T^2-트리 : 동적 주기억 데이터베이스를 위한 효율적 색인 구조

김태진* 전홍석* 이재호** 노삼혁*
*홍익대학교 컴퓨터공학과
**인천교육대학교 컴퓨터교육과

T^2-Tree : An Efficient Index Structure for Dynamic Main Memory Database

Tae Jin Kim* H. Seok Jeon* Jae Ho Lee** Sam H. Noh*
*Dept. of Computer Engineering, Hong Ik University
**Dept of Computer Education, Inchon National University of Education

요 약

주기억 데이터베이스를 위한 색인 구조는 기존의 디스크 기반 데이터베이스의 색인 구조와는 고려되어야 할 사항이 다르다 최근까지 연구된 색인 구조 중 대표적인 것은 T-트리와 T*-트리이다 비록 T*-트리가 T-트리의 단점인 범위 질의의 비효율성을 해결하고 있지만 데이터의 삽입과 삭제가 많은 시스템에서 트리 균형을 맞추기 위한 오버헤드, 회전 연산의 수행과 후위 포인터(successor pointer)의 추가적인 오버헤드가 있다 따라서 본 논문에서는 삽입과 삭제가 빈번한 동적 주기억 데이터베이스를 위해서 역제된 노드 생성 및 삭제 기법과 스레드 이진 트리의 특성을 이용한 보다 효율적인 색인 구조인 T^2-트리를 제안한다

1. 서론

주기억장치의 집적기술 향상과 가격 하락으로 인하여 현대의 컴퓨터는 대용량 주기억장치를 장착할 수 있는 환경으로 바뀌고 있다. 이러한 경향에 맞추어 데이터베이스 분야에서도 주기억장치 기반 데이터베이스가 가능하게 되었다 따라서, 주기억장치 기반 데이터베이스를 위한 효율적인 색인 구조가 필요로 하게 되었다.

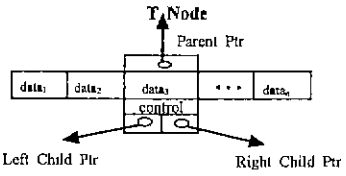
주기억장치 기반 데이터베이스를 위한 색인 구조는 디스크 기반 데이터베이스를 위한 색인 구조와는 고려되어야 할 사항이 다르다 후자가 디스크 접근횟수와 디스크 공간을 줄이는 것을 고려해서 접근한다면, 전자는 중앙처리장치의 계산횟수와 주기억장치에 차지하는 공간을 줄이는 것을 고려해서 접근한다.

최근까지 연구된 색인 구조의 대부분은 디스크 기반 데이터베이스를 위한 색인 구조이다 주기억 데이터베이스를 위한 색인 구조에 대한 연구는 활발하지 않지만 그 중 대표적인 색인 구조는 T-트리와 T*-트리다 그러나 이들 구조들은 삽입과 삭제가 많은 동적 주기억 데이터베이스에서는 빈번한 노드 생성 및 삭제에 따르는 오버헤드로 효율적이지 못하다. 따라서 본 논문에서는 동적 주기억 데이터베이스에 보다 효율적인 색인 구조인 T^2-트리를 제안한다

본 논문의 구성은 다음과 같다. 2절에서는 기존의 주기억 데이터베이스 색인 구조인 T-트리, T*-트리를 설명하고, 3절에서는 본 논문에서 제안한 T^2-트리의 구조, 탐색, 삽입, 삭제, 회전 연산을 설명하고, 마지막으로 4절에서 결론을 맺는다.

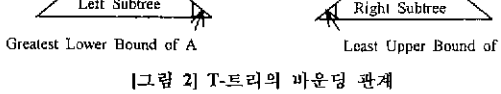
2. 관련연구

T-트리[1]는 AVL트리와 B-트리의 특성을 결합해서 새롭게 변형된 트리 구조이다. 노드 구조는 정렬된 여러 개의 아이템들로 구성되어 있으며, 두 개의 서브 트리를 가리키는 포인터가 있다 노드는 서브 트리를 가리키는 포인터의 널(nil)여부에 따라서 중간노드(internal node), 반단말노드(half-leaf node), 단말노드(leaf node)로 나누어진다. [그림 1]은 T-트리의 노드 구조를 보여주고 있다



[그림 1] T-트리의 노드 구조

T-트리의 이진 트리 특성에 의하여 [그림 2]의 바운딩(bounding) 관계가 나타난다 노드 A의 최소 아이템보다 작은 아이템으로 구성된 왼쪽 서브 트리에서 최대값을 Greatest Lower Bound (GLB)라 하고, 노드 A의 최대 아이템보다 큰 아이템으로 구성된 오른쪽 서브 트리에서 최소값을 Least Upper Bound (LUB)라 한다 또한 데이터 아이템 X가 노드 A의 최소 아이템과 최대 아이템 사이에 포함된다면 아이템 X는 노드 A에 바운드(bound) 된다고 한다



[그림 2] T-트리의 바운딩 관계

모든 노드의 삽입과 삭제는 단말노드에서만 이루어지며 이때 트리가 불균형 상태가 되면 회전 연산을 수행하여 균형 상태로 만들어 준다 T-트리에서는 중간노드에서만 데이터 아이템이 점유할

본 연구는 한국과학재단 특정기초연구과제 (과제번호: 98-0102-09-01-3)의 지원에 의해 수행되었음

수 있는 범위에 해당하는 최소수와 최대수를 가지며, 반단말노드와 단말노드는 최대수만을 가진다

T-트리가 주기억 데이터베이스에서 다른 색인 구조에 비해 성능이 우수하지만 다음과 같은 단점이 있다. 첫 번째, 노드간 순회를 중위순회로 하므로 범위 질의나 순차 질의 수행 시 불필요한 순회를 하게 된다. 두 번째, 데이터 아이템의 삽입과 삭제 시 중간노드의 점유 범위 만족을 위해서 GLB 접근이 발생할 때 중간노드를 반드시 거쳐야만 접근가능하고, GLB만을 살펴보기 때문에 새로운 노드 생성 또는 삭제 가능성이 높다. 세 번째, 노드 생성과 삭제 시 트리의 불균형 상태를 알아보기 위해서 항상 트리 깊이를 계산해 주어야 하고, 불균형 상태가 되면 회전 연산을 수행해야 한다는 단점이 있다

T^*-트리[2]는 T-트리의 구조에서 후위 포인터(successor pointer)를 첨가한 색인 구조이다. 후위 포인터는 T-트리의 바운딩 관계에서 LUB를 가리키는 포인터이다. 후위 포인터 첨가로 인하여 범위 질의나 순차 질의 수행 시 성능을 높일 수가 있다. 또한 데이터 아이템의 삽입과 삭제 시 중간노드를 거쳐서 GLB로 접근하는 T-트리와는 다르게 LUB로 직접 접근하여 불필요한 노드 순회를 막고 있다. 이때 LUB가 있는 노드 내의 데이터 아이템의 이동을 방지하기 위해서 T^*-트리는 T-트리와는 다르게 한 노드에 데이터 아이템들의 삽입 방향이 오른쪽에서 왼쪽으로 두고 있다

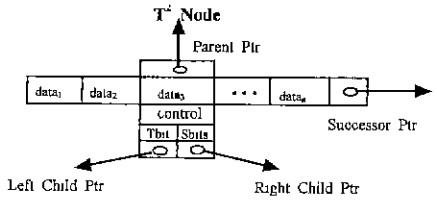
T^*-트리에는 다음과 같은 단점이 있다. 첫 번째, 후위 포인터 첨가로 인하여 노드의 삽입 및 삭제 시 포인터 연결을 위해서 자신을 가리키는 후위 포인터를 가진 노드로 거슬러 올라가야 한다. 두 번째, 데이터 아이템의 삽입과 삭제 시 LUB만을 살펴보기 때문에 여전히 노드의 삽입과 삭제 가능성이 높다. 세 번째, 노드 생성과 삭제 시 트리의 불균형 상태를 알아보기 위해서 항상 트리 깊이를 계산해 주어야 하고, 불균형 상태가 되면 회전 연산을 수행해야 한다는 단점이 있다.

3. T^2-트리

앞에서 살펴본 T-트리와 T^*-트리의 단점을 보완하기 위해서 본 논문에서는 억제된 노드 생성 및 삭제와 단말노드의 스레드 포인터를 이용한 새로운 색인 구조인 T^2-트리를 제안한다

3.1 구조

T^2-트리는 T^*-트리의 구조에서 스레드 이진 트리의 특성을 결합한 색인 구조이다. [그림 3]는 T^2-트리의 노드 구조를 나타낸다



[그림 3] T^2-트리의 노드 구조

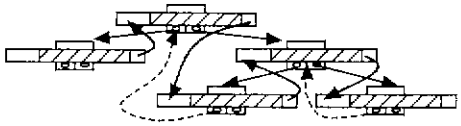
T^2-트리는 왼쪽 자식 포인터에 대해서 스레드를 적용하고 있다. 포인터를 구별하기 위해 Tbit를 컨트롤 박스에 두고 있다. 만약에 0 값을 가지면 실제 자식을 가리키는 포인터이며, 1값을 가지면 스레드 포인터가 되어서 현재 노드의 전위 포인터(predecessor pointer)가 된다. 이 전위 포인터는 T^*-트리의 단점 중 후위 포인터의 오버헤드를 해결하고 있다. 왜냐하면 Tbit가 1을 가진 노드는 단말노드이고 노드의 생성과 삭제는 단말노드에서만 이루어진다. 그래서 자신을 가리키는 후위 포인터를 가진 노드를 찾기 위해 트리를 거슬러 올라가지 않고도 직접 그 노드를 접근할 수 있다

T^2-트리는 데이터의 삽입과 삭제 시 해당 노드에서 오버플로나 언더플로가 발생하면 GLB 노드와 LUB 노드의 상태를 비교하여 적당한 노드로 접근하게 된다. 이 상태를 파악하기 위해서 Sbits와 후위 포인터를 두고 있다. 물론 후위 포인터는 T^*-트리와 마찬가지로 범위 질의와 순차 질의 수행 시에도 사용한다. Sbits는 자신을 가리키는 후위 포인터를 가진 노드의 상태를 나타내며 다음과 같은 세 가지 상태가 있다

[상태 1] 노드내의 아이템의 개수가 점유 범위 최대수인 상태
[상태 2] 노드내의 아이템의 개수가 1인 상태
[상태 3] 그 외 상태

이러한 상태 정보는 T-트리와 T^*-트리에서 데이터 아이템의 삽입과 삭제 시 보다 노드 생성과 삭제를 억제시킬 수가 있으며 노드 생성과 삭제 시 균형을 맞추기 위해서 드는 오버헤드를 줄일 수가 있다

T^2-트리에서는 T^*-트리와 마찬가지로 노드 내의 데이터 아이템의 삽입은 오른쪽에서 왼쪽으로 채워지게 된다. 이것은 LUB 노드로 현재 노드의 최대 데이터 아이템의 재삽입 시 LUB 노드에 있는 데이터 아이템의 전체 이동을 막기 위해서이다. [그림 4]은 T^2-트리의 전체적인 구성도를 나타낸 것이다



[그림 4] T^2-트리의 전체 구성도

본 논문에서는 설명을 용이하게 하기 위해서 후위 포인터가 가리키는 노드를 후위노드라고 부르고, 스레드 포인터가 가리키는 노드를 전위노드라고 부르겠다

3.2 검색 알고리즘

T^2-트리의 검색은 기존의 T-트리와 같으며 이진 트리에서의 검색과 유사하다. 다른점은 이진 트리에서 한 개의 값으로 비교하는 반면, T^2-트리에서는 최소값과 최대값을 가지고 비교한다는 점이다.

T^2-트리에서 데이터 아이템 X를 검색하는 알고리즘은 다음과 같다

```
S1  검색은 항상 트리의 루트(root)부터 시작한다.
S2  if (X < 노드의 가장 작은 값)
      then 왼쪽 서브트리로 이동, 계속 검색한다
    else if (X > 노드의 가장 큰 값)
           then 오른쪽 서브트리로 이동, 계속 검색한다
    else 현재 노드내에서 이진탐색으로 X를 찾는다
```

한 노드내에서 아이템을 찾지 못하거나 검색하려는 값이 바운드 노드를 찾지 못하게 되면 검색을 실패하게 된다.

3.3 삽입 알고리즘

T^2-트리에서 데이터 아이템 X를 삽입하는 알고리즘은 다음과 같다.

```
S1  아이템을 삽입할 위치인 바운딩 노드를 검색한다
S2  if (바운드 노드가 존재한다면)
      if (노드 내의 삽입 공간이 있으면)
          then 아이템 X를 적절 위치에 삽입 후 종료한다
      else if (Sbits가 [상태 1]이 아니고 후위노드에 삽입할 공간이
               없다면)
          then 노드내의 가장 작은 값을 제거, 임시장소에 저장하
               고 아이템 X를 노드 내의 적절한 위치에 삽입한 후,
```

중위순회를 통해 GLB 노드로 이동, 임시장소의 값을 최대값으로 삽입한다

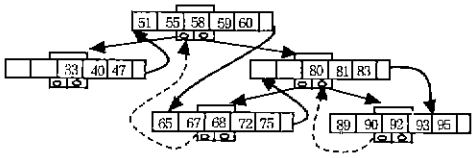
else 노드 내의 가장 큰 값을 제거, 임시장소에 저장하고 이 아이템 X를 노드 내의 적절한 위치에 삽입한 후, 후위포인터를 따라 LUB 노드로 직접 이동, 임시 장소의 값을 최소값으로 삽입한다.

S3 *else if* (검색경로상의 마지막 노드인 단말노드에 삽입할 공간이 있으면) */* 바운딩 노드를 찾지 못한 경우 */*
then 단말노드에 저장한다
else 새로운 단말노드를 생성하고, 후위 포인터 변경 또는 추가와 스레드 포인터의 변경 후, 아이템 X를 새로운 노드의 가장 오른쪽 공간에 최소값으로 저장한다

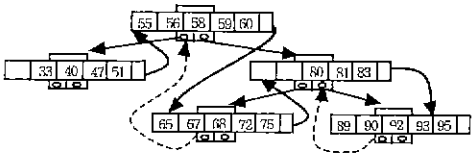
S4 *if* (새로운 단말노드 생성) *then*
while (단말노드부터 루트까지의 경로에 대하여)
do { 트리의 균형을 검사하여
if (길이의 차이가 1보다 크게되어 불균형 발생)
then 회전 연산 수행한다
}

삽입 알고리즘에서 데이터 아이템을 해당 노드에 삽입 시 노드의 상태를 체크하여 후위노드의 Sbits의 상태 값을 변경시켜 준다

[그림 5]와 [그림 6]은 T^2-트리에 56값을 삽입 시 노드 생성이 억제됨을 예를 들어 보여준다



[그림 5] 56값의 삽입 전 (최대수 = 5, 최소수=3)



[그림 6] 56값의 삽입 후 (최대수 = 5, 최소수=3)

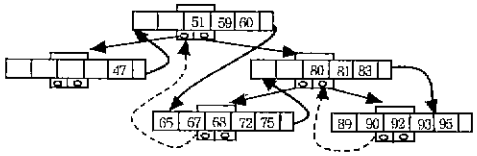
3.4 삭제 알고리즘

T^2-트리에서 데이터 아이템 X를 삭제하는 알고리즘은 다음과 같다.

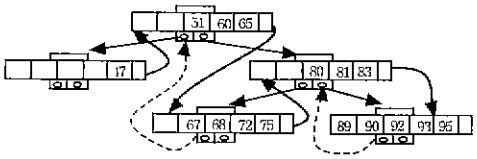
S1 삭제할 값의 바운딩 노드를 검색한다
S2 *if* (노드 안에 삭제할 값이 없다면)
else if (삭제로 인하여 언더플로가 야기되지 않는다면)
then 그 값을 삭제하고 종결한다.
else { */* 언더플로 야기 */*
if (현노드가 내부노드인 경우)
then 아이템을 삭제한다
if (Sbits가 [상태 2]가 아니고 후위노드의 아이템이 한 개 있다면)
then GLB 노드로부터 최대값을 가지고 온다
else LUB 노드로부터 최소값을 가지고 온다
else if (단말노드인 경우)
then 아이템을 삭제한다.
}
S3 *if* (반단말노드이면서 다른 단말노드와 결합 시킬 수 있다면)
then 두 개의 노드를 한 노드로 병합하고 다른 한 노드는 버린다 후위 포인터와 스레드 포인터를 조정한 후, S5를 수행한다
S4 *if* (단말노드가 삭제로 인하여 비지 않으면)
then 종결한다
else 빈노드를 소거하고 후위 포인터와 스레드 포인터를 변경시키고, S5를 수행한다
S5 *if* (단말노드가 생성) *then*
while (단말노드부터 루트까지의 경로에 대하여)
do { 트리의 균형을 검사하여
if (길이의 차이가 1보다 크게되어 불균형 발생)
then 회전 연산 수행한다
}

삭제 알고리즘에서 데이터 아이템을 해당 노드에서 삭제 시 노드의 상태를 체크하여 후위노드의 Sbits의 값을 변경시켜 준다.

[그림 7]와 [그림 8]은 T^2-트리에 59값을 삭제 시 노드 삭제가 억제됨을 예를 들어 보여준다



[그림 7] 59값의 삭제 전 (최대수 = 5, 최소수=3)



[그림 8] 59값의 삭제 후 (최대수 = 5, 최소수=3)

3.5 회전 연산

T^2-트리에서의 회전 연산은 T-트리와 같다. 이는 후위 포인터와 단말노드의 왼쪽 포인터인 스레드 포인터가 노드간의 순서 관계를 유지함으로써, 트리 균형을 위해서 회전 연산을 하더라도 이러한 관계가 계속 유지되기 때문에 별도로 고려할 필요가 없다

4. 결론

본 논문에서는 기존의 주기억 데이터베이스 색인 구조인 T-트리와 T^*-트리의 단점을 보완한 T^2-트리를 제안했다 T^2-트리는 억제된 노드 생성과 삭제로 트리 균형에 드는 오버헤드의 회전 연산을 억제한다. 후위 포인터를 통한 범위 질의, 순차 질의의 빠른 수행이 가능하며, 노드 생성과 삭제 시 후위 포인터의 조정을 단말노드의 스레드 포인터를 통해서 추가적인 오버헤드 없이 가능해졌다 또한 요구되는 메모리 공간도 T-트리와 T^*-트리에 비해서 큰 차이가 없다 따라서 T^2-트리는 동적 주기억 데이터베이스를 위한 효율적 색인 구조이다

참고 문헌

[1] T J Lehman and M J Carey, "A Study of Index Structures for Main Memory Database Management Systems", *Proc 12th Conf Very Large Data Bases*, Kyoto, August, 1986, pp 294-303
[2] 최공림, "주기억 데이터베이스의 질의처리를 위한 저장구조 및 최적화 전략"", 홍익대학교 박사학위 논문, 1996