메인 메모리에서 선반입을 사용한 확장된 R-Tree 색인 기법†

An Extended R-Tree Indexing Method using Prefetching in Main Memory

강홍구*, 김동오*, 홍동숙*, 한기준**
Hong-Koo Kang, Dong-O Kim, Dong-Sook Hong, Ki-Joon Han

요 약 최근 메인 메모리 기반에서 R-Tree의 성능을 개선하기 위해 캐시를 고려한 색인 구조들이 제안되었다. 이들 색인 구조의 일반적인 캐시 성능 개선 방법은 엔트리 크기를 줄여 펜-아웃(fanout)을 증가시키고 하나의 노드에 더 많은 엔트리를 저장함으로써 캐시 실패를 최소화하는 것이다. 그러나 이러한 방법은 갱신시 줄어든 엔트리 정보를 복원하는 추가 연산으로 갱신 성능이 떨어지고, 노드간 이동시 발생하는 캐시 실패는 여전히 성능 저하의 큰 문제가 되고 있다.

본 논문은 이러한 문제점을 개선하기 위해 메인 메모리에서 R-Tree에 선반입을 적용한 확장된 메인 메모리 기반 R-Tree 색인 기법인 PR-Tree를 제안하고 평가하였다. PR-Tree는 R-Tree의 근본적인 변형없이 노드 크기를 선반입에 최적화되도록 확장하고, 노드간 이동시 자식 노드를 선반입하여 캐시 실패를 최소화하였다. PR-Tree는 실험에서 R-Tree보다 검색 연산에서는 최대 38%의 성능 향상을 보였고, 갱신 연산에서는 최대 30%의 성능 향상을 보였고, 또한 노드 분할 연산에서는 최대 67%의 성능 향상을 보였다.

ABSTRACT *Recently, studies have been performed to improve the cache performance of* the R-Tree in main memory. A general method to improve the cache performance of the R-Tree is to reduce size of an entry so that a node can store more entries and fanout of it can increase. However, this method generally requires additional process to reduce information of entries and do not support incremental updates. In addition, the cache miss always occurs on moving between a parent node and a child node.

To solve these problems efficiently, this paper proposes and evaluates the PR-Tree that is an extended R-Tree indexing method using prefetching in main memory. The PR-Tree can produce a wider node to optimize prefetching without additional modifications on the R-Tree. Moreover, the PR-Tree reduces cache miss rates that occur on moving between a parent node and a child node. In our simulation, the search performance, the update performance, and the node split performance of the PR-Tree improve up to 38%, 30%, and 67% respectively, compared with the original R-Tree.

주요어 : 메인 메모리, R-Tree, PR-Tree, 선반입, 공간 색인, 캐시
Key word : Main Memory, R-Tree, PR-Tree, Prefetching, Spatial Index, Cache

1. 서 론

최근 프로세서 속도와 메인 메모리 속도의 차이가 커짐에 따라 메인 메모리 기반 색인에서 캐시 메모리를 얼마나 효과적으로 사용하는가 하는 문제는 전체 성능에 결정적인 영향을 미치게 되었다. 이에 캐시 성능을 개선한 색인 구조와 알고리즘에 관한 연구가 많은 연구가에 의해 다양하게 진행되어 왔다[1][2][3][4][5].

특히, 지리 정보 시스템(Geographical Information System: GIS), 위치 기반 시스템 (Location Based System: LBS), 텔레매틱스 (telematics) 등과 같

† 본 논문은 대학IT연구센터 육성·지원사업의 연구결과로 수행되었음.
* 건국대학교 컴퓨터공학과 대학원생 [hkkang.dokim.dshang@db.konkuk.ac.kr]
** 건국대학교 컴퓨터공학과 교수 kjhan@db.kankuk.ac.kr

이 복잡한 공간 데이타의 빠르고 효율적인 처리를 요구하는 데이타베이스 응용 분야의 발전에 따라 캐시를 고려한 공간 색인 구조에 대한 연구도 활발해졌다.

Rao와 Ross는 메인 메모리 색인의 설계에 있어서 캐시 성능의 중요성을 제기하고 read-only OLAP 환경에서 이진 탐색 트리나 T-Tree보다 빠른 검색 성능을 보이는 CSS-Tree(Cache-Sensitive Search Tree)를 제안하였다[3]. 그리고, 이를 확장하여 B+-Tree의 캐시 성능을 향상시키는 CSB+-Tree를 제안하였다[4].

R-Tree에 대해서는 Sitzmann과 Stuckey가 R-Tree의 노드 크기를 캐시 라인 크기에 맞추고, MBR의 불필요한 정보를 제거하여 노드에 보다 많은 정보를 저장하는 pR-Tree(partial R-Tree)를 제안하였다[5]. 그리고 Kim과 Cha는 엔트리의 MBR을 압축하여 노드에 보다 많은 엔트리를 포함할 수 있도록 하는 CR-Tree (Cache-conscious R-Tree)를 제안하였다[2].

이러한 구조들의 일반적인 캐시 성능 개선 방법은 엔트리 크기를 줄여 펜-아웃(fanout)을 증가시키고 하나의 노드에 더 많은 엔트리를 저장함으로써 캐시 실패를 최소화하는 것이다. 그러나 이러한 방법은 갱신시 줄어든 엔트리 정보를 복원하는 추가 연산으로 갱신 성능이 떨어지고, 노드간 이동시 발생하는 캐시 실패는 여전히 전체 성능 저하에 큰 영향을 미친다.

본 논문은 이러한 문제점을 개선하고 캐시 성능을 향상시키기 위해 메인 메모리에서 R-Tree에 선반입(prefetching)을 적용한 확장된 메인 메모리 기반 R-Tree 색인 기법인 PR-Tree (Prefetcing R-Tree)를 제안하고 평가하였다. 본 논문에서 제시한 PR-Tree는 R-Tree의 근본적인 변형없이 노드 크기를 선반입에 최적화되도록 확장하고, 노드간 이동시에 발생하는 캐시 실패를 줄이기 위해 접근이 필요한 자식 노드를 미리 캐시 메모리에 적재한다. 선반입을 적용한 PR-Tree의 성능 향상은 접근할 노드의 크기와 개수에 비례한다. 따라서 점 질의 보다는 영역 질의에서 보다 효율적이다. PR-Tree는 다양한 환경에서 실험을 통하여 R-Tree보다 검색 및 갱신 연산에서 좋은 성능 향상을 보였고, 노드 분할 연산에서도 최대 67%의 높은 성능 향상을 보였다.

본 논문의 구성은 다음과 같다. 1장의 서론에 이어서 2장의 관련 연구에서는 먼저 캐시 메모리와 R-Tree 색인 기법을 소개하고, 캐시를 고려한 기존의 색인 구조들을 분석하고, 선반입 기법을 설명한다. 3장에서는 본 논문에서 제시한 PR-Tree의 개요와 구조 및 알고리즘을 설명한다. 4장에서는 PR-Tree의 성능을 분석하고, 실험을 통해 PR-Tree의 성능 결과를 보인다. 마지막으로, 5장에서는 결론에 대해 언급한다.

2. 관련 연구

본 장에서는 먼저 캐시 메모리와 R-Tree를 소개한다. 다음으로 기존의 캐시를 고려한 색인 구조들을 분석하고, 마지막으로 선반입에 대해 설명한다.

2.1 캐시 메모리

캐시 메모리는 프로세서에 빠른 속도로 데이타를 제공하기 위해 사용된다. 프로세서와 메인 메모리 사이에 위치하는 캐시 메모리는 보통 2계층으로 구성되며, 각각을 L1, L2 캐시라 부른다. L1 캐시는 레지스터와 L2 캐시 사이에 위치하고, L2 캐시는 L1 캐시와 메인 메모리 사이에 위치한다[6].

프로세서의 데이타 접근에서 데이타가 캐시 메모리에 존재하는 경우에 캐시 히트(cache hit)가 발생했다고 말하고, 존재하지 않은 경우에 캐시 실패(cache miss)가 발생했다고 말한다. 캐시 실패가 발생하면 LRU(Least Recently Used) 정책에 따라 캐시 메모리에서 가장 오랫동안 사용되지 않은 데이타를 내리고 필요한 데이타를 적재한다[3].

현재의 시스템들은 속도 향상을 위해 캐시 라인 크기와 캐시 메모리 용량이 커지는 추세에 있다. 표 1은 인텔의 Pentium3와 Pentium4 그리고 Sun의 UltraSparc II 시스템의 캐시 메모리 계층 구조의 특징을 나타낸다[7].

〈표 1〉 캐시 메모리 계층 구조 명세

	L1캐시	L2캐시
전체크기	16K	256K~1M
캐시 라인 크기	32~64B	32~128B
접근 시간	1 사이클	1~4 사이클
실패 비용	7~18 사이클	74~276 사이클
다음 레벨	L2 캐시	메인 메모리

일반적으로 데이타 캐시는 데이타 지역성(locality)의 기본 원리를 따른다. 배열과 같은 구조는 연속된 저장 구조이기 때문에 데이타 지역성이 높지만 트리와 같은 구조는 참조할 데이타가 포인터를 통해 접근하기 때문에 데이타 지역성이 낮다. 따라서 트리와 같은 구

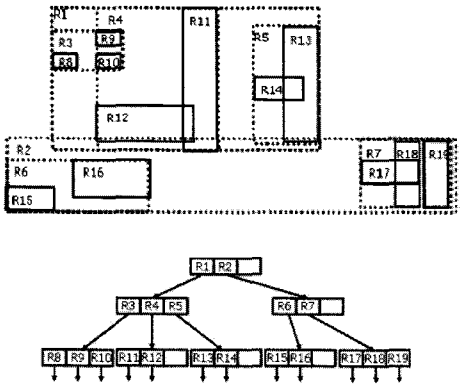
조에서 캐시 성능을 개선하기 위해서는 접근할 데이타의 양을 줄이거나 접근할 데이타를 선반입하는 방식이 필요하다.

2.2 R-Tree

R-Tree는 B-Tree를 공간 색인에 맞게 변형한 것이다. R-Tree는 B-Tree와 마찬가지로 트리 구조가 높이 균형적이며 객체에 대한 참조는 리프 노드에만 존재한다. 공간 객체를 표현하는데 최소 사각형(Minimum Bounding Rectangle: MBR)을 사용하는 R-Tree는 공간 객체를 찾기 위하여 적은 수의 노드만 방문하면 되도록 설계되었다. 또한, R-Tree에서는 트리 구조의 동적 생성이 지원되기 때문에 갱신을 검색과 혼합하여 진행할 수 있고 주기적으로 트리 구조를 재정열할 필요가 없다[8].

R-Tree에서 리프 노드의 형태는 (*I*, *tuple-identifier*)이며, 이 때, *tuple-identifier*는 공간 객체가 저장된 위치를 참조하는 포인터이고, I는 공간 객체에 대한 최소 사각형이다. 중간 노드의 형태는 (*I*, *child-pointer*)이고 이 때, *child-pointer*는 자식 노드를 가리키는 포인터이고, *I*는 자식 노드들을 완전히 포함하고 있는 MBR이다.한 개의 노드가 포함할 수 있는 최대 엔트리 수가 M이고 최소 엔트리 수를 m(m<M/2)이라 할 때 R-Tree는 다음과 같은 조건을 만족한다.

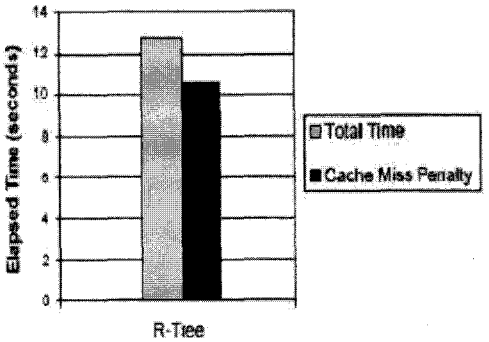
(a) 루트 노드(root node)는 리프 노드가 아니라면 적어도 2개의 자식을 가진다.
(b) 모든 중간 노드는 루트 노드가 아니라면 적어도 m에서 M개 사이의 자식을 가진다.
(c) 모든 리프 노드는 루트 노드가 아니라면 적어도 m에서 M개 사이의 자식을 가진다.
(d) 모든 리프 노드는 같은 높이에 있다.



〈그림 1〉 R-Tree 구조의 예

〈그림 1〉은 R-Tree 구조와 노드의 MBR 사이에 존재할 수 있는 포함(containment), 겹침(overlapping) 관계를 보여주고 있다. 그림 1에서 실선으로 표현된 사각형은 리프 노드의 엔트리에 의해 참조되는 실제 공간 객체의 MBR이고, 점선으로 표현된 사각형은 중간 노드의 MBR이다.

초기에 R-Tree는 디스크 기반 색인으로 디스크 I/O를 효과적으로 줄이기 위해 설계되었기 때문에 노드 크기가 디스크 블록에 최적화되었다. 그러나 이것은 블록이 작은 캐시 메모리에는 적합하지 못하다. 그림 2는 R-Tree에서 캐시 실패 비용을 보여준다.



〈그림 2〉 R-Tree의 캐시 실패 비용

〈그림 2〉에서 보듯이 R-Tree의 전체 수행 시간에서 캐시 실패로 인한 지연 시간이 매우 높은 것을 알 수 있다[7]. 특히 메인 메모리 공간 DBMS와 같이 R-Tree 색인이 메인 메모리에 상주하는 경우에는 디스크 I/O가 없어 캐시 실패가 전체 성능에 미치는 영향이 크므로 캐시 실패를 줄이는 색인 구조가 반드시 필요하다[9].

2.3 기존의 캐시 성능 향상을 위한 색인 구조

기존의 캐시 성능 향상을 위한 색인 기법에는 대표적으로 CSB+-Tree, pR-Tree, CR-Tree가 있다.

CSB+-Tree는 B+-Tree의 캐시 실패를 줄이기 위해 노드의 첫 번째 자식 노드의 포인터를 제외한 나머지 자식 노드들의 포인터를 제거하여 자식 노드를 메모리에 연속적으로 저장한 B+-Tree의 변형이다[4]. 자식 노드의 위치는 첫 번째 자식 노드로부터 메모리의 offset을 계산하여 얻는다. 그러나 이러한 포인터 제거 기법은 포인터가 차지하는 부분이 상대적으로 작은 R-Tree에서는 큰 성능 향상을 얻기 힘들다. 또한, 자식 노드가 연속적으로 저장되어 있기 때문에

갱신 연산마다 연속된 자식 노드의 재구성이 필요하고, 메모리 offset으로 데이타를 접근하므로 연속된 메모리 공간을 확보해야 하는 단점이 있다.

pR-Tree는 R-Tree의 캐시 실패를 줄이기 위해 부모 MBR의 좌표값과 겹침이 발생하는 자식 MBR 좌표값을 제거한 R-Tree의 변형이다[5]. 이 구조는 CSB+-Tree와 같이 포인터 제거 기법을 사용하였다. 그리고 엔트리에서 부모 MBR의 좌표값과 겹치는 자식 MBR의 좌표값을 4개의 비트에 저장하여 식별하고, 또한 캐시 실패를 줄이기 위해 노드 크기를 캐시 라인 크기에 맞추었다. 그러나 pR-Tree는 엔트리 수가 적을 때는 좋은 성능을 보이지만 엔트리 수가 증가할수록 부모 MBR의 좌표값과 겹치는 자식 MBR 좌표값의 비율이 크게 떨어져 성능이 나빠진다. 또한, 중복되는 자식 MBR 좌표값을 제거했기 때문에 갱신 연산에서 제거된 자식 MBR의 좌표값을 재구성하기 위한 추간 연산으로 갱신 성능이 나빠지는 단점이 있다.

CR-Tree는 R-Tree의 캐시 실패를 줄이기 위해 R-Tree에서 색인의 대부분을 차지하는 MBR을 압축하여 압축된 MBR을 키로 사용하는 R-Tree이다[2]. CR-Tree에서의 MBR의 압축은 먼저 자식 노드의 MBR을 부모 노드 MBR에 대해 상대적 좌표로 표현하고 이를 다시 일정한 비트로 표현될 수 있도록 양자화하여 이루어진다. 그러나 CR-Tree에서는 MBR을 압축하는 과정에서 원래의 MBR과 약간의 오차가 발생할 수 있고, 이러한 오차에 의해서 잘못된 결과(false hit)가 발생할 수 있다. 또한, 갱신 연산에서 압축된 MBR의 재구성을 위한 추가 연산으로 갱신 성능이 나빠지는 단점이 있다.

2.4 선반입 기법

선반입은 참조할 데이타를 미리 캐시 메모리에 적재함으로써 프로그램 수행 속도를 향상시키는 기법이다. 특히, 선반입은 캐시 메모리에 없는 데이타를 프로세서가 요구하기 전에 캐시 메모리에 적재함으로써 캐시 실패를 줄일 수 있다. R-Tree의 캐시 실패를 효율적으로 줄이기 위해서는 노드 접근시 발생하는 메모리 지연을 전체적으로 줄일 수 있도록 선반입이 적용되어야 한다. 선반입은 프로세서에 의해 자동적으로 수행되는 하드웨어적 방법과 프로그램 소스에 선반입 명령어를 삽입하는 소프트웨어적 방법으로 제어될 수 있다.

하드웨어적 선반입은 프로세서가 자동으로 데이타를 선반입하는 것으로, 선반입할 페이지를 예측하기 위해 지역성(locality)과 순차성(sequencing)과 같은 데이타 특성을 사용한다. 지역성은 참조된 캐시 블록이나 인접한 블록이 가까운 미래에도 참조될 확률일 높다는 특성이다. 순차성은 참조된 페이지의 다음 주소에 위치한 페이지가 참조될 확률이 높음을 나타내는 특성이다[10].

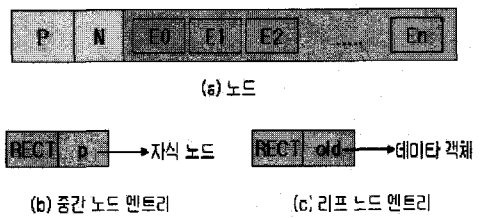
소프트웨어적 선반입은 프로그램 소스에 선반입 명령을 추가하는 것으로, 현대의 마이크로프로세서는 선반입을 수행하기 위해 반입(fetch) 명령을 지원한다. 반입 명령의 수행은 간단하고 속도도 매우 빠르며, 비-블록킹(non-blocking) 메모리 연산이기 때문에 캐시 메모리에서 동작하는 다른 메모리 연산들에 우선하여 수행된다. 따라서 성능 향상을 위해서는 프로그램 코드 사이의 적당한 위치에 반입 명령을 배치하는 것이 매우 중요하다.

3. PR-Tree

본 장에서는 본 논문에서 제안하는 선반입을 적용한 메인 메모리 기반 R-Tree 색인 구조인 PR-Tree에 대하여 기술한다. 먼저 PR-Tree의 구조와 특징을 기술하고, 다음으로 PR-Tree에서 사용하는 알고리즘을 설명한다.

3.1 PR-Tree 구조

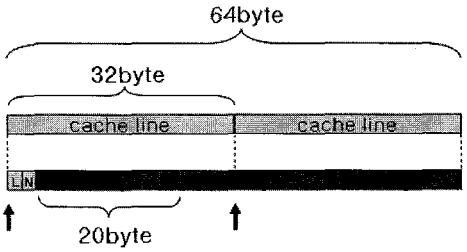
PR-Tree는 R-Tree에서와 마찬가지로 루트 노드, 중간 노드, 리프 노드를 가지고 있으며, 실제 데이타 객체에 대한 참조는 리프 노드에만 존재한다. 그림 3은 PR-Tree의 노드 구조를 보여준다.



〈그림 3〉 PR-Tree의 노드 구조

〈그림 3〉에서 (a)는 전체 노드 구조를 나타내며 P는 노드의 레벨을 나타내고, N은 엔트리 개수를 나타낸다. 그리고, En(n)=M/2)은 엔트리를 나타내고 엔트리는 중간 노드와 리프 노드에 따라 두 가지 형태가 있다. 그림 3에서 (b)와 (c)는 각각 중간 노드와 리프

노드의 엔트리 형태를 나타낸다. PR-Tree의 노드는 캐시 라인에 맞도록 엔트리의 개수를 제한한다. 즉, PR-Tree는 캐시 라인의 크기에 비례하게 노드 크기를 결정한다. 일반적으로 캐시 라인 크기는 32 바이트 또는 64 바이트이다. 예를 들어, 캐시 라인 크기가 32 바이트일 경우에는 노드의 크기는 64+160*n (n〉=0) 바이트가 되고, 64바이트일 경우에는 64+320*n (n〉=0) 바이트가 된다. 그림 4는 32 바이트 캐시 라인 2개로 구성된 PR-Tree 노드를 보여준다.



〈그림 4〉 PR-Tree의 노드 크기

〈그림 4〉에서 보듯이 노드가 가지는 엔트리 개수는 3개이다. 노드는 캐시 라인 단위로 선반입되며, 선반입이 적용되는 위치는 화살표가 가리키는 부분이다.

3.2 알고리즘

본 절에서는 PR-Tree의 검색, 삽입, 삭제, 노드 분할 알고리즘을 설명한다.

3.2.1 검색 알고리즘

PR-Tree의 검색 연산은 R-Tree와 유사하다. 루트 노드부터 시작하여 각 노드의 엔트리에 포함된 하위 노드 MBR 정보를 이용하여 하위 노드가 질의 사각형을 포함하거나 겹치는지 판단하는 과정을 리프 노드에 내려갈 때까지 반복한다. PR-Tree의 검색 알고리즘은 두 가지 방식으로 선반입 명령을 적용한다.

첫 번째 방식은 노드의 크기가 작을 경우에 현재 참조되는 노드의 모든 자식을 선반입하는 것이다. 자식 노드를 모두 선반입하면 겹침이 없는 자식 노드까지 선반입되지만 노드 크기가 작기 때문에 전체 자식 노드를 캐시 메모리에 선반입하는 시간이 매우 짧아 R-Tree 연산에 큰 영향을 미치지 않는다. 따라서, 노드 크기가 작은 경우에는 자식 노드 전체를 선반입하는 것이 메모리 지연을 보다 효율적으로 줄일 수 있다. 〈그림 5〉는 이러한 방식의 검색 알고리즘을 보여준다.

```
입력
  R: PR-Tree 노드
  W: 질의 사각형
출력
  Results: W와 겹치는 모든 객체
함수
  SEARCH: 공간을 분해하고 트리를 반복적으로 검색
알고리즘 ----------------------------------------
  begin
    Results = Φ
    if (R is not a leaf node) then // Search internal node
      prefetch (all entry's child);
      if (entry's child pointer is not NULL) then
        for each entry (p, RECT) of R do
          if (RECT overlaps W) then
            CHILD = node pointed to by p
            SEARCH (CHILD, W ∩ RECT)
          end if
        end for
      end if
    else // Search leaf node
      if (entry's child pointer is not NULL) then
        for each entry (p, RECT) of R do
          if (RECT overlaps W) then
            OBJECT = spatial object pointed to by p
            Results += OBJECT
          end if
        end for
      end if
    return Results
  end
```

〈그림 5〉 검색 알고리즘 1

두 번째 방식은 노드 크기가 큰 경우에 현재 노드의 다음에 접근할 자식 노드만 선반입하는 것이다. 노드 크기가 크면 전체 자식 노드의 개수가 늘고 노드 하나를 캐시 메모리에 선반입하는 시간도 길어지므로 노드 크기가 작은 경우와 같이 전체 자식 노드를 선반입하는 것은 매우 비효율적이다. 따라서, 노드 크기가 큰 경우에는 필요한 자식 노드만 선반입하는 것이 메모리 지연을 보다 효율적으로 줄일 수 있다. 〈그림 6〉은 이러한 방식의 검색 알고리즘을 보여준다.

```
입력
  R: PR-Tree 노드
  W: 질의 사각형
출력
  Results: W와 겹치는 모든 객체
함수
  SEARCH: 공간을 분해하고 트리를 반복적으로 검색
알고리즘 ----------------------------------------
  begin
    Results = Φ
    if (R is not a leaf node) then  // Search internal node
      if (entry's child pointer is not NULL) then
        for each entry (p, RECT) of R do
          prefetch (all entry's child);
          if (RECT overlaps W) then
            CHILD = node pointed to by p
            SEARCH (CHILD, W ∩ RECT)
          end if
        end for
      end if
    else  // Search leaf node
      if (entry's child pointer is not NULL) then
        for each entry (p, RECT) of R do
          if (RECT overlaps W) then
            OBJECT = spatial object pointed to by p
            Results += OBJECT
          end if
        end for
    end if
    return Results
  end
```

〈그림 6〉 검색 알고리즘 2

3.2.2 삽입 알고리즘

PR-Tree의 삽입 연산은 우선 검색 연산을 선행한다. 먼저 루트 노드부터 시작하여 각 노드의 엔트리에 포함된 모든 하위 노드 MBR 정보를 계산하여 하위 노드가 객체를 삽입할 때 크기 확장이 최소로 되는지를 판단하는 과정을 리프 노드에 내려갈 때까지 반복한다. 이 때, 객체를 삽입할 리프 노드에 이미 엔트리가 가득 차있으면 노드 분할이 발생한다. PR-Tree의 삽입 알고리즘에서 선반입은 삽입할 리프 노드를 찾는 과정에서 이루어진다. 〈그림 7〉은 삽입 알고리즘을 보여준다.

```
입력
  R: PR-Tree 노드
  IR: PR-Tree에 삽입될 사각형
출력
  IR을 삽입한 새로운 PR-Tree
함수
  INSERT : IR이 삽입될 곳을 찾고 해당 리프 노드에 추가
Algorithm ----------------------------------------
  begin
    if (R is not a leaf node) then  // Search internal node
      for each entry (p, RECT) of R do
        prefetch (all entry's child);
        if (RECT overlaps W) then
          CHILD = node pointed to by p
          INSERT (CHILD, IR)
        end if
      end for
    else  // Insert into leaf node
      ADD (IR, R)  // Insert IR into R
        if (R overflow) then
          SPLIT (R)
        end if
    end if
  end
```

〈그림 7〉 삽입 알고리즘

3.2.3 삭제 알고리즘

PR-Tree의 삭제 연산도 삭제할 객체를 찾는 검색 연산이 선행된다. 먼저 루트 노드부터 시작하여 각 노드의 엔트리에 포함된 하위 노드 MBR 정보를 계산하여 하위 노드가 질의 사각형을 포함하거나 겹치는지 판단하는 과정을 리프 노드에 내려갈 때까지 반복한다. 이 때, 엔트리의 삭제시 리프 노드에 최소 엔트리 개수보다 작아지면 노드가 삭제되고 남은 엔트리의 재삽입이 이루어진다.

PR-Tree의 삭제 알고리즘에서도 검색 알고리즘과 같이 노드 크기에 따라 두 가지 방식으로 선반입 명령을 적용한다.

첫번째 방식은 〈그림 8〉과 같이 노드의 크기가 작을 경우에 현재 참조되는 노드의 모든 자식 노드를 선반입하는 삭제 알고리즘이고, 두 번째 방식은 〈그림 9〉와 같이 노드 크기가 큰 경우에 현재 노드의 다음 접근할 자식 노드만 선반입하는 삭제 알고리즘이다.

```
입력
  R: PR-Tree 노드
  IR: PR-Tree로부터 제거되는 사각형
출력
  IR을 제거한 후의 새로운 PR-Tree
함수
  DELETE : IR의 위치를 찾고 해당 리프 노드에서 제거

Algorithm ----------------------------------------
  begin
    if (R is not a leaf node) then  // Search internal node
      prefetch (all entry's child);
      for each entry (p, RECT) of R do
        if (RECT overlaps IR) then
          CHILD = node pointed to by p
          DEL (CHILD, IR)
        end if
      end for
    else  // Remove from leaf node
      DEL (IR, R)   // Remove IR from R
    end if
  end
```

〈그림 8〉 삭제 알고리즘 1

```
입력
  R: PR-Tree 노드
  IR: PR-Tree로부터 제거되는 사각형
출력
  IR을 제거한 후의 새로운 PR-Tree
함수
  DELETE : IR의 위치를 찾고 해당 리프 노드에서 제거

알고리즘 ----------------------------------------
  begin
    if (R is not a leaf node) then  // Search internal node
      for each entry (p, RECT) of R do
        if (RECT overlaps IR) then
          prefetch (all entry's child);
          CHILD = node pointed to by p
          DEL (CHILD, IR)
        end if
      end for
    else  // Remove from leaf node
      DEL (IR, R)   // Remove IR from R
    end if
  end
```

〈그림 9〉 삭제 알고리즘 2

3.2.4 노드 분할 알고리즘

PR-Tree의 노드 분할은 삽입 연산 과정에서 객체를 삽입할 노드의 엔트리가 가득 찬 경우에 발생한다. 먼저 노드를 분할하기 위해 노드의 엔트리를 두 개의 그룹으로 나누고 분할 알고리즘에 따라 영역 확장이 최소가 되는 두 개의 노드로 분할된다. 노드 분할 알고리즘에서는 분할이 발생하기 전에 현재 노드를 선반입하고 엔트리를 나누기 위해 생성한 임시 버퍼를 선반입한다. 〈그림 10〉은 노드 분할 알고리즘을 나타낸다.

```
입력
  R: PR-Tree 노드
출력
  새로운 PR-Tree
함수
  SPLIT : 분할할 노드에 대한 파티션을 찾고, 새로운 두 개의
          노드를 생성. 필요시 분할이 상위나 하위 노드로 전파
알고리즘 ----------------------------------------
  begin
    prefetch(current node); prefetch(entry buffer);
    (S1, S2) = PARTITION (R)                // Find a partition
    RECT= MBROF (R);  P = POINTEROF (R);
    RECT1= MBROF (S1); RECT2= MBROF (S2);
    R1= CREATENODE (RECT1);
    R2= CREATENODE (RECT2);
    for each nodes Rk(Pk, RECTk) of R do
      if (RECT1 contains RECTk) then
        ADD(Rk, R1)
      else (RECT2 contains RECTk) then
        ADD (Rk, R2)
      end if
    end for
    if (R is root node) then
      RR = CREATENODE (RECT1 U RECT2)
      ADD (R1, RR);  ADD (R2, RR);
      ROOT = RR  // RR becomes the new root node
    else
      PR = parent node of R
      ADD (R1, PR);  ADD (R2, PR);
      if (PR overflows) then
        SPLIT (PR)
      end if
    end if
  end
```

〈그림 10〉 노드 분할 알고리즘

3.3 노드 크기 설정

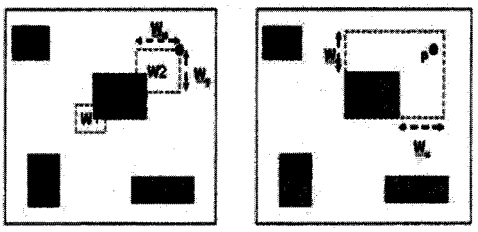
PR-Tree의 캐시 성능 향상을 위해 선반입에 최적화된 노드 크기가 필요하다. PR-Tree는 노드를 캐시 라인 단위로 선반입 하기 때문에 노드 크기에 따라 선반입 시간이 결정된다. 노드 크기가 작으면 실제로 노드가 필요한 시점보다 먼저 캐시 메모리에 적재되어 최악의 경우 다른 연산에 의해 캐시 메모리에서 밀려나는 경우가 발생할 수 있고, 반대로 노드 크기가 크면 노드가 필요한 시점보다 늦게 캐시 메모리에 적재되어 효율적으로 메모리 지연을 줄이지 못하게 된다.

노드 크기를 결정할 때에 많은 영향을 미치는 요소에는 캐시 라인 실패 시간(T_1), 파이프라인된 캐시 라인 실패 시간(T_{next})가 있다. 캐시 라인 크기의 정수배가 되는 노드는 처음 캐시 라인을 적재할 때 캐시 라인 실패가 발생하고, 다음 캐시 라인부터는 파이프라인에 적재되어 수행된다. 또한 T_1을 T_{next}로 나눈 비율이 메모리 대역폭(normalized bandwidth)라 하는데, 이 비율이 높을수록 프로세서의 병렬 처리 성능이 우수함을 알 수 있다. 〈표 2〉는 노드 크기 설정에 고려할 변수를 나타낸다.

〈표 2〉 노드 크기 설정에 고려할 변수

변수	설명
W	질의 사각형
Wx	질의 사각형의 X축 방향 길이
Wy	질의 사각형의 Y축 방향 길이
c	노드에 포함된 캐시 라인 개수
d	노드에 포함된 엔트리 개수
T_1	캐시 실패 지연 시간
T_{next}	파이프라인된 캐시 라인 지연 시간
T_{cache_miss}	전체 캐시 실패 비용
B	일반 메모리 대역폭 (T_1/T_{next})
N	전체 노드 개수
N.mbr	노드 N을 포함하는 최소 직사각형
N.mbr.xmin	N.mbr의 왼쪽 X좌표
N.mbr.ymin	N.mbr의 하단 y좌표

공간 객체들이 균등 분포되어 있고 같은 레벨의 각 노드는 대략 같은 크기를 갖는다고 가정하면 질의 사각형에 객체가 포함되는 확률은 전체 데이타 영역에서 질의 사각형이 차지하는 비율이 된다[11]. 따라서, 노드 N과 (Wx, Wy)의 크기를 가진 사각형 W를 가지고 있다고 가정하면, W의 우측 상단의 모서리로 점 질의한 확률로서 N.mbr이 W와 교차하는 확률을 평가할 수 있다. 범위 질의는 실제로 〈그림 11〉에서 (a)처럼 두 경우가 발생할 수 있다.



(a) 윈도우 질의 (b) (a)와 동등한 점 질의

〈그림 11〉 질의 사각형의 비용 측정

〈그림 11〉의 (a)에서는 질의 사각형(W1)의 우측 상단 모서리가 N.mbr에 포함되는 것을 나타내고 질의 사각형(W2)의 우측 상단 모서리가 N.mbr의 오른쪽으로 Wx만큼, 왼쪽으로 Wy만큼 팽창된 부분에 포함되는 것을 보여준다. 만약 질의 사각형 W2의 우측 상단 모서리가 사각형 W=(N.mbr.xmin + Wx, N.mbr.ymin + Wy) 안에 포함되면 W2가 N.mbr과 겹침을 알 수 있고, 그림 12의 (b)와 같이 W에 관한 점 질의로 평가될 수 있다. 정리하면 범위 질의에서 탐색되는 노드 개수 WQ(w)는 다음과 같이 계산될 수 있다.

$$WQ(w) = \sum_{i=0}^{n} (s_{i,y} + w_x) \times (s_{i,y} + w_y)$$

위 식은 R-Tree, R*-Tree의 특정 타입, 그리고 정적, 동적 구성 모드에 관계없이 R-Tree의 특성에 의존한다[11].

전체 노드 개수를 n이라 하고 하나의 노드에 d개의 엔트리를 갖는 PR-Tree는 대략 $\lceil \log_d(N/d-1)+1 \rceil$의 높이를 가진다. 그리고, 높이가 h일 때의 노드 개수는 $\lceil d^{(\log_d(N/d-1)+1)} \rceil$이 된다. 객체가 전체 영역에 균등하게 분포되어 있고 노드는 가질 수 있는 엔트리를 모두 가진다고 가정하자. 이 때, 전체 트리에서 검색으로 인한 평균 메모리 지연 시간(T_{cache_miss})은 다음과 같이 계산될 수 있다.

$$\begin{aligned} T_{cache-miss} &= WQ(w) \times (T_1 + (d-1) \times T_{next}) \\ &= T_{next} \times WQ(w)(\frac{T_1}{T_{next}} + (d-1)) \\ &= T_{next} \times WQ(w)(B + (d-1)) \end{aligned}$$

T1과 T_{next}는 상수이므로 하드웨어의 성능에 의해 결정된다. 따라서, PR-Tree의 메모리 지연 시간은 WQ(w)와 d에 의해 결정된다. WQ(w)는 질의 사각형이 클수록 높아지고 d는 노드가 포함하는 엔트리 개

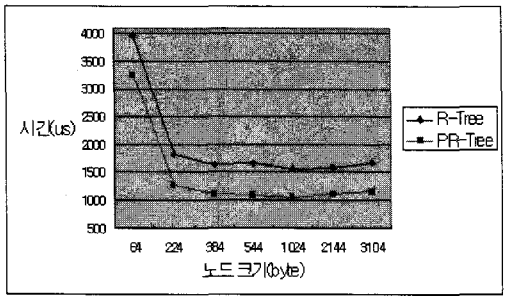
수가 클수록 높아진다.

PR-Tree에서 줄일 수 있는 메모리 지연 시간은 질의 사각형과 겹치는 노드의 개수에 비례한다. 따라서, 노드 내의 공간을 가장 많이 이용하는 조건을 만족하기 위해 캐시 라인 블록에 맞는 엔트리의 개수를 제한하도록 d, c, WQ(w)의 조합에 대해서 캐시 비용을 계산함으로써 적당한 크기를 갖는 노드를 설정할 수 있다.

4. 성능 평가

PR-Tree의 성능 평가를 위하여 리눅스 환경에서 C 언어로 R-Tree와 PR-Tree를 구현하였다. 실험에 사용된 컴퓨터 시스템은 인텔 Pentium3 1GHz이고 L1, L2 캐시의 캐시 라인 크기는 각각 32바이트이며, 메인 메모리의 용량은 1GB이다.

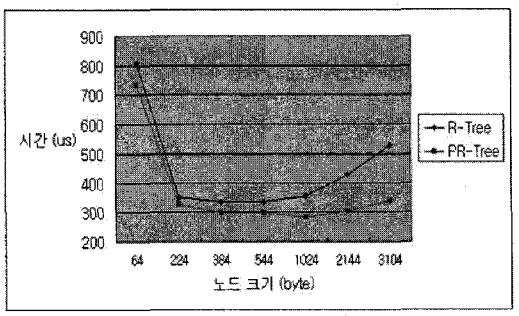
검색 연산의 실험을 위해 한 변의 길이가 1인 정사각형을 전체 공간 영역으로 설정하였다. 공간 객체는 실수형(float)으로 전체 영역에 균등하게 분포하도록 10,000개의 객체를 임의로 생성하고, 객체의 크기는 한 변의 길이가 평균 0.0001이 되도록 하였다. 검색에 사용할 질의 영역은 전체 영역의 70%를 포함하는 것으로 하였고, 임의로 10,000번의 검색 질의를 수행하여 PR-Tree 와 R-Tree간의 성능을 비교하였다. 〈그림 12〉는 검색 연산을 수행한 결과를 보여준다.



〈그림 12〉 검색 연산 성능 비교

〈그림 12〉와 같이 대체로 노드 크기가 커질수록 검색 성능이 좋아지고, 선반입을 이용한 성능 향상이 일정한 것을 볼 수 있다. 이는 노드 접근에서 발생하는 메모리 지연을 줄였기 때문이다. PR-Tree의 검색 성능은 〈그림 13〉에서 보듯이 최대 35%가 향상되었다.

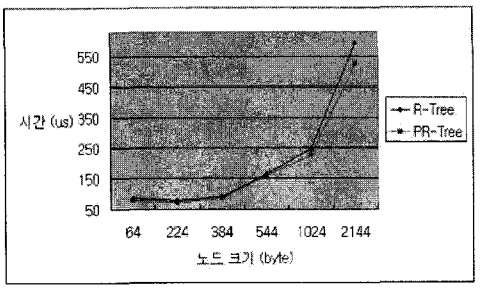
공간 객체의 불균형 분포에 대한 검색 연산 실험에서 공간 객체는 실수형(float)으로 전체 영역에 불균등하게 분포하도록 10,000개의 객체를 임의로 생성하고, 객체의 크기는 한 변의 길이가 평균 0.0001이 되도록 하였다. 검색에 사용할 질의 영역은 전체 영역의 30%를 포함하는 것으로 하였고, 임의로 10,000번의 검색 질의를 수행하여 PR-Tree와 R-Tree간의 성능을 비교하였다. 그림 13은 불균등 분포에서 검색 연산을 수행한 결과를 보여준다.



〈그림 13〉 불균등 분포에서 검색 연산 성능 비교

〈그림 13〉과 같이 대체로 노드 크기가 커질수록 검색 성능이 좋아지는 것을 볼 수 있다. 이는 공간 객체가 불균등하게 분포되어 있기 때문에 노드간 겹침이 높아지고 접근하는 노드의 개수가 증가하여 선반입으로 줄어지는 메모리 지연 시간이 상대적으로 많아졌기 때문이다. PR-Tree의 검색 성능은 그림 13에서 보듯이 최대 38%가 향상되었다.

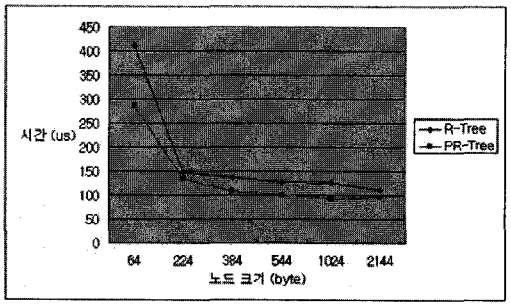
삽입 연산의 실험에서 공간 객체는 실수형(float)으로 전체 영역에 균등하게 분포하도록 10,000개의 객체를 임의로 생성하고, 객체의 크기는 전체 질의 영역의 평균 0.0001이 되도록 하였다. 삽입할 객체는 한 변이 평균 0.0001이 되도록 하였다. 그리고, 객체를 임의로 10,000번 삽입 질의를 수행하여 PR-Tree와 R-Tree간의 성능을 비교하였다. 그림 14는 삽입 연산을 수행한 결과를 보여준다.



〈그림 14〉 삽입 연산 성능 비교

〈그림 14〉에서 보듯이 대체적으로 노드 크기가 클수록 삽입되는 시간이 증가하나 선반입으로 인한 성능 향상 비율은 증가하는 것을 볼 수 있다. 이는 노드 크기가 클수록 선반입을 이용한 성능 향상이 높아지기 때문이다. PR-Tree의 삽입 성능은 실험에서 R-Tree보다 최대 11%의 성능 향상을 보였다.

삭제 연산의 실험에서 공간 객체는 실수형(float)으로 전체 영역에 균등하게 분포하도록 10,000개의 객체를 임의로 생성하고, 객체의 크기는 전체 질의 영역의 평균 0.0001이 되도록 하였다. 삭제할 영역은 한 변이 평균 0.001이 되도록 하였다. 그리고, 임의로 10,000번 삭제 질의를 수행하여 PR-Tree와 R-Tree 간의 성능을 비교하였다. 〈그림 15〉는 삭제 연산을 수행한 결과를 보여준다.

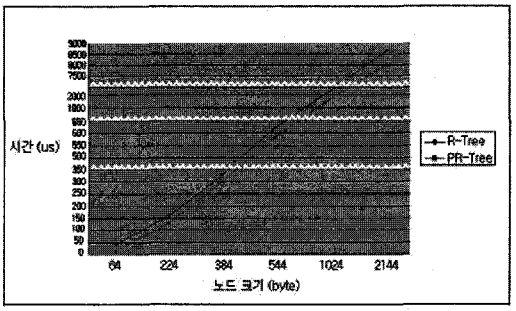


〈그림 15〉 삭제 연산 성능 비교

〈그림 15〉에서 보듯이 대체적으로 노드 크기가 클수록 삭제 성능이 좋아지고, 선반입으로 인한 성능 향상이 일정한 것을 볼 수 있다. 이는 노드 접근에서 선반입으로 줄이는 메모리 지연 시간이 일정하기 때문이다. PR-Tree의 삭제 성능은 실험에서 R-Tree보다 최대 30%의 성능 향상을 보였다.

노드 분할 연산의 실험에서 공간 객체는 실수형(float)으로 전체 영역에 균등하게 분포하도록 10,000개의 객체를 임의로 생성하고, 객체의 크기는 전체 질의 영역의 평균 0.0001이 되도록 하였다. 그리고, 임의의 객체 10,000개를 삽입 질의하여 이 때, 발생하는 노드 분할 시간을 측정하여 같은 조건에서의 R-Tree의 노드 분할 시간과 비교하였다. 〈그림 16〉은 노드 분할 연산을 수행한 결과를 보여준다.

〈그림 16〉에서 보듯이 대체적으로 노드 크기가 클수록 분할하는 시간이 증가하는 것을 볼 수 있다. 그리고, 선반입으로 인한 성능 향상 폭이 일정한 것을 볼 수 있다. 이는 노드 크기가 클수록 선반입으로 줄어지는 메모리 지연 시간이 많아지기 때문이다. PR-Tree의 노드 분할 성능은 실험에서 R-Tree보다 최대 67%의 성능 향상을 보였다.



〈그림 16〉 노드 분할 연산 성능 비교

5. 결 론

최근 프로세서의 속도와 메인 메모리의 속도의 차이가 커지면서 데이타베이스 관리 시스템을 포함하여 많은 컴퓨터 애플리케이션에서 캐시 실패로 인한 성능 저하가 문제시 되고 있다. 특히, GIS, LBS, 텔레매틱스 등과 같이 복잡한 공간 데이타의 빠르고 효율적인 처리를 요구하는 공간 응용 분야의 발전에 따라 캐시를 고려한 공간 색인 구조에 대한 연구도 활발해졌다.

최근에 메인 메모리 기반 R-Tree 색인 구조의 캐시 성능을 개선하기 위해 노드 크기를 줄이는 방안이 제안되었으나, 이러한 방안은 갱신시 엔트리 정보를 복원하는 추가 연산으로 갱신 성능이 떨어지고 여전히 노드간 이동마다 발생하는 캐시 실패가 전체 성능 저하에 큰 영향을 미치고 있다.

본 논문은 이러한 문제점을 개선하기 위해 선반입 기법을 R-Tree에 적용하여 캐시 실패를 줄이고 갱신 연산에서 추가 비용이 발생하지 않는 PR-Tree를 제안하였다. PR-Tree는 R-Tree의 근본적인 변형없이 노드 크기를 선반입에 최적화하고, 노드간 이동시 자식 노드를 선반입하여 캐시 실패를 최소화하였다. PR-Tree는 실험에서 R-Tree보다 검색 연산에서는 최대 38%의 성능 향상을 보였으며, 갱신 연산에서는 최대 30%의 성능 향상을 보였고, 노드 분할 연산에서는 최대 67%의 성능 향상을 보였다.

향후에는 종합적인 테스트 데이타를 가지고 실제 메인 메모리 공간 DBMS 시스템에 적용하여 다양한 실험을 할 계획이다.

참고문헌

[1] Chen, S., Gibbons, P., Mowry, T., and Valentin, G., Fractal Prefetching B+-Trees : Optimizing Both Cache and Disk Performances, Proceedings of ACM SIGMOD Conference, 2002, pp.157-168.

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

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

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

[5] Sitzmann, P., and Stuckey, "Compacting Discriminator Information for Spatial Trees," Proceedings of Australasian Database Conference, 2002, pp.167-176.

[6] Mowry, T., Lan, S., and Gupta, A., "Design and Evaluation of a Compiler Algorithm for Prefetching," Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems, 1992, pp.62-73.

[7] Zhou, J., and Ross, K., "Buffering Accesses of Memory-Resident Index Structures," Proceedings of VLDB Conference, 2003, pp.405-416.

[8] Guttman, A., "R-Trees: a Dynamic Index Structure for Spatial Searching," Proceedings of ACM SIGMOD Conference, 1984, pp. 47-54.

[9] 강홍구, 김동오, 홍동숙, 한기준, PR-Tree: 메인 메모리에서 선반입을 적용한 확장된 R-Tree 색인 구조, 개방형지리정보시스템학회, 2003 추계학술대회 논문집, 2003, pp.123-128.

[10] VanderWiel, S., and Lilja, D., Data Prefetch Mechanisms, ACM Computing Surveys, Vol.32, 2000, pp.174-199.

[11] Kamel, I., and Faloutsos, C., On Packing R-Trees, Proceedings of ACM CIKM Conference, 1993, pp.490-499.

강홍구
2002년 건국대학교 컴퓨터공학과 졸업(학사)
2004년 건국대학교 대학원 컴퓨터공학과 졸업(석사)
2004년~현재 건국대학교 대학원 컴퓨터공학과 박사과정
관심분야: 지리정보시스템, 공간데이타베이스, MMDB, MODB, LBS

김동오
2000년 건국대학교 컴퓨터공학과 졸업(학사)
2002년 건국대학교 대학원 컴퓨터공학과 졸업(석사)
2002년~현재 건국대학교 대학원 컴퓨터공학과 박사과정
관심분야: 지리정보시스템, GML, LBS, MODB

홍동숙
1999년 건국대학교 컴퓨터공학과 졸업(학사)
2001년 건국대학교 대학원 컴퓨터공학과 졸업(석사)
2001년~2003년 ㈜쌍용정보통신
2003년~현재 건국대학교 대학원 컴퓨터공학과 박사과정
관심분야: 지리정보시스템, LBS, MODB, XML, 데이타베이스

한기준
1979년 서울대학교 수학교육학과 졸업(학사)
1981년 한국과학기술원 전산학과 졸업(석사)
1985년 한국과학기술원 전산학과 졸업(박사)
1990년 Stanford 대학 전산학과 visiting scholar
1985년~현재 건국대학교 컴퓨터공학부 교수
2004년~현재 개방형지리정보시스템학회 회장
2004년~현재 한국정보시스템감리사협회 회장
관심분야: 데이타베이스, GIS, LBS, 텔레매틱스, 정보시스템 감리