CST-트리 인덱스의 빠른 구축

이재원[0] 이익훈 김현철 이상구
서울대학교 대학원 전기·컴퓨터 공학부
{lyonking[0], ihlee, hckim, sglee}@europa.snu.ac.kr

Seoul National University

Jae-won Lee[0] Ig-hoon Lee, Hyun-chul Kim, Sang-goo Lee

School of Computer Science & Engineering, Seoul National University

요 약

기술의 발달로 인하여 컴퓨터에 사용되는 메모리가 대용량화되고, 가격이 저렴해지면서 메인 메모리 데이터베이스 시스템이 주목을 받고 있다. 메인 메모리 데이터베이스 시스템은 디스크 기반 데이터베이스 시스템에 비해 디스크 접근을 줄임으로써, 좀 더 빠른 트랜잭션 처리를 보여주고 있다. 그러나 전원 차단과 같은 장애 발생 시, 메모리의 휘발성으로 인한 데이터 손실에 항상 대비를 해야 한다. 증권, 통신사와 같이 실시간 서비스가 이루어지고, 시스템 장애가 큰 손실로 이어지는 곳에서는 장애 발생 시 데이터의 빠른 복구를 필요로 하게 된다. 본 논문은 메인 메모리 데이터베이스 시스템에서 CSB+-트리(Cache Sensitive B-tree)보다 좋은 성능을 보이는 CST-트리(Cache Sensitive T-tree)에서 사용할 수 있는 인덱스의 빠른 구축 기법을 제안한다.

1. 서 론

일반적으로 데이터베이스 시스템은 데이터를 디스크에 저장한다. 디스크 속도는 메모리에 비해 느리지만 가격이 저렴하고 안정적이기 때문에 많이 사용되고 있다. 한편 증권, 통신사와 같이 실시간 서비스가 이루어지고 있는 곳에서는 좀 더 빠른 처리 성능을 필요로 하고 있다. 이와 같은 요구 사항은 기술의 발달과 더불어 메인 메모리 데이터베이스 시스템에 대한 관심을 불러 일으키고 있다. 메모리가 점점 대용량화되고, 가격이 저렴해 지면서 데이터베이스 시스템에서 중요한 기술로 주목을 받고 있는 것이다. 그러나 메인 메모리 데이터베이스 시스템은 전원 차단과 같은 시스템 장애가 발생했을 때, 문제의 소지가 많다. 실시간 서비스가 이루어지고 있는 곳에서는 시스템 장애 발생이 큰 손실로 이어질 수 있으므로 빠른 복구는 의미를 지닌다고 볼 수 있다.

본 논문은 메인 메모리 데이터베이스 시스템에서 장애 발생 시, 빠른 복구를 위한 알고리즘을 제안한 것으로 CST-트리(Cache Sensitive T-tree)에서 사용되는 복구 알고리즘을 제안한다.

본 논문은 다음과 같이 구성되어 있다. 2장에서 B+-트의 재구축 알고리즘 및 CST-트리에 대한 설명을 하며, 3장에서 CST-트리를 위한 인덱스의 재구축 알고리즘을 설명한다. 4장에서는 결론 및 향후 과제를 제시한다.

2. 관련 연구

2.1 B+-트리의 재구축 알고리즘

최근 CPU기술의 발달로 CPU의 속도가 메모리 속도에 비해 빨라짐에 따라 B-트리[1]의 성능이 T-트리보다 뛰어나게 되었다[2]. 즉, 메모리 속도의 향상 정도가 CPU 속도의 향상 정도를 따르지 못하여 메모리 접근 수가 적은 B-트리가 성능이 더욱 좋게 된 것이다. [3]에서는 B-트리의 계열인 B+-트리의 인덱스 구축 방법으로 순차 삽입 방식과 일괄 구성 방식을 제안하였다. CST-트리[4] 역시 빠른 복구를 위해 B+-트리와 같은 인덱스 구축 방법이 필요하다.

▸ 본 논문은 ITRC(Information Technology Research Center) 지원 프로그램에 의해 작성되었습니다.

2.1.1 순차 삽입 방식

기본적인 재구성 방법으로 B+-트리의 알고리즘에 따라 키를 하나하나 트리에 삽입하는 방식이다[3]. 이 방식은 키 삽입마다 루트로부터 자신의 위치를 찾아 들어가야 한다. 또한 말단 노드가 꽉 차면 분할을 해야 하며, 이는 다시 상위 노드까지 전파가 되어 상당한 오버헤드를 초래한다. 이 방법은 병렬 처리가 가능한데, 데이터베이스로부터 데이터를 읽어 오는 단계와 트리 구조에 데이터를 삽입하는 단계를 병렬 처리함으로써 성능을 향상 시키는 것이 가능하다.

2.1.2 일괄 구성 방식

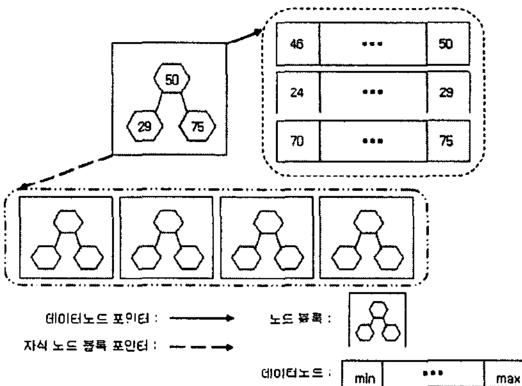
B+-트리가 루트 노드로부터 모든 말단 노드까지의 높이가 일정하다는 점을 이용하여, 일괄적으로 B+-트리를 구축하는 것이 가능하다[3]. 순차 삽입 방식과는 달리 말단 노드부터 키를 삽입하고, 그 상위 노드를 작성하는 바텀-업(bottom-up) 방식으로써 노드 분할과 상위 노드로의 전파 과정이 필요 없기 때문에 좋은 성능을 보일 수 있다. 그러나 이 방식은 데이터를 데이터베이스로부터 모두 가져온 후, 정렬을 해야 하므로 데이터를 읽어 오는 단계와 데이터를 삽입 하는 단계를 병렬 처리하는 것이 어렵다. 본 논문은 일괄 구성 방식의 장점인 노드 분할과 상위 노드로의 전파가 불필요함에 주목하여 이 방식을 CST-트리에도 적용한다.

2.2 CST-트리(Cache Sensitive T-tree)

[4]에서는 CSB+-트리[5]에 착안하여 캐시를 효율적으로 사용하는 인덱스 구조인 CST-트리를 제안하였다. CST-트리는 캐시의 효율을 높이기 위해 다음과 같은 특징을 갖는다.

- 캐시에 옮겨지는 데이터의 가능한 많은 부분을 사용
- 포인터의 사용을 최대한 제거하여 배열로 트리를 구성
- 노드 블록 크기가 캐시 블록 크기일 때, 최대의 성능을 보임

CST-트리의 구조를 도식화한 것이 [그림1]이다.



[그림1] CST-트리의 구조

[그림1]은 캐시 블록의 크기가 16 바이트인 경우이다. 노드 블록과 노드 블록 사이의 연결은 포인터를 이용하고 있으며, 노드 블록 내의 키 값들은 데이터 노드의 최대값을 갖고 있다. 그러므로 노드 블록 내의 각 노드들은 해당 데이터 노드로 가는 포인터를 가지고 있다. 본 논문은 인덱스를 재구축함에 있어 CST-트리에 사용할 수 있는 인덱스 재구축 방법을 제안한다.

3. 제안 기법

3.1 CST-트리의 인덱스 재구축 알고리즘

CST-트리에서 사용하고자 하는 인덱스 재구축 알고리즘은 3 단계로 나눈다.

1단계) 삽입할 데이터를 정렬 시킨다.

평균 시간 복잡도가 $O(nlogn)$ 인 정렬 알고리즘으로는 히프정렬, k-way 합병 정렬, 퀵정렬 등이 있다. 히프정렬의 경우 데이터를 정렬하기 위해 별도의 히프를 구성해야 하는 단점이 있다. k-way 합병 정렬은 자료의 크기의 2배의 공간이 필요하다는 단점이 있다. 반면에 퀵정렬은 별도의 구성 단계 및 추가 공간 없이도 같은 시간 복잡도를 가지므로 일반적으로 이 방법을 많이 사용한다. 본 논문은 퀵정렬을 사용하여 데이터를 정렬한다.

정렬 알고리즘의 선택은 CST-트리의 인덱스 구축 시, 고려의 대상이 아니다. 그러므로 $O(nlogn)$의 시간 복잡도를 갖는 정렬 알고리즘이라면, 어떤 정렬 알고리즘을 선택하여도 상관이 없다.

2단계) 빈 트리를 생성한다.

빈 트리를 생성하는 것은 노드 분할 과정과 상위 전파를 피하기 위해 필요한 노드 블록을 미리 생성하는 것이다. 이를 위해 필요한 정보는 다음과 같이 계산한다.

$$\#ofNodeInBlock : \frac{SizeCacheBlock}{SizeOfInt} - 1$$

$$\#ofNodeBlock : \left\lceil \frac{n}{\#\ ofNodeInBlock * \lfloor \#\ ofData * R \rfloor} \right\rceil$$

$$\text{높이} : \left\lceil \frac{\#\ ofNodeBlock}{SizeCacheBlock} * SizeOfInt \right\rceil$$

$\#ofNodeInBlock$: 노드 블록 안에 있는 노드의 개수
$SizeCacheBlock$: 캐시 블록 크기
$SizeOfInt$: integer 크기 (4바이트)
n : 삽입되는 데이터의 개수
$\#OfData$: 한 노드 안에 포함되는 데이터의 개수
R : 데이터 노드를 채우는 비율

3단계) 정렬된 데이터를 삽입하며 인덱스를 구축한다.

B+-트리는 데이터가 말단 노드에만 삽입이 되며 상위 노드는 말단 노드로 데이터를 찾아가기 위한 인덱스 값을 갖고 있다. 또한 말단 노드가 연결 리스트로 연결되어 있기 때문에 말단 노드의 왼쪽부터 오른쪽으로 데이터를 삽입하는 일괄 구성 방식을 이용한다[3]. 그러나 CST-트리의 경우, B+-트리와 달리 데이터가 중간 노드에도 삽입된다. 그러므로 삽입될 데이터의 중간 노드 위치를 찾기 위한 방법이 필요하다. 본 논문은 이를 위해 중위 순회법을 사용한다. 즉, 가장 왼쪽에 있는 말단 노드 블록으로부터 시작하여 가장 오른쪽에 있는 노드 블록까지 중위 순회법을 따라 데이터를 삽입한다.

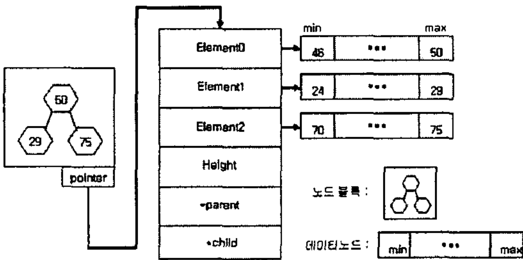
본 논문에서 이를 구현한 방법은 다음과 같다[그림2].

```
if(index == leafNodeOfNodeBlock)//인덱스가 말단노드일 때
   IndexLeftChildNodeBlock = 2*index+1-#ofNodeInBlock
   IndexRightChildNodeBlock = 2*index+2-#ofNodeInBlock
   if(leftChildNodeBlock(IndexLeftChildNodeBlock) != null)
    //왼쪽 자식 노드에 데이터가 없을때
        while(leftChildNodeBlock->height != 1)
            //왼쪽 말단 노드 블록을 계산
            temp = FCP(temp) + left
        end while
        InsertLeftChildNode(temp)//왼쪽 자식 노드 삽입
        InsertParentNode() //부모 노드에 삽입
        temp = temp->parent
        while(rightChildNodeBlock->height != 1)
            //오른쪽 말단 노드 블록을 계산
            temp = FCP(temp)+right
        end while
        InsertRightChildNode(temp)//오른쪽 자식노드 삽입
        ++index
   else //왼쪽 자식 노드에 데이터가 있을 때
        InsertParentNode()
        while(rightChildNodeBlock->height != 1)
            //오른쪽 말단 노드 블록을 계산
            temp = FCP(temp)+right
```

```
        end while
        InsertRightChildNode()
        ++index
    end if
else
    InsertParentNode() //인덱스가 말단 노드가 아닐 때
    ++index
end if
```

[그림2] 중위 순회법 알고리즘

각 노드 블록에서 데이터를 삽입하는 방법은 다음과 같다.



[그림3] 각 노드 블록의 세부 구현

[그림3]은 캐시 블록의 크기가 16바이트인 경우이다. 정렬된 데이터 배열을 데이터 노드의 크기만큼씩 분할한 후, 각각의 최대값을 노드 블록에 저장한다. 분할된 각각의 데이터 배열은 해당 데이터 노드에 배열 복사를 통해 삽입한다. 여기서 일괄 구성 방식의 병렬화가 가능하다. 즉, 노드 블록 내에서 노드의 위치를 찾는 단계와 데이터 노드에 데이터 배열을 삽입하는 단계를 병렬화 하는 것이 가능하다. 본 논문에서는 병렬화 처리를 향후 과제로 남기기로 하고 일단 논하지 않기로 한다.

3.2 각 인덱스 구조의 구축 방법 비교

B+-트리[3]의 경우 인덱스를 재구축하기 위해 루트로부터 데이터를 삽입하는 순차 삽입 방식과 말단 노드부터 삽입하여 상위 노드를 작성하는 일괄 구성 방식이 있다. 순차 삽입 방식의 경우 톱-다운(top-down) 방식을 채택하고 있으나, 노드 분할과 같은 오버헤드를 갖는다. 일괄 구성 방식은 바텀-업(bottom-up) 방식으로 데이터를 삽입하며, 데이터를 정렬해야 하는 단점이 있다.

CSB+-트리[5]의 경우, B+-트리의 일괄 구성 방식을 사용하고 있다. 미리 필요한 말단 노드의 개수를 계산한 후, 말단 노드의 왼쪽부터 오른쪽으로 데이터를 채운다. 말단 노드 블록의 최대값들은 다시 상위 노드 블록을 채우게 되며 이는 루트 노드에 도달할 때까지 반복이 된다. 이를 정리하면 [그림4]와 같다.

	B+-트리		CSB+-트리	CST-트리
	순차 삽입	일괄 구성		
구성방식	top-down	bottom-up	bottom-up	bottom-up
데이터삽입	말단노드	말단노드	말단노드	전체노드
트리성장	유	무	무	무

[그림4] 인덱스 구조의 비교

3.3 시간 복잡도

알고리즘은 크게 3 단계로 나눌 수 있다.

1단계) 빈 트리를 생성
2단계) 삽입할 데이터를 정렬
3단계) 정렬된 데이터를 삽입하며 인덱스를 구축

먼저 데이터를 정렬하는 단계에서의 시간 복잡도는 퀵정렬을 사용하고 있으므로 $O(nlogn)$ 이다. 다음으로 빈 트리 생성 단계에서의 시간 복잡도를 구한다. 각 노드 블록의 개수는

$$\left\lceil \frac{n}{\#\ ofNodeInBlock * \#\ ofData} \right\rceil \text{ 이다.}$$

이를 정리하면 $O(n)$ 이다.

정렬된 데이터를 삽입하며 인덱스를 구축하기 위한 시간 복잡도는 다음과 같다. CST-트리는 하나의 노드에 s 개의 키가 저장될 경우, n/s(n:전체 데이터 개수)로 이진 트리를 만들며, 트리의 높이는 $\log_2 \frac{n}{s}$ 이다. 데이터 노드에서 데이터를 검색하는데 걸리는 시간은 $\log_2 s$이므로 하나의 데이터를 검색하는데 걸리는 시간은 $\log_2 \frac{n}{s}$ + $\log_2 s$이다. n 개의 데이터를 삽입/재구축하는데 걸리는 전체 시간은 $n(\log_2 \frac{n}{s} + \log_2 s)$ 이며, 시간 복잡도는 $O(nlogn)$이다.

위의 3단계에서 구한 시간 복잡도를 더하면 전체 재구축에 걸리는 시간 복잡도를 구할 수 있으며, 이는 $O(n)$ + $O(nlogn)$ + $O(nlogn)$ = $O(nlogn)$이다.

4. 결론 및 향후 과제

지금까지 메인 메모리 데이터베이스 시스템에서 시스템 장애 시 빠른 복구를 위해 사용될 수 있는 CST-트리 인덱스의 재구축 기법을 보았다. 또한 각 인덱스 구조의 구축 방법 및 시간 복잡도를 분석했다. 현재 알고리즘만을 제시하였으나, 구현을 통해 실제 구축되는 성능을 측정/실험 하도록 하는 것은 향후 과제로써 남긴다. 또한 본 논문에서는 제외하였던 병렬처리를 위한 기법 역시 좀 더 연구할 과제로 남긴다.

5. 참고 문헌

[1] D. Comer, The Ubiquitous B-Tree, Computing Surveys 11, 2 (June, 1979)
[2] Jun Rao, et al, Cache Conscious Indexing for Decision-Support in Main Memory, VLDB 1999
[3]SangWook Kim, HeeSun Won, "Batch Construction of B+-trees", Proc. 2001 ACM Symposium on Applied Computing, pp. 231-235, 2001
[4] 이익훈, 캐시를 고려한 T-트리 인덱스 구조, 정보과학회 논문지, 2004 submitted
[5] Jun Rao, K. A. Ross, Making B+-Trees Cache Concious in Main Memory, ACM, 2000