■ 2003년 정보과학 논문경진대회 수상작

주기억 데이타베이스 인덱싱을 위한 CCMR-트리
(Making Cache-Conscious CCMR-trees for Main Memory Indexing)

윤 석 우 † 김 경 창 ††
(Sukwoo Yun) (Kyungchang Kim)

요 약 매년 CPU 속도가 60% 정도 증가되고, 메모리 속도가 10% 증가되는 현실에서, 캐쉬 미스(Cache miss)를 얼마나 줄이느냐 하는 문제가 현재의 주기억 데이타베이스 환경에서 가장 중요한 문제로 대두되었다. 최근 연구들에서는 R-트리의 변형 모델인 CR-트리와 같은 인덱스 구조들이 제시되었으나, 이는 손실 발생 가능한 압축 기법을 사용함으로써 검색 성능이 더 나빠질 수 있다는 문제점이 있다.

본 논문에서는 MR-트리라고 이름 붙여진 캐쉬 동작에 민감한 R-트리의 새로운 변형 모델을 제시한다. MR-트리는 리프가 아닌 중간 노드 엔트리들을 100%에 가깝게 사용하여 결과적으로 트리의 높이와 중간 노드 엔트리의 MBR을 줄여주는 효과를 준다. 이를 위해 노드 분할 발생시 입력 경로 상에 하나 이상의 빈 엔트리를 지니는 중간 노드가 존재할 경우에만, 노드 분할을 상위로 전송하고, 존재하지 않을 경우 새롭게 생성된 노드는 분할된 노드의 자식 노드가 된다. MR-트리는 이와 같은 동작으로 인해 발생 가능한 트리 불균형 문제를 높이 균형화(HeightBalance) 알고리즘을 수행함으로써 해결한다. 한편, 본 논문에서는 MR-트리를 캐쉬 동작에 더욱 민감한 트리형태로 만들기 위해 CCMR-트리를 제안한다. 본 논문의 실험과 분석 결과, 2차원의 MR-트리는 약간의 개선된 수정 속도와 비슷한 메모리 사용량을 기록하며, 기존의 R-트리에 비해 2.4배 이상의 빠른 검색 속도를 나타냈다.

키워드 : 공간 데이타베이스, 주기억 데이타베이스, 공간 인덱스, 캐쉬, MR-트리, CCMR-트리, R-트리

Abstract To reduce cache misses emerges as the most important issue in today's situation of main memory databases, in which CPU speeds have been increasing at 60% per year, and memory speeds at 10% per year. Recent researches have demonstrated that cache-conscious index structure such as the CR-tree outperforms the R-tree variants. Its search performance can be poor than the original R-tree, however, since it uses a lossy compression scheme.

In this paper, we propose alternatively a cache-conscious version of the R-tree, which we call MR-tree. The MR-tree propagates node splits upward only if one of the internal nodes on the insertion path has empty room. Thus, the internal nodes of the MR-tree are almost 100% full. In case there is no empty room on the insertion path, a newly-created leaf simply becomes a child of the split leaf. The height of the MR-tree increases according to the sequence of inserting objects. Thus, the HeightBalance algorithm is executed when unbalanced heights of child nodes are detected. Additionally, we also propose the CCMR-tree in order to build a more cache-conscious MR-tree. Our experimental and analytical study shows that the two-dimensional MR-tree performs search up to 2.4times faster than the ordinary R-tree while maintaining slightly better update performance and using similar memory space.

Key words : spatial database, main-memory database, spatial index, cache, MR-tree, CCMR-tree, R-tree

· 이 논문은 한국과학재단 특정기초연구[과제번호 : R02-2001-000-00540-0 (2002)]지원으로 수행되었음
† 학생회원 : 홍익대학교 컴퓨터공학과
cs.hongik.ac.kr
†† 종신회원 : 홍익대학교 정보컴퓨터공학부 교수
kckim@cs.hongik.ac.kr
논문접수 : 2003년 5월 14일
심사완료 : 2003년 8월 28일

1. 서 론

실시간 GIS와 같은 모든 실시간 응용들의 주 요구사항은 아주 빠른 응답시간이다. 그러나 기존의 디스크 기반 데이타베이스 시스템은 빈번한 디스크 I/O로 인해 이러한 요구사항을 만족시키지 못한다. 이런 상황에서

실시간 응용들의 위와 같은 요구사항을 위해 주기억 기반의 데이타베이스 시스템이 제안되었으며, 최근 계속되는 램 가격의 하락은 점점 더 주기억 장치 데이타베이스 시스템의 구축을 앞당겼다. 최근 연구[1]에서는 10년 이내에 수백 테라바이트(terabyte) 데이타베이스를 위해 테라바이트 단위의 주기억 장치를 제공하는 일이 흔해질 거라고 예고하고 있다.

요즘의 컴퓨터 시스템들은 CPU와 상대적으로 큰 주기억 장치사이에 캐쉬(Cache) 메모리를 사용한다. 이 캐쉬는 작고 빠른 SRAM으로 구성되며, 최근에 접근했던 데이타들을 저장함으로써 주기억 장치보다 더 빠른 접근 시간을 보장한다. 그러므로 캐쉬 미스를 줄이는 것이 현재의 프로그래밍 환경에서 가장 중요한 문제점중의 하나임을 알 수 있다.

과거 주기억 데이타베이스 시스템에서의 인덱싱 기법은 인덱스 노드를 접근하는데 걸리는 시간과 노드안의 키들을 비교하는데 걸리는 시간을 같은 정도의 비율로 생각했기 때문에, T-트리를 포함한 그의 변형 모델들이 제안되었다. 그러나 최근 CPU의 속도가 매년 60% 정도 증가하는 반면 메모리의 속도가 매년 10%정도 증가하기 때문에, 인덱스안의 키들을 비교하는데 걸리는 시간의 부담이 상대적으로 점점 더 줄어들고 있다[2-4]. 그러므로 현재의 인덱싱 기법에서는 인덱스 키 비교 시간을 줄이는 일보다 인덱스 노드 접근 시간을 줄이는 일이 더 중요하다는 것을 의미한다. CSB^+-트리(Cache sensitive B^+-트리)[5]는 이러한 환경을 고려해 B+-트리의 변형 모델로 그림 1과 같이 제시되었다.

CSB^+-트리는 첫 번째를 제외한 모든 자식 노드들에 대한 포인터들을 제거하기 위해 자식 노드들을 순서대로 메모리 상에 연속적으로 저장시킨다. i 번째 자식 노드에 접근할 경우, 해당 주소는 첫 번째 자식 노드로부터 계산된다(첫 번째 자식 노드 시작 주소 + size_of(노드)*i). 포인터 제거로 인해 발생되는 노드내의 빈공간은 인덱스 키들로 채워지기 때문에, 이러한 방법은 B^+-트리의 노드내의 저장 가능한 엔트리 수를 두 배로 늘려준다. 만약 캐쉬 블록 크기와 같은 인덱스 노드 크기를 사용한다면, 저장 가능 엔트리수가 두 배로 늘어난 인덱스 트리는 트리의 높이를 줄여주고, 트리를 순회할 때 발생 가능한 캐쉬 미스의 수를 최소로 줄여준다.

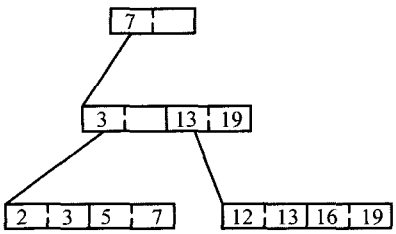


그림 1 CSB^+-트리

다차원 공간 인덱스 구조를 위해 R-트리를 주기억 장치 인덱스 기법으로 사용할 경우, 위에서 사용한 포인터 제거 기술만으로는 트리의 높이를 크게 줄여주지 못한다. 이는 MBR (Mininum Bounding Rectangle)이라 불리는 다차원 공간 인덱스 키가 포인터에 비해 훨씬 더 큰 공간을 차지하기 때문이다[6]. 이를 해결하기 위해 CR-트리(Cache-conscious R-tree)를 포함한 많은 새로운 인덱스 기법들[7-9]이 제안되었다.

캐쉬 미스의 발생 빈도를 줄이기 위해, 본 논문에서는 R-트리의 동작 모습을 다음과 같이 다시 생각해 보았다. 원래의 R-트리에서는 리프 노드의 분할시 항상 이를 상위 노드들로 전달하고, 이는 다시 중간 노드(internal node)들의 분할을 초래하기 때문에 중간 노드들은 평균 70% 정도의 엔트리들만이 채워지게 된다[6]. 또한, R-트리는 높이 균형 트리이기 때문에, 리프 노드들의 높이를 맞추기 위해 불필요하게 트리의 높이를 증가시키는 경우가 발생한다는 점을 고려하였다. 예를 들어 하나의 리프 노드 분할이 발생할 경우 이는 상위 노드로 전송되고, 다시 중간 노드들의 분할로 이어져 새로운 루트 노드가 생김으로써 트리의 높이가 증가하는 경우 발생한다. 만약 추가적인 입력이 없다면 분할이 발생한 하나의 리프 노드를 위해 검색 시 그외 다른 모든 리프 노드들에 대한 접근 노드 개수가 증가하는 문제가 발생한다. 이는 캐쉬 미스 수의 증가를 의미한다. 이러한 문제점을 수정하기 위해 본 논문에서는 트리의 높이 증가를 최대한 억제시키는 방안으로 중간 노드 엔트리들을 100%에 가깝게 사용하는 R-트리의 수정된 모델인 MR-트리(Main-memory R-tree)를 제시한다. MR-트리에서는 같은 부모 노드를 갖는 노드들의 높이 차이는 0 또는 1로 유지된다.

본 논문에서는 MR-트리에 리프 레벨에서만 CR-트리에서 사용하는 QRMBR 기술을 적용시키는 CCMR-트리 역시 제안한다. 본 논문의 실험에서는 MR-트리와 CCMR-트리가 R-트리와 그 변형들에 비해 약간 앞선 입력 및 삭제 성능을 유지하며, 훨씬 뛰어난 검색 성능을 나타낸다. MR-트리의 경우 다양한 검색 사각형 크기에 대해 검색 측면에서 기존의 R-트리와 그 변형들에 비해 2.4배에서 9.5배의 앞선 성능을 보이며, 삽입과 삭제 성능에서는 R-트리 변형 모델들에 비해 약간 앞선 속도를 나타내며, 비슷한 크기의 메모리 공간을 사용한다.

본 논문의 구성은 다음과 같다. 2장에서는 관련연구들에 대해 간략하게 언급하고, 3장에서는 본 논문에서 제시하는 새로운 인덱스 구조에 대한 검색, 삽입, 삭제 알고리즘을 제시한다. 4장에서는 MR-트리, CCMR-트리

에 대해 R-트리와의 성능을 분석하며, 5장에서는 여러 인덱싱 기법들 사이의 실험 결과를 설명한다.

2. 관련연구

2.1 전통적인 캐쉬 인덱싱 기법(Typical Cache Indexing Techniques)

클러스터링(Clustering) 기법[10,11]에서는 동시에 접근되는 데이타 구조들을 하나의 캐쉬 블록 내에 저장한다. 이 기법은 공간적 지역적으로 가까운 데이타들에 대해 한번의 캐쉬 블록 불러오기로 동시에 접근이 가능하기 때문에 암시적인 미리-불러오기(prefetch)를 제공한다. 트리 구조를 클러스터링 할 때 효율적인 방법은 인덱스 노드와 그 서브 노드들을 하나의 캐쉬 블록내에 저장하는 것이다. 그림 2에서는 이진 트리에서 서브 트리에 대한 클러스터링 기법을 보여준다.

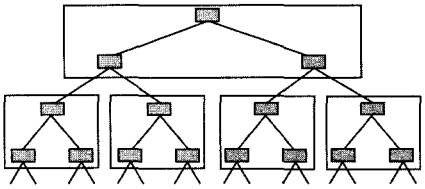


그림 2 서브 트리 클러스터링

문자와 숫자와 같은 1차원 데이타 타입을 위한 인덱스 구조를 생성할 때 이러한 클러스터링 기법은 효율적으로 적용된다. 그러나 캐쉬 블록 크기가 32~64 바이트임을 감안하면, R-트리에 클러스터링 기법을 사용하는 것은 현실적으로 불가능하다. 만약 2차원 인덱스 구조를 생성한다고 할 경우, 각 차원의 데이타 값을 표시하기 위해서는 4 바이트가 필요하기 때문에, 하나의 엔트리를 표현하기 위해서는 각 차원의 최소, 최대 값으로 이루어진 16 바이트의 MBR과 4 바이트의 포인터 저장 공간이 필요하다. 이와 같이 R-트리에서는 하나의 엔트리를 표현하기 위해 20 바이트가 필요하기 때문에, 만약 캐쉬 블록 크기가 128 바이트로 주어진다면, 클러스터링 기법을 적용하기 위해서는 하나의 노드에는 단지 2개의 엔트리만을 저장할 수 있다. 그러므로 서브 트리 클러스터링 방법을 R-트리에 적용시킬 경우에 트리의 높이는 심각하게 커질 것이며 이는 결국 캐쉬 미스 발생을 증가시킬 수 있다.

컬러링[10]이란 동시에 접근되는 인덱스 노드들을 캐쉬내에 서로 겹치지 않는 지역에 저장하는 방법이다. 만약 2-컬러링 방법을 사용할 경우, 자주 접근되는 노드들은 첫 번째 캐쉬 지역에 저장하고 나머지 노드들은 다른 캐쉬 지역에 저장한다. 이러한 메핑 방법은 자주 접근되는 노드들이 서로 충돌하지 않고, 자주 접근되지 않는 노드들에 의해 바뀌지 않는 것을 보장한다. 그러나 시스템 커널이 캐쉬 교환 정책을 수행하기 때문에, 이 방법 역시 공간 인덱싱 기법에 즉각적인 적용이 불가능하다.

압축 기법[6,10]은 하나의 인덱스 노드내에 더 많은 엔트리들을 저장할 수 있게 한다. 이러한 기법은 캐쉬 블록뿐만 아니라 공간 이용율을 증가시켜준다는 장점이 있지만, 키 값들을 압축하고 압축해제하기 위해 더 많은 프로세서 동작이 필요하다는 단점이 있다. 그러나 요즘의 환경에서는 프로세서가 계산하는데 드는 비용이 메모리 접근 비용에 비해 상대적으로 싸기 때문에 상당히 긍정적인 해결책으로 고려할 수 있다. 압축 기법은 키 압축 기법, 포인터 제거 기법 그리고 핫/콜드 노드 분할 기법 등으로 제안되었다.

2.2 SS-트리(SS-tree)

SS-트리(Similarity Search-Tree)[12]는 캐쉬 동작에 민감한 인덱싱 기법을 고려할 때 효율적인 인덱싱 기법으로 생각할 수 있다. SS-트리는 우선 그림 3과 같이 인덱싱할 지역을 나타내는 방안으로 사각형대신 구 형태를 사용한다. 그림 3에서 표현된 구의 중심은 표현된 점들의 중심점으로 나타나며, SS-트리는 트리 구성 알고리즘에서 중심점을 이용하는 방법으로 여러 점들을 여러 구 형태로 표현한다. 어떤 하나의 점을 삽입할 경우, 삽입 알고리즘에서는 새로운 점에 최대로 근접한 중심점을 갖는 서브트리를 선택하는 방법으로 가장 적합한 서브 트리를 찾는다. 만약 어떤 노드가 가득 찰 경우, 분할 알고리즘은 그 자식 노드들의 중심점을 이용해, 각 차원에 대한 편차 값을 비교한 다음, 가장 큰 차이를 갖는 차원을 선택한다. SS-트리는 인덱싱할 지역을 표현함에 있어서 사각형이 아닌 구 형태를 사용하기 때문에, 사각형으로 표현하는 것보다 거의 반 정도의 인덱스 공간을 절약할 수 있다. 이는 구 자체가 차원 수의 값과 반지름을 나타내는 값으로 표현 가능하다는 점에 기인한다. 한편, 사각형 형태로 표현할 경우 각 차원의 최소와 최대 값으로 각 차원을 표현해야 하기 때문에

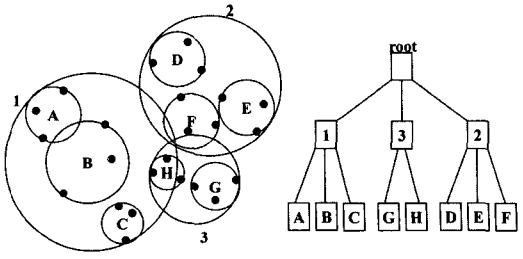


그림 3 SS-트리

차원 수의 두 배에 해당하는 공간이 필요하다. 결국, SS-트리는 R-트리에 비해 하나의 노드 안에 들어가는 엔트리 수를 늘일 수 있으며, 이는 트리의 높이 감소와 캐쉬 미스의 발생 빈도를 줄일 수 있는 장점을 지닌다. 그러나 이러한 장점에도 불구하고, SS-트리는 높은 차원으로 갈수록 사각형에 비해 구형이 불필요하게 더 넓은 공간을 나타내고 이는 검색 성능을 저하시킬 수 있다는 단점이 있다[13].

2.3 CR-트리(CR-tree)

CR-트리[6]는 R-트리의 캐쉬 동작에 민감한 변형 버전에 해당한다. 하나의 노드내에 더 많은 엔트리를 저장하기 위해, CR-트리는 QRMBR(Quantized Relative MBR)이라 불리는 손실 발생이 가능한 압축 기법을 사용한다. CR-트리의 인덱스 노드는 노드안 엔트리들을 모두 포함하는 최소 직사각형인 참조 MBR(reference MBR), 그리고 QRMBR 키와 자식 노드들에 대한 포인터로 구성된 엔트리들로 표현된다. 일반적으로 1 바이트 또는 2바이트로 되는 양자화(Quantization) 레벨을 L로 두고, 참조 MBR을 R로 둔다면, 2차원 인덱싱 기법에서 MBR의 x축 최소 경계값 xl은 다음과 같이 표현된다.

$$if(MBR.xl \le R.xl)\ QRMBR.xl = 0$$
$$else\ if(MBR.xl \ge R.xh)\ QRMBR.xl = L-1$$
$$else QRMBR.xl = |L(MBR.xl - R.xl)/(R.xh - R.xl)|$$

MBR x축의 최대 경계값 xh에 대한 표현은 다음과 같다.

$$if(MBR.xlh \le R.xl)\ QRMBR.xh = 0$$
$$else\ if(MBR.xh \ge R.xh)\ QRMBR.xl = L$$
$$else QRMBR.xh = |L(MBR.xh - R.xl)/(R.xh - R.xl)|$$

CR-트리는 검색 알고리즘에서 각 방문하는 노드의 참조 MBR을 이용해 검색 사각형을 해당 노드에 해당하는 QRMBR로 변경시킨 다음 검색을 수행한다. 삽입 알고리즘에서는 인덱스 노드를 방문할 때마다, 새롭게 삽입되는 객체에 대한 MBR은 해당 인덱스 노드의 참조 MBR을 이용해 QRMBR로 변경시킨 다음, 이를 포함시키기 위해 최소의 크기 증가가 필요하는 자식 노드를 찾는다. 만약 이러한 방법으로 리프 노드까지 찾았다면, 우선 이 리프 노드의 참조 MBR은 삽입될 객체의 MBR을 포함한 최소 사각형의 형태가 되도록 수정된다. 그런 다음, 해당 객체를 위한 새로운 엔트리가 노드내에 생성된다. 만약 노드의 참조 MBR이 삽입된 객체로 인해 커졌다면, 이전 참조 MBR를 통해 QRMBR들에 대한 원래의 MBR을 계산 한 다음, 수정된 참조 MBR을 이용해 모든 QRMBR들이 재 계산된다. 만약 삽입시 노드내에 빈 엔트리가 없다면, 해당 노드는 두개의 노드로 분할된다. 이 경우 분할에 따라 참조 MBR이 수정되며, 노드안의 QRMBR들 역시 참조 MBR에 따라 다시 재계산된다. 이 분할 정보는 필요하다면 루트까지 상위 레벨로 전달된다.

CR-트리의 문제점은 QRMBR이 재 계산될 때마다 커질 수 있다는 점에 있다. 그림 4는 그 한 예를 보여준다.

그림 4의 (a)는 어떤 중간 노드의 원래의 MBR 정보를 표현한 것이며, (b)는 이를 QRMBR 형태로 표현한 것에 해당한다. 만약 최소 값으로 (115,115)를 가지고, 최대 값으로 (120,120)를 가지는 새로운 객체 MBR이 삽입될 경우, CR-트리는 그림의 R3를 삽입 경로로 선택한다. 그런 다음, CR-트리는 이 노드 경계 MBR이 커졌기 때문에, 참조 MBR과 QRMBR을 재 계산하게 된다. 이 경우 참조 MBR을 재 계산하기위해 우선 CR-트리는 QRMBR 생성 기술을 역으로 적용시켜 원래의 MBR 값을 계산한 다음, 참조 MBR을 조정한다. 그런 다음, 조정된 참조 MBR과 계산해낸 원래의 MBR 값을 이용해, QRMBR을 재 계산한다. 그림 4의 (c)는 이러한 일련의 계산을 거쳐 새롭게 표현된 노드의 모습을 나타낸다. 여기서 R1의 경우, CR-트리는 최소 값으로 (25,25)를, 최대 값으로 (43.75,43.75)를 원래의 MBR 값으로 계산한 다음, 조정된 참조 MBR을 이용해 (c)와 같은 QRMBR을 계산한다. 그러나 여기서 R1의 바른

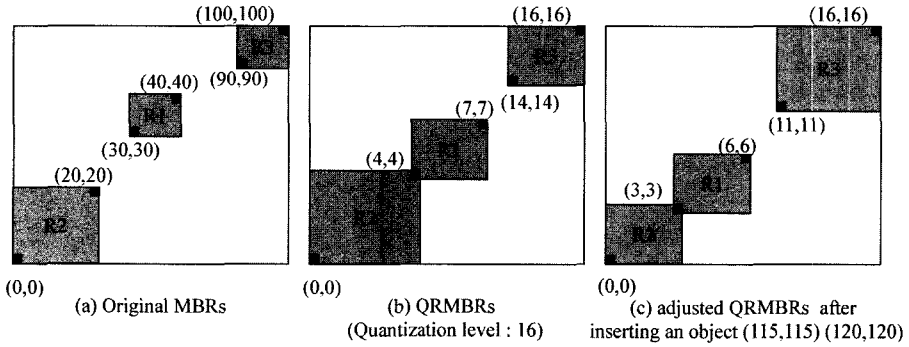


그림 4 CR-트리 이상 현상 예제

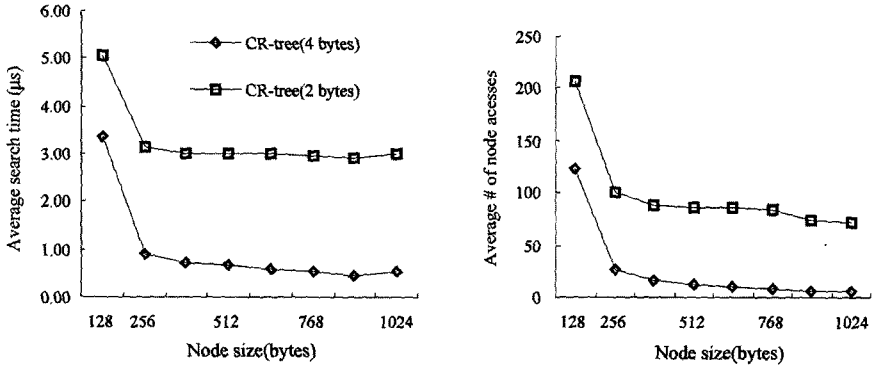


그림 5 서로 다른 QRMBR 크기를 사용할 경우 검색 성능 및 접근 노드의 수 비교

QRMBR 최소 값은 (4,4)로 구성되어야 한다. 이와 같이 잘못되게 커진 QRMBR은 CR-트리의 동작에 나쁜 영향을 미친다. 만약 위의 (c)상황에서 최소 값으로 (20,20), 최대 값으로 (23,23)을 지닌 새로운 객체 MBR이 삽입될 경우, 우선 새로운 MBR은 최소 값으로 (2,2), 최대 값으로 (4,4)를 지닌 QRMBR로 변경된다. R-트리의 경우에는 이 새로운 MBR을 삽입하기 위한 경로로 R2를 선택하지만, CR-트리는 R1을 선택할 수 있다. 이는 R1 역시 새로운 객체 QRMBR를 포함하기 위한 최소 직사각형 증가를 가져오기 때문이다. 이러한 이상현상은 루트 노드에 가까운 상위 중간 노드일수록 자주 발생할 수 있다. 왜냐하면 이러한 중간 노드들의 QRMBR은 하위 노드들의 QRMBR보다 상대적으로 더 크고 삽입하려는 객체는 이러한 노드들에서 더욱 작은 QRMBR로 표현되기 때문이다. 이러한 이상 현상은 또 노드의 분할시에도 잘못된 결과를 초래할 수 있다. 이렇게 되면 질의 사각형과 겹치는 객체들을 검색할 경우, 이러한 이상현상은 방문해야 하는 노드의 수를 상당히 증가시킬 수 있다. 결과적으로 CR-트리는 공간 사용 측면에서는 좋은 성능을 항상 보장하지만, 작은 크기의 QRMBR로 인해 검색 성능 저하를 초래할 수 있다. 서로 다른 크기의 QRMBR을 사용하는 CR-트리를 비교하기 위해 본 논문에서는 단위 크기의 정사각형 내에 측면 길이가 0.001인 백만 개의 작은 사각형을 균등 분포를 갖도록 생성했다. 그림 5는 QRMBR의 크기를 4 바이트와 8바이트로 하였을 경우를 비교한 결과이다. 여기서 사용한 질의 검색 사각형의 크기는 전체 1 평방 크기의 0.01%에 해당한다. 이 실험 결과에서는 4 바이트의 QRMBR을 사용하였을 경우가 8 바이트의 QRMBR을 사용하였을 경우보다 접근해야 되는 노드의 수가 더 많음을 알 수 있으며, 이는 앞의 이상 현상으로 설명할 수 있다.

3. MR-트리

앞에서도 언급한 바와 같이 이 논문의 목적은 원래의 R-트리보다 더 캐쉬 동작에 민감한 변형된 R-트리를 만드는 것이다. 원래의 R-트리는 높이 균형 트리이기 때문에, 항상 노드 분할을 상위 노드로 전달하므로, 결국 노드안에는 엔트리들이 평균 70% 정도만이 차있게 된다. 다시 말해, R-트리는 리프 노드들의 높이 균형을 위해 불필요하게 트리의 높이를 증가 시키고, 이는 결국 불필요한 캐쉬 미스 발생을 초래한다. 그러므로 본 논문에서는 캐쉬 미시 발생을 줄이기 위해 R-트리와 AVL-트리의 특성을 결합시킨 MR-트리(Main-memory R-tree)라는 새로운 인덱스 구조를 제안한다.

3.1 인덱스 구조(Index Structure)

MR-트리는 R-트리의 인덱스 구조와 같은 기본적인 인덱스 구조를 사용하지만, 다음과 같은 특성을 지닌다. 첫째, R-트리는 모든 리프 노드들의 높이가 같은 균형 트리의 형태를 지니지만, MR-트리는 서브 트리들 사이의 높이가 불균형 할 수 있으며 그 차이는 최대 1 이하로 된다. 만약, 서브 트리들 사이의 높이가 1 이상이 될 경우, 높이 균형화(HeightBalance) 알고리즘을 수행한다. 둘째, R-트리와 달리 분할 발생시 삽입 경로상에 빈 엔트리를 지닌 노드가 존재할 경우에만, 분할을 상위 노드로 전달한다. 마지막으로 삭제의 경우에도 선택적으로 재삽입 여부를 판단한다는 특성을 지닌다. 이를 위한 MR-트리의 구조는 그림 6과 같다.

MR-트리의 인덱스 노드들은 그림 6과 같이 중간 노드(internal node)들, 리프 노드(leaf node)들, 그리고 반-리프 노드(half-leaf node)들로 분류된다. 여기서 반-리프 노드라 함은 리프 노드들과 데이타 객체에 대한 엔트리들을 모두 지니는 노드를 의미한다. 만약 새로운 객체가 계속해서 이 반-리프 노드에 삽입된다면, 이 노

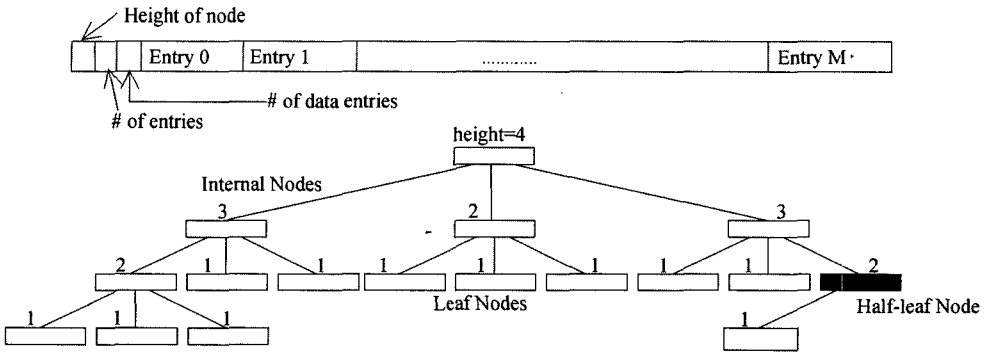


그림 6 MR-트리 구조

드는 높이 2를 지니는 중간 노드로 변경된다. MR-트리의 노드 구조는 R-트리의 노드 구조와 구별되는 두 가지의 추가적인 필드가 존재한다. 하나는 노드의 높이(height of node)를 나타내는 필드로 자식 노드들의 높이들 중 가장 큰 높이의 값에 1을 더한 값을 지니며, MR-트리의 불균형 상태를 검색하는데 사용된다. 반-리프 노드에 의해 사용되는 데이타 엔트리의 수(# of data entries) 필드는 노드 안에 저장하고 있는 데이타 엔트리의 수를 저장한다. 한편, 반-리프 노드내의 엔트리들이 자식 노드를 위한 엔트리인지, 데이타 객체를 위한 엔트리인지를 추가적인 필드 없이 식별하기 위하여, CR-트리에서는 데이타 엔트리들을 자식 노드를 위한 엔트리들 다음에 배치시킨다. 그러므로 데이타 객체들을 위한 첫 번째 엔트리 번호는 엔트리의 시작 번호를 0으로 생각할 때 노드안 엔트리의 수(# of entries) 필드에 해당한다.

3.2 검색(Search)

MR-트리의 검색 알고리즘은 R-트리의 검색 알고리즘과 비슷하다. 단지 차이점이라면 반-리프 노드를 검색할 경우 질의 사각형을 자식 노드를 위한 MBR들 뿐만 아니라 데이타 객체를 위한 MBR들과도 비교를 한다는 점을 들 수 있다.

```
알고리즘 Search(N, Q)  // Q : 질의 사각형, N : MR-트리의 노드
  for (all entry E in the node N) {
      if(E.MBR overlaps Q) {
          if(E.height==1) Return E.
          elsif(E.height==2 && offset_of_E
              ≥ #_of_entries(N)) Return E.
          else Search(E.child, Q).
      }
  }
```

3.3 삽입(Insertion)

삽입 알고리즘에서는 노드 분할 발생시만 다르게 동작한다. 루트에서 리프 노드까지 내려오면서 MR-트리는 각 내부 노드가 빈 엔트리를 지니는지 여부를 검사한다. 만약 그 중 하나 이상의 노드가 빈 엔트리를 지닌다면, 노드 분할 발생시 이를 상위로 전달한다. 만약 모든 내부 노드들의 엔트리가 가득 차있다면, 분할된 노드는 새롭게 생성된 노드와 데이타 객체들을 위한 엔트리를 지니는 반-리프 노드가 된다. MR-트리는 삽입 순서에 따라 트리의 높이가 커지고, 중간 노드들의 높이 차이가 증가할 수 있다. 만약 MR-트리의 트리조정(AdjustTree) 알고리즘 내에서 어떤 내부 노드의 서브 트리들 간의 높이차이가 2가 되는 서브 트리를 발견하면, 높이균형(HeightBalance) 알고리즘이 호출되고, 이는 중간 노드의 서브 트리들 간의 높이 차이가 최대 1이 되도록 MR-트리를 재 균형화시킨다.

```
알고리즘 Insert(N,E)
// MBR E를 지는 새로운 객체 O를 MR-트리 N에 삽입
 I1. L = ChooseLeaf(N,E,&flag)
        //If one of the internal nodes on the path
          has empty entry
          then flag=1 else flag=0.
 I2. if (L has room for another entry) install E;
 I3. elsif (L is a half-leaf node && full)      {
        Make new child node CL of L and fill
        CL by all data entries in L;
        install E in L;
 I4. } else {                          // L is a leaf node
        LL = splitNode(L,E); //Node L is split into L,LL
        if (flag==0) {
              Create a new entry ELL
              ELL.child=LL; Adjust(ELL.MBR);
              Add ELL to L;
        }
     }
 I5. if (split was not performed !!flag==0)
        AdjustTree(L,NIL)
```

I6. **else AdjustTree(L,LL);**

알고리즘 경로선택(ChooseLeaf). 경로선택(Choose-Leaf) 알고리즘은 R-트리에서 사용하는 것과 비슷하다. 차이점은 중간 노드 검색시 빈 엔트리가 존재하면 플래그(flag)를 1로 지정하고, 아닐 경우 0으로 지정한다. MR-트리가 반-리프 노드에 도착할 경우, 우선 데이타 객체들의 모든 엔트리 MBR들을 포함하는 MBR을 생성한 다음, 그 MBR과 다른 서브 트리에 대한 엔트리 MBR들중 최소 증가 크기를 이루는 하나의 엔트리를 선택한다.

알고리즘 분할(Split). MR-트리에서는 R-트리와 R*-트리[11,14,15]를 포함한 모든 R-트리 변형들에서 사용되는 분할 알고리즘들이 적용 가능하다. 단지 분할 발생시 분할 발생 노드와 생성된 노드의 높이가 재 계산되어야 한다는 차이가 있다. 본 논문의 실험에서는 R-트리의 선형비용(linear-cost) 분할 알고리즘을 사용했다.

알고리즘 AdjustTree(N, NN)

A1. **if** (N is a root) **return**;

A2. Let P be a parent of N; Let EN be a N's entry in P; PP=NIL;
Adjust(EN.MBR); *// to encloses all entry MBRs in N.*

A3. **if**(NN <> NIL) {
Create a new entry ENN; ENN.child=NN;
Adjust(ENN.MBR);
if (P has an empty room) add ENN to P;
elsif(P is a half-leaf **&&** #_of_data_entries(P) > 0) **AdjustHalfLeaf(P, E_{NN});**
else PP = **splitNode(P,E_{NN});**

A4. } **elsif**(N.height ≥ P.height) {
P.height = N.height +1;
if(exists a child node C of P such that P.height==C.height+3)
HeightBalance(P,N);
}

A5. **AdjustTree(P, PP)**

Algorithm AdjustHalfLeaf(N,E)

L1. Create a new node L; Move all data entries in N into L;

L2. Create a new ET ; ET.child=L;
Adjust(ET.MBR); *// to enclose all MBRs in L*

L3. Add ET to N;

L4. Add E to N;

중간 노드들의 높이 차이는 MR-트리가 삽입 순서에 따라 커질 수 있다는 특징 때문에 증가될 수 있다. 높이 균형(HeightBalance) 알고리즘은 중간 노드의 서브 트리들 사이의 높이 차이가 2가 될 경우 트리조정(AdjustTree) 알고리즘에 의해 호출된다. 그림 7은 MR-트리에서 높이 균형화시키는 예를 보여준다.

그림 7의 (a)는 높이 균형화 작업 전의 MR-트리를 보여준다. 여기서 노드 P의 높이는 P의 자식 노드 높이가 증가하는 만큼 따라서 증가된다. (a)에서는 P의 자식들 사이의 높이 불균형이 발생됨을 알 수 있다. 트리조정(AdjustTree) 알고리즘은 이러한 상황을 감지하면 원인을 제공한 노드 N_1을 높이균형(HeightBalance) 알고리즘을 호출할 때 파라미터 값으로 넘긴다. 여기서 노드 N_1 역시 자신의 자식 노드들 중 하나 때문에 그 높이가 증가되었기 때문에, 높이균형 알고리즘은 우선 이 원인 제공 노드인 C_1을 찾는다. 그런 다음, C_1을 위한 N_1 내의 엔트리는 E_T라는 임시 엔트리에 저장시킨 다음 N_1으로부터 제거한다. 이제 C_1을 위한 엔트리가 임시 엔트리 E_T로 옮겨졌기 때문에, MR-트리는 N_1의 높이와 P의 높이를 줄인다. 두 번째로 P의 자식 노드들과 C_1중에서 서로 합칠 때 최소의 MBR 증가 크기를 갖는 한 쌍의 노드를 찾는다. 그런 다음, 선택된 두 노드의 엔트리들은 하나의 엔트리로 병합되고 임시 엔트리 E_T는 병합과정이 끝난 다음 다시 P에 추가된다. 여기서 병합을 위한 한 쌍의 노드를 찾을 때, 만약 후보 노드들이 모두 높이로 P.height-1의 크기를 갖는다면, 두 후보 노드에 대한 엔트리들을 지니는 새롭게 생성될 P의 자식 노드는 그 높이로 P.height를 갖게 되고, 이는 다시 P의 높이 값을 증가시켜야 하는 문제점이 생긴다. 이러한 문제

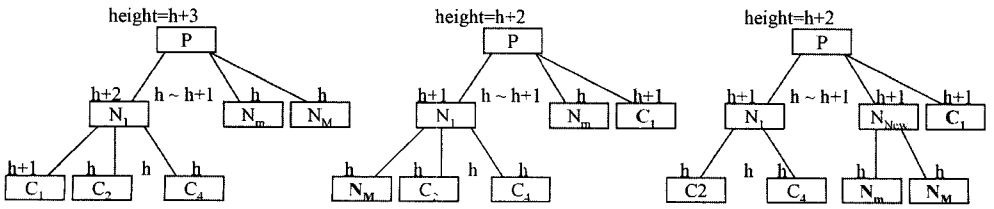


그림 7 MR-트리에서 높이 균형화 과정

점은 하나의 후보 노드의 높이가 P.height-2이고, 다른 하나의 후보가 빈 엔트리가 없는 높이가 P.height-1인 노드일 경우 역시 발생 할 수 있다. 이러한 이유로 한 쌍의 후보 노드를 선정함에 있어서 하나는 P.height-2 높이를 지녀야 하며, 다른 하나는 P.height-2 높이를 갖거나, 빈 엔트리를 갖는 P.height-1 높이의 노드가 되어야 한다. 그림 7의 (b)와 (c)는 높이균형 알고리즘 수행 후의 가능한 MR-트리 모습을 나타낸다.

알고리즘 **HeightBalance(P,N)**

H1. Search the entry E_C in N, where E_C.child ->Height+1 == N.height;
H2. Create a new entry E_T copy E_C to E_T; Remove E_C from N;
H3. N.height--; P.height--;
H4. **for**(each pair of entries <E_i,E_j> from the entries in P including E_T){
  Let N_{Ei} be the node pointed by E_i.child and N_{Ej} by E_j.child;
  if(N_{Ei}.height==P.height-2 **&&** (N_{Ej}.height ==P.height-2 !! N_{Ej} has empty room)) {
    Compose a rectangle J including E_i.MBR and E_j.MBR;
    Calculate d_{ij} = area(J)-area(E_i.MBR)-area(E_j.MBR);
  }
}
H5. Choose the pair <E_i,E_j> with the smallest d_{ij};
H6. **if**(both N_{Ei} and N_{Ej} have same Heights){
  //height==P.height-2
  Create a node N_{New} Add entries E_i, E_j to N_{New};
  Create an entry E_{New} for N_{New};
  Remove the entry E_j from P; Add E_{New} to P; *// 그림7(c)*
H7. } **else** { *// NEi.height== NEj.height-1*
H8. Add E_i to N_{Ej}; Adjust(E_j.MBR);
  // to encloses Ei.MBR; 그림7(b)
}
H9. Remove E_i from P; Add E_T to P;

3.4 삭제(Deletion)

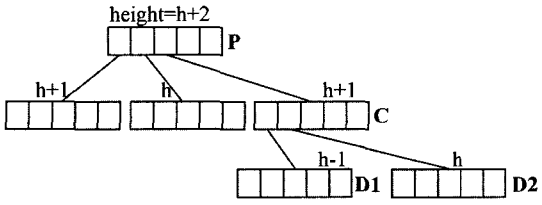
MR-트리에서는 R-트리 변형들에서 사용하는 것과 같은 삭제 알고리즘을 사용한다. 그러나 R-트리 변형들의 삭제 알고리즘과 비교하여, 트리압축(CondenseTree) 알고리즘에서 재 삽입 수를 줄이기 위해 약간의 수정을 가한다. 그림 8은 그 예를 보여준다.

원래의 R-트리 계열에서, 삭제(Delete) 알고리즘에 의해 호출되는 트리압축(CondenseTree) 알고리즘은 만약 노드 N이 지정한 최소 엔트리 수보다 적은 수의 엔트리를 갖게 될 경우 이 노드 N을 집합 Q에 삽입한다. 그런 다음, 마지막 단계에서 Q안의 모든 엔트리들은 R-트리로 재 삽입된다. 만약 R-트리의 트리압축 알고리즘이 그림 8의 (a)와 같은 상황을 만난다면, 노드 C는 단순히 Q로 삽입될 것이고, 마지막 단계에서 노드 D1과 D2는 재 삽입 과정을 거치게 될 것이다. 그러나 MR-트리의 트리압축(CondeseTree) 알고리즘에서는 우선 노드 P안에 빈 엔트리가 있는지 여부를 검사한다. 만약 빈 엔트리가 있다면, 다시 노드 C의 자식 노드들에 대한 높이를 조사한다. 그림 8의 (a)에서 D1의 높이는 h-1이고 D2의 높이는 h이기 때문에, 노드 D2는 재삽입 없이 P의 자식 노드로 연결이 가능하다. 이는 P와 D2의 높이 차이가 2에 불과하기 때문이다. 그러므로 트리압축 알고리즘은 그림 8의 (b)에서와 같이 노드 D2를 P의 자식 노드로 지정하고, 단지 D1만을 Q로 삽입한다. 이와 같은 방법으로 MR-트리는 R-트리에 비해 재 삽입되어야 할 엔트리의 수를 줄일 수 있다.

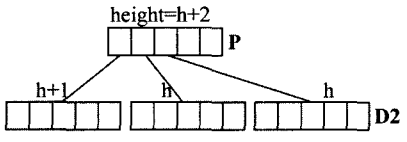
알고리즘 Delete. MR-트리에서 인덱스 엔트리 E를 삭제한다. MR-트리는 R-트리 변형 모델들에서 사용하는 것과 같은 삭제 알고리즘을 사용한다.

알고리즘 **CondenseTree(N)**

C1. Set Q, the set of eliminated nodes, to be empty;
C2. **While**(N is not the root) {
C3. Let P be the parent of N and let EN be N's entry in P;
C4. **if**(N has more than m entries) Adjust(EN.



(a) The MR-tree before condensing.



(b) The MR-tree before reinsertion.
(Only D1 is added to set Q)

그림 8 MR-트리에서 트리 압축과정 예

```
        MBR); // to tightly contain all entries in N.
C5.   else {
C6.     Delete EN from P; recalculate P.height;
C7.     for(each entry E of N) {
                  Let T be the node pointed by
                  E.child;
                  if(P.heightT.height < 3 && P
                  has empty room) add T to P;
                  else add T to set Q;
        }
      }
    }
C8. Insert all entries of nodes in set Q if Q is
    not empty;
```

3.5 CCMR-트리

원래의 CR-트리 경우, QRMBR들로부터 중간 노드에 대한 원래의 MBR들을 계산해 낼 수 없었기 때문에 잘못된 삽입 경로가 선택될 수 있었다. 이것은 결국 CR-트리가 항상 좋은 검색 성능을 보장하지 못함을 야기했고, 양자화(quantization) 레벨 선택에 있어서 신중함을 기해야 하는 이유가 되었다. 본 논문에서는 2.3장에서 서로 다른 양자화 레벨을 사용했을 경우의 검색 성능을 비교 실험했다. 이 실험에서는 4 바이트의 QRMBR을 사용하는 것이 8 바이트의 QRMBR을 사용하는 것에 비해 오히려 더 나쁜 성능을 나타냄을 보였다.

QRMBR 기술은 본 논문에서 제시하는 MR-트리에 직접적인 적용이 가능하다. 그러나 이 역시 CR-트리와 같은 이유로 좋은 성능을 항상 보장하지 못한다. 그러므로 본 논문에서는 QRMBR 기술을 CR-트리와는 다르게 단지 리프 노드들에게만 적용시킨다. 만약 리프 노드의 참조 MBR이 조정되었을 경우, 이 리프 노드의 QRMBR들은 QRMBR 기술을 역으로 적용시켜서가 아니라, 주기억 데이타베이스에 저장된 데이타 객체 MBR들을 이용해 직접적으로 재 계산한다. 이러한 객체 MBR들은 리프 노드 분할이 발생했을 경우 역시 적용된다. 이렇게 수정된 MR-트리를 이 논문에서는 CCMR-트리(Cache Conscious MR-tree)라 부른다.

CCMR-트리의 검색 알고리즘은 MR-트리의 검색 알고리즘과 비슷하다. 단지 차이점은 리프 노드에 도착했을 때, 질의 사각형은 리프 노드의 참조 MBR을 이용해 QRMBR로 변경되고, QRMBR을 이용해 리프 노드의 각 QRMBR들과 중복 여부를 검사한다.

새로운 객체를 삽입할 경우, CCMR-트리는 MR-트리와 같이 루트에서부터 자식 노드를 선택해 나간다. 그리고 리프 노드에 도착했을 때, CCMR-트리는 우선 빈 엔트리의 존재 여부를 검사하고, 만약 존재하지 않으면, 각 QRMBR들에 대한 원래의 MBR들을 이용해 분할 알고리즘을 실행한다. 여기서 원래의 MBR은 데이타 객체와 그에 대한 MBR을 저장하고 있는 주기억 데이타베이스로부터 얻어 진다. 만약 빈 엔트리가 존재한다면, CCMR-트리는 참조 MBR이 조정될 필요가 있는지를 검사하고, 만약 조정된다면, 원래의 MBR들을 이용해 리프 노드의 QRMBR들을 재 계산한다. CCMR-트리의 높이균형 알고리즘은 기본적으로 MR-트리와 같은 골격을 갖는다.

MR-트리와 같이, CCMR-트리는 삭제될 엔트리를 지니는 리프 노드를 검색한다. 삭제 후엔 만약 필요하다면, 참조 MBR을 데이타베이스에 저장된 원래의 MBR들을 이용해 재 계산한다. 그런 다음, 리프 노드의 모든 QRMBR들이 참조 MBR과 원래의 MBR들을 이용해 재 계산된다. CCMR-트리는 MR-트리와 같은 트리압축(CondenseTree) 알고리즘을 사용한다.

4. 분 석

제안한 MR-트리와 R-트리를 분석적으로 비교하기 위해, 본 논문에서는 다른 논문들과 같은 전제 조건을 이용해 분석한다[6,16-18]. 첫 번째로, 데이타 도메인이 정 사각형 형태의 단위 크기 내에 존재하고, 데이타 객체들은 이 도메인 내에 균등하게 분포하고 있고 가정한다. 두 번째로 R-트리의 리프 노드들이 거의 같은 크기의 정사각형 같은 MBR을 지닌다고 가정하며, 세 번째로, MR-트리 역시 R-트리와 같은 경로선택(choose-Leaf)과 분할(Split) 알고리즘을 사용하기 때문에 R-트리와 같은 조건을 갖는다고 가정한다. 마지막으로 질의 MBR 역시 정사각형의 모양을 갖는다고 가정한다.

f를 R-트리의 최대 수용 가능한 엔트리의 수로 가정할 경우, 노드의 평균 엔트리의 수는 $0.7f$로 생각할 수 있으며, N을 도메인내의 데이타 MBR들의 수로 가정할 경우, 이를 이용해 리프 노드의 수를 $\lfloor N/0.7f \rfloor$로 정의할 수 있다. 리프 노드의 수는 또 다른 방식으로 측정할 수 있다. 만약 s를 리프 노드 MBR의 측면 길이로 정의할 경우, 두 번째 가정으로부터 리프 노드의 수는 차원 수 d를 이용해 $1/s^d$로 정의할 수 있다. 그러면, 이 두 가지의 정의를 이용해 다음과 같은 식을 구성할 수 있다.

$$\left\lfloor \frac{N}{0.7f} \right\rfloor = \frac{1}{s^d} \quad , \quad s = \left(\frac{1}{N/(0.7f)} \right)^{1/d} \tag{1}$$

q를 질의 MBR의 측면 길이라 정의할 경우, 위의 식을 이용해 질의 수행시 접근해야 되는 리프 노드의 수를 다음과 같이 측정할 수 있다.

$$\left\lfloor \frac{N}{0.7f} \right\rfloor (s + q^{1/d})^d \tag{2}$$

모든 높이의 노드들의 평균 엔트리 사용율을 0.7f 가정하고, 같은 높이의 MBR들이 같은 크기를 갖는다고

가정할 경우, R-트리의 전체 접근 노드의 수는 각 높이의 접근 노드의 수를 더하는 방식으로 다음과 같이 구할 수 있다.

$$\sum_{h=1}^{H}\left|\frac{N}{(0.7f)^h}\right|(s_h+q^{1/d})^d \quad (3)$$

위 식에서 H는 R-트리의 높이를 의미하며 $\lceil \log_{0.7f} N \rceil$으로 정의할 수 있으며, S_h는 다음 식과 같이 주어진다.

$$s_h=\left(\frac{1}{N/(0.7f)^h}\right)^{1/d},\ h=1,\ldots,H$$

MR-트리의 경우, 리프 노드의 수와 각 리프 노드의 측면 길이 s 역시 위의 식 (1)과 (2)로부터 정의할 수 있다. 만약 삽입 경로상의 어떤 중간 노드도 빈 엔트리를 가지지 않는다면, MR-트리는 리프 노드 분할을 상위 노드로 전달하지 않는다. 그러므로 MR-트리의 모든 중간 노드는 거의 다 차있게 된다. 만약 반-리프 노드가 존재하지 않는다고 가정한다면, 높이 2를 가지는 중간 노드의 수는 다음과 같이 정의할 수 있다.

$$\left|\frac{\lfloor N/(0.7f)\rfloor}{f}\right| \quad (4)$$

위의 식 (4)에서 내림(ceiling) 함수를 사용한 이유는 높이 3의 중간 노드들이 높이 2의 중간 노드뿐만 아니라 리프 노드 역시 자식 노드로 가질 수 있기 때문이다. I_2를 높이 2의 중간 노드의 한 차원에 대한 측면 길이로 두면, 높이 2를 가지는 중간 노드의 수를 $1/I_2^d$ 로 다시 정의 가능하다. 그러면 I_2를 다음과 같은 식으로 측정할 수 있다.

$$\left|\frac{\lfloor N/(0.7f)\rfloor}{f}\right|=\frac{1}{I_2^d},\ I_2=\left(1/\left|\frac{\lfloor N/(0.7f)\rfloor}{f}\right|\right)^{1/d} \quad (5)$$

H_M을 MR-트리의 높이로 두면, 앞에서 설명한 이유로 다음과 같이 정의된다.

$$H_M=\lceil \log_f \lfloor N/(0.7f)\rfloor \rceil+1 \quad (6)$$

MR-트리의 전체 접근 노드 수는 앞의 식 (1), (4), (5), (6)을 합함으로써 다음과 같이 정의된다.

$$\left|\frac{N}{0.7f}\right|(s+q^{1/d})^d+\sum_{h=2}^{H_M-1}\left|\frac{\lfloor N/(0.7f)\rfloor}{f^{h-1}}\right|(I_h+q^{1/d})$$

여기서 I_h은 다음과 같이 정의된다.

$$I_h=\left(1/\left|\frac{\lfloor N/(0.7f)\rfloor}{f^h}\right|\right)^{1/d},\ h=2,\ldots,H_M-1$$

CCMR-트리의 리프 노드는 MBR 대신 QRMBR을 사용한다. F를 CCMR-트리의 최대 수용가능한 엔트리 수라 두면, 전체 리프 노드의 수는 $\lfloor N/0.7F \rfloor$로 정의되며, 위의 식 (1)을 적용시켜 s를 다음과 같이 정의할 수 있다.

$$\left|\frac{N}{0.7F}\right|=\frac{1}{s^d},\ s=\left(\frac{1}{\lfloor N/(0.7F)\rfloor}\right)^{1/d} \quad (8)$$

CCMR-트리의 중간 노드의 수는 MR-트리와 같은 방식으로 측정가능하다. 그러므로 CCMR-트리에서 전체 접근 인덱스 노드 수는 다음과 같이 정의할 수 있다.

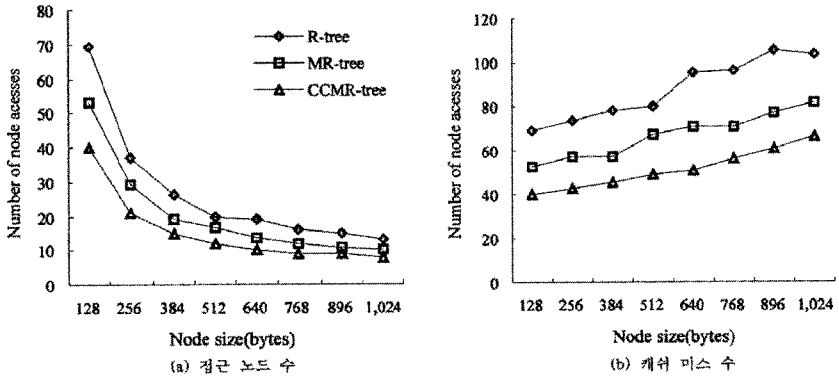
$$\left|\frac{N}{0.7F}\right|(s+q^{1/d})^d+\sum_{h=2}^{H_{CCM}-1}\left|\frac{\lfloor N/(0.7F)\rfloor}{F^{h-1}}\right|(I_h+q^{1/d})^d+1 \quad (9)$$

여기서 I_h와 H_{CCM}은 다음과 같이 각각 주어진다.

$$I_h=\left(1/\left|\frac{\lfloor N/(0.7F)\rfloor}{f^h}\right|\right)^{1/d},\ h=2,\ldots,H_{CCM}-1$$

$$H_{CCM}=\lceil \log_f \lfloor N/(0.7F)\rfloor \rceil+1$$

그림 9는 백만 개의 데이타(N)에 대해 질의 사각형의 크기를 전체 도메인의 0.01%로 두었을 때 식 (3), (7), 그리고 (9)를 비교 분석한 결과이다. 본 논문에서는 분석 시에 포인터 크기를 4 바이트로 가정하였고, MR-트리와 CCMR-트리의 높이 2의 노드들은 데이타 엔트리를 지니지 않는 중간 노드들로 구성된다고 가정하였다. 2차원 모델에서 MBR의 크기는 16 바이트이며, QR-MBR의 크기는 MBR 크기의 1/4로 하였다. 그림 9의 (a)에서 노드 크기가 커짐에 따라 접근해야 하는 노드의



(a) 접근 노드 수 (b) 캐쉬 미스 수

그림 9 R-트리, MR-트리, CCMR-트리에 대한 분석적 비교

수가 감소함을 볼 수 있으며, 모든 노드 크기에 대해 MR-트리와 CCMR-트리가 R-트리보다 더 적은 접근 노드의 수를 나타냄을 알 수 있다. 식 (4)에서 MR-트리의 접근할 리프 노드 개수를 나타내는 첫 번째 수식은 R-트리와 같은 수를 나타내지만, 중간 노드에 대한 접근 노드 개수를 나타내는 두 번째 수식에서 R-트리의 경우에 비해 적은 수를 나타내기 때문에 그림 9와 같은 결과를 나타낸다. CCMR-트리의 접근 노드 개수를 나타내는 식 (9)는 하나의 리프 노드에 저장 가능한 엔트리 개수가 R-트리와 MR-트리에 비해 약 4배 가깝기 때문에 리프 노드에 대한 접근 노드 개수를 나타내는 첫 번째 수식에서 훨씬 적은 수의 결과를 가져온다. 접근 중간 노드 개수를 나타내는 두 번째 수식은 MR-트리의 그것과 같은 방식으로 계산되지만, MR-트리에 비해 접근해야 될 리프 노드의 개수가 훨씬 적기 때문에 더 적은 수 중간 노드 접근이 필요하다는 계산이 나온다. 캐쉬 미스의 수를 구하기 위해서는 식 (3), (7)과 (9)를 하나의 노드를 접근할 때 발생되는 캐쉬 미스의 수와 곱해진다. 그림 9의 (b)에서 본 논문은 L2 캐쉬 블록 크기를 128 바이트를 사용하였기 때문에, 하나의 인덱스 노드에 접근할 때 발생하는 캐쉬 미스의 수는 노드크기/128로 계산할 수 있다. 이 그림에서 역시 MR-트리와 CCMR-트리가 R-트리보다 더 뛰어난 성능을 보임을 알 수 있다.

5. 실 험

본 논문의 실험에서는 리눅스 운영체계에서 GNU의 gcc 로 컴파일 되는 인덱스 구조들을 구현하였다. 하드웨어 환경은 1.6GHz의 CPU와 256K의 L2 캐쉬, 128 바이트의 캐쉬 블록 크기를 사용하는 펜티엄 4 컴퓨터를 사용하였다.

본 실험에서는 R-트리, PE R-트리, CR-트리, PE CR-트리, MR-트리, 그리고 CCMR-트리를 2차원 인덱스 구조로 구현하였다. PE R-트리(Pointer Eliminated R-tree) 는 중간 노드들에서 맨 처음의 하나를 제외한 모든 자식 노드들에 대한 포인터들을 제거한다[6]. PE CR-트리 역시 PE R-트리와 마찬가지로 첫 번째 하나를 제외한 모든 자식 노드 포인터들을 제거한다. 최적의 노드 크기를 찾기 위해, 노드의 크기는 128 바이트에서 1,024 바이트까지 변경시켰다. 한편, R-트리와 MR-트리를 위한 MBR의 크기는 16 바이트를 사용했으며, CR-트리와 PE CR-트리를 위해서는 QRMBR의 크기를 8 바이트로 사용했다. 이는 앞의 그림 5에서 설명한 것과 같이 너무 작은 QRMBR의 크기는 오히려 성능 저하를 나타냄을 보였기 때문이다. 그러나 CCMR-트리에 대해서는 CR-트리에서 발생 가능한 어떤 이상 현상도 발생하지 않기 때문에 4 바이트의 QRMBR을 사용한다. 본 실험에서는 이 모든 인덱스 구조들에 원래의 R-트리에서 사용했던 선형-비용(linear-cost) 분할 알고리즘을 적용 시켰다.

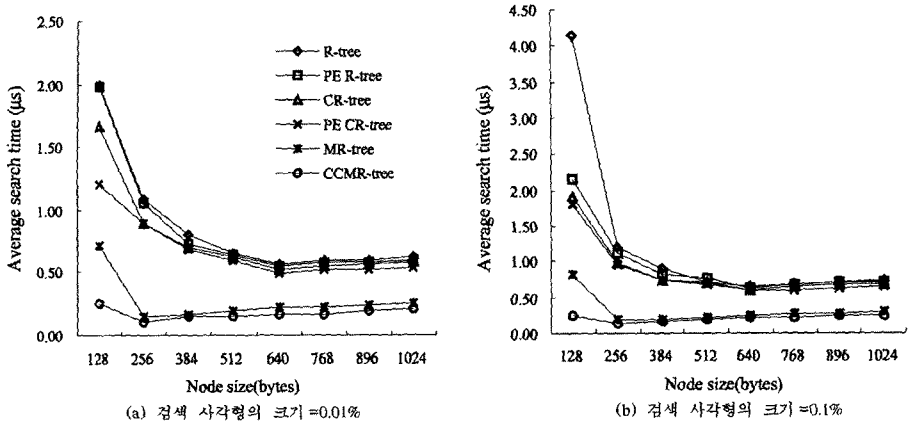
본 실험에서는 [6]과 같이 2개의 데이타 집단을 인위적으로 생성 시켰다. 이는 단위 정사각형 안에 위치한 백만 개의 작은 사각형으로 구성된다. 첫 번째 데이타 집단은 단위 정사각형 안에서 균등 분포를 나타내며, 두 번째는 중심점 (0.5, 0.5)와 표준편차 0.25를 갖는 가우스 분포를 나타낸다. 각각의 데이타 사각형의 평균 측면 길이는 0.001로 지정하였다. 한편, 모든 인덱스 구조의 성능이 입력 순서에 영향을 받기 때문에, 본 실험에서는 데이타 집단들안 데이타 객체 MBR의 입력 순서를 난수 함수를 사용해 설정했다.

5.1 검색 성능(Search performance)

본 실험에서는 여러 인덱스 구조들에 대한 검색 성능을 비교하기 위해, 2차원의 사각형 검색 영역에 대한 처리 시간을 비교했다. 이를 위해 본 실험에서는 가우스 분포와 균등 분포를 갖는 각각의 10,000개의 서로 다른 질의 사각형을 생성했다. 각각의 데이타 집합을 위한 질의 사각형은 단위 정사각형의 0.01%에서 1%로 다양화하였다.

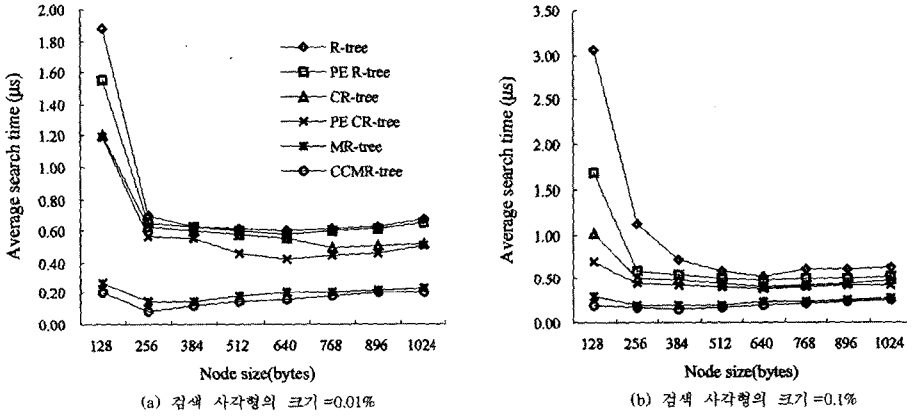
그림 10은 가우스 분포를 갖는 데이타 집단에 대해 여러 비교 인덱스들의 평균 검색 시간을 나타내며, 그림 11은 균등 분포를 갖는 데이타 집단에 대한 평균 검색 시간을 나타낸다. 이 그래프에 대해 다음과 같은 관찰 결과를 얻을 수 있다.

- 노드의 크기가 커지면서, 검색시간은 빠르게 최소 상태에 도달하며, 다시 천천히 증가함을 알 수 있다. 이러한 경향은 검색 사각형의 크기를 크게 했을 경우에도 같은 결과를 보여준다. 본 논문에서는 이러한 경향의 원인을 4장에서 설명한 노드의 크기가 커져도 접근해야 하는 노드의 수는 크게 줄어들지 않고, 캐쉬 미스의 수는 점점 증가한다는 분석으로부터 찾았다.
- MR-트리와 CCMR-트리는 R-트리, PE R-트리, CR-트리, PE CR-트리와 비교해 더 빠른 검색 성능을 보여준다. 특히, 본 논문에서 제시하는 두 기법은 데이타 집단이 가우스 분포를 가질 경우와 검색 사각형이 작을수록 더 좋은 검색 성능을 보여 준다.
- 본 실험에서 CCMR-트리는 분석 결과와 비교해 실제 그 만큼의 더 좋은 성능을 보여 주지 못한다. CCMR-트리와 MR-트리의 유일한 차이점은 CCMR-트리가 리프 노드에 더 많은 엔트리들을 저장하는 방법으로 접근해야 하는 노드의 수를 줄인다는 점을 감안한다면, 공간 인덱스에 있어서 키 비교가 일차원 인덱스만큼 간단하지 않고, 결국 공간 인덱스의 키 비교



(a) 검색 사각형의 크기 =0.01% (b) 검색 사각형의 크기 =0.1%

그림 10 검색 성능 비교(평균 <0.5,0.5>, 표준편차 0.25의 가우스 분포)



(a) 검색 사각형의 크기 =0.01% (b) 검색 사각형의 크기 =0.1%

그림 11 검색 성능 비교 (균등 분포)

시간을 아직까지는 무시하지 못한다는 결론에 도달한다. 그러나 CPU의 속도가 점점 더 빨라지는 상황을 고려한다면 가까운 미래에 CCMR-트리가 더 좋은 성능을 보일 것으로 예상한다.

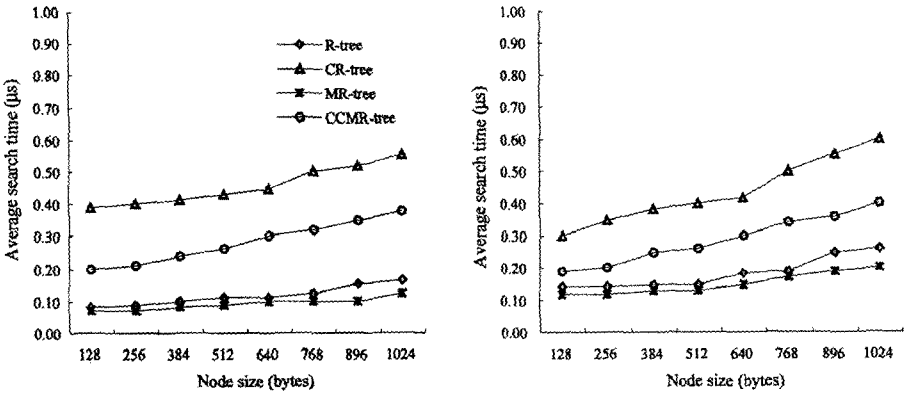
본 논문에서는 이밖에도 여러 불균형한 분포를 갖는 데이타 집단들에 대해 같은 실험을 수행했지만, 그림 10과 그림 11로부터 확연하게 구별되는 특별한 차이점을 찾지 못했다.

5.2 갱신 성능(Update Performance)

갱신 성능을 비교하기 위해, 본 실험에서는 백만 개의 균등분포 데이타 집단을 로딩한 후, 10,000개의 데이타 객체를 삽입하고, 전체 데이타 집단으로부터 임의로 선정한 10,000개의 데이타 객체를 삭제했다.

그림 12의 (a)와 (b)는 각각 삽입 및 삭제에 대한 평균 처리 시간을 나타낸다. 인덱스들의 삽입시간에서는 R-트리와 MR-트리에 비해 CR-트리와 CCMR-트리가 더 나쁜 성능을 보여준다. 이것은 노드 분할이 발생하거나, 참조 MBR의 변경에 있어서 CR-트리와 CCMR-트리은 QRMBR에 대한 재 계산 과정이 필요하기 때문이다. 한편, MR-트리와 CCMR-트리는 R-트리 및 CR-트리와의 비교에 있어서 각각 더 좋은 성능을 나타낸다. 이는 다음과 같이 설명된다.

MR-트리와 CCMR-트리의 경우 삽입 경로상의 중간 노드에 빈 엔트리가 있을 경우에만 리프 노드의 분할이 상위 노드로 전달된다. 그러므로 MR-트리와 CCMR-트리는 전체 노드 분할 수를 확연히 줄일 수 있다. 그러나 MR-트리와 CCMR-트리는 삽입 성능에 악영향을 미는 높이 균형화 과정이 발생할 수 있다. 본 실험에서는 두 개의 데이타 집단에 대해 삽입시의 높이 균형화 발생수를 측정하였고, 그 결과 이는 노드 분할 수에 비해 극히 미미했기 때문에 전체 삽입 성능에 별다른 영향을 미치지 않는다는 점을 발견했다. 이러한 점은 불균등한 데이타 집단에 대한 실험에서도 같은 결과를 얻었고, 최악의 경우, 높이 균등화 발생은 R-트리의 중간 노드 분할과



(a) 10,000 데이터에 대한 삽입시간 (b) 10,000 데이터에 대한 삭제시간

그림 12 갱신 성능 비교(균등 데이타 분포)

비슷한 수로 발생되었다.

인덱스 삭제의 경우 역시, CR-트리와 CCMR-트리가 R-트리와 MR-트리에 비해 더 나쁜 성능을 나타낸다. 이 이유는 삽입의 경우와 같은 방식으로 설명할 수 있다. MR-트리의 경우 R-트리와 비교해서, 서로 다른 트리압축(CondenseTree) 알고리즘의 사용으로 인해 재삽입되어야 하는 노드의 수를 확연히 줄인다. 그러므로 MR-트리와 CCMR-트리는 R-트리와 CR-트리에 비해 각각 더 좋은 성능을 나타낸다.

5.3 공간 효율성(Space efficiency)

인덱스들에 차지하는 공간을 비교하기 위해, 본 실험에서는 각 인덱스 구조를 위한 인덱스 노드의 수를 계산한다. 그림 13은 균등 분포의 백만 데이타 집단을 로딩시킨 후, 각 인덱스들의 공간 사용을 비교한다. CCMR-트리와 CR-트리를 비교하면, CCMR-트리가 MBR의 1/4 크기 QRMBR을 사용하기 때문에 2/4 크기 QRMBR을 사용하는 CR-트리에 비해 더 좋은 공간 효율성을 보인다. 그리고 R-트리와 비교해 MR-트리 역시 더 적은 공간을 사용한다. 이 것은 MR-트리의 거의 모든 중간 노드들이 가득 차 있기 때문에 R-트리에 비해 더 적은 수의 노드로 인덱스 트리를 구성할 수 있기 때문이다.

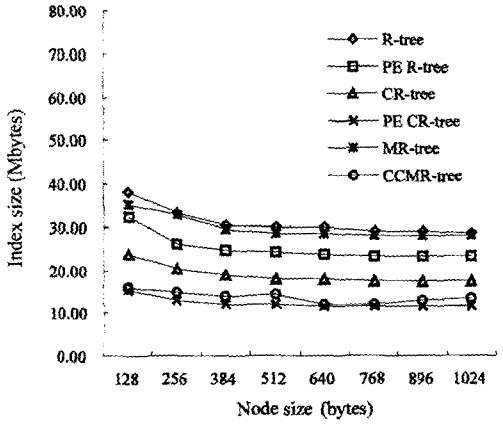


그림 13 공간 효율성(균등 분포)

6. 결 론

본 논문에서는 주기억 장치 접근을 위해 MR-트리라는 R-트리의 수정된 인덱스 구조를 제시했다. 본 논문은 R-트리를 주기억 인덱싱 기법으로 직접적으로 적용시켰을 때 발생할 수 있는 문제점들에 대한 인식을 출발한다. 그 문제점들은 R-트리의 중간 노드들은 약 70% 정도의 엔트리들만을 사용하고, R-트리 계열의 어떠한 노드 분할 알고리즘도 어떤 공간을 인접하지 않는 두개의 공간으로 항상 분리시켜 주지 못한다는 점에 있다. 그러므로 R-트리는 많은 처리 시간을 요구하는 반면 좋은 검색 성능을 언제나 보장하지 못한다는 문제점이 있다.

이를 해결하기 위해 제안된 MR-트리는 중간 노드들의 이용률을 100%에 가깝게 높임으로써 트리 높이를 줄여주고 이와 함께 중간 노드 MBR의 크기를 줄여 검색시 기존의 R-트리 계열에 비해 더 좋은 성능을 나타낸다. MR-트리에서는 삽입 경로상의 중간 노드들 중 하나 이상이 빈 엔트리를 지닐 때만, 노드 분할을 상위 노드로 전달한다. 만약 빈 엔트리가 전혀 없다면, 새롭게 생성된 리프 노드는 단지 분할된 노드의 자식 노드가 된다. 이는 중간 노드에 대한 엔트리 공간 이용률을 100% 에 가깝게 높임으로써 트리의 높이 증가를 최대한 억제해 캐쉬 미스 발생 횟수를 최대한 줄여준다는 장점이 있다. MR-트리의 높이는 데이타의 삽입 순서에

영향을 받는다. 그러므로 어떤 중간 노드의 자식 노드들의 불균형을 발견할 경우, 높이 균형화 알고리즘이 실행되어 자식 노드들의 높이를 균형화 시킨다. 이 밖에 본 논문에서는 리프 레벨에서만 QRMBR 기술을 적용시키는 CCMR-트리를 제안했다. CCMR-트리의 제안 동기는 2.3절에서 설명한 것과 같이 모든 레벨에서 QRMBR 기술을 적용시키는 CR-트리는 여러 이상 현상을 발생시킬 수 있다는 점이다.

본 논문의 실험에서는 MR-트리와 CCMR-트리가 R-트리와 CR-트리에 비해 검색 시간에서 2.4배 이상의 좋은 성능을 나타냈다. MR-트리와 CCMR-트리는 검색 사각형의 크기가 작을수록, 그리고 데이타 집단이 가우스 분포를 형성할 때 더 좋은 성능을 나타냈다. 수정 성능 측면에서 역시 MR-트리와 CCMR-트리가 더 좋은 성능을 나타냈다. MR-트리와 CC-MR-트리를 비교할 경우, MR-트리가 CCMR-트리에 비해 갱신 성능 측면에서 더 나은 성능을 나타냈다.

본 논문은 캐쉬 동작에 민감한 주기억 데이타베이스 인덱싱에서 MR-트리라는 새로운 R-트리 변형 모델을 제시하였다. 향후 연구과제로 실제 데이타를 이용해 MR-트리의 성능을 평가할 계획이며, MR-트리를 발전시켜 빠른 갱신 성능이 필요한 실시간 이동 객체에 대한 인덱싱 기법을 연구할 계획이다.

참 고 문 헌

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

[2] J. Rao, K. A. Ross, "Making B+-trees Cache Conscious in Main Memory," *Proceedings of ACM SIGMOD Conference*, 2000, pp. 475-486.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "DBMSs on a Modern Processor: Where Does Time Go?," *Proceedings of VLDB Conference*, 1999, pp. 267-277.

[4] P. Bones, S. Manegold, and M. Kersten, "Database Architecture Opimized for the New Bottleneck: Memory Access," *Proceedings of VLDB Conference*, 1999, pp. 54-65.

[5] J. Rao, K. A. Ross, "Cache Conscious Indexing for Decision-support in Main Memory," *Proceedings of VLDB Conference*, 1999, pp. 78-89.

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

[7] V. Gaede and O. Gnther, "Multidimensional Access Methods," *Computing Surveys*, 30(2), 1998, pp. 170-231.

[8] Kenneth A. Ross, Inga Sizmann and Peter J. Stuckey, "Cost-based Unbalanced R-trees," *Technical Report*, CSSE, The university of Melbourne, 2000.

[9] K. Ravi Kanth, D. Agrawal, and A. E. Abbadi, "Indexing non-uniform spatial data," *Proceedings of IDEA*, 1997, pp 289-298.

[10] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus, "Making Pointer-Based Data Structures Cache Conscious," *IEEE Computer*, TBD 2000 pp. 67-74.

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

[12] D. A. White, R. Jain., "Similarity Indexing with the SS-tree," *Proceedings of the Int. Conf. On Data Engineering*, 1996, pp. 516-523.

[13] N. Katayama, S. Satoh, "The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries," *Proceedings of ACM SIGMOD Conference*, 1997, pp. 369-380.

[14] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles," *Proceedings of ACM SIGMOD Conference*, 1990, pp. 322-331.

[15] T. Sellies, N. Roussopoulos, and C. Faloutsos, "The R+-tree: A Dynamic Index for Multi-dimensional Objects," *Proceedings of VLDB Conference*, 1987, pp. 507-518.

[16] C. Faloutsos, I. Kamel, "Beyond Uniformity and Independence: Anaylysis of R-trees Using the Concept of Fractal Dimension," *Proceedings of ACM PODS Symposium*, 1994, pp. 4-13.

[17] I. Kamel and C. Faloutsos, "Hilbert R-tree: An Improved R-tree Using Fractals," *Proceedings of VLDB Conference*, 1994, pp. 500-509.

[18] I. Kamel and C. Faloutsos, "On Packing R-trees," *Proceedings of ACM CIKM Conference*, 1993, pp. 490-499.

[19] B.-U. Pagel, H.-W. Six, H. Toben, and P. Widmayer, "Towards an analysis of range query performance in spatial data structures," *Proceedings of ACM PODS*, 1993, pp. 214-221.

[20] J. M. Hellerstein, "Indexing Research: Forest or Trees?," *Proceedings of ACM SIGMOD Conference*, 2000, pp574, Panel.

윤 석 우

1994년 홍익대학교 컴퓨터공학과 졸업(학사). 1996년 홍익대학교 전자계산학과 졸업(석사). 2002년~현재 홍익대학교 컴퓨터공학과 재학(박사). 관심분야는 실시간 데이타베이스, 지리정보시스템, 모바일 데이타베이스

김 경 창

1978년 홍익대학교 전자계산학과 졸업(학사). 1980년 한국과학기술원 전산학과 졸업(석사). 1990년 University of Texas at Austin 전산학과 졸업(박사) 1991년~현재 홍익대학교 정보컴퓨터공학부 교수. 관심분야는 객체지향 데이타베이스, 주기억 데이타베이스, OLAP 및 데이타 웨어하우징, Web 데이타베이스