RFID 태그 추적을 위한 캐시 & 메인 메모리 기반의 색인 기법(CSTmr-tree)

홍진숙*, 윤성대**
*부경대학교 전산정보학과
**부경대학교 정보공학과
e-mail:*hongcine@naver.com,**sdyoun@pknu.ac.kr

Indexing Scheme based on the Cache & Main Memory for RFID tag Tracing (CSTmr-tree)

Jin-Suk Hong*, Sung-Dae Youn**
*Dept of Computer and Information, Pukyong National University
**Dept of Information Engineering, Pukyong National University

요 약

주기억 색인 기법인 Tmr-트리가 R-트리에 비해서 삽입시간이 오래 걸린다는 단점이 있다. 본 논문은 L2 캐시를 최대한 활용하여 기존 Tmr-트리의 장점을 가지는 새로운 CSTmr-트리(Cache Sensitive Tmr-트리)구조를 제안하고, 이 구조에 삽입, 삭제 등의 알고리즘을 제안하였다. 제안한 구조와 알고리즘을 다른 인덱스 구조와 비교하여 CSTmr-트리의 우수성을 보인다.

1. 서론

지난 10년간 CPU의 속도는 메모리의 속도에 비해 급속한 속도로 발전하였다. 그 결과 데이터베이스 시스템을 포함한 다른 컴퓨터 응용분야에서 메모리의 접근이 병목현상을 일으키게 되었다. 메모리의 접근 속도를 줄이기 위해 캐시 메모리가 도입되었다. 하지만 캐시 메모리는 원하는 데이터가 캐시에 옮겨져 있어야 메모리 접근 속도를 줄일 수 있다. 따라서 응용프로그램에서 데이터를 어떤 순서로 액세스 하느냐에 의해 캐시의 활용도가 달라지고 응용프로그램의 성능이 달라지게 된다. 그리고 최근 메모리 가격의 하락과 본격적인 64비트 운영체제로 인해 대용량의 램을 사용하는 주기억 데이터베이스 시스템의 구축이 실현 가능하게 되었다. 유비쿼터스 컴퓨팅, 와이브로, 텔레매틱스, RFID 등의 이동체 데이터베이스들이 실시간 데이터 처리에 기반하고 있어 주기억 데이터베이스에 대한 관심과 수요는 폭발적으로 증가 할 것으로 전망된다.

그러나 기존의 디스크 기반 공간색인기법은 디스크 접근 시간만을 주로 고려하기 때문에, 주기억 색인기법으로 디스크 기반 색인 기법을 직접적으로 적용시키는 것은 부적절하다. 주기억 장치 색인 기법은 모든 색인 노드들이 주기억 장치에 상주하기 때문에 노드에 대한 접근 시간이 디스크 기반 기법에 비해 상당히 미미하고, 결국 효율적인 색인 기법을 위해서는 노드 접근시간 뿐 만 아니라 노드 내의 키 비교시간을 고려해야 한다.[1,2,3,4,5,6,7,8]

본 논문은 주기억 색인 기법인 Tmr-트리가 캐시를 효율적으로 사용하도록 새로운 구조를 제안하였다. 시스템의 L2 캐시를 최대한 활용하며 기존 Tmr-트리의 장점을 가지는 새로운 CSTmr-트리(Cache Sensitive Tmr-트리)를 설계 개발하고, 실험을 통해 다른 인덱스 구조에 비교하여 CSTmr-트리의 우수성을 보인다.

2. 관련연구

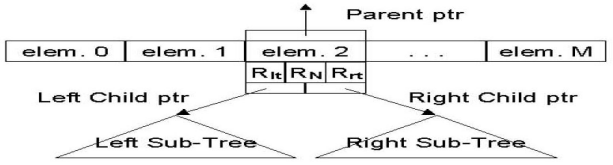
2.1 유비쿼터스 RFID 이동체 데이터베이스

기존이동객체의 위치추적은 시간이 지남에 따라 위치가 변화하는 객체 이동객체 데이터모델로 이산적 모델을 사용하여 표현하며 이동객체의 샘플링된 위치를 점(x,y)로 나타내고 궤적을 다중선(polyline)으로 표현하며 3DR-Tree, TB-Tree 이동체의 최근보고 이전의 이동궤적을 검색하고 3차원 MBB(Minimum Bounding Box)형태로 색인에 저장하며 질의종류는 영역질의, 궤적질의, 복합질의처리가 있다. RFID 태그객체의 위치추적은 시간의 변화에 따라 위치가 변화하며 이동객체와 유사하지만 중요한 지점에 설치된 판독기에서만 위치를 보고하고 판독기의 인식영역의 크기와 수에 따라 위치의 정확성이 결정되며 RFID 태그객체의 위치 추적을 위한 색인이 필요하고 새로운 과거 이력 표현 방법이 필요하며 순서와 시간을 기반으로 하는 데이터모델과 색인구조가 필요하다[1,2,3]

2.2 메인메모리 기반 Tmr-트리

공간 데이터를 둘러싸는 MBR로 공간 데이터를 표현할 때 메인메모리 기반 Tmr-트리 색인 구조는 높이 균형

트리인 T-트리 구조를 기본으로 사용하며 각 트리 노드의 구조는 (그림 1) 과 같이 색인 노드의 구조는 T-트리 노드와 같이 부모노드에 대한 포인터(Parent ptr), 왼쪽 자식 노드에 대한 포인터(Left child ptr), 오른쪽 자식 노드에 대한 포인터(Right child ptr), 여러 개의 엔트리들, 그리고 새롭게 정의되는 왼쪽 서브트리의 MBR(Rlt), 자신 노드 N의 엔트리들에 대한 MBR(RN), 오른쪽 서브트리의 MBR(Rrt)으로 구성된다.



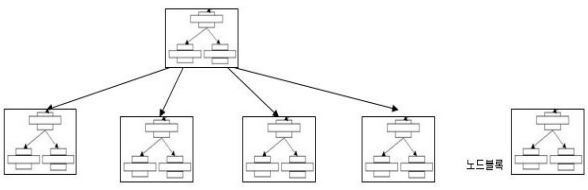
(그림 1) 공간 색인 구조 (노드 N)

삽입시간에서 R-트리에 비해서 Tmr-트리가 약간 나쁜 성능을 보여주는데, 이것은 Tmr-트리 경우 노드 분할 발생 시 로테이션 수행과 같은 처리 지연 효과를 그 원인으로 설명할 수 있다. Tmr-트리 색인 구조는 R-트리와 달리 이진 검색에 의해 내부 노드에서 완료될 수 있기 때문에 빠른 검색이 가능하다. 2차원 이상의 공간 데이터의 경우, 삽입 및 삭제와 같은 연산이 자주 발생하지 않으므로 색인 구조는 주기억 장치 데이터베이스에서 공간 데이터를 다루는 데에 효율성을 가지고 있다고 볼 수 있다. Tmr-트리 색인 구조를 질의 연산인 조인과 같은 질의처리에 적용하는 방법이 R-트리에 비해서 삽입시간이 오래 걸린다는 단점이 있다.[4,5,6,7,8]

3. CSTmr-트리의 제안

3.1 CSTmr-트리 구조

Tmr-트리를 모아 이진트리를 구성한다. 노드블록 사이즈가 캐시 블록 사이즈 일 때 가장 효율 적이다. Tmr-트리 노드를 캐시 블록 크기만큼 모아 만든 이진트리 블록을 '노드 블록'이라 부르기로 한다. 노드 블록 내에서는 캐시 미스 없이 자식 노드를 읽을 수 있다. 노드 블록과 노드 블록 사이의 연결은 포인터를 이용한다. (그림 2)는 CSTmr-트리의 논리적 구조이며 하나의 노드 블록에 달리는 자식 노드의 개수는 4개이다. 이 자식 노드의 공간을 할당 받을 때 각각을 할당 받게 되면 4개의 포인터가 필요하지만, 인덱스의 개수가 4인 노드 블록의 배열로 공간을 할당 받으면 포인터가 하나만 있어도 하나의 포인터와 배열의 인덱스로 각각의 자식 노드 블록을 액세스할 수 있다.



(그림 2) CSTmr-트리의 구조

3.2 CSTmr-트리 검색 알고리즘

(그림 3)은 CSTmr-트리 검색 알고리즘으로 색인 구조를 사용한 검색을 위해서는 노드 블록을 B, 루트 노드는 N, 자신노드의 엔트리들에 대한 nt, 왼쪽 서브트리 lt, 오른쪽 서브트리 rt를 이용하여 객체 식별자(tid)를 검색 시, 자신 노드와 교차되는 지점이 있는지 검색하여 존재하면 자신 노드 엔트리들을 검색한다. 왼쪽 서브트리 교차점이 있는지 검색 후, 존재하면 왼쪽 자식 노드를 방문하고, 같은 방식으로 오른쪽 서브트리의 검색작업을 수행한다.

```
search(B, N) // B:node block, N:root node of CSTmr-tree
  if overlap(B, nt of N),
    for (each entry e of node N) if overlap(B, e), result=e
  if overlap(B, lt of N),
    result += search(B, N.left_child_ptr)
  if overlap(B, rt of N),
    result += search(B, N.right_child_ptr)
```

(그림 3) CSTmr-트리 검색 알고리즘

3.3 CSTmr-tree 삽입 알고리즘

(그림 4)의 CSTmr-트리 삽입 알고리즘(insert)은 검색을 하여 삽입할 노드를 찾아 삽입하고 삽입으로 인해 노드의 오버플로가 발생할 경우, 분할하여 엔트리들을 기존의 노드와 새롭게 생성한 노드에 분산 배치한다. 새롭게 생성된 노드는 분할이 발생된 노드를 중심으로 크기가 최소가 되는 왼쪽 서브트리 또는 오른쪽 서브트리로 찾아들어간 후, 리프 혹은 반-리프 노드의 자식 노드가 되도록 배치한다.

```
insert(E, N) // E : object, N : root node of CSTmr-tree
if (minA(E,N)=nt of N),{
 if (N has empty room),
     insert E into N, and adjust nt of N
 else {
     N′ = split(N, E)
     insertN(N′, N)
   }
  else if(minA(E,N)=lt of N),{
     insert(E, N.left_child_ptr), and adjust lt of N
     if (N.left_child_ptr is unbalanced),
         N.left_child_ptr = balance(N.left_child_ptr)
     }
   else {
     insert(E, N.right_child_ptr), and adjust rt of N
     if (N.right_child_ptr is unbalanced),
         N.right_child_ptr=balance(N.right_child_ptr)
  }
 }
```

(그림 4) CSTmr-트리 삽입 알고리즘(insert)

3.4 CSTmr-트리 삭제 알고리즘

(그림 5)의 CSTmr-트리 삭제 알고리즘(delete)은 색인 노드 안의 엔트리를 삭제를 할 경우에는 해당 엔트리를 포함하는 노드를 검색해 삭제한다. 본 논문에서 제시하는 트리구조는 리프뿐만 아니라 모든 위치의 노드에 데이터를 저장하기 때문에 삭제는 모든 노드에서 이루어질 수 있다. 만약, 중간노드에서 삭제로 인해 언더플로가 발생할 경우에는 최소 직사각형의 면적 증가크기를 비교하여 엔트리들 중 가장 증가크기가 최소인 노드 엔트리를 검색하여 삭제 발생 노드로 이동시킨다.

```
delete(E, N) // E: object, N : root node of CSTmr-tree
 if (N is underflow), {
  if (N is a leaf node), return
  else if(minA(E,N)=lt of N), {
       E = choose(nt, N.left_child_ptr)
       if (left sub-tree is unbalanced),
          N.left_child_ptr = balance(N.left_child_ptr)
       } else {
       E = choose(nt, N.right_child_ptr)
       if(right sub-tree is unbalanced),
         N.right_child_ptr=balance(N.right_child_ptr)
    }
   insert E into node N
 }
```

(그림 5) CSTmr-트리 삭제 알고리즘(delete)

3.5 CSTmr-트리 노드밸런스 알고리즘

(그림 6)은 CSTmr-트리는 검색 과정에서 성능에 가장 크게 영향을 미치는 것은 캐시 미스이다. 따라서 캐시 미스를 발생하게 하는 노드 블록 간의 밸런싱에 더욱 신경을 쓸 필요가 있다. 이 노드 블록 간의 트리 밸런싱은 Tmr-트리의 알고리즘을 변형하여 고안하였다. 노드 블록 간의 밸런싱 만큼이나 노드 블록 내의 이진트리 밸런싱 알고리즘도 중요하다. 한 노드 블록 안에 최대한 많은 수의 노드가 포함되어 있어야 검색이 빨라지기 때문이다.

노드 블록 간의 밸런싱 알고리즘은 노드 블록 내의 이진트리는 캐시 블록 크기에 의해 그 크기가 제한되어 있으므로 밸런스가 깨질 때 마다 인덱스를 재배열하는 방법으로 밸런싱을 조절한다. 노드 블록 내의 이진트리는 한 번에 캐시 블록으로 복사가 되므로 밸런싱 작업을 하는 중에 추가적인 캐시 미스는 발생하지 않는다.

```
balance(B) // B : node block
if (maxH(B) - minH(B) > 1), {
 if (maxI(B) > minI(B)),  {
     B′ = maxI(B)
     if (maxI(B) - minI(B) != 1), {
         e = maxI(B) -1
         s = minI(B)
         for ( i = s ; i < e; i++) {
            rotation(B, i+1, i)
         }
      }
     if (maxI(B′) != child), {
         for ( i = maxI(B′) ; i < child ; i++ ) {
            rotation(B′ , i , i+1)
         }
     }
         rotation(B , maxI(B) , minI(B))
else
   }
}
/* minH(B) : min Height(child of B),
maxH(B) : max Height(child of B),
minI(B) : min Index(child of B),
maxI(B) : max Index(child of B) */
```

(그림 6) 노드 블록 밸런싱 알고리즘

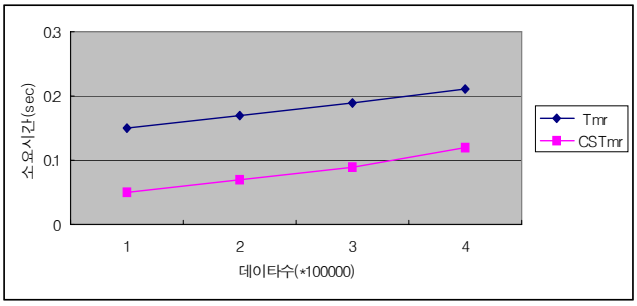
4. 성능 평가

4.1 실험 환경

CSTmr-트리는 C 프로그래밍 언어로 구현되었고 Linux 운영체제에서 GNU gcc컴파일러로 컴파일했다. 실험을 한 컴퓨터는 Intel Pentium(R) IV 3.0Ghz CPU와 1.0GB DDR RAM의 컴퓨터로 1M의 L2 캐시 메모리를 가지고 실험하였다.

4.2 검색(Search) 성능

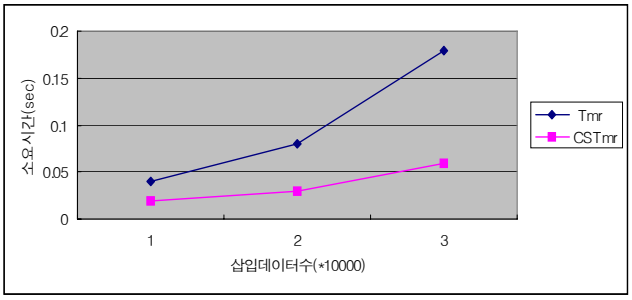
(그림 7)은 트리가 각각 100,000, 200,000, 300,000, 400,000 개의 키를 가지고 있을 때 200,000 건의 임의검색(Random Search)을 수행했을 때 걸린 시간을 그래프로 표시한 것이다. Tmr-트리보다 CSTmr-트리가 좋은 성능을 보이고 있다.



(그림 7) 검색의 수행 시간

4.3 삽입(Insertion) 성능

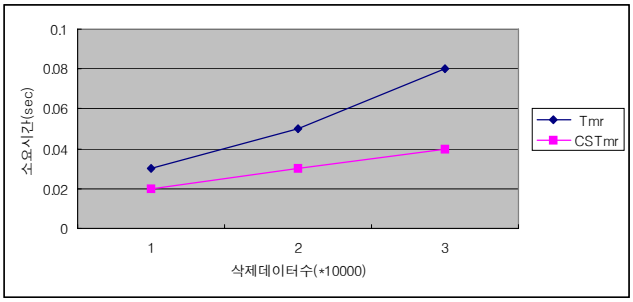
(그림 8)은 10,000, 20,000, 30,000 개의 키를 임의로 생성하여 삽입했을 때 걸린 시간을 그래프로 표시한 것이다. Tmr-트리보다 CSTmr-트리가 좋은 성능을 보이고 있다.



(그림 8) 삽입의 수행 시간

4.4 삭제(Deletion) 성능

(그림 9)는 200,000 개의 키를 임의 생성하여 트리를 구축한 후, 10,000, 20,000, 30,000 개의 키를 임의로 삭제했을 때 걸린 시간을 그래프로 표시한 것이다. Tmr-트리보다 CSTmr-트리가 좋은 성능을 보이고 있다.



(그림 9) 삭제의 수행 시간

5. 결론 및 향후연구

본 논문에서는 캐시를 고려한 주기억 데이터베이스에서 이동체 데이터를 위한 효율적인 색인 구조인

CSTmr-트리를 제안하였다. 캐시를 고려한 인덱스 설계로 성능을 높이고자 하는 아이디어를 Tmr-트리에 적용하여 만들어졌다. 지금까지 제안된 이동체 데이터에 대한 색인 구조는 디스크 기반의 데이터베이스를 위한 구조였기 때문에 디스크 접근 횟수의 최소화와 디스크 저장 공간의 효율적 사용에 그 초점이 맞춰져 있었다. 그러나 주기억 데이터베이스에서는 디스크 기반 데이터베이스와는 달리 빠른 처리 속도와 메모리 공간 사용의 최적화라는 목적을 두고 있다.

시스템 L2 캐시 활용도를 최적화할 수 있도록 Tmr-트리의 구조를 새로이 설계하고 그에 따른 검색 알고리즘을 고안하였으며, 삽입/삭제 연산에서의 노드블록 밸런싱을 위해 제안한 트리에 맞게 새로운 회전 알고리즘을 제안하였다. 제안한 알고리즘의 타당성을 위해 실험을 하였다. 실험의 결과에 의하면, Tmr-트리보다 CSTmr-트리가 검색, 삽입, 삭제 시에 성능이 우수함을 알 수 있었다.

본 논문에서 제안한 새로운 CSTmr-트리에 대해 다음의 향후 추가 연구가 필요하다. DBMS의 새로운 대안으로 하이브리드MMDBMS 등장시점에 맞추어 캐시를 고려한 주기억 데이터베이스 색인에서 더 발전한 캐시를 고려한 주기억 데이터베이스 색인과 디스크 기반의 데이터베이스 색인의 혼용 구성 연구가 필요하다.

참고문헌

[1] S. E. Sarma, S. A. Weis, and D. W. Engels, “RFID Systems and Security and Privacy Impl -ications ,” Springer-Verlag, 2002, pp454-469.

[2] K. Romer, T. Schoch, “Infrastructure Concepts for Tag-Based Ubiquitous Computing Applica- tions ,” Workshop on Concepts and Models for Ubiquitous Computing, 2002.

[3] EPC Tag Data Standard Work Group, "EPC Tag Data Standards Version 1.23," EPC Glo- bal, 2005.

[4] A. Guttman, “R-Tree: A dynamic index struc- cture for spatial searching ,” Proc. of the 1984 ACM SIGMOD on Management of Data, 1984, pp47-57.

[5] Tobin J. Lehman, “A Study of Index Structures for Main Memory Database Management System”, VLDB 1986.

[6] Jun Rao, et al, “Cache Conscious Indexing for Decision-Support in Main Memory”, VLDB 1999.

[7] Hongjun Lu, Yuet Yeung, Ng Zengping, “T-Tree or B-Tree : Main Memory Database Index Structure Revisited”, Australasian Data- base Conference, 2000.

[8] 윤석우, 김경창 ,“Tmr-트리:주기억 데이터베이스에서 효율적인 공간 색인 기법”, 정보처리학회논문지D 제 12-권 제 4 호, 2005.