대용량 공간 데이터의 빠른 검색을 위한 해시 기반 R-Tree†

A Hash based R-Tree for Fast Search of Mass Spatial Data

강홍구* · 김정준 · 신인수 · 한기준
Hong-Koo Kang* · Joung-Joon Kim · In-Su Shin · Ki-Joon Han
건국대학교 컴퓨터·정보통신공학과
{hkkang*, jjkim9, isshin, kjhan}@db.konkuk.ac.kr

요 약

최근, GIS 분야에서 RFID와 GPS 센서 같은 위치 및 공간 데이타를 포함하는 다양한 GeoSensor의 활용으로 수집되는 공간 데이타가 크게 증가하면서, 대용량 공간 데이타의 빠른 처리를 위한 공간 인덱스의 중요성이 높아지고 있다. 특히, 대표적인 공간 인덱스인 R-Tree를 기반으로 검색 성능을 높이기 위한 연구가 활발히 진행되고 있다. 그러나, 기존 연구는 R-Tree에서 노드의 MBR 간의 겹침이나 트리 높이를 어느 정도 줄임으로써 다소 검색 성능을 향상시켰지만, 트리 검색에서 발생하는 불필요한 노드 접근 비용 문제를 효율적으로 해결하지 못하고 있다.

본 논문에서는 이러한 문제를 해결하고 R-Tree에서 대용량 공간 데이타의 빠른 검색을 제공하는 인덱스인 HR-Tree(Hash based R-Tree)를 제시한다. HR-Tree는 트리 검색 없이 R-Tree 리프 노드를 직접 접근할 수 있는 해시 테이블을 이용함으로써 R-Tree의 검색 성능을 높인다. 해시 테이블은 데이타 영역을 차원에 따라 반복적으로 분할한 Partition과 대응되는 R-Tree 리프 노드의 MBR과 포인터들로 구성된다. 각 Partition은 생성 과정에서 고유의 식별 코드를 갖기 때문에 Partition 코드가 주어지면 해시 테이블에서 해당 레코드를 쉽게 접근할 수 있다. 또한, HR-Tree는 R-Tree 구조의 변경없이 다양한 R-Tree 변형 구조에 쉽게 적용할 수 있는 장점이 있다. 마지막으로 실험을 통하여 HR-Tree의 우수성을 입증하였다.

1. 서론

u-GIS란 독립적으로 발전해온 GIS 기술과 USN 기술을 접목하여 유비쿼터스 시대에 필요한 공간 정보와 센서 정보의 융·복합 저장 및 처리를 통한 차세대 시각화 및 사용자의 다양한 요구에 적정한 맞춤형 정보 서비스 기술이다[1]. 이러한 u-GIS 환경에서 RFID와 GPS 센서 같은 위치 및 공간 데이타를 포함하는 다양한 GeoSensor의 활용으로 수집되는 공간 데이타가 크게 증가하면서, 대용량 공간 데이타의 빠른 처리를 위한 공간 인덱스의 중요성이 높아지고 있다[2].

특히, 가장 널리 사용되고 있는 공간 인덱스인 R-Tree[3]를 기반으로 검색 성능을 향상시키기 위한 연구가 활발히 진행되고 있다[4,5,6,7,8,9]. 제시된 대표적인 인덱스에는 R^+-Tree[4], R^*-Tree[5], Hilbert R-Tree[6], X-Tree[7], QR-Tree[8], iQR-Tree[9] 등이 있다. 그러나 이들 인덱스는 노드의 MBR간 겹침이나 트리 높이를 어느 정도 줄임으로써 검색 성능을 향상시켰으나, 트리 검색에서 발생하는 불필요한 노드 접근 비용 문제를 효율

† 본 연구는 건설교통부 첨단도시기술개발사업-지능형국토정보기술혁신 사업과제의 연구비지원(07국토정보C05)에 의해 수행되었음.

적으로 해결하지 못하고 있다.

본 논문에서는 이러한 문제를 해결하고, R-Tree에서 대용량 공간 데이타의 빠른 검색을 제공하는 새로운 인덱스인 HR-Tree(Hash based R-Tree)를 제시한다. HR-Tree는 데이타 영역을 차원에 따라 서로 다른 크기를 갖는 여러 Partition으로 분할한다[11]. 분할 과정에서 각 Partition은 고유의 식별 코드를 갖게 되며, Partition 간에는 계층 구조 형태를 갖게 된다. HR-Tree에서 해시 테이블은 데이타 영역을 차원에 따라 반복적으로 분할한 Partition과 대응되는 R-Tree 리프 노드의 MBR과 포인터들로 구성된다.

따라서, HR-Tree는 R-Tree와 같이 루트 노드에서 리프 노드까지의 트리 검색을 피하고, 주어진 Partition 코드를 해싱하여 해시 테이블에서 해당 R-Tree의 리프 노드를 빠르게 접근할 수 있다. 또한, HR-Tree에서 해시 테이블은 R-Tree의 기본 연산 과정에서 쉽게 구성할 수 있으므로 다양한 R-Tree 변형 구조에 적용할 수 있는 장점이 있다. 마지막으로 성능 실험을 통해 기존 연구에 비해 HR-Tree의 검색 성능이 우수함을 입증하였다.

본 논문의 구성은 다음과 같다. 2장의 관련 연구에서는 R-Tree와 R-Tree의 검색 성능을 향상시키는 기존 연구에 대해 설명한다. 3장에서는 본 논문에서 제시하는 HR-Tree에 대해 기술한다. 4장에서는 실험을 통한 성능 평가를 보인다. 마지막으로 5장에서는 결론에 대해 언급한다.

2. 관련 연구

본 장에서는 R-Tree와 R-Tree의 검색 성능을 향상시키는 기존 연구들에 대해 설명한다.

2.1 R-Tree

R-Tree는 B-Tree를 공간 인덱스에 맞게 변형한 것으로서, 공간 객체를 표현하기 위해 MBR을 사용하는 높이 균형 트리이다. R-Tree는 공간 객체에 대한 참조가 리프 노드에만 존재하고, 트리 구조의 동적인 생성을 지원하기 때문에 갱신을 검색과 혼합하여 진행할 수 있다[3].

R-Tree의 중간 노드 형태는 (p, RECT)로 나타낼 수 있다. 여기서 p는 자식 노드에 대한 포인터이고, RECT는 자식 노드에 대한 MBR이다. R-Tree의 리프 노드 형태는 (oid, RECT)로 나타낼 수 있다. 여기서 oid는 공간 객체가 저장된 디스크 페이지에 대한 포인터이고, RECT는 공간 객체에 대한 MBR이다. 노드의 최대 엔트리 수가 M이고 최소 엔트리 수를 m($m \le M/2$)이라 할 때, R-Tree는 다음과 같은 특성을 가진다.

(a) 루트 노드는 리프가 아니면, 적어도 2개의 자식을 가진다.
(b) 모든 중간 노드는 루트가 아니면, m에서 M개 사이의 자식을 가진다.
(c) 모든 리프 노드는 루트가 아니면, m에서 M개 사이의 자식을 가진다.
(d) 모든 리프 노드는 높이가 같다.

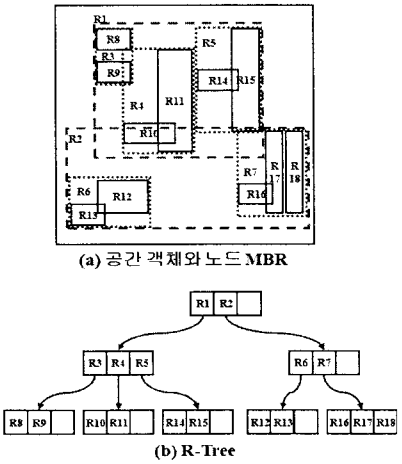


그림 2. R-Tree 예

그림 1은 R-Tree의 예를 보여준다. 그림 1(a)에서 사각형 R1~R7은 R-Tree 노드의 MBR이고, 사각형 R8~R18은 공간 객체의 MBR이다. 이와 같이 R-Tree는 노드의 MBR이 서로 겹치기 때문에 루트 노드에서 리프 노드에 도달하는 검색 경로가 하나 이상 존재하며, 검색 경로 상의

모든 노드를 접근해야 한다[3,10].

R-Tree에서 노드의 빈번한 접근은 디스크 I/O가 높은 것을 의미하므로 검색 비용을 줄이기 위해 많은 인덱스 구조가 제안되었다. 대표적인 인덱스로 R^+-Tree, R^*-Tree, Hilbert R-Tree, X-Tree, QR-Tree, iQR-Tree 등이 있다.

2.2 iQR-Tree

가장 최근에 제시된 iQR-Tree는 Quad-Tree와 R-Tree를 결합한 인덱스로 Quad-Tree 노드에 R-Tree를 연계시킨 구조를 가진다[9]. 그림 2는 iQR-Tree의 예를 보여준다.

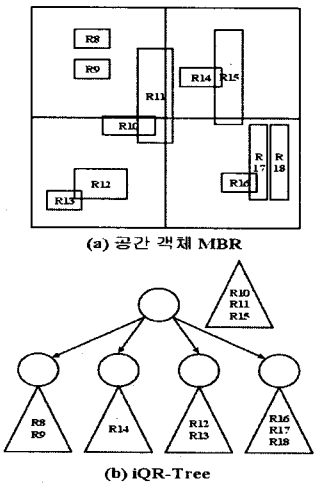


그림 2. iQR-Tree 예

그림 2에서 구(Sphere)는 Quad-Tree의 노드를 나타내고, 삼각형은 R-Tree를 나타낸다. iQR-Tree에서 삽입되는 공간 객체는 Quad-Tree의 하위 경계와 겹치는 경우, 공간 객체를 포함할 수 있는 상위 노드와 연계되는 R-Tree에 저장되고, 그렇지 않은 경우에는 하위 노드와 연계되는 R-Tree에 저장된다. 예를 들어, 공간 객체 MBR인 R10, R11, R15는 Quad-Tree의 루트 노드의 하위 경계와 겹치기 때문에 루트 노드와 연계된 R-Tree에 저장되고 나머지는 하위 노드와 연계된 R-Tree에 저장된다.

iQR-Tree는 단순히 데이타 영역을 구분하는 Quad-Tree를 메인 메모리에 구성하고, 실제 공간 객체를 저장하는 R-Tree를 디스크에 구성하는 이중 구조를 가지고 있다. 이를 통해 R-Tree 높이를 줄이고, 검색시 디스크 접근을 줄임으로써 검색 성능을 향상시켰다.

그러나 인덱스 확장으로 Quad-Tree의 높이가 커지면, 하위 R-Tree를 재구성하는 오버헤드가 발생하고, R-Tree 영역간 겹침이 증가하여 검색 성능이 급격히 떨어지는 문제가 있다[8,9].

3. HR-Tree

본 장에서는 본 논문에서 제시한 HR-Tree에 대하여 설명한다.

3.1 Partition 생성

HR-Tree는 R-Tree의 리프 노드를 직접 접근하게 해주는 해시 테이블을 구성하기 위해 전체 데이타 영역을 서로 다른 크기의 여러 Partition으로 분할한다. Partition들은 분할되면서 계층적인 형태를 갖는다. 따라서, Partition은 분할이 이루어지지 않은 최하위 Partition 즉, 리프 Partition과 분할이 이루어진 Partition 즉, 중간 Partition으로 구분할 수 있다.

중간 Partition은 분할 경계와 겹치는 하나 이상의 R-Tree 리프 노드와 연계되고, 리프 Partition은 분할 경계가 없기 때문에 리프 Partition에 완전히 포함되는 하나의 R-Tree 리프 노드와 연계된다. 만일 리프 Partition에 리프 노드의 추가로 인해 오버플로우(Overflow)가 발생되면, 리프 Partition은 분할된다. 이러한 Partition의 주요 특징은 다음과 같다.

(a) 데이타 영역은 서로 다른 크기를 가지는 Partition들로 분할된다.
(b) Partition은 계층적인 형태를 가진다.
(c) 각 Partition은 식별 가능한 비트 스트링 형태의 코드를 가진다.
(d) Partition 코드를 통해 Partition의 분할 차원과 위치를 알 수 있다.

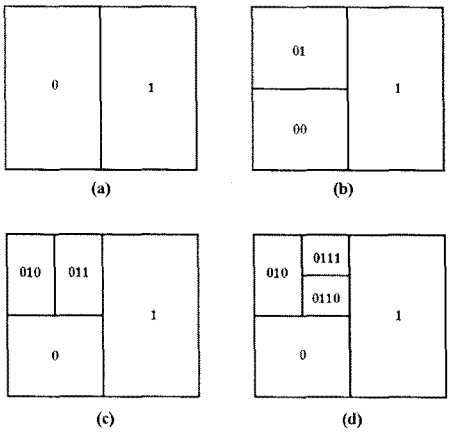


그림 3. Partition 생성 과정

그림 3은 x-y 평면에서 Partition의 생성 과정을 보여준다. 그림 3(a)는 전체 데이타 영역을 나타내는 Partition에 오버플로우가 발생하여, x축을 분할 축으로 하여 같은 크기를 갖는 두 개의 하위 Partition으로 분할된 것을 보여준다. 분할된 Partition의 코드는 각각 0과 1이다. 그림 3(b)는 코드가 0인 Partition에 오버플로우가 발생하여 y축을 분할 축으로 하여 같은 크기를 갖는 두 개의 하위 Partition으로 분할된 것을 보여준다. 분할된 Partition의 코드는 각각 00과 01이다. 그림 3(c)와 그림 3(d)도 마찬가지로 분할 축을 재순환(Recycle)하면서 Partition이 생성되는 것을 보여준다.

3.2 해시 테이블

해시 테이블은 R-Tree의 리프 노드를 직접 접근하게 해주는 기능을 가지고 있으며, Partition과 연계된 R-Tree 리프 노드 정보를 이용하여 구성된다. 중간 Partition에는 분할 경계와 겹치는 R-Tree 리프 노드가 연계되고, 리프 Partition에는 Partition에 완전히 포함되는 R-Tree 리프 노드가 연계된다. 그림 4는 Partition에 연계되는 R-Tree 리프 노드를 보여준다.

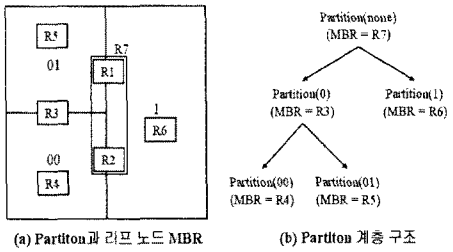


그림 5. Partition과 리프 노드의 연계

그림 4(a)에서 중간 Partition의 코드는 none과 00이고, 리프 Partition의 코드는 1, 01, 11이다. 그리고, 그림 4(b)와 같이 Partition은 계층적인 구조를 가지며 각 Partition은 자신과 연계된 R-Tree 리프 노드들을 모두 포함하는 MBR을 가진다. HR-Tree의 해시 테이블은 Partition이 가지는 MBR과 리프 노드를 접근하기 위한 포인터를 가지는 구조체 배열로 구성된다. 그림 5는 해시 테이블 구조를 나타낸다.

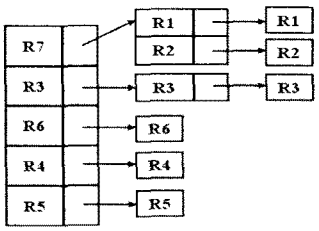
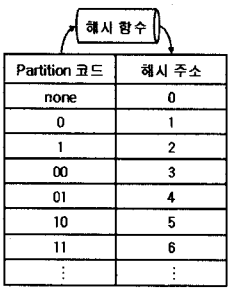


그림 6. 해시 테이블 구조

그림 5는 그림 4(a)에서 나타내는 Partition과 연계된 리프 노드에 대한 해시 테이블을 나타내고 있다. 그림 5와 같이 해시 테이블은 Partition과 연계되는 R-Tree 리프 노드를 모두 포함하는 MBR과 R-Tree 리프 노드를 가리키는 포인터들로 구성되어 있다. 해시 테이블의 레코드는 해시 주소인 배열 변수로 식별이 가능하므로 간단히 Partition 코드를 해싱하여 접근할 수 있다. 그림 6은 Partition 코드와 대응되는 해시 주소를 나타낸다.



Partition 코드	해시 주소
none	0
0	1
1	2
00	3
01	4
10	5
11	6
⋮	⋮

그림 7. Partition 코드와 해시 주소

그림 6과 같이 Partition 코드는 해시 함수를 통해 해시 테이블의 배열 변수, 즉 해시 주소를 나타낸다. 이 때, Partition 코드 none은 비트가 하나도 없는 것을 나타내며, Partition은 전체 데이타 영역을 나타낸다.

해시 함수는 Partition 코드를 구성하는 비트의 개수에 따른 초기값(Initial Value)과 비트의 조합에 따른 코드값(Code Value)의 합인 산술식으로 나타낼 수 있다. 따라서, Partition 코드의 비트 개수를 n, i번째 비트 값을 Bi일때, 초기값과 코드값은 각각 식1과 식2로 표현할 수 있다. 단, 비트 개수 n이 0이면 초기값과 코드값은 0이 되는 것으로 가정한다.

$$\sum_{i=1}^{n} 2^{i-1} \ (n \geq 1) \qquad \text{- (식1)}$$

$$\sum_{i=1}^{n} Bi \times 2^{i-1} (n \geq 1) \qquad \text{- (식2)}$$

결과적으로 해시 함수는 식3으로 표현할 수 있다.

$$\sum_{i=1}^{n} (Bi+1) \times 2^{i-1} (n \geq 1) \ \text{- (식3)}$$

3.3 HR-Tree 구조 및 알고리즘

HR-Tree는 R-Tree에서 리프 노드를 직접 접근할 수 있는 해시 테이블을 이용하여 검색 성능을 향상시킨 인덱스 구조이다.

HR-Tree는 전체 데이타 영역을 차원에 따라 서로 다른 크기의 여러 Partition으로 분할하고, R-Tree의 리프 노드를 각 Partition과 연계시킨다. 그리고 Partition에 주어진 고유한 식별 코드를 통해 R-Tree의 리프 노드를 접근하는 해싱 기법을 이용한다. 그림 7은 HR-Tree의 전체 구조를 나타낸다.

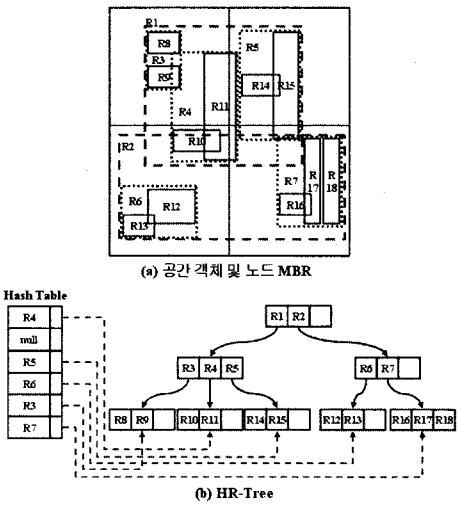


그림 8. HR-Tree의 전체 구조

그림 7(a)은 공간 객체와 노드의 MBR을 나타내며, 이들 MBR 중에서 리프 노드 MBR은 사각형 R3~R7이다. 그림 7(b)는 그림 7(a)에 대한 R-Tree와 R-Tree 리프 노드를 접근하기 위한 해시 테이블을 나타낸다. 해시 테이블의 각 레코드는 R-Tree 리프 노드를 참조하고 있다.

HR-Tree에서 해시 테이블의 갱신은 R-Tree의 삽입 및 삭제 과정과 연결된다. R-Tree에서 공간 객체가 리프 노드에 삽입되거나 삭제될 때, 리프 노드 MBR의 크기 변화가 있는 경우에만 해시 테이블에서 이전 리프 노드 정보를 삭제하고 새로운 리프 노드 정보를 삽입한다. 만일 리프 노드 MBR의 크기 변화가 있으면 해시 테이블의 갱신은 필요하지 않다. 그림 8은 HR-Tree에서 리프 노드 정보를 삭제하는 알고리즘을 나타낸다.

```
Algorithm : Delete(R, LN, PC)
 1: HA ← GetHashAddress(PC);
 2: MBR ← mbr in a record accessed by HA;
 3: If (R contained by MBR) Then
 4:   DeleteLeafNode(R, PTR, LN);
 5: Else
 6:   If (R contained by left partition) Then
 7:     Delete(R, LN, PC*10);
 8:   Else If (R contained by right partition) Then
 9:     Delete(R, LN, PC*10+1);
10:   End If
11: End If
```

그림 9. 삭제 알고리즘

그림 8의 삭제 알고리즘에서 입력 값은 삭제할 R-Tree 리프 노드 MBR인 *R*, 리프 노드 포인터인 *LN*, Partition 코드를 나타내는 *PC*이다. 먼저 GetHashAddress(*PC*)에서 입력된 *PC*를 해싱하여 해시 주소 *HA*를 구하고, 해시 테이블에서 *HA*에 해당하는 레코드의 MBR인 *MBR*을 읽는다. 만일 *MBR*이 *R*을 완전히 포함하면, *R*과 *LN*이 같은 리프 노드를 찾아 삭제한다. 그렇지 않으면, *R*이 포함되는 자식 Partition에서 삭제 알고리즘을 반복 수행한다.

```
Algorithm : Insert(R, LN, PC)
 1: HA ← GetHashAddress(PC);
 2: PTR ← pointer of a record in hash table by HA;
 3: If (two mbrs of child records are null) Then
 4:   AddLeafNode(R, PTR, LN);
 5: Else
 6:   If (R overlaps split line) Then
 7:     AddLeafNode(R, PTR, LN);
 8:   Else If (R contained by left partition) Then
 9:     Insert(R, LN, PC*10);
10:   Else If (R contained by right partition) Then
11:     Insert(R, LN, PC*10+1);
12:   End If
13: End If
```

그림 10. 삽입 알고리즘

그림 9는 HR-Tree에서 리프 노드 정보를 삽입하는 알고리즘을 나타낸다. 그림 9의 삽입 알고리즘에서 입력 값은 삽입할 리프 노드 MBR인 *R*, 리프 노드 포인터인 *LN*, Partition 코드를 나타내는 *PC*이다. 먼저 GetHashAddress(*PC*)에서 입력된 *PC*를 해싱하여 해시 주소 *HA*를 구하고, 해시 테이블에서 *HA*에 해당하는 레코드의 포인터인 *PTR*을 읽는다. 만일 레코드가 리프 Partition에 해당하면, 바로 리프 노드 정보를 저장한다. 그렇지 않고 레코드가 중간 Partition에 해당하면, *R*이 Partition의 분할 선과 서로 겹치는지 검사한다. 만일 겹치면 리프 노드 정보를 저장하고, 그렇지 않으면 자식 Partition에서 삽입 알고리즘을 반복 수행한다.

HR-Tree는 R-Tree와 같이 트리 검색을 수행하지 않고, 해시 테이블을 이용하여 검색을 수행한다. HR-Tree에서 검색 알고리즘은 그림 10과 같다.

```
Algorithm : Search(QW, PC)
 1: RESULT ← Ø;
 2: HA ← GetHashAddress(PC);
 3: MBR ← mbr of a record in hash table by HA;
 4: If (MBR is null) Then
 5:   Return RESULT;
 6: End If
 7: If (MBR overlaps QW) Then
 8:   PTR ← pointer of a record in hash table by HA;
 9:   RESULT ← SearchField(QW, PTR, PC);
10: End If
11: RESULT ← Search(QW, PC*10);
12: RESULT ← Search(QW, PC*10+1);
13: Return RESULT;
```

그림 11. 검색 알고리즘

그림 10의 검색 알고리즘에서 입력 값은 질의 영역인 *QW*, Partition 코드를 나타내는 *PC*이다. 먼저 GetHashAddress(*PC*)에서 입력된 *PC*를 해싱하여 해시 주소 *HA*를 구하고, 해시 테이블에서 *HA*에 해당하는 레코드의 MBR인 *MBR*을 읽는다. 만일 *MBR*과 *QW*가 서로 겹치면 해당 레코드의 R-Tree 리프 노드를 접근한다. 그렇지 않으면 자식 Partition에서 검색 알고리즘을 반복 수행한다.

4. 성능 평가

본 장에서는 대용량 공간 데이타를 가지고 실험을 수행하여 HR-Tree의 검색 성능을 평가한다.

4.1 실험 환경

실험에 사용된 시스템의 하드웨어 사양은 Intel Core 2.4GHz CPU, 2GB RAM이며, 운영체제는 Windows XP를 사용하였다. 실험을 위해 Visual C++ 6.0으로 R-Tree, iQR-Tree, HR-Tree를 구현하였다.

실험에 사용된 공간 데이타는 랜덤 데이타와 실제 데이타를 사용하였다. 랜덤

데이타는 Uniform 분포에서 한 변의 길이가 전체 데이타 영역의 한 변의 길이의 0.01%가 되는 정사각형 10,000~50,000개를 이용하였고, 실제 데이타는 서울시에 존재하는 주택, 상점, 학교 등의 건물 데이타 650,000개를 이용하였다.

4.2 실험 분석 결과

검색 성능 실험에서는 점 질의와 범위 질의를 수행하였으며, 범위 질의는 전체 데이타 영역의 1%가 되는 사각형 영역으로 질의하였다. 성능 비교를 위해 1,000번의 질의를 수행하는 동안 R-Tree, iQR-Tree, HR-Tree의 평균 노드 접근 횟수를 비교하였다. 그림 11은 랜덤 데이타에서 점 질의를 수행한 결과를 나타낸다.

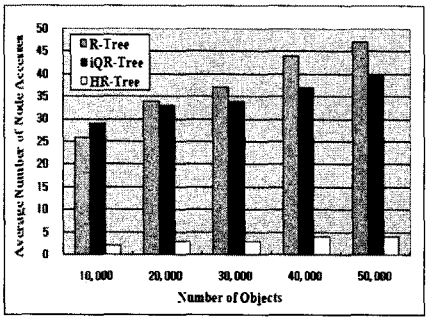


그림 12. 점 질의 결과 (랜덤 데이타)

그림 11와 같이 HR-Tree의 노드 접근 횟수가 가장 적었으며, iQR-Tree 보다 평균 8배, R-Tree 보다 평균 10배가 적었다. 그림 12는 랜덤 데이타에서 범위 질의를 수행한 결과를 나타낸다.

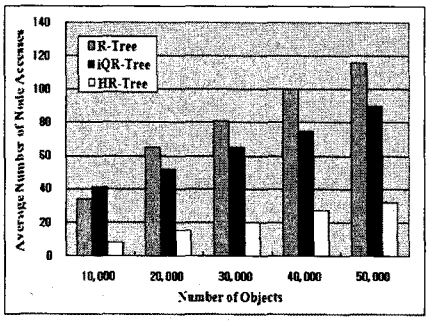


그림 13. 범위 질의 결과 (랜덤 데이타)

그림 12와 같이 점 질의와 비슷하게 HR-Tree의 노드 접근 횟수가 가장 적었으며, iQR-Tree 보다 평균 3배, R-Tree 보다 평균 4배가 적었다. 그림 13은 실제 데이타인 서울시 건물 데이타에서 점 질의와 범위 질의를 수행한 결과를 나타낸다.

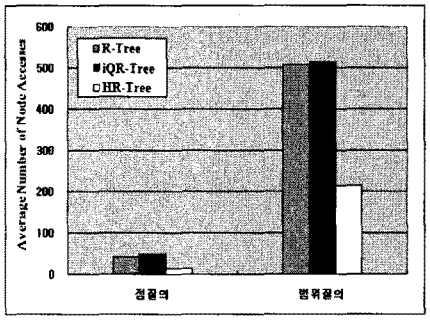


그림 14. 질의 결과 (서울시 건물 데이타)

그림 13과 같이 점 질의와 범위 질의에서 모두 HR-Tree의 노드 접근 횟수가 가장 적었다. 점 질의에서는 HR-Tree의 노드 접근 횟수가 R-Tree 보다 평균 10배, iQR-Tree 보다 평균 8배가 적었으며, 범위 질의에서는 R-Tree, iQR-Tree 보다 평균 2배가 적었다.

5. 결론

본 논문에서는 R-Tree에서 검색시 불필요한 노드 접근을 줄이기 위해 리프 노드를 직접 접근하게 해주는 해시 테이블을 이용하는 인덱스인 HR-Tree를 제안하였다. HR-Tree는 데이타 영역을 차원에 따라 서로 다른 크기를 갖는 여러 Partition으로 분할하고, 각 Partition에 R-Tree 리프 노드를 연계시킴으로써, 검색 영역에 대한 Partition 코드를 해싱하여 리프 노드를 빠르게 접근하도록 하였다.

성능 실험을 통해 HR-Tree의 검색 성능이 기존 인덱스보다 우수함을 입증하였다. 향후 연구는 검색 성능뿐만 아니라 삽입 및 삭제 성능도 고려한 하이브리드 공간 인덱스 구조로 HR-Tree를 확장하는 것이다.

참고문헌

[1] 김형복, "u-GIS 기반의 u-City 구축 방안," 한국GIS학회 공동춘계학술대회, 2008, pp.293-297.

[2] 박종현, "Ubiquitous Sensor Network and u-GIS 기술 특집," 인터넷정보학회지, 9권 1호, 2008, pp.3-3.

[3] Guttman, A., "R-Trees: A Dynamic Index Structure for Spatial Searching," In Proc. of the ACM SIGMOD Int. Conf. on Management of Data, 1984, pp.47-57.

[4] Sellis, T., K., Roussopoulos, N., and Faloutsos, C., "The R^+-Tree: A Dynamic Index for Multi-dimensional Objects," In Proc. of the 13th Int. Conf. on Very Large Data Bases, 1987, pp.507-518.

[5] Beckmann, N., Kriegel, H., P., Schneider, R., and Seeger, B., "The R^*-Tree: An Efficient and Robust Access Method for Points and Rectangles," In Proc. of ACM SIGMOD Int. Conf. on Management of Data, 1990, pp.322-331.

[6] Kamel, I. and Faloutsos, C., "Hilbert R-Tree: An Improved R-tree using Fractals," In Proc. of 20th Int. Conf. on Very Large Data Bases, 1994, pp.500-509.

[7] Berchtold, S., Keim, D. A. and Kriegel, H., "The X-Tree : An Index Structure for High-Dimensional Data," In Proc. of 22th Int. Conf. on Very Large Data Bases, 1996, pp.28-39.

[8] Fu, Y., C., Hu, Z., Y., Guo, W., and Zhou, D., R., "QR-Tree: A Hybrid Spatial Index Structure," In Proc. of the 2nd Int. Conf. on Machine Learning and Cybernetics, 2003, pp.459-463.

[9] Bo, H., and Qiang, W., "A Spatial Indexing Approach for High Performance Location Based Services," The Journal of Navigation, Vol.60, No.1, 2007, pp.83-93.

[10] Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A. N. and Theodoridis, Y., "R-trees Have Grown Everywhere," Submitted to ACM Computing Surveys, http://www.rtreeportal.org/pubs/MNPT03.pdf, 2003.

[11] Kumar, A., "G-Tree: A New Data Structure for Organizing Multidimensional Data," IEEE Transactions on Knowledge and Data Engineering, Vol.6, No.2, 1994, pp.341-347.