이동체의 현재 위치 색인을 위한 동적 해쉉 구조의 설계 및 구현

전봉기*

Design and Implementation of the dynamic hashing structure for indexing the current positions of moving objects

Bong-Gi Jun*

요 약

위치 기반 서비스는 이동체의 위치에 종속적인 결과를 얻는 위치 기반 질의를 필요로 한다. 이동체의 위치는 연속적으로 변하기 때문에, 이동체의 색인은 변경된 위치 정보를 유지하기 위하여 빈번한 갱신 연산을 수행해야 한다. 기존의 공간 색인들(그리드 파일, R-트리, KDB-트리 등)은 정적 데이터를 검색하는데 효과적인 색인이다. 이들 색인은 연속적으로 위치 데이터가 변경되는 이동체 데이터베이스의 색인으로는 적합하지 않다.

본 논문에서는 삽입/삭제 비용이 적은 동적 해쉉 색인을 제안한다. 동적 해쉉 색인 구조는 해쉬와 트리를 결합한 동적 해쉉 기술을 공간 색인에 적용한 것이다. 실험 결과에서 동적 해쉉 색인은 R*-tree와 고정 그리드 보다 성능이 우수하였다.

ABSTRACT

Location-Based Services(LBS) give rise to location-dependent queries of which results depend on the positions of moving objects. Because positions of moving objects change continuously, indexes of moving object must perform update operations frequently for keeping the changed position information. Existing spatial index (Grid File, R-Tree, KDB-tree etc.) proposed as index structure to search static data effectively. There are not suitable for index technique of moving object database that position data is changed continuously.

In this paper, I propose a dynamic hashing index that insertion/delete costs are low. The dynamic hashing structure is that apply dynamic hashing techniques to combine a hash and a tree to a spatial index. The results of my extensive experiments show the dynamic hashing index outperforms the R*-tree and the fixed grid.

키워드

Moving objects databases, Spatio-temporal indexing, Hash, GIS

Ⅰ. 서 론

위치 기반 서비스(LBS: Location Based Service)는 무선 통신 서비스의 핵심 응용으로 대두되고 있다. 이동체의 이동과 관련된 위치 기반 질의를 처리하기 위해서는 이동체의 위치 데이터를 이동체 데이터베이스(Moving Objects Databases)에 저장하고 관리해야 한다. 연속적으로 이동하는 이동체

*신라대학교 컴퓨터정보공학부

접수일자 : 2004. 09. 17

의 위치를 검색하는 것은 위치 기반 서비스에서는 중요한 응용 중의 하나이다[1].

이동체는 공간적으로 두 가지 특징이 있다. 첫째, 연속적으로 이동하기 때문에 빈번한 갱신 연산이 수행된다. 위치 변경은 색인 구조의 변경을 초래하기 때문에 색인 구조가 변하지 않는 색인 구조가 적합하다. 둘째, 이동체를 차량이나 사람이라고 본다면 도심이나 교차로 등에서 밀집되는 특징이 있다. 밀집화 현상은 색인의 성능을 저하시키는 요인이 된다.

이동체를 효율적으로 검색하기 위해서는 공간 색인 구조가 필요하다. 공간 색인 구조는 해쉬 기반의 그리드 구조와 트리 기반의 R-Tree[2]가 대표적인 색인 구조라 할 수 있다. 트리 기반에는 Quad-Tree[3], K-D-B Tree[4] 등도 있으나, 균형 트리(Balanced Tree) 구조가 아니기 때문에 밀집 지역에서 성능 저하를 초래한다. R-Tree는 균형 트리 구조를 가지고 있지만 중간 노드에서 하위노드의 포함하는 최소 경계 사각형(MBR: Minimum Boundary Rectangle)을 유지하기 때문에 이동 객체의 위치 변경이 이동체의 방향성으로 인하여 MBR의 확장을 초래하여 중복 검색의 원인이 된다[5].

해쉬 기반의 공간 색인에서 밀집 지역에 이동체가 집중되면 색인의 버켓에 오버플로우가 발생한다. 버켓의 오버플로우는 색인에서 버켓 분할을 초래하여 색인의 성능을 저하시킨다. 이 논문에서는 기존의 해쉬 기법에서 사용하는 오버플로우 처리 기법을 공간 색인에 적용하여 새로운 동적 해쉬 색인 방법을 제안한다.

이 논문의 구성은 다음과 같다. 2장에서는 이동체에 관한 관련 연구를 기술하고, 3장에서는 빈번한 위치 변경에 따른 균형 트리 색인의 문제점과 이동체의 밀집화 문제에 대해 기술한다. 4장에서는 빈번한 갱신 연산을 효과적으로 처리할 수 있는 동적 해쉬 색인 구조와 알고리즘들을 제시한다. 5장에서는 실험을 수행하여 제안하는 동적 해쉬 색인의 성능을 비교하였다. 마지막으로 6장에서는 결론과 향후 과제를 기술한다.

II. 관련 연구

기존의 데이터베이스 기술에서는 저장 객체는 고정된 값을 저장하는데 목적을 두고 있다고 한다면 이동체는 변화하고 있는 값을 저장한다. 이동체에 관한 연구는 이동 단말기의 위치 정보 전송에 관한 연구와 이동체의 저장 방법에 관한 연구로 나뉘어지며, 정확한 위치 정보를 저장하기 위한 효율적인 이동 객체의 색인 구조에 관한 연구가 활발히 진행 중에 있다[5, 6, 7, 8].

이동체의 현재 및 미래 위치 색인에 관한 연구는 이동체가 시간이 지남에 따라 연속적으로 이동하기 때문에, 색인의 연속적인 색인의 갱신 연산을 최소화하는 색인 구조에 관한 연구와 이동중인 이동체의 현재 또는 미래의 위치를 예측하는 것을 목적으로 하는 색인 연구로 구분되어 질 수 있다.

LUR-tree(Lazy Update R-tree)[9]는 다차원 색인인 R-tree을 이용한 이동체 색인으로 갱신 비용을 줄이는 것을 목적으로 한다. Lazy 갱신은 이동체의 위치 변경으로 발생하는 갱신 연산을 노드 내에서의 갱신으로 제한함으로써 노드의 구조 변경에 따른 비용을 감소시켰다.

TPR-tree[5]는 이동체의 위치를 시간에 대한 선형 함수로 표현하여 현재위치 및 미래위치에 대한 질의를 가능하게 한 구조이다. TPR-tree는 이동체의 위치가 갱신될 때 시간 경계 간격을 최소화하였으며, 이동체의 속도와 방향을 시간에 대한 함수의 매개 변수로 사용하여 R*-tree에서 색인화하였다. 그리고, 이동체의 방향과 속도가 특정 임계값 이상 변화하지 않으면 무시함으로써 갱신 횟수를 줄였다.

LUR-tree와 TPR-tree는 이동체의 위치 변경을 단말 노드내의 이동으로 제한하여 갱신 연산으로 발생하는 색인의 구조 변경을 최소화하였으나, 이는 단말 노드의 최소경계사각형의 확대로 인하여 검색 성능이 저하되고, 임계치 이후의 색인 변경에 따른 비용이 발생한다.

해쉬 기반 색인은 셀 내의 이동에서 색인 구조의 변경이 없다는 특징을 이용한다.[8, 13] 하지만 해쉬 기반 색인은 균형 트리가 아니기 때문에, 이동체는 연속적으로 위치가 변하기 때문에 시간에 따라 분포성이 변하는 문제를 고려해야 한다. 관련 연구[8]에서는 Quad-tree를 색인으로 사용하기 때문에 비균등 분포에서 경사 트리(Skewed tree)가 될 수 있다. 특히, 이동체는 특정시간에 도심지, 교차로에서 밀집될 수 있다. 해쉬 기반 인덱스에서 이동체의 밀집화는 셀 버켓(bucket)의 오버플로우를 유발하여 색인의 성능을 저하시킨다.

III. 문제 정의

3.1 위치 변경에 따른 균형 트리의 문제점

이동체의 위치 변경으로 인한 빈번한 색인의 변경 문제를 해쉬(Hashing) 함수를 기반으로 접근한

이동체 색인 연구[8, 13]가 있다. 이동체의 위치 변경은 기존 공간 객체의 변경으로 볼 수 있다. 기존의 공간 색인은 대부분 갱신 연산보다는 검색 연산을 중점적으로 고려하였다. 그러나 이동체 데이터베이스에서는 빈번한 갱신 연산을 고려한 색인 구조가 필요하다. 공간 객체의 위치 변경은 트리 기반의 색인구조에서 삭제 후 재 삽입 연산으로 이루어진다. 삭제와 재 삽입 연산은 색인에서 병합과 분할을 초래하기 때문에 위치 변경이 빈번한 이동체 색인에서는 적합하지 않다.

해쉬 기반의 색인 구조에서 셀 내에서 위치를 변경할 경우, 색인 구조는 변경할 필요가 없다.[13] 이동체의 위치가 다른 셀로 이동할 경우, 이동체의 위치 정보는 갱신이 되어야 한다. 하지만 해쉬 기반의 색인 구조에서는 삭제와 재 삽입 연산 비용이 트리 계열보다 적은 장점이 있다.

3.2 이동체의 분포에 따른 해쉬 기반 색인의 문제점

이동체는 연속적으로 위치가 변화하기 때문에 시간에 따라 분포성이 변한다. 특히, 특정시간에 도심지, 교차로에서 밀집될 수 있다. 해쉬 기반 색인에서는 이동체의 밀집화로 인하여 셀 버켓(bucket)이 오버플로우가 발생한다. 이 논문에서 고려하는 해쉬 방법은 버켓을 사용하는 해쉬 방법이다. 버켓(bucket)을 사용하는 해쉬 방법에서 오버플로우를 처리하는 방법은 별도의 오버플로우 영역(Separate Overflow Area)을 사용하는 방법, 확장 해쉉 (Extensible hashing) 방법, 동적 해쉉 (Dynamic Hashing) 방법이 있다.

확장 해쉉 방법은 다른 방법에 비해 색인 변경 비용이 크다는 문제점이 있다. 이동체의 이동으로 인한 색인 변경시에 셀의 분할 또는 병합시에 색인 테이블 전체를 확장/축소해야 하기 때문이다.

IV. 동적 해쉉 색인 구조

이 논문에서는 기존의 고정 그리드를 사용하는 방법에서 발생하는 오버플로우 버켓의 순차검색 문제를 해결하기 위하여 새로운 동적 해쉉 색인을 제안한다. 이 장에서는 제안하는 동적 해쉉 색인의 자료구조와 알고리즘들을 소개한다.

4.1 자료 구조

제안하는 동적 해쉉 색인은 그림 1(b)와 같이 1차원 색인에서의 전통적인 동적 해쉉과 같이 1차 주소를 얻기 위해 해쉬 함수를 사용하고, 오버플로우 버켓들을 트리 구조의 색인을 사용하여 색인화 하였다. 오버플로우 버켓은 Z-ordering 방법을 사용하여 분할번호를 부여한다.

동적 해쉉 색인에서 공간은 1차적으로 고정 그리드로 나뉘어 진다. 이동체의 삽입 또는 이동으로 인하여 고정 그리드의 셀이 오버플로우가 발생하면 이 셀은 이진 분할된다. 이진 분할된 결과는 고유한 분할번호가 부여된다. 그림 1에서 오버플로우가 발생하여 4번의 분할이 발생한 셀의 예와 분할번호가 저장된 구조를 보인다.

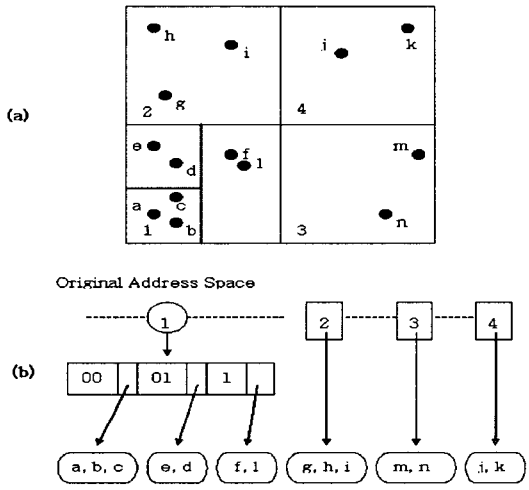


그림 1. 동적 해쉉 색인의 예. (a) 지도 상의 분할 예, (b) 동적 해쉉 색인의 구조

Fig. 1. An example of a dynamic hashing index. (a) An example of the split on a map, (b) Structure of a dynamic hashing index.

분할번호는 분할횟수만큼의 이진수를 가지며, 수직분할에서 왼쪽은 0값을 오른쪽은 1값을 가진다. 수평분할에서 아래쪽은 0값을 위쪽은 1값을 가진다.

4.2 검색 알고리즘

그림 1(a)에서 객체 a를 검색하고자 한다. 우선 해쉬 함수를 사용하여 객체 a가 그리드 1번에 속한다는 것을 구한다. 그리드 1번은 오버플로우 그리드이므로 객체 a를 검색하기 위해서는 객체 a의 Z-ordering 번호를 구하여야 한다.

그림 1에서 오버플로우 그리드의 분할횟수는 2이다. 2차원 셀에서 분할은 수직, 수평으로 순차적으로 진행되기 때문에 객체 a의 Z-ordering 번호는

00이다. 00 값을 이용하여 그림 1(b)의 트리를 검색하여 해당 객체를 검색할 수 있다.

```
Algorithm assign(X1, X2, b) {
        // b is most significant bits
  P = 0;                        // zero
  one_bit = 0x01 << MAX_BITS -1;
        // first bit is one
  for ( k=1; k<=b; k++ )        // repeat b times
  {
     i = k mod n;  // n-dimensional space
     if ( Xi < (Li + Hi ) / 2)
        // range of i-dimension is [Li, Hi]
        Hi = (Li + Hi) / 2;
     else  {
        P = P | one_bit;
        Li = (Li + Hi) / 2;
     }
     one_bit = one_bit >> 1;
  }
  return (P);
}
```

그림 2. Z-Ordering 번호를 구하는 알고리즘.
Fig. 2. Z-Ordering Numbering Algorithm.

오버플로우 셀에서 Z-ordering 번호를 구하는 알고리즘은 그림 2와 같다. 검색 알고리즘에서 X1, X2는 객체의 공간키 값이며, P는 노드내의 포인터 값이다. b는 해당 오버플로우 그리드의 분할 횟수이다.

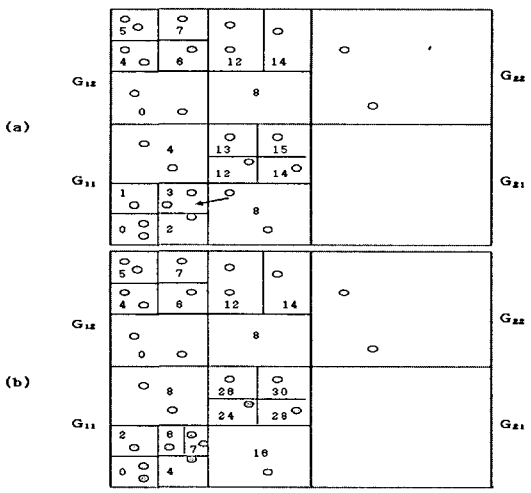


그림 3. 이동체의 이동으로 인한 셀 분할의 예. (a) 분할 전의 예, (b) 분할 후의 예
Fig. 3. An example of splitting cells due to the movement of moving objects. (a) An example before split. (b) An example after split

4.3 삽입 알고리즘

그림 3에서 G11, G12, G21, G22는 고정 그리드의 그리드 번호이다. 그림 3(a)에서 이동체가 그리드 G11내의 셀 8번에서 셀 3번으로 이동하게 되면 셀 3번은 오버플로우가 발생한다. 예에서 셀의 최대허용개수는 2이다.

새로운 이동체의 삽입으로 셀이 오버플로우가 발생하면 셀은 그림 3(b)와 같이 2개의 셀로 분할된다. 새로운 셀 분할로 인하여 그리드 G11에서 분할횟수는 4에서 5로 증가하게 된다. 분할 횟수의 증가는 셀 번호를 2배로 증가시키는 결과가 된다. 하지만 분할 횟수의 증가는 논리적인 값의 변화로 트리의 키값들은 변하지 않는다.

```
Algorithm Insert(X, Y, b)  {
   P_in = assign(X, Y, b);
   P_act = search(P_in);
   if ( num_points(P_act) < max_entries )
         insert (X, Y) into partition P_act;
   else  {
         P0 = split_partition(P_act);
         Insert (X, Y, size(P0));
   }
}
```

그림 4. 트리의 삽입 알고리즘.
Fig. 4. Algorithm for inserting in the tree.

그림 4는 트리의 삽입 알고리즘이다. 그림 5에서와 같이 분할이 발생하면 이전 키 값을 삭제하고 분할된 새로운 2개의 키값을 삽입해야 한다. X와 Y는 좌표값이며, assign 함수는 Z-ordering 번호를 구하는 함수이다. max_entries는 셀의 최대허용개수이다.

```
Algorithm split_partition(P_act)  {
   P1 = P1 | FIRST_BIT >> size(P_act);
   reallocate all points in P_act to P-act and P1;
   if ( num_points(P1) > 0 )
      insert P1 into tree;
            // if P1 is a nonempty partition
   else
      split_partition(P_act);
   if ( num_points(P_act) == 0 )  {
      delete P_act from tree;
      split_partition(P1);
   }
}
```

그림 5. 오버플로우가 발생한 셀의 이진 분할 알고리즘.
Fig. 5. The binary splitting algorithm of the overflow cell.

P_in는 삽입 객체의 Z-ordering 번호이며, P_act

는 실제 포함하는 셀의 분할번호이다. search 함수는 트리에서 객체의 Z-ordering 번호를 이용하여 객체를 포함하고 있는 셀을 찾는 함수이다.

트리에서 오버플로우가 발생하면 이진 분할을 수행한다. 이진 분할은 공간적으로 같은 크기로 2개의 영역으로 나누어지는 것을 의미한다. 나누어진 영역에 속하는 이동체가 없는 경우에는 트리에 삽입되지 않고, 다시 이진 분할을 수행한다. 그림 5는 오버플로우가 발생한 셀의 이진 분할 알고리즘이다.

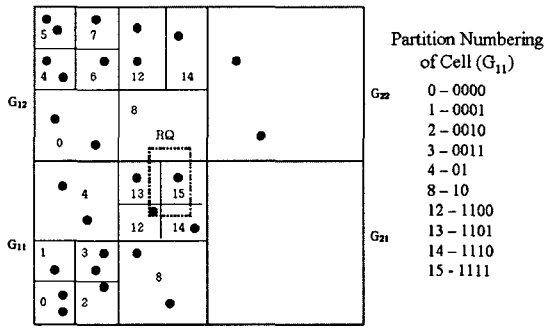


그림 6. 영역 질의의 예.
Fig. 6. An example of the range query.

4.4 영역 질의 알고리즘

이 절에서는 동적 해쉬 색인의 영역 질의 처리 방법에 대해 소개한다. 동적 해쉬 색인은 4.1절에서 소개한 것과 같이 해쉬 함수를 이용하여 1차적인 주소를 번역하고, 오버플로우 셀은 트리 탐색을 수행해야 한다. 트리의 하단은 분할 번호가 정렬되어 있어야 한다.

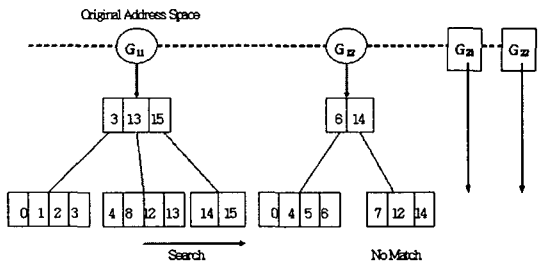


그림 7. 영역 질의 처리를 위한 탐색의 예
Fig. 7. An example of the search for processing the range query.

그림 6은 영역 질의의 예를 보이고 있다. 그림 6에서 질의 RQ는 그리드 G11과 G12와 중첩된다. 해쉬 함수를 이용하여 질의 영역과 중첩되는 그리드를 구할 수 있다. 그리드 G11과 G12는 모두 오버플로우 그리드이기 때문에 트리 탐색을 통하여 영역 질의 내의 이동체를 찾아야 한다.

먼저 그리드 G11에서 영역 질의 RQ의 좌측 하단 점과 우측 상단 점의 분할 번호를 구한다. 분할 번호는 12와 15이다. 그림 7에서와 같이 분할 번호는 트리의 단말 노드에서 정렬되어 있기 때문에 12와 15 사이의 모든 분할 번호를 탐색한다. 그림 7에서 그리드 G11내의 분할 번호 12,13,14,15의 페이지가 영역 질의 RQ와 중첩 또는 포함되는 페이지이다.

```
Algorithm range_query(x1, y1, x2, y2) {
  Hlo = grid_cell_number(x1, y1);
  Hhi = grid_cell_number(x2, y2);
  for(i=Hlo.x; i<=Hhi.x; i++)
    for(j=Hlo.y; i<=Hhi.y; i++)
      if( DIR[i,j].tag == GRID ) {
        read page DIR[i,j].PageID;
        examine each point in the page and
        output it if it satisfies query;
      }
      else
        region_query_tree(x1, y1, x2, y2,
        DIR[i,j].PageID);
}

Algorithm region_query_tree(x1, y1, x2, y2,
page_id)
{
  b = read_tree(page_id);
  Plo = assign(x1, y1, b);
  Phi= assign(x2, y2, b);
  Rlo = Rank(Plo, low);  Rhi = Rank(Phi, high);
  while ( Plo <= P <= Phi ) {
    Tlo = Rank(P,low);   Thi = Rank(P,high);
    if(Rlo.x < Tlo.x AND Thi.x < Rhi.x AND
        Rlo.y < Tlo.y AND Thi.y < Rhi.y) {
        // P is contained in query region
      output(all points in P);
    else  if (Rlo.x >= Tlo.x AND Thi.x >= Rhi.x
        AND Rlo.y >= Tlo.y AND Thi.y >= Rhi.y)
        // P overlaps query region
      examine each point in P and output it if it
      satisfies query;
    }
    P = Pnext          // next partition in tree
  }
}
```

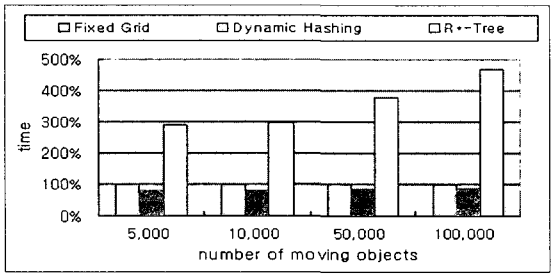
그림 8. 영역 질의 알고리즘.
Fig. 8. Algorithm of the range query.

그림 8은 동적 해쉬 색인의 영역 질의 처리 알고리즘이다. 함수 range_query에서 1차적으로 해쉬 함수를 사용하여 영역 질의와 중첩되는 그리드들

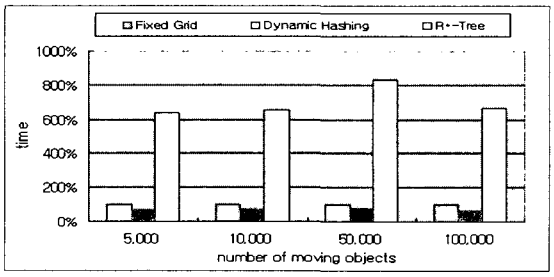
을 찾는다. 이들 그리드 중에서 오버플로우 그리드들은 함수 region_query_tree를 호출하여 트리 탐색을 수행한다.

함수 region_query_tree에서 b는 해당 오버플로우 그리드의 분할 횟수이다. 함수 Rank는 주어진 분할 번호에 해당되는 트리의 조각 셀의 좌측 하단 또는 우측 상단의 점 좌표를 구하는 함수이다. Rlo와 Rhi는 영역 질의 RQ의 좌측 하단 점과 우측 상단 점이며 Tlo와 Thi는 각 셀의 좌측 하단 점과 우측 상단 점이다.

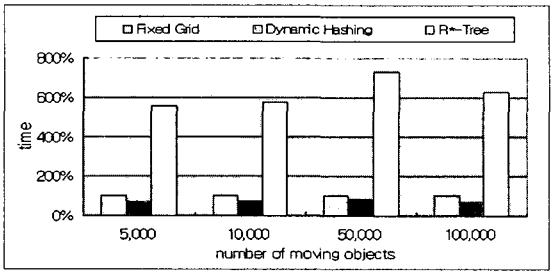
V. 실 험



(a)



(b)



(c)

그림 9. 이동체의 개수에 따른 성능 비교. (a) 초기 색인 구축 비용, (b) 색인 변경 비용 (c) 초기 색인 구축 및 색인 변경 비용의 합

Fig. 9. Comparison of performance versus the number of objects. (a) the cost for initial building index, (b) the cost of updating index, (c) the total cost for initial building & updating index

이 장에서는 논문에서 제안하는 동적 해쉬 색인의 성능을 평가하기 위해 기존의 공간 색인 중에서 고정 그리드[10], R*-tree[11]와 성능을 비교하고자 한다. 실험은 정규분포의 이동체를 자유 이동시켜 색인 구축 비용과 색인 변경 비용을 비교하였다.

현재까지 실제적인 이동체 데이터 집합에 대한 공개된 자료가 존재하지 않기 때문에, 이동체 생성에 대한 대표적인 도구인 GSTD 알고리즘[12]을 사용하여 실험 데이터를 생성하였다. 동적 해쉬 색인, 고정 그리드, R*-tree는 각각 C 언어로 구현하였다.

그림 9는 이동체의 수를 5000에서 10만까지 달리하여 이동체의 이동에 따른 색인의 성능을 비교하였다. R*-tree는 색인의 빈번한 삽입 삭제 연산으로 인하여 해쉬 기반의 공간 분할 방식의 색인에 비해 상대적으로 연산비용이 높았다. 제안하는 동적 해쉬 색인은 색인 구축 비용은 고정 그리드에 비해 10%-20% 성능 향상이 있었다. 나머지 실험에서는 20%-30% 성능 향상이 있었다.

VI. 결 론

이 논문에서는 이동체의 현재 위치를 저장하는 해쉬 기반의 새로운 동적 공간 색인을 제안하였다. 이동체의 위치 정보를 저장한 이동체 색인에서 빈번한 위치 변경으로 인한 균형 공간 트리의 문제점과 이동체의 분포 변경에 따른 해쉬 기법의 오버플로우 처리 문제를 설명하였다.

제안하는 동적 해쉬 색인에서는 이동체의 현재 위치를 2차원의 점으로 정의하고, 빈번한 이동체의 위치 변경으로 발생하는 색인의 갱신 연산 비용을 해쉬 기법을 사용하여 줄였으며, 해쉬 기법에서 발생하는 오버플로우 처리를 새로운 트리 구조와 알고리즘을 제시하여 갱신 성능을 향상시켰다. 실험 평가에서 기존의 고정 그리드에 비해 20% 이상의 성능 향상이 있었다.

향후 다양한 실험 데이터를 이용한 성능 평가를 진행할 계획이며, 이동체의 수와 고정 그리드 셀의 총 개수가 성능이 미치는 비용 모델을 연구하고자 한다.

참고문헌

[1] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha, "Updating and querying databases that track mobile units," Distributed and Parallel Databases, vol. 7, no. 3, pp. 257-387, 1999.

[2] A. Guttman, "R-trees: A dynamic index structure for spatial searching," Proc. of the ACM SIGMOD Int'l Conf. on Management of Data, pp. 47-54, 1984.

[3] H. Samet, The Design and Analysis of Spatial Data Structures, Addision-Wesley, Reading, MA, 1990.

[4] J. T. Robinson, "The K-D-B-tree: A search structure for large multidimensional dynamic indexes," Proc. of the ACM SIGMOD Int'l Conf. on Management of Data, pp. 10-18, 1981.

[5] S. Saltenis, C. S. Jensen, S.T. Leutenegger, and M. A. Lopez, "Indexing the positions of continuously moving objects," Proc. of the ACM SIGMOD Int'l Conf. on Management of Data, pp. 331-342, 2000.

[6] G. Kollios, D. Gunopulos, and V. J. Tsotras. "On indexing mobile objects," Proc. of the ACM Symposium on Principles of Database Systems, pp. 261-272, 1999.

[7] J. Tayeb, O. Ulusoy, and O. Wolfson. "A quadtree based dynamic attribute indexing method," The Computer Journal, vol. 41, no. 3, pp. 185-200, 1998.

[8] Z. Song and N. Roussopoulos, "Hashing moving object," Int'l. Conf. on Mobile Data Management, pp. 161-172, 2001.

[9] D. Kwon, S. Lee and S. Lee, "Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree", Int'l. Conf. on Mobile Data Management, pp. 113-120, 2002.

[10] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "Then Grid Files: An adaptive, symmetric multikey file structure," ACM Transactions on Database Systems, vol.9, no. 1, pp. 38-71, 1984.

[11] N. Beckmann and H. P. Kriegel, "The R*-tree: An efficient and robust access method for points and rectangles," Proc. of the ACM SIGMOD Int'l Conf. on Management of Data, pp. 332-331, 1990.

[12] Y. Theodoridis, J. R. O Silva, and M.A Nascimento, "On the generation of spatiotemporal datasets," Proc. of Int'l Symposium on Spatial Databases, pp. 147-164, 1999.

[13] 전봉기, 홍봉희, "이동체 데이타베이스를 위한 해쉬 기반의 공간 색인," 한국정보과학회 학술발표 논문집, 제28권, 제2호, pp. 205-207, 2001.

저자소개

전봉기(Bong-Gi Jun)

1991년 부산대학교 컴퓨터공학과 공학사
1993년 부산대학교 컴퓨터공학과 공학석사
2003년 부산대학교 컴퓨터공학과 공학박사
1993년~1998년 한국통신 연구소 전임연구원
2003.9~현재 신라대학교 컴퓨터정보공학부 전임강사
※관심분야 : 데이터베이스, 공간 데이터베이스, 이동체 데이터베이스