*, 2단계 반복. 자식 엔트리가 없다면, 6단계 수행
6. *RightOffset = I + 1*
7. 마지막 *LR* 값을 기준으로 *'L'* 일 때 *LeftOffset*을, *'R'*일 때 *RightOffset*을 반환.

### 3.4 분할정책과 전파과정

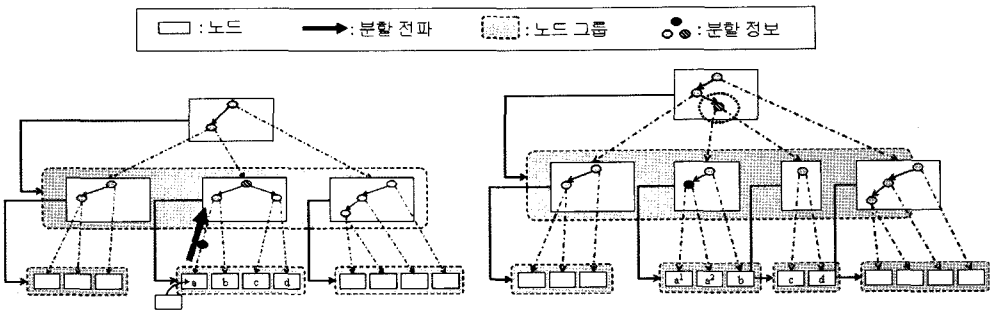
KDBcs-트리의 분할은 크게 단말노드 분할과 노드그룹의 분할로 나눌 수 있다. 오버플로우가 발생한 단말노드는 FD-분할 정책을 이용하여 두 개의 노드로 분할한다. 이때, 분할된 노드와 함께 노드 그룹과 부모 노드가 동시에 고려되어야 한다. 분할 정보는 부모 노드의 엔트리로 추가되고, 새로 생성된 분할 노드는 현재 노드 그룹에 삽입을 시도한다. 만약 삽입이 가능하다면 알고리즘을 종료하고, 그렇지 않으면, 분할이 상위 부모 노드로 전파된다. 중간 노드의 분할은 동시에 하위 노드 그룹의 분할을 의미하며, 중간 노드의 첫 번째 엔트리 노드를 기준으로 FD-분할 정책에 따라 두 개의 노드와 두 개의 그룹으로 분할된다.

그림 10은 분할되는 과정의 예이다. 그림 10(a)처럼 단말노드에서 분할이 발생된 경우, 노드를 분할하고, 노드 분할로 발생한 새로운 노드를 단말 노드 그룹에 삽입하게 된다. 하지만 그림 10(a)처럼 더 이상 노드를 삽입할 수 없는 경우, 그림 10(b)처럼 상위 노드의 첫 번째 키 엔트리를 기준으로 단말 노드 그룹을 둘로 분할하고, 상위 노드 역시 둘로 분할하게 된다. 그 다음 분할된 상위 노드의 정보는 해당 노드 그룹에 삽입을 시도한다. 만약 상위 노드가 포함된 그룹 더 이상 노드를 삽입할 수 없는 경우, 분할 정보의 전파와 함께 중간노드와 그룹의 분할이 발생한다.

중간노드와 그룹의 분할 역시 단말 노드의 분할 과정과 동일하며, 아래로부터 전파된 분할 정보를 현재 노드 그룹에 삽입가능하거나 최종 루트 노드에 도달할 때까지 이 과정이 반복된다. 최종 루트 누드에 도달할 때까지 삽입이 불가능한 경우, 루트 노드와 분할된 정보를 포함하는 새로운 노드 그룹을 하나 생성하고, 이 노드 그룹의 상위 노드를 생성하고, 이 노드를 새로운 루트 노드로 변경한다.

### 3.5 KDBcs-트리의 주요 연산

KDBcs-트리의 검색, 삽입, 삭제 연산은 KDBKD-트리와 유사하며, 노드 구조 변경에 따른 분할값의 양자화와




(a) 분할 전 (b) 분할 후

그림 10 분할의 예

복원이 필요하다는 것이 차이점이다.

3.5.1 검색 알고리즘

범위 질의를 수행하기 위해서 질의 영역과 교차하는 공간 영역을 찾아야 한다. 질의 영역과 교차하는 영역을 찾기 위해서 스택을 이용하여 기존의 KDB-트리와 동일한 방법으로 질의를 처리한다.

**Algorithm *Search***
*질의 영역 Q에 대해 Q와 겹치는 모든 색인 레코드들을 검색*

1. 초기에 빈 스택 $S$에 루트 노드와 실제 영역 정보 $R$을 삽입
2. 만약 스택 $S$가 비었다면, 종료
3. 스택 $S$로부터 노드 $N$과 실제 영역 정보 $D$를 꺼냄.
4. 만약 노드 $N$이 단말 노드가 아니고, 엔트리 수가 0이면, 현재 노드 그룹의 첫번째 노드와 영역 정보 $D$를 스택 $S$에 삽입
5. 만약 노드 $N$이 단말 노드가 아니고, 엔트리 수가 0이 아니면, 각 엔트리 E의 DV정보를 $((D/[\text{양자화레벨}] \times DV)$식에 의해 역양자화를 수행한 분할축 값을 복원하고 $D$를 분할하는 부분 영역 $R$을 구한 뒤, 각 부분 영역 $R$이 $Q$와 겹치는지 검사. 만약 $Q$와 겹친다면, 자식 노드와 $R$를 스택 $S$에 삽입.
6. 만약 노드 $N$이 단말 노드라면, 결과셋에 겹치는 엔트리를 추가한다.
7. 2단계부터 반복

3.5.2 삽입 알고리즘

새로운 엔트리를 $KDB_{CS}$-트리에 삽입하기 위해서 먼저 트리를 검색하여 새로운 엔트리가 삽입될 단말 노드를 선택하고, 그 다음 선택된 단말 노드에 엔트리를 추가하게 된다. $KDB_{CS}$-트리의 단말 노드를 검색하는 과정은 KDB-트리와 유사하지만 노드내에 계층적인 정보를 이용하여 검색한다는 차이점이 있다. 만약 삽입을 시도하고, 오버플로우가 발생하게 되면 분할 알고리즘을 수행하게 된다.

**Algorithm *Insert***
*새로운 레코드 O를 삽입하기 위해서 ChoosePage 알고리즘을 호출하고, 분할의 필요에 따라 SplitPointPage 알고리즘과 SplitRegionPage 알고리즘을 호출*

1. 삽입하고자 하는 레코드 $O$를 포함할 수 있는 단말 노드 $N$을 찾기 위해서 *ChoosePage* 알고리즘을 호출
2. 레코드 $O$가 1단계에서 찾은 단말 노드 $N$에 삽입 가능한지 검사
   2.1 그렇다면, 레코드 $O$를 삽입하고 종료
   2.2 그렇지 않다면, *SplitPointPage* 알고리즘 호출
3. 단말 노드 $N$ 으로부터 루트 노드 방향으로 필요에 따라 상위 노드로 분할 정보를 전파. 이때, 중간 노드가 분할 정보를 저장할 수 없는 경우, *SplitRegionPage* 알고리즘을 호출

3.5.3 삭제 알고리즘

$KDB_{CS}$-트리로부터 레코드를 삭제하는 방법은 먼저 해당 레코드가 포함된 단말 노드를 검색 알고리즘을 통해 검색하고, 단말 노드가 레코드를 포함하고 있는지 여부를 검사하게 된다. 만약 포함하고 있다면, 레코드를 삭제하고, 노드의 NE 값을 감소시킨다.

**Algorithm *Delete***
*$KDB_{CS}$-트리로부터 레코드 E를 제거*

1. 레코드 $E$를 포함할 수 있는 단말 노드 $N$을 찾기 위해서 *ChoosePage* 알고리즘을 호출
2. 1단계에서 찾은 단말 노드 $N$이 레코드 $E$를 포함하고 있는지 검사하고, 그렇다면, 레코드를 삭제한 후, $NE$의 값을 감소

3.5.4 분할 알고리즘

앞에서 언급했듯이 분할 알고리즘은 크게 단말 노드와 중간 노드의 분할로 나눌 수 있다. 분할이 발생한 경우, 노드의 분할과 함께 부모 노드와 현재의 노드 그룹을 동시에 고려해야한다. FD-분할 정책을 변형하여 사용하되, 분할과정에서 역시 계층정보를 유지할 수 있도록 엔트리의 구성 요소인 *PI*, *LR* 값을 적절히 변경해야한다. 단말 노드의 분할이 완료되지 않은 경우, 분할정보는 부모 노드로 전파된다. CR-트리의 경우, 분할정보를 부모노드로 전파하고, 다시 형제 노드로 전파하여 MBR 정보를 조정하는 과정 필요하다. 이와 달리 제안하는 KDBcs-트리는 형제 노드로 전파하는 과정이 필요없다.

**Algorithm *SplitPointPage***
*KDBcs-트리는 KDBKD-트리의 분할 알고리즘을 사용할 수 있으며, 성능 평가에서는 FD(First Division) 정책을 사용하였다.*

**Algorithm *SplitRegionPage***
*오버플로우가 발생한 영역 페이지를 분할*

1. 부모 노드의 첫 번째 엔트리 $V$ 를 기준으로 현재 노드 그룹 $G$ 를 두 개의 노드 그룹 $G$ 와 $G'$ 로 분할. 이때, 부모노드 역시 $V$ 를 기준으로 $P$와 $P'$ 로 분할.
2. 자식 노드로부터 전파된 분할 정보 $PI$ 를 $P$ 또는 $P'$ 에 삽입하고, 전파된 분할 노드 $PN$ 을 $G$ 또는 $G'$ 에 삽입.
3. 부모 노드의 첫 번째 엔트리 $V$ 정보가 부모 노드의 부모 노드 $GP$ 에 삽입 가능한지 검사.
   3.1 만약 가능하다면 $V$ 를 $GP$에 삽입하고 종료.
   3.2 그렇지 않다면 분할 정보 $V$와 분할된 부모노드 $P'$ 를 각각 $PI$, $PN$으로 하여 상위 노드로 전파하고 *SplitRegionPage* 를 호출.

## 4. 실험 및 성능 평가

### 4.1 실험 환경

본 논문의 목적은 데이타 갱신이 빈번한 상황에서의 다차원 데이타를 효율적으로 처리하기 위한 방법을 연구하는 것이므로 기존의 대표적인 캐시를 고려한 다차원 색인 구조인 FF(False-hit Free) CR-트리와 SE(Space-Efficient) CR-트리와 실험을 통해 성능을 비교한다. 실험환경은 256kb의 L2캐시를 사용하는 인텔 펜티엄III 700Mhz프로세서와 128MBytes의 메모리를 갖는 시스템에서 윈도우 2000을 운영체제로 사용한다.

FF CR-트리는 단말노드의 양성 오류(False-positive) 제거 연산 비용을 줄이기 위한 CR-트리의 변형이다. CR-트리는 손실 압축 기법을 사용하고 있기 때문에 결과셋으로 양성 오류를 포함할 가능성이 있다. 양성 오류를 제거하기위해서 최종 단말 노드에서 실제 데이타 값과의 비교 연산이 필요하다. FF CR-트리는 비교 연산 비용을 줄이기 위해서 중간노드만 MBR 압축 기법을 적용하고, 단말 노드는 기존 R-트리 노드와 동일한 구조를 가진다.

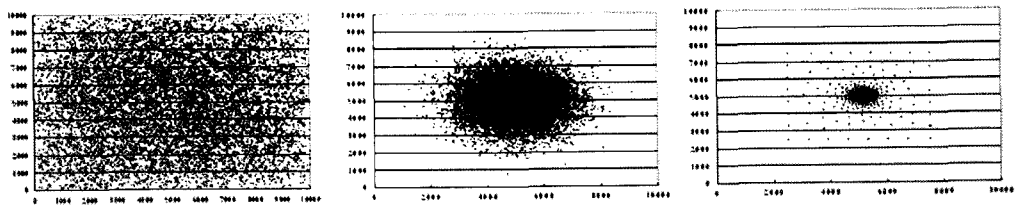
SE CR-트리는 중간노드, 단말노드 모두 압축 기법을 사용하며, CSB$^+$-트리와 같이 자식 노드를 그룹으로 할당하여 노드에 대한 포인터를 제거한 형태의 구조이다. 또한, 노드의 MBR에 대한 정보를 별도로 저장하지 않으며, 노드의 MBR정보는 전체 영역을 기준으로 트리 탐색 과정에서 MBR정보를 이용하여 계산된다. 정확한 색인구조의 성능 비교를 위해서 본 논문에서는 SE CR-트리의 단말 노드 구조를 FF CR-트리와 동일하게 수정하여 양성 오류 제거 연산을 위한 메모리 접근을 제거하였고, 데이타의 연산에 있어서 가능한 한 동일한 코드를 사용하도록 구현하였다.

다양한 실험 평가를 위해서 그림 11과 같은 균등분포 데이타, 가우시안 분포 데이타, TIGER 데이타를 사용한다. TIGER 데이타는 웹사이트[13]에서 제공하는 2차원 분포 데이타를 (0, 10000) 사이의 정수 값으로 적절히 변환하여 사용하였고, 분포 데이타는 산술적인 방법을 통해 생성하였다.

제안하는 색인구조의 우수성을 입증하기 위해서 팬-아웃, 삽입성능, 검색성능, 갱신성능을 비교했다. 첫 번째, 팬-아웃은 산술적인 계산을 통해 평가했고, 두 번째, 삽입성능은 노드의 크기를 변화시키면서 100,000개의 데이타를 순차적으로 삽입하면서 측정했다. 세 번째, 검색성능을 측정하기 위해서 전체 영역의 0.01%, 0.25% 크기에 해당하는 10,000개의 범위 질의를 사용하였으며, 노드의 크기와 차원수를 변경시키면서 접근하는 노드수, 검색 시간 그리고 캐시 접근 실패 수를 측정했다. 네 번째, 갱신성능을 측정하기 위해서 모든 데이타가 현재 위치에서 전체영역의 10%, 1%, 0.1%의 범위로 랜덤하게 갱신되도록 했으며, 노드의 크기와 차원수를 변경시키면서 동일한 데이타에 대한 갱신 시간을 측정했다. 트리의 경우, 기본 R-트리의 갱신 정책인 엔트리를 삭제 후 삽입하는 방식으로 구현했으며, KDBcs-트리의 경우, 앞서 3장에서 제안한 갱신 알고리즘을 적용했다. 모든 그래프에서는 측정한 값의 평균값을 사용하였다.

### 4.2 팬-아웃

기존의 색인 기법과 KDBcs-트리의 팬 아웃은 표 1의 수식 기호를 이용하여 표 2와 같이 계산할 수 있다. 노드의 크기에 따라 선형적으로 모든 노드의 팬-아웃이



(a) 균등 분포 데이터 (b) 가우시안 분포 데이터 (c) 실제데이터(TIGER)

그림 11 성능 평가 데이타 모델

증가한다. FF CR-트리와 SE CR-트리의 경우, MBR 또는 QMBR 의 크기가 차원수에 비례하기 때문에 차원이 커짐에 따라 팬-아웃이 감소한다. 하지만, KDBcs-트리의 경우, MBR 정보 대신 Round-Robin 방식으로 분할 축에 해당하는 정보만 저장하기 때문에 차원과 상관없이 정보를 표현 가능하다.

표 1 팬-아웃을 나타내기 위한 수식 기호

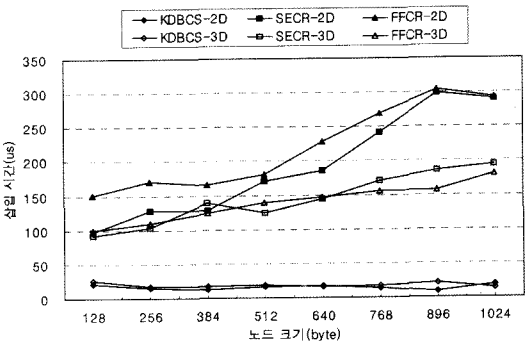
| 기호 | 크기 | 설명 |
|---|---|---|
| S | 128byte~1024byte | 전체 노드의 크기 |
| RPNE | 2byte | RP=중간/단말노드<br>NE=엔트리수 |
| P | 4byte | 포인터 |
| d | - | 차원수 |
| MBR | 4byte × d | MBR |
| QMBR | 1byte × d | 양자화된 MBR |
| PILR | 2byte | PI=부모엔트리의 순서 번호<br>LR= Left / Right |
| DV | 1byte | 양자화된 분할축 정보 |

표 2 팬-아웃 비교(CR-트리 vs. KDBCS-트리)

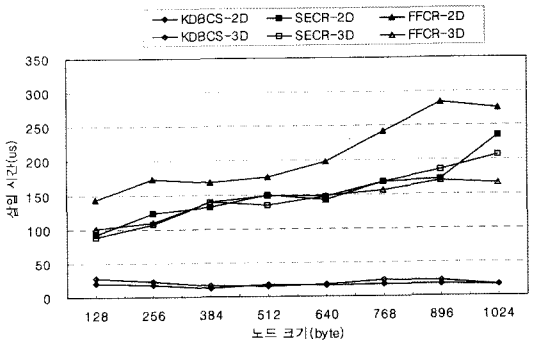
| | 최대 팬-아웃 | |
|---|---|---|
| | 중간 노드 | 단말 노드 |
| SE CR-트리 | (S-RPNE-P)/<br>QMBR | (S-RPNE)/<br>(MBR+P) |
| FF CR-트리 | (S-RPNE-MBR)/<br>(QMBR+P) | (S-RPNE-MBR)/<br>(MBR+P) |
| KDBCS-트리 | (S-RPNE-P)/<br>PILR+DV | (S-RPNE)/<br>(MBR+P) |

### 4.3 삽입 성능

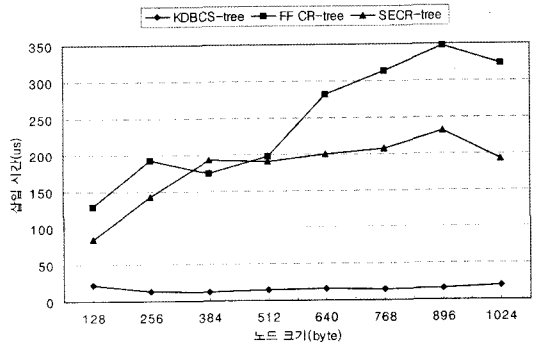
그림 12는 100,000개의 데이타(각각 균등 분포 데이타, 가우시안 분포 데이타, TIGER 데이타)를 삽입할 때, 노드 크기를 변화시키면서 차원 수에 따라 삽입하는데 드는 시간 비용을 측정한 결과이다. SE CR-트리와 FF CR-트리의 경우, 차원이 높아짐에 따라 분할과 MBR 조정에 따른 연산비용이 커지기 때문에 더 많은 시간이 필요하다. 노드 크기가 늘어남에 따라 팬-아웃이 증가하여 노드 접근수가 줄어들기 때문에 단말 노드 검




(a) 균등 분포 데이타

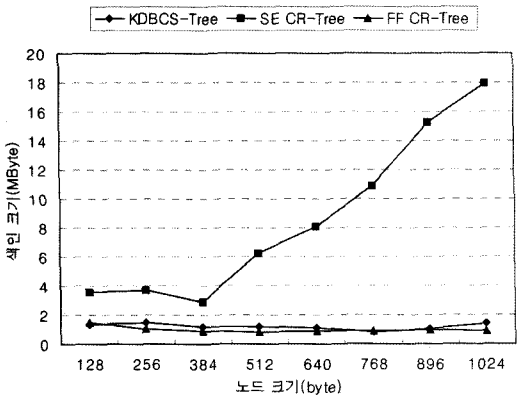



(b) 가우시안 분포 데이타

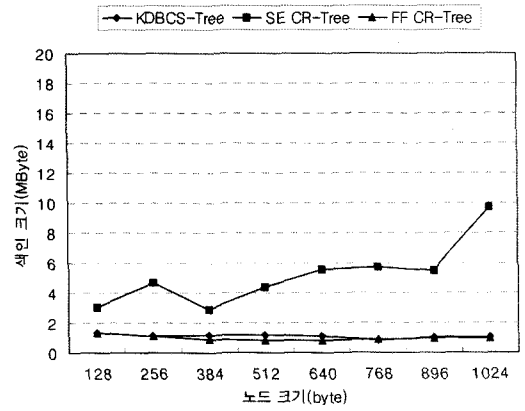



(c) TIGER 데이타

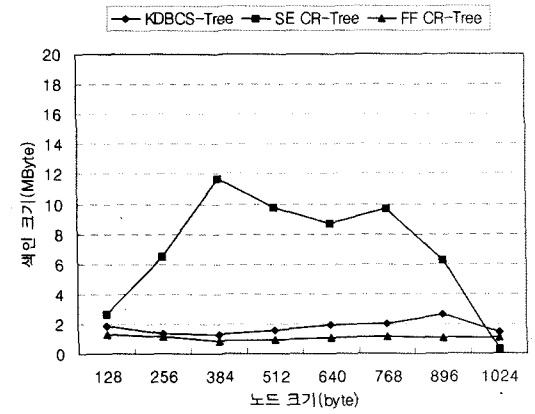
그림 12 삽입시간




(a) 균등 분포 데이타




(b) 가우시안 분포 데이타




(c) TIGER 데이타

그림 13 데이타 삽입에 따른 색인 구조의 크기

색 시간이 단축된다. 하지만 새로운 노드의 삽입과 함께 노드 내 엔트리의 MBR을 조정하고 조정을 전파하는 연산 비용이 많아지기 때문에 삽입시간이 증가하는 성향을 볼 수 있다. 256~384byte, 896~1024byte 구간처럼 일부 노드 크기 구간에서는 전체적인 결과 추세와 다른 예외적인 결과를 보이는데, 이것은 노드의 크기와 삽입하는 데이타의 특성에 따라 노드의 활용도가 달라지기 때문이다.

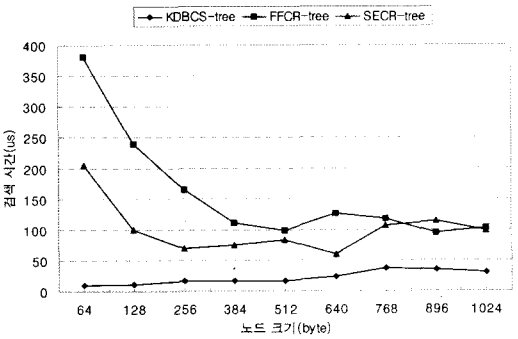
같은 맥락으로 차원을 비교해 보면, 3차원 색인구조의 삽입시간이 2차원 색인구조의 삽입시간보다 짧다는 것을 알 수 있다. KDBcs-트리의 경우, 차원수와 상관없이 거의 균등한 성능을 보여주는 것을 알 수 있다.

그림 13은 10,000개의 데이타를 삽입했을 때 메모리상 색인 구조의 크기를 나타낸다. FF CR-트리에 비해서 KDBcs-트리와 SE CR-트리의 색인 구조가 더 많은 공간을 차지하는 결과를 보여주는데, 이것은 포인터 제거 기법을 위해서 노드의 그룹을 사용하고, 노드가 분할이 이루어질 때, 그룹을 미리 생성하기 때문이다. 데이타가 계속 삽입될 때, FF CR-트리는 분할과 함께 새로운 노드 생성을 요구하지만, KDBcs-트리와 SE CR-트리는 미리 생성해 놓은 노드를 사용하기 때문에 삽입 성능이 향상된다. KDBcs-트리와 SE CR-트리를 비교할 때, KDBcs-트리의 색인 크기가 현저히 작은 것을 알 수 있다.

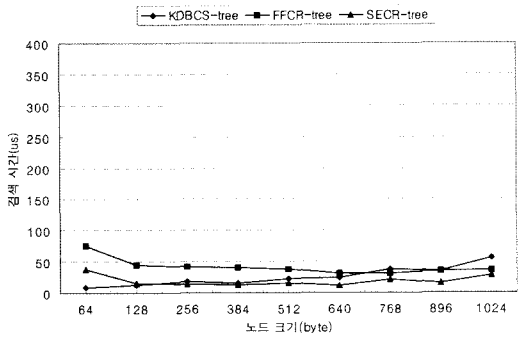
### 4.4 검색 성능

그림 14와 15는 검색 성능을 평가하기 위해서 2차원 색인구조를 대상으로 일정 크기의 범위질의를 수행한 결과이다. 노드의 크기와 검색의 성능은 관련성이 깊다. 일반적으로 노드의 크기가 커짐에 따라 색인구조의 팬-아웃이 커지기 때문에 전체 색인구조의 높이가 낮아진다. 또한, 그림 15와 같이 노드 접근수도 함께 감소하기 때문에 그림 14와 같이 전체 검색 시간이 단축되는 것을 알 수 있다.

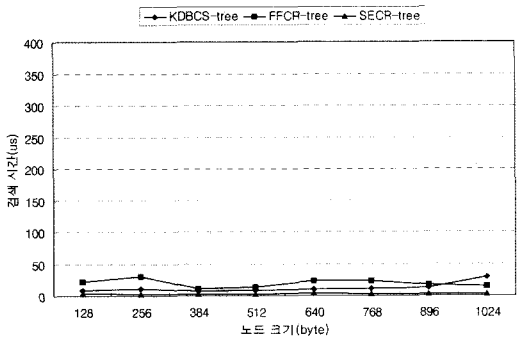
실험 결과, KDBcs-트리는 노드의 크기, 삽입 데이타의 특성과 상관없이 일정한 성능을 보여주지만, CR-트리의 경우, 노드의 크기, 삽입 데이타의 특성에 따라 검색 성능이 크게 달라진다. 이것은 공간 분할 기법을 사용하는 KDBcs-트리의 특성상 데이타 공간의 중첩이 발생하지 않기 때문에 그림 15와 같이 일정한 노드 접근수가 보장되는데 반해 CR-트리의 경우 MBR의 중첩이 발생하기 때문에 노드의 접근수가 일정하지 않게 된다. 따라서 균등분포 데이타와 가우시안 분포데이타의 경우, KDBcs-트리가 좋은 성능을 보여주고, TIGER 데이타의 경우, 오히려 CR-트리가 더 좋은 성능을 보여준다. 이것은 CR-트리는 데이타 분할 방식을 사용하기 때문에 데이타의 분포 특성에 따라 검색과정 중 중간 노드의 Pruning을 통해 검색 성능이 향상되기 때문이다. 반면 KDBcs-트리는 검색 과정 중 Pruning이 불가능하며, 단말 노드까지 무조건 접근해야하기 때문에 데이타의




(a) 균등 분포 데이타

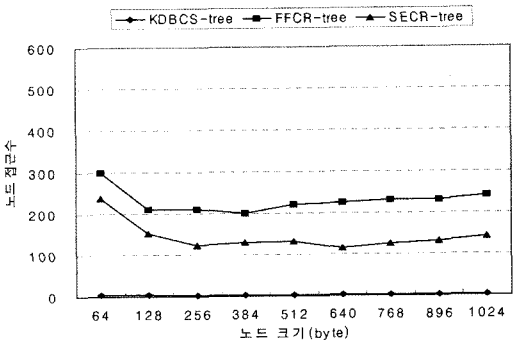



(b) 가우시안 분포 데이타

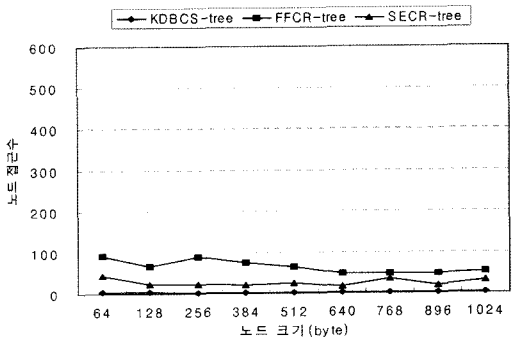



(c) TIGER 데이타

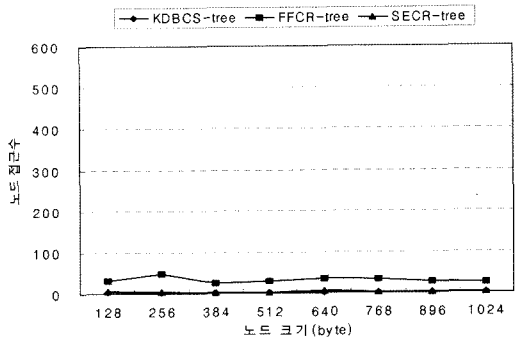
그림 14 검색 시간




(a) 균등 분포 데이타




(b) 가우시안 분포 데이타



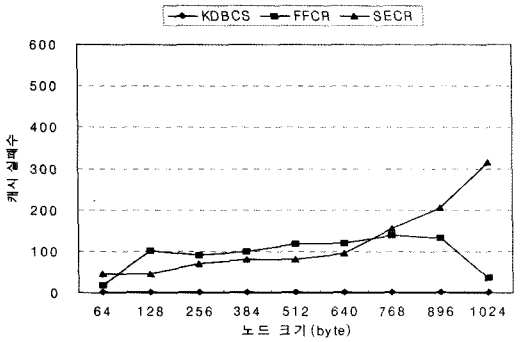

(c) TIGER 데이타

그림 15 노드 접근수

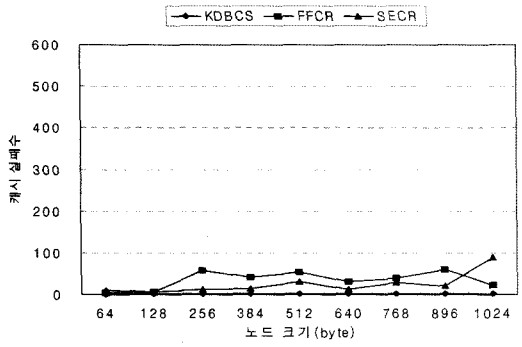
특성과 상관없이 일정한 성능을 보인다.

### 4.5 캐시 성능

그림 16과 표 3은 각각 균등 분포 데이타, 가우시안 분포 데이타, TIGER 데이타에 대한 캐시 실패수를 나타낸 결과이다. 노드의 크기가 증가하면 트리의 높이가 낮아지고 접근하는 노드 수는 줄어들지만, 하나의 노드를 읽기 위해 여러 번의 캐시 접근 실패를 초래하게 된다. 노드 접근 비용이 일정한 $KDB_{CS}$-트리의 경우, 낮고 일정한 캐시 실패수를 보여주며, 데이타의 특성에 따라 노드 접근 비용이 다른 CR-트리의 경우, 데이타 특성과 노드 크기에 따라 캐시 실패수가 다양하게 나타난다. 대체적으로 팬-아웃이 큰 SE CR-트리가 FF CR-트리보다 캐시 실패수가 작은 것을 알 수 있고, 노드의 크기가 커짐에 따라 노드 내부의 엔트리 연산이 증가하여 캐시 실패수가 증가하는 것을 알 수 있다. 그림 16(a)에서 노드의 크기가 768byte이상인 경우 SE CR-트리보다 FF CR-트리가 좋은 성능을 보여주는데 이것은 SE CR-트리가 팬-아웃이 커진 대신 중첩되는 공간이 많고, 노드 내부 연산 비용이 커져 FF CR-트리의 노드 간 이동 비용보다 전체 연산 비용이 상대적으로 커지기 때문이다.

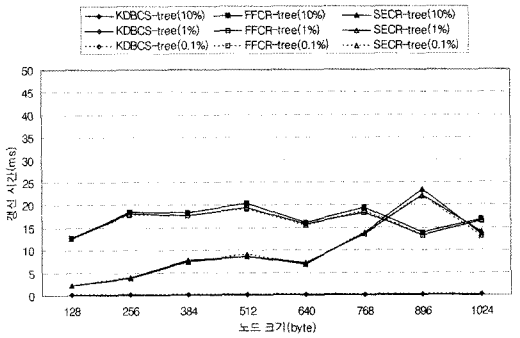



(a) 균등 분포 데이타

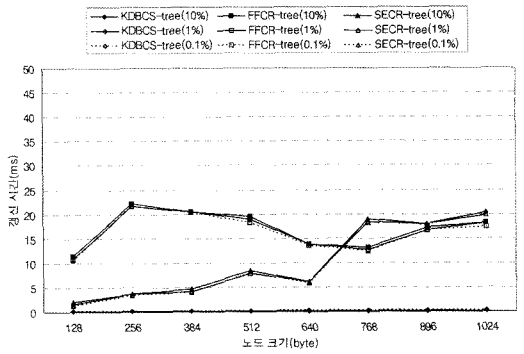



(b) 가우시안 분포 데이타

그림 16 캐시실패수




(a) 균등 분포 데이타




(b) 가우시안 분포 데이타

그림 17 갱신 시간(ms)

표 3 캐시실패수(TIGER 데이타)

| 노드 | KDB$_{CS}$-트리 | FF CR-트리 | SE CR-트리 |
|---|---|---|---|
| 128 | 1.046 | 1.192 | 3.116 |
| 256 | 1.018 | 1.186 | 9.465 |
| 384 | 1.017 | 1.136 | 3.583 |
| 512 | 1.080 | 1.241 | 17.119 |
| 640 | 1.042 | 1.521 | 42.229 |
| 768 | 1.061 | 1.486 | 50.307 |
| 896 | 1.130 | 1.506 | 36.005 |
| 1024 | 1.243 | 1.786 | 5.466 |

표 4 갱신 시간(ms) (TIGER 데이타)

| 노드 | KDB$_{CS}$-트리 | FF CR-트리 | SE CR-트리 |
|---|---|---|---|
| 128 | 0.09 | 11.35 | 2.01 |
| 256 | 0.09 | 22.18 | 3.66 |
| 384 | 0.10 | 20.55 | 4.67 |
| 512 | 0.12 | 19.58 | 8.35 |
| 640 | 0.15 | 13.78 | 6.13 |
| 768 | 0.18 | 13.16 | 18.91 |
| 896 | 0.17 | 17.23 | 17.99 |
| 1024 | 0.20 | 18.25 | 20.35 |

### 4.6 갱신 성능

그림 17과 표 4는 각각 균등 분포 데이타, 가우시안 분포 데이타, TIGER 데이타에 대한 갱신 성능을 평가한 결과이다. 전체 영역의 10%, 1%, 0.1%에 해당하는 범위로 갱신이 이루어지도록 데이타를 만들고, 10,000개의 데이타를 모두 갱신하는데 걸리는 시간을 측정했다. 갱신 성능을 평가한 결과, KDB$_{CS}$-트리가 모든 경우에서 FF CR-트리, SE CR-트리보다 좋은 성능을 보여주었으며, 데이타의 특성과 노드의 크기에 상관없이 거의 일정한 갱신 성능을 보여주었다. 갱신의 경우, 데이타의 삭제와 삽입이 연속적으로 이루어지는 과정이다. 따라서 앞 절에서 분석한 데이타 검색 성능과 삽입 성능에 따라 갱신의 성능이 결정된다.

## 5. 결 론

본 논문에서는 데이타에 대한 갱신이 빈번한 환경에서 캐시 라인의 활용도를 높인 주기억장치 기반의 다차원 색인 기법인 KDB$_{CS}$-트리를 제안했다. KDB$_{CS}$-트리는 KDB$_{KD}$-트리를 기반으로 공간 압축 기법과 노드의 그룹화를 통한 포인터 제거 기법을 활용하여 영역 페이지의 팬-아웃을 크게 증가시켰다. 팬-아웃의 증가로 트리의 높이가 낮아지고 노드 접근 수의 감소와 캐시 실

패 수의 감소를 가져왔다. 또한 차원의 변화와 무관하게 중간 노드의 팬-아웃이 일정하도록 확장성을 보장하여 성능을 향상하였다.

성능 평가를 위해서 다양한 환경에서 삽입, 검색, 갱신과 관련된 기본 알고리즘의 수행 시간과 접근 노드수, 캐시 실패수를 측정했다. 삽입 시간에 대해서는 기존의 색인 구조들보다 모두 뛰어난 결과를 보였고, 검색 시간의 경우, 균등 분포 데이타에 대해서는 뛰어난 성능을 보였지만 가우시안 분포와 실제 데이타를 사용한 경우에는 공간 분할 색인 기법의 단점을 보여주었다. 캐시 실패수와 갱신 성능 부분에서는 기존 색인들보다 모두 월등히 뛰어난 결과를 보여주었다.

본 논문에서는 기존의 캐시를 고려한 다차원 색인 구조들에 대한 자세한 분석을 통해 향상된 성능의 색인 구조를 제안하였고, 제안하는 색인 구조가 기존의 색인 구조들보다 우수함을 보였다. 향후 연구방향으로, 초기 데이타 삽입을 위한 벌크로딩 기법을 제안하고, 다중 사용자 환경에 적합하도록 동시성제어 알고리즘을 적용하는 것이다.

## 참 고 문 헌


[1] Phil Bernstain. et al., "The Asilomar report on database research," Sigmod Record, 27(4), 1998.

[2] Peter A. Boncz, et al., "Database architecture optimized for the new bottleneck: Memory access," Proceedings of the 25th VLDB Conference, pp.54-65, 1999.

[3] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill and David A. Wood, "DBMSs On A Modern Processor: Where Does Time Go?," Proceedings of the 25th VLDB Conference, pp.266-277, 1999.

[4] Jun Rao and Kenneth A. Ross, "Cache Conscious Indexing for Decision-Support in Main Memory," Proceedings of the 25th VLDB Conference, pp.78-89, 1999.

[5] Jun Rao and Kenneth A. Ross, "Making B+-Trees Cache Conscious in Main Memory," Proceedings of the ACM SIGMOD Conference, pp.475-486, 2000.

[6] Kihong Kim, Sang K. Cha and Keunjoo Kwon, "Optimizing Multidimensional Index Trees for Main Memory Access," Proceedings of the ACM SIGMOD Conference, pp.139-150, 2001.

[7] Kaushik Chakrabarti and Sharad Mehrotra, "The Hybrid Tree: An Index Structure for High Dimensional Feature Spaces," Proceedings of the International Conference on Data Engineering, pp. 440-447, 1999.

[8] John T. Robinson, "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes," Proceedings of the ACM SIGMOD Conference, pp.10-18, 1981.

[9] R. Orlandic and B. Yu, "Estimating the Probability of Overlap between Multi-imensional Rectangles in the Analysis of Spatial Structures," Information Sciences, 2001.

[10] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," Proceedings of ACM SIGMOD Conference, pp.47-57, 1984.

[11] Ratko Orlandic, Byunggu Yu, "Implementing KDB-Trees to Support High-Dimensional Data," Proceedings of the International Database Engineering & Applications Symposium, IEEE, pp.58-67, 2001.

[12] Byunggu Yu, Tomas Bailey, Ratko Orlandic, Jothi Somavaram, "KDBKD-Tree: A Compact KDB-Tree Structure for Indexing Multidimensional Data," Proceedings of the International Conference on Information Technology: Coding and Computing[Computers and Communications], IEEE, pp.676-680, 2003.

[13] S.T. Leutenegger, "Multi Dimensional Data Sets," http://www.cs.du.edu/~leut/MultiDimData.html, 1996.



여 명 호

2004년 2월 충북대학교 정보통신공과 공학사. 2006년 2월 충북대학교 정보통신공학과 공학석사. 2006년~현재 충북대학교 정보통신공학과 박사과정. 관심분야는 메인 메모리 기반 데이타베이스 시스템, 시공간 데이타베이스 시스템, 무선 센서 네트워크 등

민 영 수

1998년 2월 충북대학교 정보통신공과 공학사. 2001년 2월 충북대학교 정보통신공학과 공학석사. 2006년 2월 충북대학교 정보통신공학과 공학박사. 2006년~현재 한국전자통신연구원 Postdoc. 관심분야는 데이타베이스 시스템, 위치 기반 서비스, 분산 컴퓨팅, 저장 시스템 등

유 재 수

1989년 2월 전북대학교 컴퓨터공학과 공학사. 1991년 2월 한국과학기술원 전산학과 공학석사. 1995년 2월 한국과학기술원 전산학과 공학박사. 1995년 2월~1996년 8월 목포대학교 전산통계학과 전임강사. 1996년 8월~현재 충북대학교 전기전자컴퓨터공학부 및 컴퓨터정보통신연구소 정교수. 관심분야는 데이타베이스 시스템, 정보검색, 멀티미디어 데이타베이스, 분산 객체 컴퓨팅 등