고정 그리드를 이용한 이동객체의 위치 색인 기법

이양구, 이응재, 류근호
충북대학교 데이터베이스 연구실
{leeyangkoo, eungjae, khryu}@dblab.chungbuk.ac.kr

An Indexing Method for Location of Moving Objects Using the Fixed Grid

Yang-Koo Lee, Eung-Jae Lee, Keun-Ho Ryu
Database Laboratory, Chungbuk National University, Korea

요 약

최근 무선/이동 통신 기술과 GPS 기술의 발달은 휴대폰을 소지하고 이동하는 사람이나 GPS 수신기를 탑재한 차량과 같은 이동객체의 위치 정보와 관련된 서비스의 제공을 가능하게 하였다. 이러한 환경에서 연속적으로 변경되는 이동객체의 위치 정보는 데이터베이스에 빈번한 갱신 연산을 요구하게 되고, 이는 전체 시스템의 성능을 저하시키는 원인이 된다. 이러한 문제를 해결하기 위하여 R-Tree와 같은 공간색인 구조를 확장하여 갱신 효율을 높이기 위한 연구가 진행되어 왔지만, 시스템의 전체 성능은 오히려 저하되는 문제를 가져왔다.

이 논문에서는 이동객체의 질의 처리 성능뿐만 아니라 객체의 빈번한 위치 갱신을 효율적으로 처리할 수 있는 방법으로 고정 그리드와 R-Tree를 혼합한 형태의 색인 기법을 제안한다. 제안된 색인 기법은 R-Tree에서 색인의 재조직화로 인해 갱신 성능이 저하되는 문제를 해결하기 위하여 셀 기반 색인 기법인 고정 그리드를 이용하여 이동객체의 위치 정보를 저장하고, 고정 그리드에서 객체의 편중 분포로 인한 오버플로 문제를 처리하기 위하여 오버플로가 발생한 각각의 셀들을 R-Tree로 관리한다. 또한, 객체의 밀도가 낮은 셀들을 하나의 버켓으로 공유하여 관리함으로써 저장 공간을 효율적으로 활용한다. 제안된 방법을 다양한 평가 요소를 통해 실험한 결과, 기존의 R-Tree보다 뛰어난 갱신 성능을 보였으며, 질의 처리에 대해서도 성능이 향상되었음을 보였다.

1. 서 론

최근 무선/이동 통신 기술의 발달과 초고속 인터넷의 보급은 휴대폰, 노트북, PDA 등과 같은 무선 통신 장치를 휴대하고 이동하는 모바일 사용자들에게 다양한 응용 서비스의 제공을 가능하게 하였다[1]. 특히, GPS를 이용하여 이동객체의 위치 정보와 관련된 서비스를 제공하는 위치기반서비스[2, 3]는 현재 활발히 연구되고 있는 응용 분야이며 교통 통제 시스템, 항법 시스템, 전장 분석 시스템 등의 분야에 응용되고 있다[4, 5].

기존의 공간 데이터베이스 환경에서, 객체의 위치 정보는 매우 오랫동안 지속되기 때문에 데이터베이스에서 갱신은 드물게 발생하고, 사용자의 질의 요청은 매우 빈번하게 발생된다. 이러한 이유로 인해, 공간 객체를 관리하는 색인 기법들은 주로 사용자의 질의에 대한 빠른 응답 시간에 초점을 맞추어 연구되어 왔다. 그러나 이동객체는 시간이 경과함에 따라 자신의 위치를 연속적으로 변경시키기 때문에 객체의 위치 갱신 요구는 매우 지속적으로 발생되고, 이는 시스템에 심각한 부담을 준다. 특히, R-Tree 기반의 색인 구조에서는 위치 정보의 갱신으로 인한 색인의 재조직화가 빈번하게 발생하며, 이러한 문제는 색인의 갱신 성능을 크게 저하시키는 원인으로 작용한다. 최근에는 LUR-Tree[6], Bottom-Up Update[7] 등과 같이 R-Tree를 단말 노드로부터 루트 노드로 상향 접근함으로써 갱신 성능을 개선하는 방법이 제안되었으나, Tree의 상향 접근을 위한 부가적인 구조를 유지해야 하는 부담과 질의 성능이 오히려 기존의 R-Tree보다 저하되는 문제를 가져왔다.

이 논문에서는 이동객체의 질의 성능뿐만 아니라 갱신 비용을 현저히 줄일 수 있는 방법으로써 고정

이 연구는 과학기술부 • 한국과학재단 지정 청주대학교 정보통신연구센터(ICRC)의 지원에 의한 것입니다.

그리드와 R-Tree를 혼합한 형태의 색인 기법을 제안한다. 제안된 색인 기법은 R-Tree 기반의 색인 구조에서 색인의 재조직화로 인해 갱신 성능이 저하되는 문제를 해결하기 위하여 셀 기반 색인 기법인 고정 그리드를 이용하여 이동객체의 위치 정보를 저장하고, 고정 그리드에서 객체의 편중 분포로 인한 오버플로 문제를 처리하기 위하여 오버플로가 발생한 각각의 셀들을 R-Tree로 관리한다. 또한, 객체의 밀도가 낮은 셀들을 하나의 버켓으로 공유하여 관리함으로써 저장 공간이 효율적으로 활용되도록 한다.

이 논문의 구성은 다음과 같다. 먼저 2장에서는 기존에 제안된 색인 기법들에 대해 소개하고, 문제점을 분석한다. 3장에서는 이 논문에서 제안하는 색인 기법을 자세히 기술하고, 4장에서는 자세한 알고리즘을 기술한다. 5장에서는 기존의 색인과 제안된 색인과의 비교를 통해 성능을 분석하고, 마지막으로, 6장에서 결론 및 향후 연구 과제를 제시한다.

2. 관련 연구

기존의 이동객체의 정보를 다루는 색인 연구들을 살펴보면 크게 이동객체의 과거 이력 정보 및 궤적을 다루는 색인에 대한 연구와 현재 위치 및 가까운 미래 위치를 다루는 색인에 대한 연구로 나눌 수 있다.

전자의 경우에 해당하는 색인 연구로는 3DR-Tree[8], STR-Tree, TB-Tree[9] 등이 있다. 3DR-Tree는 기존의 R-Tree에 시간 차원을 추가하여 3차원으로 표현한 색인으로 시간 관련 질의에는 좋은 성능을 보이지만, 궤적 관련 질의에는 성능이 떨어지는 단점이 있다. STR-Tree와 TB-Tree는 특히, 이동객체의 궤적과 관련된 질의를 빠르게 처리하기 위하여 궤적을 보호하며 색인을 구성함으로써, 궤적 관련 질의에 좋은 성능을 보인다.

후자의 경우에 해당하는 색인 연구로는 대표적인 공간 색인 기법인 R-Tree[R/R+/R*] 등과, R-Tree를 확장시킨 TPR-Tree[1], LUR-Tree, Bottom-Up Update 등이 있다.

R-Tree는 객체의 갱신으로 인해 노드의 분할 및 합병이 빈번하게 발생하고, 이에 따른 색인의 재구성으로 인해 Disk I/O 횟수가 현저히 증가하여 전체 색인의 성능을 크게 악화시킨다. 따라서 R-Tree를 이용해 연속적인 위치 갱신이 발생하는 이동객체의 동적 속성을 관리하기는 매우 어렵다.

TPR-Tree는 이동객체의 갱신 비용을 감소시키기 위하여 객체의 이동을 선형 함수로 표현하는 방법을 이용하였지만, 객체의 이동이 매우 복잡할 경우, 간단한 선형 함수는 객체의 이동을 저장하는데 적합하지 않고, 많은 갱신 연산을 요구하게 된다. 실제로, 객체의 위치를 서술하기에 적합한 선형 함수는 거의 존재하지 않기 때문에, 선형 함수는 객체의 위치 정보를 정확하게 저장할 수 없는 문제를 포함하고 있다.

LUR-Tree와 Bottom-Up Update 등은 R-Tree의 갱신 비용을 감소시키기 위하여 보조 색인 구조를 이용하여 R-Tree의 단말 노드에 직접 접근하고, 상위 노드를 bottom-up 방법으로 접근하여 색인의 갱신을 처리한다. 그러나 이와 같은 bottom-up 방식은 갱신 효율을 위하여 MBR을 확장함으로써, 노드의 겹침을 허용하였기 때문에, 질의 성능은 R-Tree보다 오히려 나빠지는 단점이 있다.

3. 제안된 색인 기법

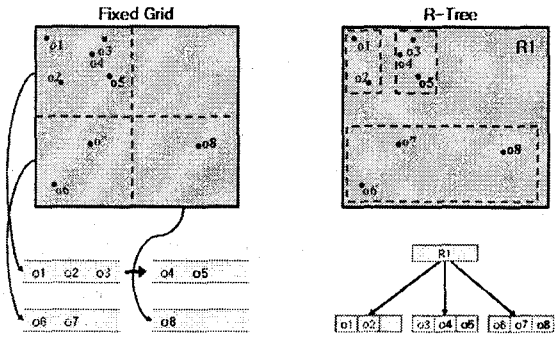
3.1 고정 그리드와 R-Tree

앞에서 언급한 바와 같이, 이동객체 환경에서 색인에 저장되는 객체의 위치 정보는 매우 빈번하게 갱신된다. 따라서 이러한 이동객체의 동적 속성을 R-Tree를 이용하여 관리할 경우, 대부분의 갱신에서 노드의 분할 및 합병으로 인한 색인의 재구성을 유발하고, 이로 인해 갱신 비용은 급격히 증가하게 된다. 특히, 객체의 양이 커질수록 색인의 깊이는 깊어지며, 동시에 색인의 재구성 비용 또한 증가하는 단점이 있다.

이에 반해 고정 그리드는 전체 공간 영역을 일정한 크기의 셀로 분할하고, 각각의 셀에 대응되는 버켓을 할당하여 색인을 구성한다. 이러한 방법은 R-Tree와 같이 노드의 탐색으로 인한 Disk I/O를 요구하지 않으므로 색인의 재구성으로 인한 갱신 성능이 저하되는 문제가 발생하지 않는다. 따라서 색인 구조의 유지가 쉽고, 셀에 대응되는 버켓에 간단한 연산을 통해 직접 접근할 수 있으므로, Disk I/O를 크게 줄일 수 있는 장점이 있다. 그러나 고정 그리드는 이동객체의 분포가 특정 지역에 편중되어 오버플로가 발생하는 경우에 새로운 버켓을 할당하고 연결리스트로 연결시킴으로써 처리되는데, 이런 경우, 연결된 버켓에 대해 순차 탐색을 수행해야 하기 때문에 질의 성능이 저하되는 문제가 발생한다.

<그림 1>은 객체의 분포에 따라 R-Tree와 고정 그리드 색인을 비교한 것이다. 그림에서 고정 그리드를 보면, 객체 o5의 위치는 오버플로가 발생된 셀이고, 객체 o5를 검색하기 위해서는 두개의 버켓을 순차 탐색해야 한다. 그러나 R-Tree에서 객체 o5를 검색할 경우에는 객체 o1,o2가 포함된 노드를 탐색하지 않기 때문에 R-Tree가 고정 그리드보다 효율적이다. 이와 반대로 객체 o8의 경우, 고정 그리드는 한번의 디스크 접근으로 객체 o8의 위치를 찾는 반면, R-Tree는

객체 o8을 검색하기 위하여 루트부터 노드를 탐색하기 때문에 고정 그리드가 R-Tree보다 효율적이다.



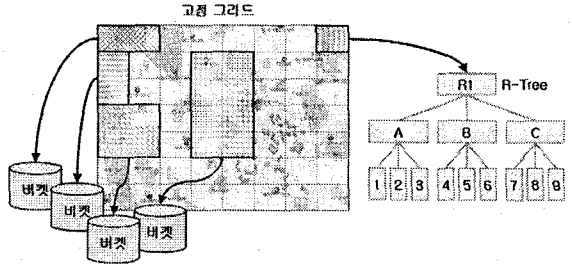
<그림 1> 고정 그리드와 R-Tree의 비교

이와 같은 분석으로 볼 때, R-Tree가 갖는 단점은 고정 그리드의 장점으로 작용하고, 고정 그리드의 단점은 R-Tree의 장점으로 작용됨을 알 수 있다.

3.2 전체 색인 구조

제안된 색인 기법에서 고정 그리드는 초기에 전체 공간 영역을 $N_x \times N_y$ 크기의 셀로 분할한다. 여기서, 초기의 셀의 개수와 크기는 사용자에 의해 미리 정의된 값이다. 따라서 셀 하나의 크기는 ($Domain_x/N_x$, $Domain_y/N_y$)의 값을 갖는다.

기존의 고정 그리드는 모든 셀과 버켓에 대해 1:1로 매핑되어 각각의 셀은 자신의 버켓을 할당받았다. 이런 경우, 객체가 분포되어 있지 않은 셀까지도 버켓이 할당되어 색인의 크기가 증가하는 문제가 있다. 따라서 제안된 색인은 저장 공간을 효율적으로 활용하기 위하여 객체의 밀도가 낮고 서로 인접한 여러 개의 셀들이 하나의 버켓을 공유하도록 한다. 그리고 하나의 셀을 포함하는 버켓이 오버플로일 경우에만 해당하는 셀 영역을 전체 범위로 갖는 R-Tree를 생성한다. 여기서 생성된 R-Tree는 버켓의 순차 탐색을 제거하고, Tree의 깊이는 전체 영역에 대한 R-Tree보다 매우 낮아지기 때문에 탐색해야 하는 노드의 수를 줄일 수 있고, 갱신 성능 또한 향상되는 장점이 있다.

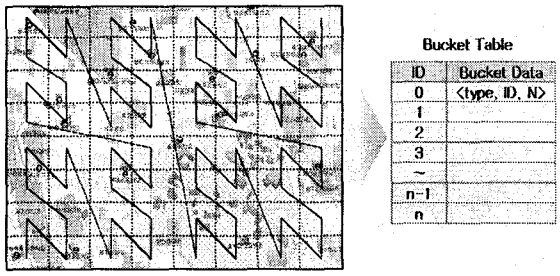


<그림 2> 전체 색인 구조

<그림 2>는 제안된 색인의 전체 구조를 나타내고 있다. 그림에서 보는바와 같이 각각의 버켓은 두 개 이상의 셀들을 관리하고, 오버플로가 발생된 하나의 셀에 대해서만 R-Tree를 구성한다. 만약, 객체의 삽입과 삭제로 인해 버켓에 오버플로나 언더플로 등이 발생하면 버켓은 저장된 셀들이 포함하는 객체의 개수에 따라 분할과 합병 과정을 거치게 된다.

3.3 Cell ID 관리와 버켓의 매핑

Cell ID는 Z-Ordering(Peano curve) 방법으로 각각의 셀을 순회하면서 순차적으로 생성된 고유 번호로 할당된다. 이렇게 셀들이 순서를 갖도록 함으로써 여러 개의 셀 집합을 포함하고 있는 버켓이 분할될 때, 셀들을 적절하게 두 개의 버켓에 순서를 갖도록 분배할 수 있게 된다. <그림 3>은 전체 공간에서 모든 셀들에 대해 Z-Ordering 방법으로 ID를 부여하는 과정을 나타낸 것이다.

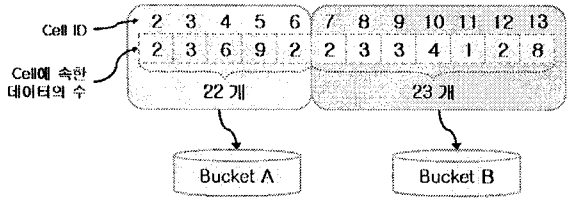


<그림 3> Z-Ordering과 Bucket Table

그림에서 Cell ID를 관리하는 Bucket Table은 Cell ID와 ID에 할당된 Bucket Data로 구성된다. Bucket Table에서 엔트리 속성 type은 셀에 포함된 객체가 버켓에 저장되어 있는지, R-Tree에 저장되어 있는지를 나타내는 상수이다. ID 속성은 셀에 포함된 객체를 저장하고 있는 Disk page ID를 말한다. 마지막으로 N 속성은 셀에 포함된 객체의 개수이다. 이 속성은 버켓에서 오버플로가 발생할 때, 어느 셀을 기준으로 분할하고, 어느 셀들을 통합할 것인가를 결정하기 위하여 사용한다.

객체를 저장하고 있는 각각의 버켓은 객체의 밀도가 작거나 0이고, 순차적으로 연속된 Cell ID를 갖는 셀들의 집합을 공유한다. 만약, 여러 개의 셀들이 공유하고 있는 버켓에 오버플로가 발생했을 경우에는 새로운 버켓을 할당하고, 셀들의 집합을 두개의 버켓에 적절하게 분포하여 저장되도록 한다. <그림 4>는 오버플로가 발생한 버켓을 분할하는 방법을 나타낸다. 그림에서 버켓은 셀의 개수와 상관없이 오버플로인 버켓에 포함된 셀들이 갖고 있는 객체의 개수를

분석하고, 분할 기준이 되는 하나의 셀을 선택하여 두개의 버켓으로 분할한다.



<그림 4> 버켓의 분할

반대로 언더플로가 발생했을 경우에 버켓을 합병하는 방법은 간단하다. 먼저, 언더플로가 발생한 셀을 기준으로 앞과 뒤의 Cell ID를 검사하여, 가장 적은 객체를 포함하고 있는 버켓을 찾아 합병을 수행한다.

4. 알고리즘

4.1. 삽입 알고리즘

<그림 5>는 자세한 삽입 알고리즘을 나타내고 있다. 그림과 같이 먼저, 객체의 삽입이 발생하면 객체의 위치 좌표 x, y를 사용하여 객체가 위치할 Cell ID를 획득하고, Bucket Table을 통해 획득된 Cell ID에 대응하는 Bucket ID를 얻은 후 해당 버켓을 접근한다. 만약, 접근한 Bucket Type이 R-Tree라면, 주어진 셀은 오버플로 상태이고, 하나의 셀 크기를 갖는 R-Tree가 구성되어 있게 된다. 따라서 R-Tree의 삽입 알고리즘에 따라 객체를 삽입한다.

```
Algorithm Insert (e : NewEntry)
Input : NewEntry (새로운 이동객체)
Begin
  cid = Cell ID using location(e)
  bid = Bucket ID in Bucket Table pointed by cid
  bk = ReadBucket(bid)
    if(Bucket Type in bk is RTROOT) then
      InsertRtree(e)
    else
      if(bk is not overflow) then
        InsertData(bk, e)
      else
        if(the same value between startCell
                        and endCell in bk) then
          InsertRtree(e)
        else
          nbk = CreatNewBucket()
          SplitCellSet(bk, nbk)
          UpdateBucketTable(bk, nbk)
        end if
      end if
    end if
end
```

<그림 5> 삽입 알고리즘

만약, Bucket Type이 Bucket이고 오버플로 상태가 아니라면, 해당 버켓에 객체를 삽입한다. 그렇지 않고 Bucket Type이 overflow라면 먼저, 셀과 버켓이 1:1로 매핑되는가를 체크하고, 만약, 매핑된다면 셀에서 오버플로가 발생한 것이기 때문에 새로운 R-Tree를 생성하여 객체를 삽입한다. 그렇지 않으면 버켓은 여러 개의 셀들을 포함하고 있는 것이므로 새로운 버켓을 할당하고 셀 집합을 적절하게 분할한 후, Bucket Table을 갱신한다.

4.2 삭제 알고리즘

객체의 삭제는 <그림 6>과 같이 객체의 위치 좌표 x, y를 사용하여 객체가 위치할 Cell ID를 획득한 후, Bucket Table을 통해 획득된 Cell ID에 대응하는 Bucket ID를 얻고, 해당하는 버켓을 접근한다. 버켓에 접근한 후에는 접근한 Bucket Type에 따라 두 가지의 경우로 객체의 삭제가 발생한다.

```
Algorithm Delete (e : LeafEntry)
Input : LeafEntry (삭제할 이동객체)
Begin
  cid = Cell ID using location(e)
  bid = Bucket ID in Bucket Table pointed by cid
  bk = ReadBucket(bid)
    if(Bucket Type in bk is RTROOT) then
      DeleteRtree(e)
      if(Root is the same Leaf) then
        Change Bucket Type in bk
      end if
    else
      DeleteData(bk, e)
      if(bk is Underflow) then
        Merge bucket Cell IDs in bk
      end if
    end if
end
```

<그림 6> 삭제 알고리즘

먼저, Bucket Type이 RTROOT라면, R-Tree로부터 삭제할 객체를 찾아 삭제한다. 객체를 삭제한 후에 R-Tree가 오직 루트 노드만을 포함하고 있다면, R-Tree는 더 이상 유지되지 않고, Bucket page로 변환하게 된다. 만약, 객체의 삭제 이후에도 여전히 R-Tree를 유지하고 있다면, 기존의 R-Tree의 삭제 알고리즘에 따라 객체를 삭제한다.

또 다른 경우로, Bucket Type이 Bucket이라면 해당하는 버켓에서 객체를 찾아 삭제한다. 만약, 객체를 삭제한 후에 버켓에서 언더플로가 발생하면, 언더플로인 버켓의 셀은 연속적으로 인접해 있는 Cell ID가 속한 버켓으로의 합병을 수행한다. 만약, 객체를 삭제한 후에 언더플로가 발생하지 않으면, 삭제 연산은 종료된다.

4.3 검색 알고리즘

영역 질의를 처리하기 위한 검색 알고리즘은 검색 영역을 포함하는 모든 Cell ID를 확보하고, Bucket Table을 통해 획득된 Cell ID에 대응하는 Bucket ID를 얻은 다음 각각의 모든 Bucket IDs를 검색한다. 그런 후에, 질의 영역을 포함하는 모든 Bucket IDs를 검색할 때까지 각각의 버켓에 접근하고, 만약, Bucket Type이 R-Tree이라면 R-Tree의 검색 알고리즘에 따라 질의 범위에 일치하는 객체를 검색하고, Bucket Type이 Bucket이라면 버켓 안에 있는 객체를 검색함으로써 범위 질의를 처리한다. <그림 7>은 자세한 검색 알고리즘을 기술하고 있다.

```
Algorithm Search (R : Range)
Input : Range (검색 영역)
Output : oIDs (검색된 이동객체 IDs)
Begin
  while(Cell IDs are included Range) do
    cid = Cell ID using location(R)
    bid = Bucket ID in Bucket Table pointed by cid
    bk = ReadBucket(bid)
    if(Bucket Type in bk is RTROOT) then
      result += SearchRtree(R)
    else
      SearchData(R)
    end if
  end while
  return result
end
```

<그림 7> 검색 알고리즘

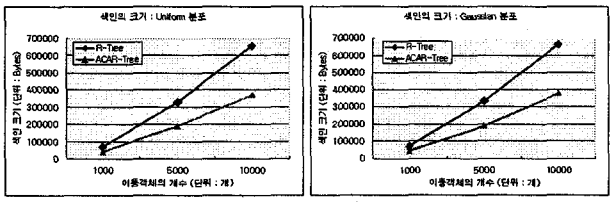
5. 실험 및 평가

5.1 실험 환경

제안된 색인 기법 및 비교 평가된 R-Tree는 C++ 언어를 이용하여 구현하였고, 펜티엄 IV 2.6GHz, RAM 1GB 메모리를 사용하는 MS Windows Server 2003 시스템에서 실험 평가하였다. 실험에 사용된 데이터는 이동객체 환경에서 모의실험 데이터로 많이 사용되고 있는 GSTD[10]를 이용하였고, 이동객체의 다양한 위치 변화에 대한 성능을 평가하기 위하여 Uniform, Gaussion 분포를 갖는 실험 데이터를 생성하였다. 또한 이동객체 수의 변화에 따른 색인의 성능 변화를 확인하기 위하여 1000, 5000, 10000개의 객체를 생성하여 실험하였다.

5.2 실험 결과

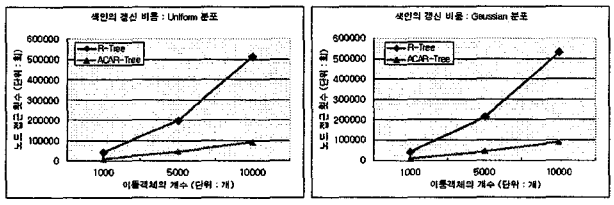
<그림 8>에서 보는 바와 같이 색인 크기의 경우는 제안된 방법이 R-Tree보다 약 40% 정도 색인의 크기가 감소되었음을 보였다. 이는 고정 그리드에서 버켓에 저장되는 객체의 정보는 점 데이터 형태로 저장되지만, R-Tree의 경우는 MBR로 저장되기 때문에 동일한 버켓 크기에서 저장될 수 있는 데이터의 수가 R-Tree가 더 적어지기 때문이다.



<그림 8> 데이터 분포별 색인의 크기

갱신 연산은 객체의 이전 위치의 삭제 연산과 새로운 위치의 삽입 연산을 통해 접근한 Disk I/O 횟수의 합을 비교함으로써 평가된다. 실험 결과, 제안된 방법이 R-Tree보다 뛰어난 갱신 성능을 보였다.

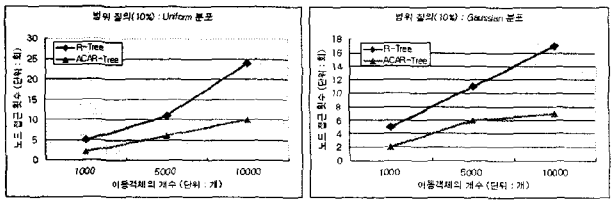
갱신 연산에서 R-Tree는 루트 노드부터 노드를 탐색하기 때문에 색인의 크기가 클수록 탐색해야 하는 노드의 수가 많아지고, 변경된 노드 정보로 인한 색인의 재조직화가 빈번하게 발생한다. 따라서 Disk I/O 횟수는 현저히 증가하게 된다. 반면에, 제안된 방법은 고정 그리드를 통해 객체의 위치에 직접 접근하므로 노드의 탐색 시간을 제거할 수 있고, 버켓에 오버플로가 발생하여 R-Tree로 관리된다고 하더라도, 셀에 생성된 R-Tree의 크기는 매우 작다. 따라서 전체 색인의 구축 시간은 R-Tree보다 현저히 줄어드는 것을 알 수 있다. <그림 9>는 데이터 분포별 색인의 갱신 성능을 비교한 결과를 나타낸다.



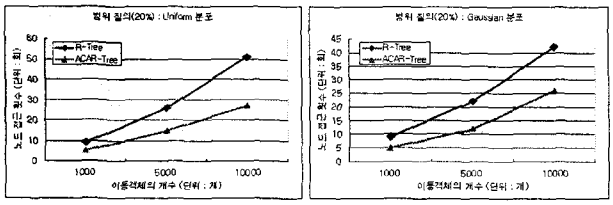
<그림 9> 데이터 분포별 색인의 갱신 성능

질의 성능의 경우는 전체 공간 영역에 대하여 각각 10%, 20%의 영역을 데이터 분포에 따라 임의로 생성하여 실험하였다. 질의 성능 역시 각각의 실험에서 제안된 방법이 R-Tree보다 좋은 성능을 보였다. 그 이유는 제안된 방법의 경우, 고정 그리드를 통해 R-Tree가 객체를 포함하는 단말 노드를 검색하기 위하여 루트 노드부터 노드를 탐색하는 과정이 제거되거나, 매우 작기 때문인 것으로 예상된다. 특히, R-Tree의 경우에 색인 내에 오버랩 되는 데이터가 많아질 경우, 질의 처리를 위하여 방문해야할 노드가 증가하기 때문에 질의 성능은 크게 저하된다. <그림

10>과 <그림 11>은 영역 질의 범위가 각각 10%, 20%일 때의 평가 결과를 나타낸다.



<그림 10> 데이터 분포별 영역 질의 성능 (10%)



<그림 11> 데이터 분포별 영역 질의 성능 (20%)

6. 결론 및 향후 연구

모바일 사용자에게 위치 정보와 관련된 다양한 서비스의 제공을 가능하게 하기 위해서는 이동객체의 연속적인 위치 변경을 수용할 수 있는 저장 공간과 새로운 색인 기술이 필요하다. 특히, 모바일 환경에서 객체의 빈번한 위치 이동은 시스템에 잦은 갱신을 요구하게 되며, 이는 전체 시스템의 성능을 저하시키는 문제가 된다.

이 논문에서는 이러한 이동객체의 빈번한 위치 갱신 요구를 효율적으로 처리하기 위한 구조로써 고정 그리드와 R-Tree 구조를 이용한 색인 방법을 제안하였다. 제안된 방법은 이동객체의 위치를 빠르게 갱신하기 위하여 고정 그리드를 이용하여 셀 단위로 객체 정보를 저장하고, 오버플로가 발생한 셀에 대해서만 R-Tree로 관리함으로써 색인의 갱신 효율을 높였고, 다양한 데이터 분포에도 적용할 수 있도록 하였다.

또한 여러 개의 셀들이 하나의 버켓을 공유할 수 있도록 함으로써, 저장 공간의 효율성과 더불어 데이터 밀도가 낮은 지역에서의 정보를 효율적으로 관리하도록 하였고, 다양한 실험 및 분석을 통하여 제안된 방법이 기존의 R-Tree보다 갱신 성능과 질의 처리 성능이 현저히 향상되었음을 보였다.

향후 연구로는 실세계의 데이터를 사용해서 기존에 존재하는 다양한 이동객체 색인 기법과의 평가를 통한 성능 분석이 필요하다. 또한 다른 형태의 Cell ID 부여 방법을 통하여, 버켓을 공유하는 셀들의 클러스터링과 색인의 성능과의 상관관계에 대한 연구도 진행할 예정이다.

[참 고 문 헌]

[1] S. Saltenis, C. Jensen, S. Leutenegger and M. Lopez. "Indexing the Positions of Continuously Moving Objects," In Proc. of the 19th ACM-SIGMOD Conference, Dallas, Texas, pp.331-342, 2000.

[2] E. Pitoura and G. Samaras, "Locating Objects in Mobile Computing," IEEE Transactions on Knowledge and Data Engineering. Vol. 13, No. 4, pp. 571-592, 2001.

[3] M. Erwig, R. H. Guting, M. Schneider and M. Vazirgiannis, "Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Object in Databases," CHOROCHRONOS Technical Report CH-97-08, December, 1997.

[4] R. H. Guting, M. H. Bohlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, M. Vazirgiannis, "A foundation for representing and querying moving objects," ACM Transactions on Database Systems, Vol.25, pp. 1-42, No.1, 2000.

[5] 류근호, 안윤애, 이준욱, 이용준, "이동객체 데이터베이스와 위치기반서비스의 적용," 데이터베이스연구회지, Vol.17, pp. 57-74, No.03, 2001.

[6] D. Kwon, S. Lee, and S. Lee, "Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree," In Mobile Data Management, MDM, pp.113-120, 2002.

[7] M. Lee, W. Hsu, C. jensen, B. Cui, and K. Teo, "Supporting Frequent Updates in R-Trees: A Bottom-Up Approach," In Proceedings of the 29th International Conference on VLDB, September. 2003.

[8] Y. Theodoridis, M. Vazirgiannis, T. Sellis, "SpatioTemporal Composition and Indexing for Large Multimedia Applications," ACM Multimedia Systems, pp. 284-298, 1998.

[9] D. Pfoser, C. S. Jensen, and Y. Theodoridis, "Novel Approaches in Query Processing for Moving Objects," CHOROCHRONOS Technical Report CH-00-3, February, 2000.

[10] Y. Theodoridis, Jefferson R. O. Silva, and Mario A. Nascimento, "On the Generation of Spatiotemporal Datasets," In Proceedings of the 6th Int'l Symposium on Large Spatial Databases (SSD'99), Hong Kong, China, July 1999. Springer-Verlag LNCS Series.