메인메모리에서 캐시를 고려한 LUR-tree

°이현진*, 장용일*, 박순영*, 오영환**, 배해영*
*인하대학교 컴퓨터 · 정보공학과
**나사렛대학교 정보통신학과
e-mail: *hyunjin@dblab.inha.ac.kr*

Cache-Conscious LUR-tree in Main Memory

°Hyun-Jin Lee*, Yong-Il Jang*, Soon-Young Park*, Young-Hwan Oh**, Hae-Young Bae*
*Dept. of Computer Science & Information Engineering, Inha University
**Dept. of Information & Communication, Korea Nazarene University

요 약

이동객체의 위치 정보는 데이터의 양이 방대하고, 객체의 위치가 변경될 때마다 계속적인 갱신연산이 요구되어진다. 이러한 갱신 연산에서 디스크 접근 비용을 최소화하기 위해 최근 Lazy Update R-tree(LUR-tree)가 제안되었다. 그러나 디스크 기반의 색인은 검색 및 갱신 연산의 실시간 처리를 보장할 수 없기 때문에 메인 메모리에서 이동객체의 위치 정보를 유지하는 것이 필요하다.

본 논문에서는 디스크 기반의 LUR-tree를 MBR 압축을 통해 캐시에 최적화되도록 변형한 색인 기법을 제안한다. MBR 압축기법은 부모 노드로의 상대적 위치로 표현된 엔트리의 MBR을 변환함수를 통해 2, 4, 8 바이트의 정수로 변환한다. 제안된 색인은 변환된 MBR의 크기에 따라 엔트리를 동적 할당함으로써, 상위노드에서는 키 비교 회수를 줄이고, 단말 노드로 갈수록 키 비교 횟수는 늘어나지만, 캐시 미스를 줄일 수 있다는 장점으로 인해 검색 및 갱신 성능을 전체적으로 향상시킨다

1. 서 론

최근 사용자의 위치 정보를 이용하여 부가 정보 서비스를 제공하는 위치 기반 서비스(LBS:Location Based Services)에 대한 요구가 증대되고 있다. 위치 기반 서비스를 효과적으로 제공하기 위해서는 이동객체의 위치에 대한 효과적인 저장 및 관리가 필요하다. 그러나 이동객체의 위치 정보는 데이터의 양이 방대하고, 객체의 위치가 변경될 때마다 계속적으로 변경되는 특징을 가지므로, 전통적인 R-tree와 같은 색인 구조를 사용할 경우, 계속적인 이동객체의 위치변경으로 인해 R-tree의 분할과 합병이 빈번하게 발생하고, 이로 인한 추가적인 오버헤드가 불가피하다. 따라서 이러한 오버헤드를 발생시키는 갱신 연산을 줄이기 위해 이동 객체의 새로운 위치가 여전히 기존의 위치가 속해 있던 MBR 내에 위치하면, 단지 위치의 변경만 일어나도록 하는 Lazy Update R-tree(LUR-tree) 기법이 제안되었고, 이를 이용함으로써 갱신 연산에 의한 색인의 재구성 비용을 크게 줄일 수 있었다[1]. 그러나 이러한 디스크 기반의 색인은 이동객체의 수가 많아지거나, 데이터가 보고되는 횟수가 증가하게 되면, 질의에 대한 빠른 처리가 불가능하기 때문에 이를 해결하기 위하여 이동객체 색인을 메인 메모리에 상주시키는 방법이 연구되었다[2].

그러나 갱신 및 검색 연산이 많을 경우, 메인메모리와 CPU 사이의 속도차로 인한 병목 현상이 나타나고, 이로 인해 갱신과 검색에 대한 실시간 처리가 불가능해짐에 따라 최근 메인 메모리에서 캐시를 고려한 색인에 대한 연구가 중요하게 부각되고 있다[1]. 캐시를 고려한 색인은 캐시 공간의 효율적인 활용을 위하여 노드의 크기를 캐시의 배수가 되도록 하고, 포인터 제거, 키의 압축 등을 통해 노드에 들어가는 키의 개수를 최대화함으로써 메모리 접근을 최소화 하도록 제안된 색인 기법이다[3][4].

B+-tree의 변형인 CSB+-tree는 포인터 제거 방법을 통해 노드에 저장할 수 있는 키의 개수를 2배 가까이 늘림으로써 캐시미스를 효과적으로 줄였다. 또한 다차원 색인인 R-tree를 캐시에 최적화한 CR-tree는 MBR을 압축하여 같은 크기의 노드에 많은 키를 저장할 수 있도록 하였다[3]. 그러나 CR-tree의 MBR 압축 기법으로 사용된 QRMBR(Quantized Relative MBR)은 루트 노드로 갈수록 더 많은 오차가 발생하는 문제점이 있다[4].

따라서 본 논문에서는 기존의 MBR 압축 기법인 QRMBR에서 발생했던 오차를 줄이기 위해 새로운 MBR 압축방법인 TRMBR(Transformed Relative MBR)을 제안하고, 기존의 LUR-tree의 엔트리에 MBR 대신 TRMBR을 사용하여 캐시에 최적화한 Cache-conscious Lazy Update R-tree(CLUR-tree)를 제안한다. TRMBR은 부모 노드로의 상대적 위치로 표현된 엔트리의 상대적 MBR을 변환함수를 통해 2, 4, 8 바이트의 정수로 변환한 값으로 QRMBR에 비해 압축 시 적은 오차를 가지고 이로 인해 검색 성능 향상에 기여한다. CLUR-tree는 TRMBR의 크기에 따라 노드 내 엔트리를 동적 할당하여 키 비교 횟수와 캐시미스를 줄임으로써 검색 및 갱신 성능을 향상시킨다.

본 논문은 다음과 같이 구성되어 있다. 2장에서는 현재 위치 데이터 관리 기법과 캐시를 고려한 색인 기법을 논하고, 3장에서는 본 논문에서 사용된 MBR 압축기법인 TRMBR과 이를 이용한 색인 기법인 CLUR-tree를 제안하고, 검색, 삽입, 삭제, 갱신 알고리즘을 살펴본다. 4, 5장에서는 성능평가 및 결론을 맺는다.

2. 관련연구

관련연구는 크게 이동체의 현재 위치 데이터 관리를 위한 색인 기법과 캐시를 고려한 색인 기법으로 나누어 설명한다.

2.1 이동체의 현재 위치 데이터 관리를 위한 색인 기법

이동체의 현재 위치 데이터 관리를 위한 색인 기법으로 Lazy Update R-tree(LUR-tree)를 들 수 있다. LUR-tree는 객체의 ID와 객체가 속한 노드의 포인터 쌍을 구성 요소로 가지는 Direct Link라고 하는 보조 인덱스를 두어, 갱신이 일어나면 우선 Direct Link를 통해 객체가 속하는 단말 노드를 검색하고, 객체의 새로운 위치가 여전히 검색된 단말 노드의 MBR 내에 위치하면, 동일 노드 내에서 위치만 갱신하고, 그렇지 않다면 표준 R-tree와 마찬가지 방법으로 삭제 및 삽입을 통해 갱신연산을 수행한다[1].

2.2 캐시를 고려한 색인 기법

캐시를 고려한 색인은 크게 포인터 제거에 의한 방법과 MBR 압축에 의한 방법으로 나눌 수 있다.

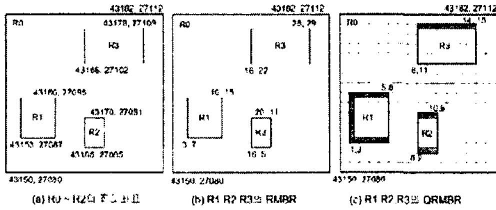
CSB+-tree는 B+-tree의 변형으로써 포인터 제거를 통해 노드에 들어갈 수 있는 키의 개수를 2배 가까이 늘렸다. CSB+-tree는 한 부모 노드에 속한 자식들을 메모리에 연속적으로 저장하고 첫 자식 노드의 포인터를 제외한 나머지 자식 노드들의 포인터는 제거하고, 질의 처리 시 간단한 연산을 통해 복구하도록 하였다[3]. 그러나 CSB+-tree에서 사용된 포인터 제거방법은 R-tree와 같은 다차원 색인에서는 압축의 효과가 적기 때문에 MBR 압축을 통해 R-tree를 캐시에 최적화되도록 변형한 CR-tree가 제안되었다[4].

CR-tree는 다차원에서 키로 사용되는 MBR을 압축하여 노드에 들어가는 키의 개수를 늘리고, 캐시미스를 효과적으로 줄였다. CR-tree에서는 [그림 2]에서와 같이 부모 노드의 MBR에 상대적으로 표현된 MBR을 RMBR으로,

본 연구는 대학 IT연구센터 육성 · 지원사업의 연구결과로 수행되었음

이를 일정한 비트를 사용하여 저장하도록 양자화한 것을 QRMBR이라 정의하였다. [그림 2]의 (c)는 16단계로 양자화된 QRMBR을 나타내고 있으며, R1, R2, R3의 MBR은 각각 2바이트로 압축이 되었음을 볼 수 있다. 그러나 R1, R2, R3는 실제 좌표를 QRMBR의 좌표로 변환하는 과정에서 각각 (c)에서 음영 부분만큼의 오차가 발생하며, 이러한 오차는 루트 노드로 갈수록 더욱 커진다.

CR-tree는 MBR 대신에 QRMBR을 사용하도록 변형된 구조로, 위와 같은 이유로 MBR 압축률이 커질수록 정확도가 떨어지게 된다. 이는 CR-tree의 성능평가에서 4, 8 바이트로 MBR을 압축했을 때에 비해 2바이트로 압축했을 때 적중 오류(false hit) 비율이 크게 증가한 것에서 확인할 수 있다.



[그림 2] RMBR과 QRMBR

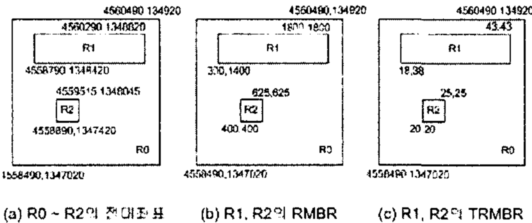
3. 압축 기법과 캐시를 고려한 LUR-tree

본 장에서는 압축 오차를 최소화한 압축 기법인 TRMBR을 제안하고, 이를 사용하여 LUR-tree를 캐시에 최적화되도록 변형한 CLUR-tree를 제안한다. 또한 제안된 색인의 검색, 삽입, 삭제, 갱신 등의 알고리즘을 제안한다.

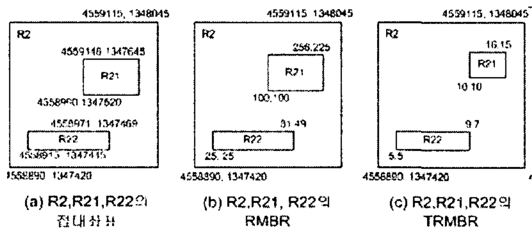
3.1 TRMBR을 이용한 MBR의 압축

R-tree의 속성에 따라 자식 노드의 MBR은 부모 노드의 MBR을 포함한다. 따라서 본 기법에서는 자식 노드를 부모 노드의 상대적 MBR인 RMBR로 표현하고, RMBR을 변환함수를 통해 변환한 값을 TRMBR (Transformed Relative MBR)이라 정의한다.

[그림 3]에서 (a)는 실제 거리가 5Km * 5Km 인 지도의 절대 좌표를 나타내고 있으며, R1과 R2는 R0의 자식 노드이다. (b)는 부모노드 R0에 대한 자식노드 R1, R2의 RMBR을 나타낸다. 본 논문에서는 변환함수로 주어진 값의 제곱근을 구하고, 구한 값보다 작지 않은 최소의 정수를 값으로 취하는 $f(t) = RMBRtoTRMBR(t)$을 사용하였다. (c)는 위의 변환함수를 사용하여, RMBR을 TRMBR로 변환한 값을 나타낸다. R2의 경우를 살펴보면, 실제 좌표는 (4558490, 1347420, 4559515, 1348420)으로 16바이트의 저장 공간이 필요하지만, TRMBR을 이용할 경우 MBR을 (20, 20, 25, 25)로 나타낼 수 있어 4바이트에 저장 가능하다.



[그림 3] R0의 RMBR과 TRMBR



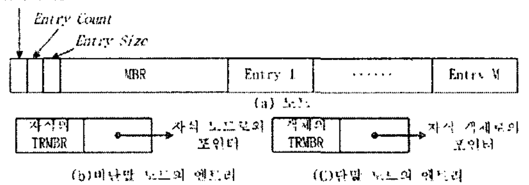
[그림 4] R2의 RMBR과 TRMBR

[그림 4]의 (a), (b), (c)는 R0의 자식노드인 R2를 확대하여 표현한 것이다. 그림에서 R2는 부모노드이고, 자식노드로 R21, R22를 갖는다. R21, R22는 TRMBR로 각각 (5, 5, 9, 7), (10, 10, 16, 15)를 가지며, 이는 2바이트에 표현될 수 있는 값으로, [그림 3]에서의 R1, R2보다 2배의 압축 효과를 가진다. 이처럼 노드 내 모든 엔트리의 MBR의 값 중 가장 큰 값인 최대 TRMBR([그림 4(c)]에서16)이 16 이하일 경우에는 MBR을 2바이트로 나타낼 수 있으며, 17보다 크고 256 이하일 경우에는 4바이트, 257 이상이고, 65536 이하일 경우에는 8바이트로 나타낼 수 있다. 이를 실제 거리로 나타내보면, 2, 4, 8바이트에 표현될 수 있는 실제거리는 각각 0.64㎞, 164㎞, 10737418㎞가 되므로, 대부분의 지역을 8바이트만 사용하여 표현할 수 있다.

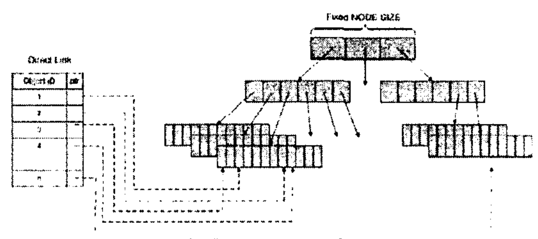
TRMBR 기법은 노드를 정해진 단계로 양자화하여 나타낸 QRMBR에 비해 계산과정이 간단하고, TRMBR로 변환했을 때 오차가 적어 검색 시 정확도를 향상시킬 수 있다.

3.2 CLUR-tree

CLUR-tree는 캐시의 활용도를 높이기 위해 노드의 크기를 캐시 크기의 배수로 고정시키고, 고정된 크기를 가진 노드의 fan-out을 늘리기 위해 MBR 대신 위에서 설명한 TRMBR을 사용하여 MBR을 압축한다. CLUR-tree의 노드 구성은 [그림 5]와 같으며, 검색, 삽입, 삭제 질의 시 엔트리의 MBR 재계산을 위해 노드는 자신의 MBR을 저장하고, 비단말 노드 및 단말노드의 엔트리는 변환함수에 의해 압축된 TRMBR을 저장한다. 그러나 CR-tree에서 QRMBR의 크기가 고정되어있던 것과 달리, TRMBR은 노드 내 엔트리의 영역에 따라 다르게 계산될 수 있으며, CLUR-tree는 계산된 TRMBR의 크기에 따라 노드 내 엔트리를 동적으로 할당한다. 노드는 캐시의 배수로 고정되어 있기 때문에 CLUR-tree의 구조는 [그림 6]와 같이 나타난다. 이는 루트노드는 포함하는 영역이 크기 때문에 정해진 크기의 노드 안에 8Byte의 엔트리 n개를 포함하고, 하위 레벨에서는 4Byte의 엔트리 2n개를, 그리고 단말 노드에서는 포함하는 영역이 작아지므로 2Byte의 엔트리 4n개를 포함하고 있음을 보여주고 있다. 또한 동일 레벨의 노드라 할지라도 포함하고 있는 영역이 다를 경우, 노드에 포함되는 엔트리의 수 역시 다르게 나타날 수 있다. 따라서 노드마다 다르게 나타나는 엔트리의 크기를 명시하기 위해 [그림 5]의 (a)에서와 같이 엔트리의 크기(Entry Size)를 저장한다. CLUR-tree의 이러한 구조는 검색 및 갱신 연산 시 상위 노드에서는 키 비교 횟수를 줄이고, 하위 노드에서는 캐시미스를 작게 하여, 전체적인 성능 향상을 가져온다.



[그림 5] 계층적 R-tree의 노드 구조



[그림 6] CLUR-tree 의 구조

3.3 CUR-tree에서의 이동체 처리 알고리즘

본 장에서는 이동객체의 검색, 삽입, 삭제, 갱신 알고리즘을 살펴본다.

3.3.1 검색

검색 알고리즘은 LUR-tree의 검색 알고리즘과 유사하며, 단지 다른 점은 질의 사각형은 MBR이고, CLUR-tree의 엔트리는 TRMBR 형태로 저장되어 있다는 것이다. 따라서 검색 시 질의사각형의 MBR을 TRMBR로 변환한 후 각 엔트리가 TRMBR로 표현된 질의사각형과 겹치는지를 결정한다.

3.3.2 삽입과 삭제

CLUR-tree는 TRMBR에 따라 노드 내 엔트리의 크기가 변경된다. 즉 [그림 4]의 (c)와 같이 노드 내 최대 TRMBR이 42가 되어 엔트리를 4바이트로 할당하였으나, 이후 새로운 객체가 삽입되어 최대 TRMBR이 256 이상이 된다면, 엔트리 하나의 크기를 8바이트로 재할당하기 위해 노드의 재구성이 일어난

다. 본 절에서는 CLUR-tree에서의 노드의 재구성을 최소화하기 위한 삽입 알고리즘을 제안한다.

R-tree는 객체 삽입될 위치를 결정할 때 객체를 포함하는 노드가 최소한의 확장을 가지도록 한다. 그러나 CLUR-tree는 노드의 재구성을 최소화하기 위하여 엔트리의 크기(Entry Size)가 바뀌지 않도록 하는 것을 우선으로 한다. 즉 N1, N2 두개의 노드의 엔트리의 크기는 각각 2바이트, 4바이트이고, 객체가 삽입되었을 경우, N1은 최소 확장을 갖지만 엔트리의 크기가 4바이트로 바뀌어야 하고, N2는 N1에 비해 MBR이 조금 더 확장되어야 하지만 엔트리가 여전히 4바이트로 유지된다면, N1보다는 N2를 선택한다. 이를 삽입 알고리즘에 반영하기 위해 R-tree에서 객체가 삽입될 노드를 찾는 함수로써 정의된 ChooseLeafNode 함수를 [알고리즘 1]과 같이 재정의한다.

```
 1  Algorithm ChooseLeafNode(Node)
 2  input: Node
 3  output: Node : 객체가 삽입될 노드
 4  Begin
 5      if Node == LeafNode
 6          return Node;
 7      else
 8          Entry[] = GetEntrysNotExtendTRMBR(Node);
 9          if(Entry[].num!=0)
10              Entry = GetLeastEnlargementEntry(Entry[]);
11          else
12              Entry = GetLeastEnlargementEntry(Node);
13      ChooseLeafNode(Entry.ChildNode);
14  End
```

[알고리즘 1] CLUR-tree 의 삽입 알고리즘

ChooseLeafNode은 만약 노드가 단말 노드가 아니라면, GetEntrysNotExtendTRMBR 함수를 이용하여 Node 내 TRMBR이 확장되지 않는 엔트리들을 구하고, 만약 TRMBR이 확장되지 않는 엔트리들이 존재하면, 그 엔트리들 중에서 MBR이 가장 적게 확장되는 엔트리를 선택한다. 그렇지 않다면, 기존의 알고리즘과 같이 MBR이 최소 확장이 되는 엔트리를 선택한다. 엔트리가 선택되면, 엔트리의 자식노드를 input으로 하여 다시 ChooseLeafNode 함수를 호출하고, 단말 노드에 이르면 이를 리턴한다. 객체가 삽입될 노드를 검색한 후의 삽입 과정은 엔트리의 MBR이 TRMBR로 변환되는 것을 제외하고는 디스크 기반의 LUR-tree와 유사하다. 또한 CLUR-tree의 삭제 알고리즘 역시 LUR-tree의 삭제 알고리즘과 유사하다.

3.3.3 갱신

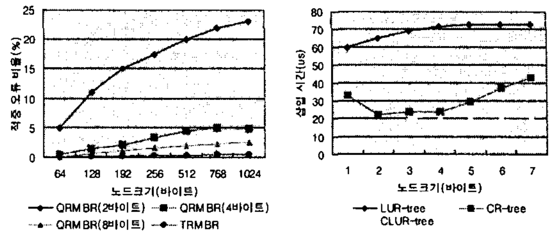
갱신 알고리즘은 디스크 기반의 LUR-tree 알고리즘을 캐시를 고려한 CLUR-tree의 구조에 맞도록 수정하였으며, Direct Link를 두어 갱신 질의의 성능 향상이 이루어졌다. CLUR-tree 알고리즘은 [알고리즘 2]와 같으며 LUR-tree 갱신 알고리즘과 유사하게 동작한다.

```
 1  Algorithm LazyUpdate(oid, newpos)
 2  input: oid : 객체의 ID, newpos:객체가 이동한 위치
 3  Begin
 4      Node = readDirectLink(oid)
 5      Entry = getEntry(Node, oid)
 6      if newpos ∈ Node.MBR then
 7          Entry.pos = newpos;
 8          writeNode(Node);
 9      else
10          Delete(oid, Entry.pos)
11          ptr = Insert(oid, new pos)
12          UpdateDirectLink(oid, ptr)
13  End
```

[알고리즘 2] CLUR-tree 의 갱신 알고리즘

4. 성능 평가

본 연구의 성능 평가는 Pentium 4, 2.4GHz CPU, 1GB의 메인메모리, 64바이트의 캐시블럭을 가진 Windows XP 운영체제 환경에서 2차원 데이터에 대해 실험을 수행하였다. 실험에 사용된 데이터는 인천광역시 지도데이터이고, 이동 객체는 자체 개발한 생성기로 시뮬레이션하였다.



(a) 적중 오류 비율 (b) 갱신 성능

[그림 7] 검색 및 삽입 성능

[그림 7]의 (a)는 기존에 제안되었던 MBR 압축 방법인 QRMBR을 이용한 CLUR-tree와 본 논문에서 제안된 TRMBR을 이용한 CLUR-tree의 적중 오류율(false hit)을 비교한 것이다. 노드의 크기는64바이트부터 1024바이트까지 변경시키며 실험을 수행하였고, 이를 반복적으로 수행한 후 평균 수행시간으로 나타내었다. QRMBR을 이용한 CLUR-tree의 경우 QRMBR의 크기에 따라 적중 오류 바율이 크게 차이가 나기 때문에, 2, 4, 8 바이트로 나누어 실험하였다. 성능평가 결과, TRMBR을 이용한 CLUR-tree가 더 적은 적중 오류 비율을 가졌으며, 이는 압축 시 TRMBR이 QRMBR에 비해 오차가 적기 때문으로 보여진다. 이로 인해는 검색 성능을 향상시켰다.

[그림 7]의 (b)는 기존에 제안되었던 LUR-tree, CR-tree 그리고 본 논문에서 제안된 CLUR-tree의 갱신 성능 평가를 나타낸 것이다. CR-tree와 CLUR-tree는 캐시를 고려함으로써 LUR-tree에 비해 전체적으로 최대 2.5배 이상의 성능 향상을 보이고 있으며, TRMBR의 크기에 따른 엔트리 동적 할당을 사용함으로써, 상위 노드로 갈수록 키 비교회수를 줄이고, 하위노드로 갈수록 캐시미스를 줄여 삽입 성능이 향상되었음을 볼 수 있다.

5. 결론 및 향후연구

본 논문에서는 이동객체의 현재 위치 데이터의 검색 및 갱신 성능의 향상을 위하여 기존에 갱신 성능 향상을 위해 제안된 LUR-tree를 캐시에 최적화한 색인 기법인 CLUR-tree를 제안하였다.

CLUR-tree는 MBR 압축을 위하여 TRMBR을 사용하였으며, 이를 사용한 CLUR-tree는 기존의 MBR 압축 기법을 사용한 색인에 비해 검색 시 적중 오류율이 낮았다. 또한 CLUR-tree는 TRMBR의 크기에 따라 노드 내 엔트리를 동적 할당함으로써 단말 노드로 갈 수록 더 많은 엔트리를 포함하고, 이는 상위노드에서는 키 비교 회수를 줄이고, 단말 노드로 갈수록 키 비교 횟수는 늘어나지만, 캐시 미스를 줄일 수 있다는 장점으로 인해 검색 및 갱신 성능을 전체적으로 향상시켰다.

향후 연구로 RMBR을 TRMBR로 변환할 때 사용할 수 있는 가장 효과적인 변환 함수에 대한 연구가 이루어져야 할 것이다.

참고 문헌

[1] D. Kwon, S. Lee, Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. MDM, 2002

[2] 이창우, 안경환, 홍봉희, 이동체 데이터베이스를 위한 메인 메모리 색인의 성능 결정 요소에 관한 연구, 정보과학회, 2003.5

[3] J. Rao, K. A. Ross, Making B+-trees Cache Conscious in Main Memory, ACM SIGMOD, 2000

[4] Kihong Kim, Sang K. Cha, Keunjoo Kwon, Optimizing Multidiminsional Index Trees for Main Memory Access, ACM SIGMOD, 2001

[5] I. Sitzmann and P. J. Stuckey, Compacting Discriminator Information for Spatial Trees, In Proceedings of the Thirteenth Australasian Database Conference, 2002

[6] Mohamed F. Mokbel, Thanaa M. Ghanem, Walid G. Aref, Spatio-Temporal Access Methods, IEEE, 2003

[7] A. Guttman, R-trees:A Dynamic Index Structure for Spatial Searching, ACM SIGMOD, 1984