Tmr-트리 : 주기억 데이터베이스에서 효율적인 공간 색인 기법

윤 석 우[†] · 김 경 창[††]

요 약

최근 들어 계속되는 램 가격 하락으로 인해 대용량의 램을 사용하는 주기억 데이터베이스 시스템의 구축이 실현 가능하게 되었다. 그러나 기존의 디스크 기반 공간 색인 기법은 디스크 접근 시간만을 주로 고려하기 때문에, 주기억 색인 기법으로 디스크 기반 색인 기법을 직접적으로 적용시키는 것은 부적절하다. 주기억 장치 색인 기법은 모든 색인 노드들이 주기억 장치에 상주하기 때문에 노드에 대한 접근 시간이 디스크 기반 기법에 비해 상당히 미미하고, 결국 효율적인 색인 기법을 위해서는 노드 접근시간 뿐만 아니라 노드내의 키 비교시간을 고려해야 한다. 이러한 주기억 장치 색인 기법의 특성을 고려하여, 본 논문에서는 Tmr-트리라는 새로운 색인 기법을 제시한다.

Tmr-트리는 T-트리의 장점과 R-트리의 장점을 결합한 이진 색인 구조로서, 색인 노드는 데이터 객체들을 위한 엔트리들, 왼쪽/오른쪽 자식 노드에 대한 포인터, 그리고 3개의 추가 필드들로 구성된다. 여기서 3개의 추가 필드들은 현재 노드에 저장된 키 값들의 범위를 포함하는 MBR과 왼쪽 서브트리에 저장된 키 값들의 범위를 포함하는 MBR, 오른쪽 서브트리에 저장된 키 값들의 범위를 포함하는 MBR에 해당한다.

본 논문의 실험에서 Tmr-트리는 R-트리와 달리 검색 시 항상 리프노드까지 방문할 필요가 없기 때문에 모든 데이터 분포에서 R-트리에 비해 더 나은 실험 결과를 보여주었다. 노드 크기 측면에서 노드안 엔트리 수를 증가시킨 초반에 상당한 검색성능 향상을 보여주었으며, 그 후로 약간씩 검색시간 증가를 나타냈다. 한편, 삽입시간 측면에서 Tmr-트리는 R-트리에 비해 약간의 더 많은 삽입시간이 필요했다.

키워드 : 주기억 데이터베이스, 공간색인, T-트리, R-트리

Tmr-Tree : An Efficient Spatial Index Technique in Main Memory Databases

Suk-Woo Yun[†] · Kyung-Chang Kim[††]

ABSTRACT

As random access memory chip gets cheaper, it becomes affordable to realize main memory-based database systems. The disk-based spatial indexing techniques, however, cannot direct apply to main memory databases, because the main purpose of disk-based techniques is to reduce the number of disk accesses. In main memory-based indexing techniques, the node access time is much faster than that in disk-based indexing techniques, because all index nodes reside in a main memory. Unlike disk-based index techniques, main memory-based spatial indexing techniques must reduce key comparing time as well as node access time.

In this paper, we propose an efficient spatial index structure for main memory-based databases, called Tmr-tree. Tmr-tree integrates the characteristics of R-tree and T-tree. Therefore, Nodes of Tmr-tree consist of several entries for data objects, main memory pointers to left and right child, and three additional fields. First is a MBR of a self node, which tightly encloses all data MBRs (Minimum Bounding Rectangles) in a current node, and second and third are MBRs of left and right sub-tree, respectively.

Because Tmr-tree needs not to visit all leaf nodes, in terms of search time, proposed Tmr-tree outperforms R-tree in our experiments. As node size is increased, search time is drastically decreased followed by a gradual increase. However, in terms of insertion time, the performance of Tmr-tree was slightly lower than R-tree.

Key Words : Main-memory Database, Spatial Index, T-tree, R-tree

1. 서 론

최근 들어 컴퓨터 하드웨어 기술 혁신으로 대용량의 메모리 칩이 계속 개발되고 있으며, 이와 함께 산업 자동화, 초고속 정보통신등의 여러 분야에서 지리정보시스템(GIS), CAD와 같은 실시간 응용시스템의 요구가 증대되고 있다. 이러한 실시간 응용시스템을 위한 처리방법으로 주기억 데이터베이스(Main Memory Database, MMDB)가 제안되었다.

이러한 주기억 데이터베이스에 대한 접근 방법으로 두 가지 형태의 구조가 제안되었는데, 그 하나는 기존의 디스크를 기반으로 한 데이터베이스 시스템에 의한 접근 방식으로

※ 이 논문은 2004학년도 홍익대학교 교내연구비에 의하여 지원되었음.
† 준 회 원 : 홍익대학교 컴퓨터 공학과 박사과정
†† 정 회 원 : 홍익대학교 컴퓨터 공학과 교수
논문접수 : 2004년 4월 24일, 심사완료 : 2005년 6월 8일

보다 큰 버퍼풀(buffer pool)을 가지게 함으로서 버퍼풀 내에서 각 트랜잭션이 필요로 하는 데이터의 대부분 혹은 전부를 적재할 수 있도록 하는 것이며[1], 또 다른 방법은 디스크 대신에 데이터베이스의 주된 저장 공간을 주기억 장치로 하는 방법이다. 따라서 기존의 디스크를 기반으로 한 데이터베이스에서는 디스크 저장 공간의 효율적인 사용과 디스크 접근 횟수를 최소화하는 것이 주된 문제점인 반면에, 주기억 데이터베이스 시스템에서는 처리속도와 주기억 장치의 효율적인 사용이 성능에 관련한 중요한 문제로 제기된다[8]. 이러한 주기억 데이터베이스 시스템을 위한 색인 기법으로는 AVL-트리, B-트리[9, 10], T-트리[6, 9], T*-트리[7] 등이 제안되었으나, 이러한 색인 기법은 정수, 실수, 문자 등과 같은 1차원 데이터에만 가능하고 2차원 이상의 데이터를 포함하는 공간 데이터에는 적용이 불가능하다.

공간 데이터는 1차원 데이터뿐만 아니라 2차원 이상의 데이터를 포함하는 데이터로서 이에 대한 색인 기법으로는 가장 대표적인 R-트리[3], 이에 대한 변형모델인 R*-트리[4], R+-트리[5] 등이 제안되었다. 그러나 가장 많이 쓰이는 이런 R-트리 계열의 색인 구조 역시 디스크를 기반으로 한 데이터베이스 시스템을 위해 제안되었기 때문에, 디스크 접근의 최소화에 그 초점이 맞춰져 주기억 데이터베이스에서 처리속도 최적화와 효율적인 메모리사용을 보장하지 못한다. 그러므로 주기억 데이터베이스에서 공간 데이터의 효율적인 색인 구조에 대한 연구가 절실히 요구된다.

본 논문에서는 주기억 장치 내에 전체 데이터베이스를 상주시키는 것으로 가정하고, 이 상황에 적합한 새로운 공간 데이터에 대한 색인 구조를 제안하고자 한다. 제안하는 색인 구조인 Tmr-트리는 R-트리와 T-트리의 장점들을 포함하는 통합 개념에 기반을 둔다.

본 논문의 구성은 2장에서 관련연구로 디스크 기반 공간 데이터에 대한 색인 구조인 R-트리와 R*-트리에 대해 알아보고, 주기억 데이터베이스의 1차원 데이터에 대한 색인 구조인 T-트리와 T*-트리에 대해 알아본다. 3장에서는 본 논문에서 제시하는 새로운 주기억 데이터베이스에서 공간 데이터의 효율적인 색인 구조인 Tmr-트리를 제안하고, 4장에서는 실험 결과로 Tmr-트리 색인 구조와 R-트리와의 검색 및 삽입 시간을 비교하며, 5장에서 결론 및 향후 연구를 제시한다.

2. 관련 연구

2.1 디스크 기반의 공간 데이터 색인 기법

디스크 기반 공간 데이터에 대한 색인 기법은 해쉬 기반과 트리 기반 기법으로 분류될 수 있다[2]. 해쉬 기반 기법은 그리드 파일을 기반으로 하며 대표적으로 그리드 파일, 그리드 파일의 변형인 EXCELL 기법, 두 레벨 그리드(two-level grid) 파일, 트윈 그리드(twin grid)등이 있다. 트리 기반 방법은 계층화된 검색 트리를 기반으로 하며 대표적으로 R-트리[3], 이에 대한 변형인 R*-트리[4], R+-트리[5], 그리고 Quad 트리, k-d 트리 등이 있다. 이 중 해쉬 기반 기법은 균등하지 않은 분포를 지니는 공간 데이터에 대해서는 좋지 않은 결과를 가져온다는 점과 오버플로(overflow) 발생 및 이에 따라 효율이 저하되는 문제점을 지닌다. 따라서 많은 디스크 기반 공간 데이터베이스에서는 트리 기반 색인 기법을 선호하고 있으며 이중 R-트리와 그 변형 모델이 가장 많이 사용되고 있다.

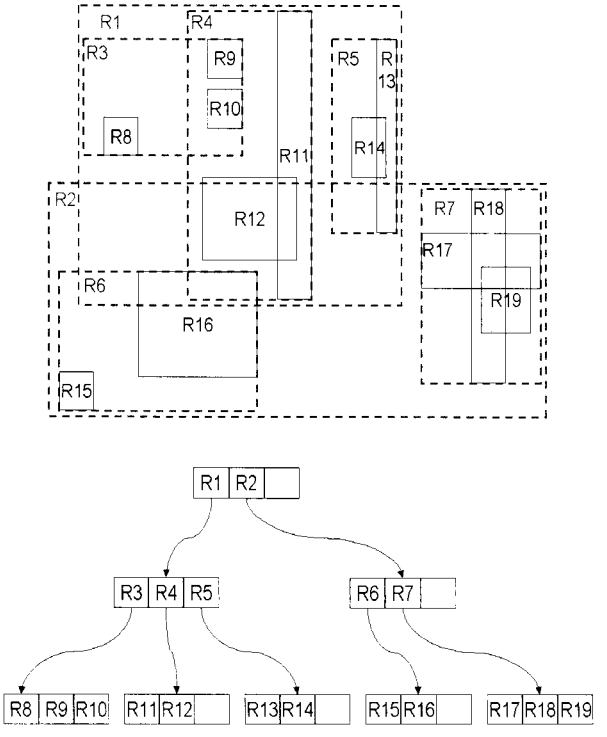
2.1.1 R-트리

R-트리는 트리 기반 방법 중 디스크에 저장되는 데이터 객체를 최소 직사각형(MBR : Minimum Bounding Rectangle)으로 표현하며 B 트리에 대한 k 차원 확장 모델에 해당된다[3]. 데이터 구조는 중간 노드와 리프 노드로 구성된 높이 균형 트리로서 각 리프 노드에는 실제 디스크 페이지에 대한 색인 값을 저장하고, 중간 노드는 하위 레벨에 있는 노드들에 대한 정보를 MBR 형태로 저장한다.

R-트리 중간 노드의 엔트리들은 (ref, rectangle) 형태를 지니며, 이때 ref는 자식 노드에 대한 포인터이고 rectangle은 자식 노드에 있는 모든 엔트리를 포함하는 MBR이다. R-트리 리프 노드의 엔트리 형태는 (tuple-id, rectangle)이고, 이때 tuple-id는 실제 공간 데이터에 대한 주소 값이고 rectangle은 이 공간 데이터를 포함하는 MBR을 의미한다.

2차원 데이터에 대한 R-트리 예제는 (그림 1)과 같다.

(그림 1)에서 실선으로 표현된 사각형은 리프 노드에 있는 엔트리들로서 실제 저장된 데이터에 대한 MBR을 의미하고, 점선으로 표현된 사각형은 루트와 중간노드의 엔트리로서 하위 노드를 포함하는 MBR에 해당된다.



(그림 1) 2차원 데이터에 대한 R-트리

2.1.2 R*-트리

R*-트리는 R-트리의 삽입 알고리즘의 단점들을 보안하기 위해 고안되었다[4]. R*-트리는 새로운 삽입 알고리즘을 제시하며, 이로 인해 R-트리에 비해 검색시 높은 성능향상을 나타낸다. R*-트리의 주된 관점은 어떻게 하면 형제 노드들 간의 교차정도를 최소화하는가에 있다. 이렇게 교차정도를 최소화하면 트리 검색시 불필요한 검색을 최소화함으로써 결국 디스크 접근 횟수를 최소화로 줄일 수 있다.

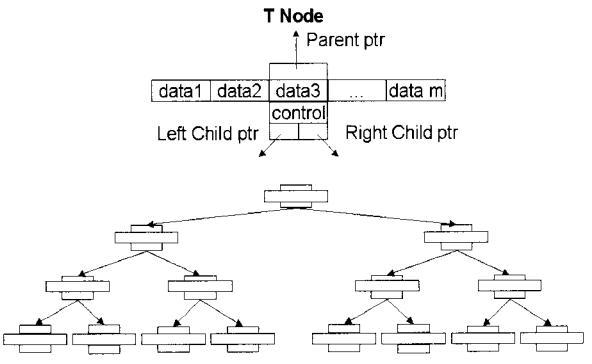
2.2 주기억 데이터베이스에서의 색인 기법

2.2.1 T-트리

T-트리는 AVL-트리와 B-트리의 특성을 결합해서 새롭게 변형되어 나온 트리 구조로서 빠른 처리속도와 주기억 장치 사용의 최적화라는 주기억 데이터베이스의 특성을 위해 Lehman이 제안했다[10]. T-트리는 한 노드안의 여러 개의 데이터들을 가지는 B-트리의 특성과 이진검색 및 높이 균형의 AVL-트리 특성을 결합하여 만든 색인 구조이다. (그림 2)는 T-트리의 한 노드의 구조와 T-트리의 전체 구성을 보여주고 있다.

T-트리 노드는 (그림 2)에서와 같이 여러 개의 엔트리(entry)들, 부모 노드에 대한 포인터, 왼쪽 자식 노드 및 오른쪽 자식 노드에 대한 포인터, 그리고 각종 조작 정보 값들로 구성된다. 각 엔트리의 구성은 키 값인 데이터 값과 그 키 값을 지니는 실제 아이템에 대한 주소를 표시하는 포인터의 쌍으로 구성된다. 한 노드내의 모든 엔트리들은 키 값의 크기에 따라 정렬되어 있고 가장 왼편의 엔트리가 가장 적은 키 값을 가지며, 가장 오른편의 엔트리가 가장 큰 키 값을 갖게 된다.

T-트리의 한 노드 A는 그 성격에 따라 내부 노드(internal node), 반-리프노드(half-leaf node)와 리프노드(leaf node)와 같이 3종류의 노드로 분류된다. 트리 구성은 노드 내에서 가장 작은 아이템보다 작은 값을 갖는 아이템들로 구성되는 왼쪽 서브트리와 가장 큰 값보다 큰 값을 가지는 아이템들로 구성되는 오른쪽 서브트리로 구성된다. 이때 왼쪽 서브트리에서 가장 큰 값을 GLB(Greatest Lower Bound)라 하고, 오른쪽 서브트리에서 가장 작은 값을 LUB(Least Upper Bound)라 한다.



(그림 2) T-트리

아이템의 삽입과 삭제는 모든 노드에서 가능하며 불균형 상태가 되면 로테이션 연산을 수행해 균형 상태로 만든다.

T-트리는 B-트리와 같은 디스크 기반 색인 방법들에 비해 빠른 검색 속도와 효율적인 메모리 사용을 보장해주는 색인 기법이지만, 정수, 실수, 문자와 같은 1차원 데이터에 대해서만 적용이 가능하다는 문제점을 지닌다.

2.2.2 T*-트리

T*-트리 색인 구조는 범위 질의와 삽입/삭제 시 불필요한 노드를 순회하는 T-트리의 문제를 해결할 수 있도록 기존 T-트리 색인 구조 및 색인 조작 알고리즘을 개선한 색인 구조이다. 이를 위해 T*-트리는 T-트리 구조에 후위 포인터를 추가하여 빠르게 다음 순서의 노드에 접근할 수 있도록 노드 내에서의 데이터 아이템의 저장순서를 개선하고, 중간노드에서 데이터 아이템의 삽입으로 인한 오버플로나 삭제로 인한 언더플로(underflow)시 이동 대상 아이템을 변경하여 추가된 후위 포인터를 최대한 활용 가능케 함으로서, 순차연산과 범위질의에서의 성능향상은 물론 삽입과 삭제 연산시 처리속도를 개선한다.

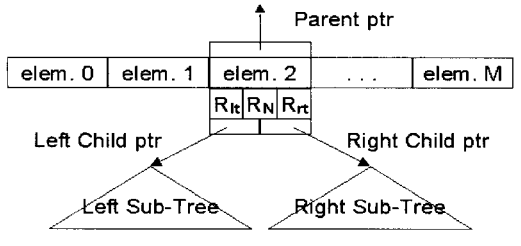
3. Tmr-트리

본 논문에서 제시하는 새로운 색인 구조는 T-트리를 기반으로 한다. 그러나 기존의 T-트리는 공간 데이터를 표현할 수 없기 때문에 T-트리 구조에 디스크 기반 공간 데이터베이스에서 가장 효율적이며 많이 사용되는 R-트리의 개념을 도입한다.

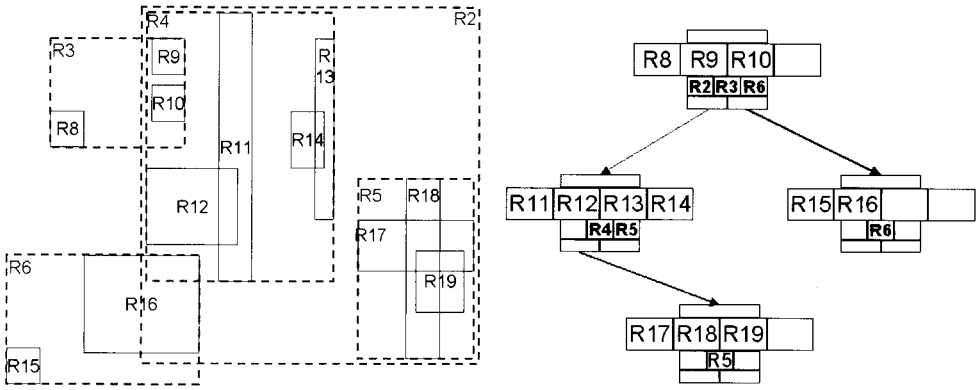
3.1 색인 구조

공간 데이터를 둘러싸는 MBR로 공간 데이터를 표현 할 때 본 논문에서 제시하는 새로운 색인 구조는 높이 균형 트리인 T-트리 구조를 기본으로 사용하며 각 트리 노드의 구조는 다음 (그림 3)과 같다.

(그림 3)과 같이 제시되는 색인 노드의 구조는 T-트리 노드와 같이 부모노드에 대한 포인터(Parent ptr), 왼쪽 자식 노드에 대한 포인터(Left child ptr), 오른쪽 자식 노드에 대한 포인터(Right child ptr), 여러 개의 엔트리들, 그리고 새롭게 정의되는 왼쪽 서브트리의 MBR(Rlt), 자신 노드 N의 엔트리들에 대한 MBR(RN), 오른쪽 서브트리의 MBR(Rrt)으로 구성된다.



(그림 3) 공간 색인 구조 (노드 N)



(그림 4) 새 트리 구조

왼쪽 서브트리 MBR이란 왼쪽 서브 트리내의 모든 엔트리들을 포함하는 최소 직사각형을 의미하고, 오른쪽 서브트리 MBR 역시 오른쪽 서브트리내의 모든 엔트리들에 대한 최소 직사각형을 의미한다. 자신 노드에 대한 MBR이란 자신 노드의 모든 엔트리들을 포함하는 최소 직사각형을 의미한다.

노드의 각 엔트리들은 R-트리의 리프 노드 엔트리와 마찬가지로 (tuple-id, rectangle)의 형태로 이루어진다. 이때 tuple-id는 주기억 데이터베이스이기 때문에 주기억 장치 내에 위치한 공간 데이터에 대한 포인터로서 주기억 장치내의 데이터에 대한 완전 주소 값 또는 데이터를 포함하는 페이지 번호와 페이지 내에서의 오프셋으로 표현될 수 있다. rectangle은 해당 데이터를 포함하는 MBR에 해당된다. T-트리 노드안의 엔트리들은 최소 값에서 최대 값 순서로 정렬되어 저장되는 반면, 새롭게 제시하는 트리구조에서는 노드안 엔트리들 간의 정렬이 이루어지지 않는다. 이는 공간 데이터를 문자, 정수, 실수와 같이 1차원적으로 정렬시킬 수 없음에 기인한다.

그 이외에 본 논문에서 제시하는 트리 노드의 구조는 T-트리 노드구조와 일치한다. 노드의 종류는 자식 포인터의 존재유무에 따라 중간노드, 반-리프노드, 리프노드로 구별되며, 트리의 구조는 T-트리와 같은 높이 균형의 2진 트리 형태를 지닌다. 그러므로 높이의 불균형이 발생되면 로테이션을 수행해서 트리의 높이가 균형이 되도록 유지한다.

3.2 기본 연산

본 절에서는 위에서 제시한 색인 구조에 대한 기본적인 연산 기법인 검색, 삽입, 삭제, 재균형 계산 기법들을 제시한다.

3.2.1 검색 기법

앞에서 설명한 바와 같은 색인 구조를 사용한 포인트, 지역에 대한 검색을 위해서는 자신노드의 엔트리들에 대한 MBR(RN), 왼쪽 서브트리 MBR(Rlt), 오른쪽 서브트리 MBR (Rrt)을 이용한다. 어떤 지역에 대한 검색 요청이 들어올 경우 우선, 자신 노드 MBR과 교차되는 지점이 있는지 검사 후, 존재 시엔 자신 노드안 엔트리들에 대한 구체적인 검색이 시작된다. 그런 다음, 왼쪽 서브트리 MBR과의 교차점이 있는지 검색 후, 존재하면 왼쪽 자식 노드를 방문하고, 같은 방식으로 오른쪽 서브트리 MBR과의 검색작업을 수행한다. (그림 1)에서 예로 들은 2차원 데이터를 본 논문에서 제시하는 트리 구조로 전환할 경우 다음 (그림 4)와 같다.

(그림 4)에서 만약 R17 지역 안의 한 지점을 검색할 경우, 우선 루트 노드에서 자신(R3)과의 교차 여부를 확인 후, 왼쪽 서브트리(R2)와의 교차 여부, 오른쪽 서브트리(R6)와의 교차 여부를 검색한다. 그러면, 해당 지점은 왼쪽 서브트리와 교차점이 있는 것을 알 수 있고, 왼쪽 자식 노드(R4)를 루트로 하여 다시 검색 루틴을 수행해 오른쪽 서브트리(R5)를 방문한다. 이렇게 해서 방문한 오른쪽 자식 노드에서 원하는 엔트리 R17을 검색해 낼 수 있다.

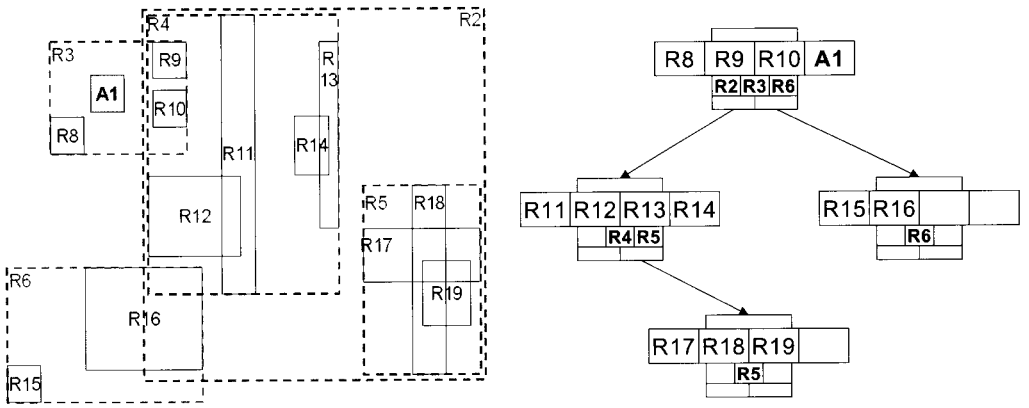
검색 알고리즘은 다음과 같다.

search(W, N, R) // W : search window, N : root node of Tmr-tree, R : result set //

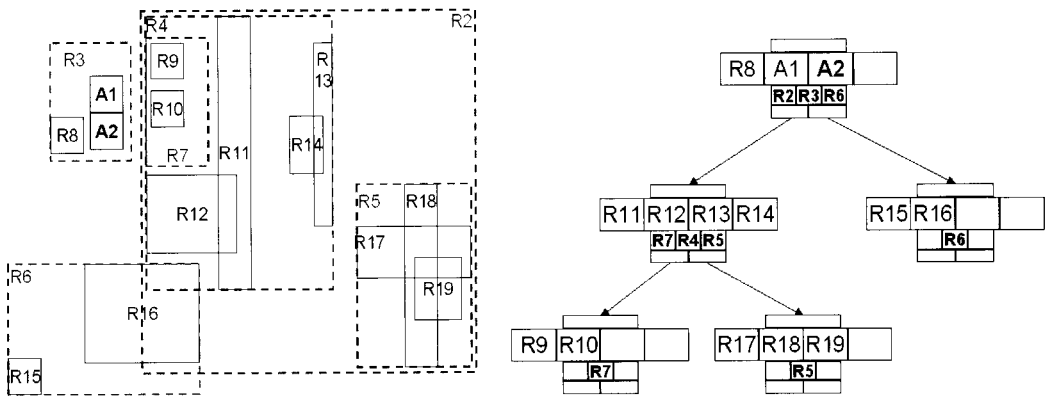
1. if overlap(W, RN of N),
   for (each entry e of node N) if overlap(W, e), R := RUe;
2. if overlap(W, Rlt of N),
   search(W, N.left_child_ptr); // N의 왼쪽 자식 노드를 루트로 하여 검색 //
3. if overlap(W, Rrt of N),
   search(W, N.right_child_ptr); // N의 오른쪽 자식 노드를 루트로 하여 검색 //

3.2.2 삽입 기법

삽입 기법은 R-트리의 삽입 기법과 비슷하다. 차이점은 R-트리에서는 삽입시 후보 노드는 무조건 리프 노드가 되어야 하지만 본 논문에서 제시된 기법에서는 리프가 아닌 노드에서도 삽입이 가능하다. 만약 삽입으로 인해 노드의 오버플로가 발생할 경우, R-트리 혹은 R*-트리의 분할 알고리즘을 이용해 엔트리들을 기존의 노드와 새롭게 생성한 노드에 분산 배치한다. 새롭게 생성된 노드는 분할이 발생된 노드를 중심으로 MBR의 증가크기가 최소가 되는 왼쪽



(그림 5) A1 삽입시 구조



(그림 6) A2 삽입시 구조

서브트리 또는 오른쪽 서브트리로 찾아 들어간 후, 리프 혹은 반-리프 노드의 자식 노드가 되도록 배치한다. 이는 검색시 MBR의 크기를 줄여주고, 트리의 불균형 발생시 T-트리의 로테이션 알고리즘 이용을 원활히 하기 위한 방안으로 고안되었다.

(그림 4)에서 제시된 색인 구조에 새롭게 A1, A2를 삽입할 경우 색인 구조는 다음과 같은 변환 과정을 거친다.

A1이 삽입될 경우 우선 루트노드를 시작으로 자신(R3)에 삽입할 경우와 왼쪽 서브트리(R2)에 삽입할 경우, 오른쪽 서브트리(R6)에 삽입할 경우를 비교해 MBR이 제일 적게 증가하는 위치를 선택한다. 그러므로 새로 삽입된 A1은 루트에 삽입되고 (그림 5)와 같은 색인 구조를 생성한다.

(그림 5)에서 다시 A2를 삽입할 경우에는 위와 같은 방법으로 최적의 색인 노드(R3)를 선택한다. 그러나 A2를 R3에 삽입하기 위한 빈 엔트리가 존재하지 않기 때문에 노드 분할이 발생한다. 그러므로 R-트리나 R*-트리의 분할 알고리즘을 이용해 새로운 노드(R7)를 생성한 다음, 새로 생성된 노드(R7)의 연결 위치를 결정한다. 그 과정은 분할이 발생된 노드(R3)를 루트로 새 노드(R7)를 포함할 경우 MBR이 적게 증가되는 왼쪽(R2) 오른쪽(R6) 서브트리를 검색해 들어간다. 선택한 자식 노드(R4)가 리프 또는 반-리프 일 경우, 트리의 높이를 고려해 높이균형이 이루어지도록 새 노드(R7)를 선택노드(R4)의 자식노드가 되도록 연결한다(그림 6).

삽입 알고리즘(insert)은 다음과 같다.

insert(E, N) // E : object MBR, N : root node of Tmr-tree //

1. calculate d1= area(RN of N + E) - area(RN of N),
   d2= area(Rlt of N + E) - area(Rlt of N),
   d3= area(Rrt of N + E) - area(Rrt of N),
2. if (d1≤d2 and d1≤d3), {
3. if (N has empty room), insert E into N, and adjust RN of N;
4. else { N′ := split(N, E); // split N into N and N′ //
5. **insertNode**(N′, N);
6. }
7. } else if (d2≤d1 and d2≤d3), { // insert E into left child //
8. adjust Rlt of N, and **insert**(E, N_left_child_ptr);
9. if (N.left_child_ptr is unbalanced), N.left_child_ptr = **rotation**(N.left_child_ptr);
10. } else { // insert E into right child //
11. adjust Rrt of N, and **insert**(E, N_right_child_ptr);
12. if (N.right_child_ptr is unbalanced), N.right_child_ptr = **rotation**(N.right_child_ptr);
13. }

```
insertNode(D, N) // D : new node, N : node of Tmr-tree //
 1. if (N.left_child_ptr is NULL) N.left_child_ptr := D;
 2. else if (N.right_child_ptr is NULL) N.right_child_ptr := D;
 3. else {
 4.       calculate d1= area(Rlt of N + D) - area(Rlt of N),
                  d2 = area(Rrt of N + D) - area(Rrt of N).
 5.       if (d1 < d2), {
 6.             insertNode(D, N.left_child_ptr);
 7.             if (N.left_child_ptr is unbalanced), N.left_
                child_ptr = rotation(N.left_child_ptr);
 8.       } else {
 9.             insertNode(D, N.right_child_ptr);
10.             if (N.right_child_ptr is unbalanced), N.
                right_child_ptr = rotation(N.right_child_ptr);
11.       }
12. }
```

3.2.3 재균형 계산

AVL-트리, -트리와 마찬가지로 새로 제안하는 Tmr-트리 역시 연산에 의하여 불균형 트리가 되면 LL, LR, RR, RL 로테이션 연산을 수행함으로써 균형을 유지하도록 한다. 여기서 기존 AVL 트리와의 차이점은 로테이션 수행시 노드의 왼쪽 서브트리 MBR(Rlt)과 오른쪽 서브트리 MBR(Rrt)을 재계산하는 과정이 새롭게 추가된다. 재균형(rotation) 알고리즘은 다음과 같다.

```
rotation(A) // A : node of Tmr-tree //
 1. if (bf(A) = 2), {
 2.   B := A.left_child_ptr;
 3.   C := B.right_child_ptr;
 4.   if (bf(B) = 1), { // LL rotation //
 5.    Do LL rotation of AVL-tree;
 6.    copy Rrt of B into Rlt of A;
 7.    adjust Rrt of B := area(Rrt of B + RN of A + Rlt
       of A + Rrt of A);
 8.   } else { // LR rotation //
 9.    Do LR rotation of AVL-tree;
10.    copy Rlt of C into Rrt of B;
11.    copy Rrt of C into Rlt of A;
12.    adjust Rlt of C := area(Rlt of C + RN of B + Rlt
       of B + Rrt of B);
13.    adjust Rrt of C := area(Rrt of C + RN of A + Rlt
       of A + Rrt of A);
14.   }
15. } else if (bf(A) = -2) {
16.   B := A.right_child_ptr;
17.   C := B.left_child_ptr;
16.   call RR or RL rotation of AVL-tree, if needed;
17.   adjust Rrt of A and Rlt of B, when RR or RL;
18.   adjust Rlt and Rrt of C, when RL;
19 }
```

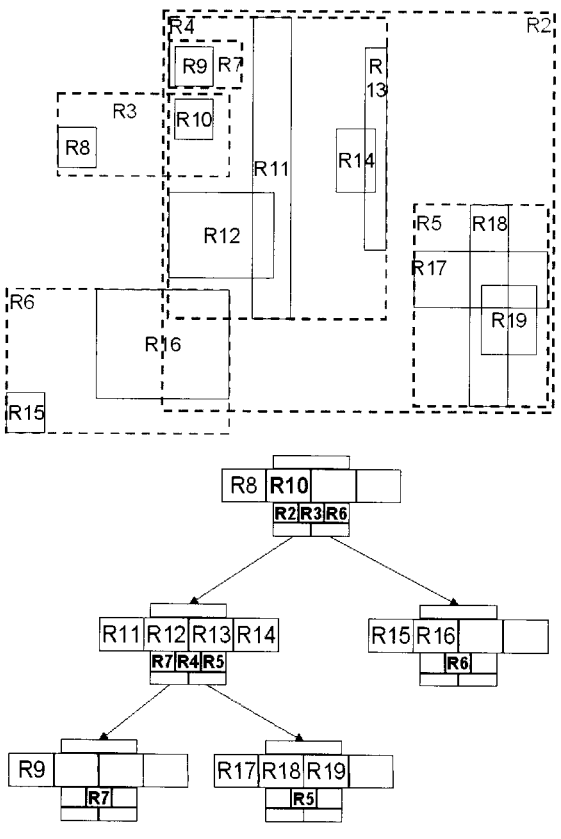
```
bf(N) // N : node of Tmr-tree //
1. if N is NULL, return 0;
2. return height(N.left_child_ptr) - height(N.right_child_ptr);
```

3.2.4 삭제 기법

색인 노드 안의 엔트리를 삭제를 할 경우에는 해당 엔트리를 포함하는 노드를 검색해 삭제한다. 본 논문에서 제시하는 트리구조는 리프뿐만 아니라 모든 위치의 노드에 데이터를 저장하기 때문에 삭제는 모든 노드에서 이루어질 수 있다. 만약, 중간노드에서 삭제로 인해 언더플로가 발생할 경우에는 최소 직사각형의 면적 증가크기를 비교하여 엔트리들 중 가장 증가크기가 최소인 노드 엔트리를 검색하여 삭제 발생 노드로 이동시킨다.

노드 안의 최소 엔트리 수를 2로 정의할 때, (그림 6)에서 A1, A2를 삭제할 경우의 트리 구조는 다음 (그림 7)과 같다.

A1이 삭제될 경우에는 단순히 노드에서 해당 엔트리를 삭제한 후 알고리즘이 종료되지만, A2를 삭제할 경우에는 루트 노드에서 언더플로가 발생한다. 이런 경우에는 루트안의 엔트리 R8과 최소의 직사각형을 유지하는 노드 R7를 검색한 후, R7에서 최소의 직사각형을 형성하는 엔트리 R10을 선택해 루트로 이동시킨다.



(그림 7) A1, A2 삭제시 구조

삭제(delete) 알고리즘은 다음과 같다.

```
delete(E, N) // E: MBR for data object o, N : root node of Tmr-tree //
 1. N := search node N, which include data object o; // modified search algorithm //
 2. if (N is underflow), {
 3.   if (N is a leaf node), return;
 4.   else if (N is a half-leaf node), {
 5.     if (N can be merged with N's left or right child node), merge N with one child node;
 6.   } else { // N is a internal node //
 7.     calculate d1 = area(Rlt + RN) - area(RN); d2 = area(Rrt + RN) - area(RN);
 8.    if (d1 < d2), {
 9.      e := chooseEntry(RN, N.left_child_ptr);
10.      if (L's left sub-tree is unbalanced), L.left_child_ptr := rotation(L.left_child_ptr);
11.    } else {
12.      e := chooseEntry(RN, N.right_child_ptr);
13.      if (L's right sub-tree is unbalanced), L.right_child_ptr := rotation(L.right_child_ptr);
14.    }
15.    insert e into node N;
16. }
```

```
chooseEntry(R, N) // R: node MBR of underflow node, N : node of Tmr-tree //
 1. calculate d1 = area(RN of N + R) - area(RN of N);
              d2 = area(Rlt of N + R) - area(Rlt of N);
              d3 = area(Rrt of N + R) - area(Rrt of N);
 2. if (d1 ≤ d2 and d1≤d3), {
 3.    search entry e in node N, which needs least enlargement including R;
 4.   ret_entry := remove e from N;
 5.   if (N is underflow), delete(e's MBR, N);
 6.   return ret_entry;
 7. } else if (d2 ≤ d1 and d2≤d3),
 8.      ret_entry := chooseEntry(R, N.left_child_ptr);
 9. else ret_entry := chooseEntry(R, N.right_child_ptr);
10. return ret_entry;
```

4. 실 험

기존의 주기억 장치 데이터베이스 환경 하에서 효율적인 색인 기법은 많이 제시되었으나, 이는 모두 1차원 데이터를 위한 색인 구조이므로, 본 논문에서 제시하는 색인 구조와는 비교가 불가능하다. 그러므로 본 논문에서는 기존에 공간 데이터에 가장 많이 사용되는 R-트리 계열 색인 기법과 제안하는 Tmr-트리 색인 기법의 성능을 비교한다. 객관적인 성능 비교를 위해 우선 R-트리 계열의 색인 구조를 디스크가 아닌 주기억 데이터베이스 환경으로 변경시켜서 성능을 비교한다.

4.1 구현 환경

본 논문에서 제안한 색인의 구현 환경으로 Linux 운영체제에서 GNU gcc 컴파일러를 통해 C언어로 작성하였다. 하드웨어로는 Intel Pentium(R) IV 2.0Ghz CPU와 256 DDR RAM을 사용하였다.

4.2 실험 결과

본 절에서는 시스템의 성능 평가를 위해서 새롭게 제시하는 Tmr-트리 색인 구조와 기존의 색인 구조(R-트리)를 구현하여 검색 성능과 삽입 시간을 비교한다. 객관적인 성능 분석을 위하여 이 두개의 색인 구조는 주기억 데이터베이스 환경에서 같은 조건을 가지고 실험을 하였다. 즉, R-트리를 주기억 데이터베이스에 맞게 구현하여 새로운 색인 구조와 비교를 한 것이다. MBR의 크기는 16바이트(2차원 색인)를 사용하여 실험하였다.

본 실험에서는 2개의 데이터 집단 백만 개를 임의적으로 생성하여 삽입하였는데, 하나의 데이터 집단은 단위 정사각형 안에서의 균등 분포를 나타내며, 또 다른 하나는 중심점(0.5, 0.5)과 표준편차 0.25를 갖는 가우스 분포를 나타낸다.

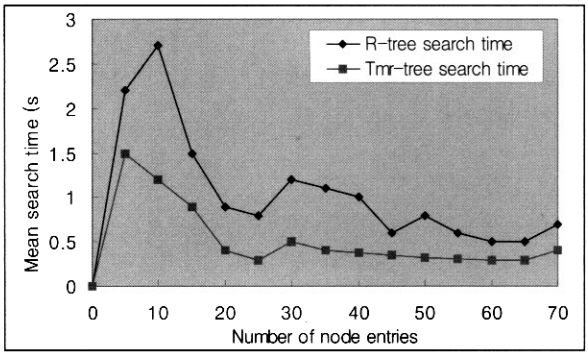
4.2.1 검색 성능

R-트리와 새로운 색인 구조인 Tmr-트리의 검색 성능을 비교하기 위해서 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 데이터 집합을 위한 질의 사각형은 단위 정사각형의 0.01%에서 0.1%로 다양화 하였다.

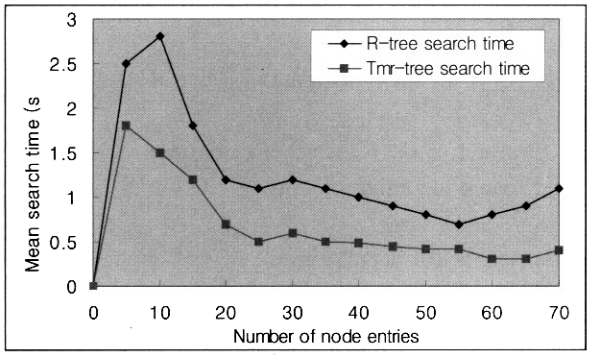
(그림 8)은 균등 분포를 갖는 데이터 집단에 대한 R-트리와 Tmr-트리의 평균 검색 시간을 나타내며, (그림 9)는 가우스 분포를 갖는 데이터 집단에 대한 R-트리와 Tmr-트리의 평균 검색 시간을 나타낸다. 평균 검색 시간은 노드의 접근 시간과 노드내에서의 엔트리 비교 시간을 합한 시간을 의미한다. 이 그래프에 대해서 다음과 같은 관찰 결과를 얻을 수 있다.

- 노드의 크기가 커지면, 검색시간은 빠르게 최소 상태에 도달하며, 다시 천천히 증가함을 알 수 있다. 이러한 경향은 검색 사각형의 크기를 크게 했을 경우에도 같은 결과를 보여준다.
- Tmr-트리와 R-트리를 비교해 볼 때 Tmr-트리는 더욱 빠른 검색 성능을 보여준다. 특히 데이터 집단이 가우스 분포를 가질 경우와 검색 사각형이 작을수록 더 좋은 검색 성능을 보여준다.

이러한 검색 성능 결과는 Tmr-트리가 이진 검색 트리의

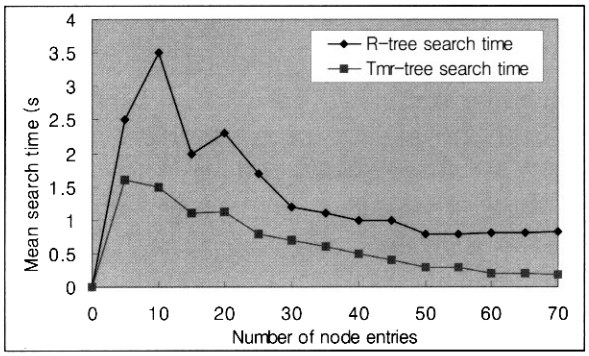


(a) 검색 사각형의 크기 = 0.01%

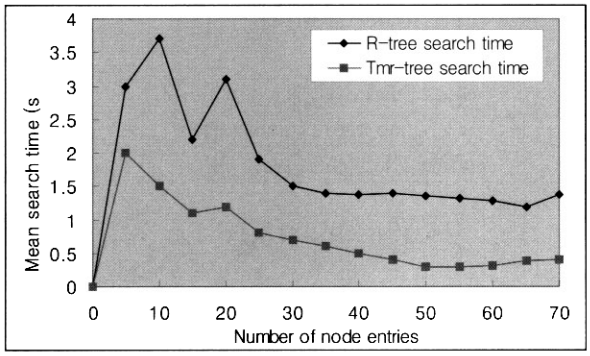


(b) 검색 사각형의 크기 = 0.1%

(그림 8) 검색 성능 비교(균등 분포)

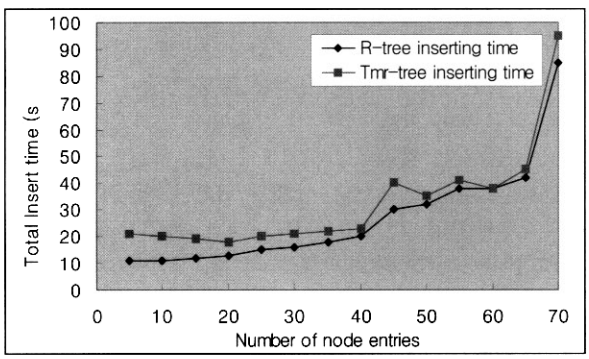


(a) 검색 사각형의 크기 = 0.01%

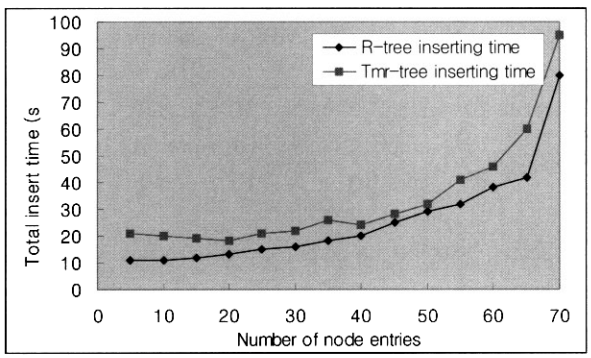


(b) 검색 사각형의 크기 = 0.1%

(그림 9) 검색 성능 비교(평균<0.5, 0.5>,표준편차 0.25의 가우스 분포)



(a) 균등 분포 데이터 삽입



(b) 가우스 분포 데이터 삽입

(그림 10) 삽입 성능 비교

형태로, 이진 검색에 의해 내부 노드에서 검색을 완료하기 때문에 빠른 검색이 가능하다.

본 논문에서는 이 외에도 여러 불균형한 분포를 갖는 데이터 집단을 생성하여 같은 실험을 수행하였지만, (그림 8)과 (그림 9)로부터 확연하게 구별되는 특별한 차이점을 찾지 못했다.

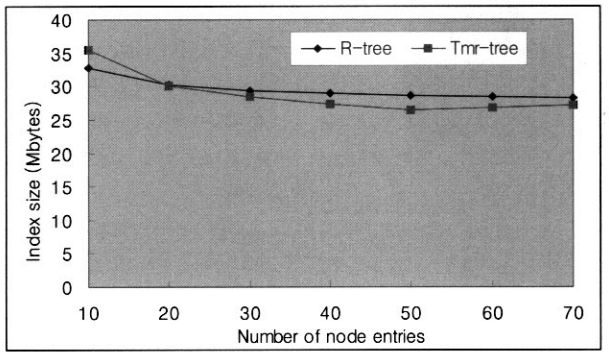
4.2.2 삽입 성능

삽입 성능을 비교하기 위해, 본 실험에서는 백만 개의 균등분포 데이터 집단과 백만 개의 가우스 분포 데이터 집단을 두개의 색인에 삽입하여 삽입 시간을 측정하였다.

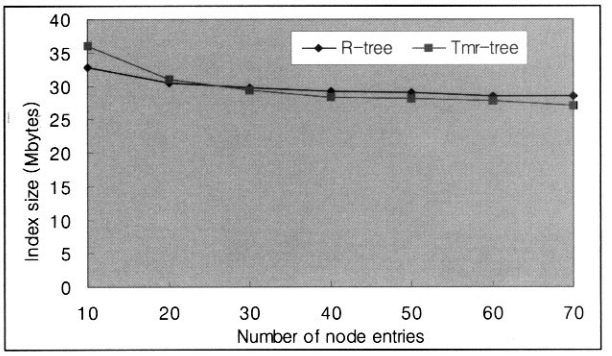
(그림 10)은 각 트리의 삽입에 대한 평균 처리 시간을 나타낸다. 삽입시간에서는 R-트리에 비해서 Tmr-트리가 약간 나쁜 성능을 보여주는데, 이것은 Tmr-트리 경우 노드 분할 발생시 로테이션 수행과 같은 처리 지연 효과를 그 원인으로 설명할 수 있다.

4.2.2 공간 성능

두 색인 구조가 주기억 장치에서 사용하는 저장 공간 크기 비교는 각각 백만 개의 균등 및 가우스 분포 데이터 집합에서의 저장 공간 크기를 비교하였으며, 실험 결과는 다음 (그림 11)과 같다.



(a) 균등 분포 경우



(b) 가우스 분포 경우

(그림 11) 저장 공간 성능 비교

실험 결과 위의 (그림 11)과 같이 Tmr-트리가 R-트리에 비해 약간 더 좋은 공간 성능을 보였는데, 이는 R-트리 경우 실제 데이터 엔트리들은 모두 리프 노드에 저장되고, 상위 레벨의 색인 노드들은 이들을 포함하는 엔트리들로 노드를 구성하는 반면, Tmr-트리는 모든 레벨 색인 노드들의 엔트리들이 실제 데이터에 대한 키 값들이라는 이유로 설명된다.

5. 결론 및 향후 연구

본 논문에서는 주기억 데이터베이스에서 공간 데이터를 위한 효율적인 색인 구조인 Tmr-트리를 제안하였다. 지금까지 제안된 공간 데이터에 대한 색인 구조는 디스크 기반의 데이터베이스를 위한 구조였기 때문에 디스크 접근 횟수의 최소화와 디스크 저장 공간의 효율적 사용에 그 초점이 맞춰져 있었다. 그러나 주기억 데이터베이스에서는 디스크 기반 데이터베이스와는 달리 빠른 처리 속도와 메모리 공간 사용의 최적화라는 목적을 두고 있다.

기존에 가장 널리 알려진 R-트리 계열의 색인 구조는 모든 실제 데이터에 대한 색인이 리프 노드에 존재하고 있기 때문에 주기억 데이터베이스에 적용시 중간 노드들의 크기만큼 주기억 장치 낭비를 초래한다. 그리고 검색시 항상 리프 노드까지 방문해야 하기 때문에 트리의 높이가 커질 경우 빠른 처리 속도를 보장하지 못한다. 트리 높이의 축소를 위해 한 노드안에 들어가는 엔트리 수(M)를 크게 증가시킬 경우에는 노드안의 검색시간 증가로 인해 오히려 더 많은 처리시간 증가를 유발한다.

기존에 주기억 데이터베이스를 위해 많은 T-트리 계열의 색인 구조가 제안되었지만, 이 역시 1차원 데이터를 위한 색인 구조로 공간 데이터에는 부적합하다.

본 논문에서는 이러한 문제점에 기인해서, T-트리에 R-트리 개념을 도입한 새로운 색인 구조인 Tmr-트리를 제안하였다. Tmr-트리 구조는 높이 균형의 이진 검색 트리 형태로, 하나의 노드에는 여러 개의 데이터 엔트리들을 저장하며, 추가적으로 각 자신 노드 MBR(RN)과 왼쪽 서브트리 MBR(Rlt), 오른쪽 서브트리 MBR(Rrt)을 저장한다.

본 논문에서 제시하는 Tmr-트리 색인 구조는 R-트리와 달리 이진 검색에 의해 내부 노드에서 완료될 수 있기 때문에 빠른 검색이 가능하다. 또한 본 논문에서는 성능 평가를 통해 이를 입증하였다.

2차원 이상의 공간 데이터의 경우, 삽입 및 삭제와 같은 연산이 자주 발생하지 않으므로 새로 제안하는 색인 구조는 주기억 장치 데이터베이스에서 공간 데이터를 다루는 데에 효율성을 가지고 있다고 볼 수 있다.

향후 연구과제로는 Tmr-트리 색인 구조를 질의 연산인 조인과 같은 질의처리에 적용하는 방법과 R-트리에 비해서 긴 삽입시간에 대한 보완 및 연구가 필요하다.

참 고 문 헌

[1] David J. Randy H. Katz, Frank Olken, Leonard D. Shapiro, Micheal R. Stonebraker, David Wood "Implementation Techniques for Main Memory Dtabase Systems", In Proc. of ACM SIGMOD, Boston, 1984.

[2] Voler Gaede, Oliver Gunther, "Multidimensional access methods", ACM Computing Surveys, Vol.30, No.2, June, 1998.

[3] Guttman, A. "R-tree: A dynamic index structure for spatial searching", In Proceedings of the ACM SIGMOD International conference on Management of data, 1984.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An Efficient and Robust Access method for Points and Rectangles", In Proc. ACM SIGMOD International Conference on management of Data, pp.332-331, 1990.

[5] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-tree: A dynamic index for multidimensional objects," In Proc. 13th Int. conference on Very Large Databases, pp.507-518, 1987.

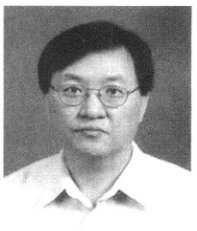
[6] Tobin J. Lehman, Michael J. Carey "A Study of Index Structures for Main Memory Database Management Systems", in Proceddings 12th Int'l. conf. on Very Large Databases, kyoto, pp.294-303, Aug., 1986.

[7] 최공림, 김경창, 김기룡, "T*-트리 : 주기억 데이터베이스에서의 효율적인 색인기법" 한국통신학회 논문지, 제 21권 제 10호, 1996.

[8] P. Bones, S. Manegold, and M. Kersten, "Database Architecture Optimized for the New Bottleneck: Memory Access", *Proceedings of VLDB Conference*, 1999, pp.54-56

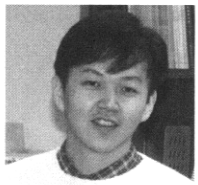
[9] Hongjun Lu, Yuet Yeung, Ng Zengping, "T-Tree or B-Tree : Main Memory Database Index Structure Revisited", Australasian Database Conference, 2000.
[10] J. Rao, K.A. Ross, "Making B+-Tree Cache Conscious in Main Memory", ACM SIGMOD, May, 2000.

김 경 창



e-mail : kckim@cs.hongik.ac.kr
1978년 홍익대학교 전자계산학과(학사)
1980년 한국과학기술원 전산학과(석사)
1990년 University of Texas at Austin 전산학과(박사)
1991년~현재 홍익대학교 정보컴퓨터공학부 교수
관심분야 : 객체지향 데이터베이스, 주기억 데이터베이스, OLAP 및 데이터 웨어하우징, Web 데이터베이스

윤 석 우



e-mail : sw0305@cs.hongik.ac.kr
1994년 홍익대학교 컴퓨터공학과(학사)
1996년 홍익대학교 전자계산학과(석사)
2002년~현재 홍익대학교 컴퓨터공학과 박사과정
관심분야 : 실시간 데이터베이스, 지리정보시스템, 모바일 데이터베이스