최대키 값을 이용한 CST-트리 인덱스의 빠른 재구축

이재원° 이익훈 이상구
서울대학교 대학원 전기컴퓨터 공학부
{lyonking°, ihlee, sglee}@europa.snu.ac.kr

Fast Construction of CST-Tree Indexs Using Max-Key Values

Jae-won Lee° Ig-hoon Lee, Sang-goo Lee
School of Computer Science & Engineering, Seoul National University

요 약

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 백업 시에 인덱스의 최대키 값을 따로 저장하는 추가 작업을 통하여, 병렬처리가 가능한 CST-트리(Cache Sensitive T-tree)의 인덱스 복구 알고리즘을 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 불러일으키고 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 위해 백업 시에 최대키 값을 따로 저장하는 추가 작업을 감수하지만 병렬 처리가 가능한 CST-트리(Cache Sensitive T-tree)의 인덱스 복구 알고리즘을 제안한다.

본 논문은 다음과 같이 구성되어 있다. 2장에서 B+-트의 복구 알고리즘에 대한 설명을 하며, 3장에서 CST-트리를 위한 인덱스의 복구 알고리즘을 설명한다. 4장에서 이에 대한 실험 결과를 보인다. 마지막으로 5장에서 결론을 제시한다.

2. 관련 연구

2.1 B+-트리의 복구 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 재구축 방법으로 순차 삽입 방식과 일괄 구성 방식을 제안하였으며, [4]에서는 병렬 처리를 이용한 인덱스 재구축 방법을 제안하였다. CST-트리[5] 역시 빠른 복구을 위해 B+-트리와 같은 인덱스 재구축 방법이 필요하다.

2.1.1 순차 삽입

기본적인 재구성 방법으로 B+-트리의 알고리즘에 따라 키를 하나하나 트리에 삽입하는 방식이다[3]. 이 방식은 키를 삽입할 때마다 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 키 값을 읽어 오는 단계와 트리 구조에 키 값을 삽입하는 단계를 병렬 처리함으로써 성능을 향상 시키는 것이 가능하다.

2.1.2 일괄 구성

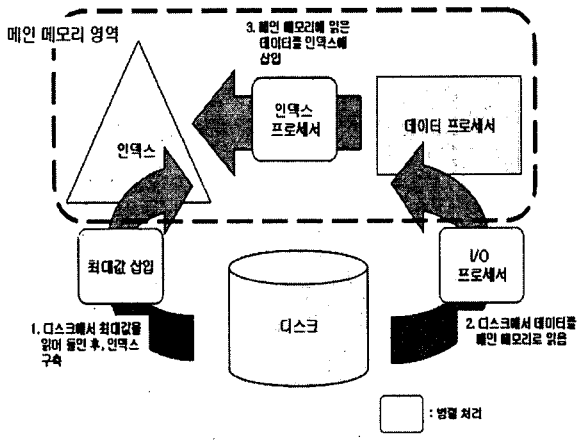
B+-트리가 루트 노드로부터 모든 말단 노드까지의 높이가 일정하다는 점을 이용하여, 일괄적으로 B+-트리를 재구축하는 것이 가능하다[3]. 순차 삽입 방식과는 달리 말단 노드부터 키를 삽입하고, 그 상위 노드를 작성하는 바텀-업(bottom-up) 방식으로써 노드 분할과 상위 노드로의 전파 과정이 필요 없기 때문에 좋은 성능을 보일 수 있다. 그러나 이 방식은 키 값을 데이터베이스로부터 모두 가져온 후, 정렬을 해야 하므로 키 값을 읽어 오는 단계와 키 값을 삽입 하는 단계를 병렬 처리하는 것이 어렵다.

2.1.3 병렬 삽입

백업 시에 인덱스의 각 말단 노드 내의 최대키 값들을

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

별도의 디스크 공간에 저장한다. 인덱스 재구축 시에는 저장된 말단 노드의 최대키 값을 메모리에서 읽어 와서 말단 노드의 공간을 확보한 후, 말단 노드의 상위 노드 단계를 구축한다. 트리의 루트 노드까지 구축되면, 말단 노드 내부에 키 값과 포인터가 빠진 인덱스 구조가 만들어진다. 인덱스를 구축하기 위해 빠진 키 값과 포인터를 채워야 하는데 이는 키 값을 구축하면서 처리한다. 이 방식은 디스크로부터 키 값을 구축하는 단계와 키 값을 인덱스 구조에 반영하는 단계를 병렬화 하는 것이 가능하다. 병렬 삽입은 순차 삽입에 비해 노드 분할에 따른 오버헤드가 감소하며, 일괄 구성에 비해 정렬을 위한 비용이 줄어들기 때문에 효율적이다. 이를 도식화하면 다음과 같다.



[그림1] 인덱스 병렬 복구

3. 제안 기법

CST-트리 인덱스 순차 삽입 알고리즘은 [5]에서 제안하고 있으며, 기본 알고리즘은 다음과 같다.

1단계) 검색을 통해 삽입할 노드를 찾아 삽입한다.

2단계) 삽입할 노드를 찾았으나, 삽입할 공간이 없으면 최소값을 빼고 그 자리에 삽입한다. 최소값을 현재 노드의 왼쪽 서브 트리에 삽입하며, 노드가 없으면 노드를 추가한다.

3단계) 노드가 추가된 단말 노드 블록 내에서 노드 밸런스를 검사한다. 만약 노드 추가로 새 노드 블록이 생성된 경우, 노드 블록 밸런스를 검사한다.

3.1 CST-트리 인덱스 일괄 삽입 알고리즘

CST-트리 인덱스의 일괄 삽입 알고리즘은 B+-트리의 일괄 삽입 알고리즘과 유사하며, 3 단계로 나눈다.

1단계) 삽입할 데이터를 정렬 시킨다.

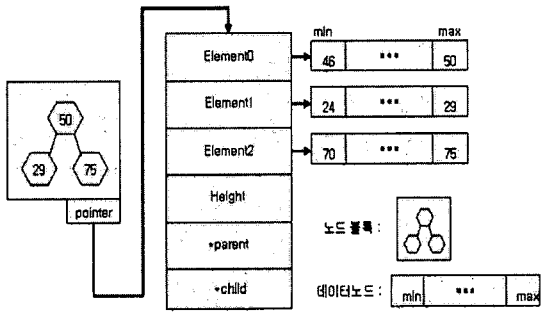
평균 시간 복잡도가 $O(nlogn)$ 인 정렬 알고리즘으로는 히프정렬, 합병 정렬, 퀵정렬 등이 있다. 히프정렬의 경우 데이터를 정렬하기 위해 별도의 히프를 구성해야 하는 단점이 있다. 합병 정렬은 자료의 크기의 2배의 공간이 필요하다는 단점이 있다. 반면에 퀵정렬은 최악의 경우, $O(n^2)$ 의 시간 복잡도를 가지지만, 별도의 구성 단계 및 추가 공간 없이도 같은 평균 시간 복잡도를 가지므로 이 정렬 알고리즘을 사용한다.

2단계) 인덱스 구조를 생성한다.

인덱스 구조를 미리 생성하는 것은 노드 분할 과정과 상위 전파를 피하기 위해 필요한 노드 블록을 미리 생성하는 것이다.

3단계) 정렬된 데이터를 삽입하며 인덱스를 재구축한다.

B+-트리는 데이터가 말단 노드에만 삽입이 되며, 상위 노드는 말단 노드로 데이터를 찾아가기 위한 인덱스 값을 가지고 있으므로 바텀-업(bottom-up) 일괄 구성 방식을 이용한다. 그러나 CST-트리의 경우, B+-트리와 달리 데이터가 중간 노드에도 삽입된다. 그러므로 삽입 위치를 찾기 위한 방법이 필요하다. 본 논문은 이를 위해 중위 순회법을 사용한다. 각 노드 블록에서 데이터를 삽입하는 방법은 다음과 같다.



[그림2] 각 노드 블록의 세부 구현

[그림2]는 캐시 블록의 크기가 16 바이트인 경우이다. 정렬된 데이터를 데이터 노드의 크기만큼씩 분할한 후, 각각의 최대키 값을 노드 블록에 저장한다. 분할된 각각의 데이터 배열은 해당 데이터 노드에 배열 복사를 통해 삽입한다.

3.2 CST-트리 인덱스 병렬 삽입 알고리즘

병렬 삽입 알고리즘은 병렬 처리를 하기 위해 백업 시, 인덱스의 최대키 값을 별도의 저장 공간에 저장하는 추가 작업을 한다. 즉, 인덱스 구축을 좀 더 빠르게 하기 위해 필요한 추가 정보를 백업 할 때 저장한다.

병렬 삽입 알고리즘은 순차 삽입 알고리즘에 비하여 노드 분할에 대한 처리를 해야 할 필요가 없어지고, 일괄 삽입 알고리즘에 비하여 정렬을 위한 비용이 줄어들기 때문에 효율적이다. 병렬 삽입 알고리즘은 다음과 같이 4 단계로 나누며, 2,3 단계를 병렬 처리한다.

1단계) 최대키 값을 이용하여 인덱스를 재구축한다.

백업할 때 별도로 저장한 최대키 값들을 이용하여, 일괄 삽입과 마찬가지로 노드 블록을 생성한다. 각 노드 블록에 디스크로부터 읽은 최대키 값을 삽입하며, 데이터 노드는 비워 둔다.

2단계) 디스크로부터 데이터를 복구한다.

디스크로부터 데이터를 읽어 데이터 배열에 저장한다. 병렬 삽입은 일괄 구성과 같이 데이터를 정렬할 필요가 없다.

3단계) 복구된 데이터를 인덱스 구조에 삽입한다.

순차 삽입하는 것처럼 루트 노드에서부터 검색을 하여 삽입할 노드의 위치를 찾는다. 최대키 값만을 이용하여 삽입할 노드의 위치를 검색하므로 다음과 같은 방법을 이용한다.

- 삽입할 노드의 위치를 찾으면 마크를 한다.
- 마크한 노드의 왼쪽 서브 노드의 최대키 값과 삽입할 키 값을 비교한다. 만약 왼쪽 서브 노드의 키 값보다 작으면 위의 단계를 반복 수행하며, 키 값보다 큰 경우, 마크한 노드 위치에 키 값을 삽입한다.

데이터 노드에 키 값을 삽입할 때 이진 검색을 통해 위치를 찾을 수도 있지만 매번 위치를 검색하여 삽입하는 것은 비용이 상당히 크다. 그러므로 데이터 노드에 들어오는 순서대로 삽입하도록 한다.

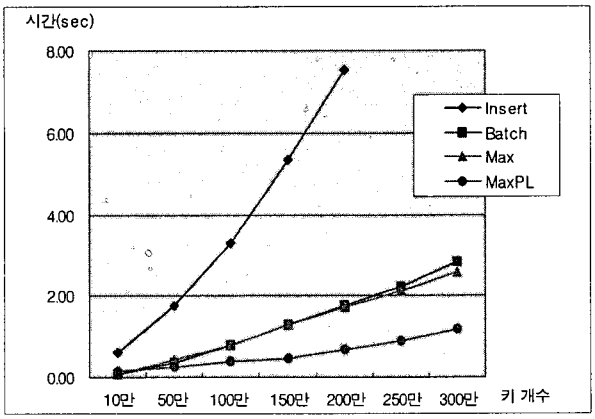
4단계) 각 노드블록의 데이터 노드에 삽입된 키 값들을 정렬한다.

각 데이터 노드에 키를 삽입할 공간이 더 이상 없을 경우, 정렬을 수행한다. 이때, 사용하는 정렬 알고리즘은 일괄 삽입과 마찬가지로 퀵정렬을 사용하여 정렬한다.

4. 실험 결과

본 실험은 두 가지로 나누어서 진행한다. 실험에 사용된 서버는 SUN-Fire-280R 이며, 중앙처리장치는 Ultra-SPARC III Cu 1.2 GHz, 메인메모리의 크기는 4GB, 캐시 라인의 크기는 512 Byte, 운영체제는 sun 5.9 이다.

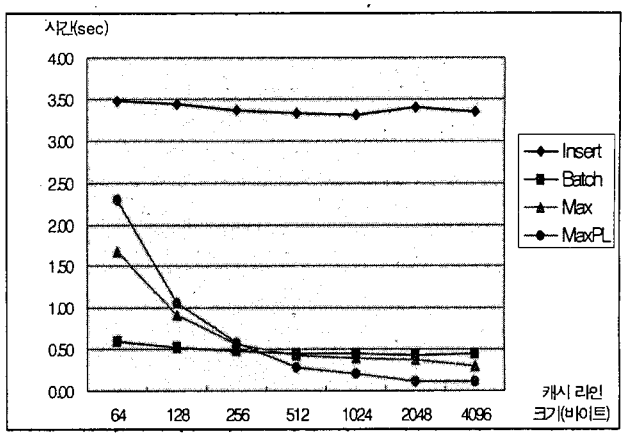
실험1) 키 값의 개수에 따른 재구축 성능을 비교한다. 이를 위해 캐시 라인의 크기를 1024 바이트로 고정시키고, 삽입하는 키 값의 개수를 변경한다.



[그림3] 삽입 키의 개수에 따른 성능 비교

[그림3]에서 순차 삽입 알고리즘은 키 값을 삽입할 때마다 밸런스를 검사하여야 하므로 삽입 시간이 오래 걸린다. 최대키 값을 이용한 삽입 알고리즘은 일괄 삽입 알고리즘과 비슷한 성능을 보이거나 좋은 성능을 보이고 있다. 그러나 최대키 값을 이용한 재구축 방법은 병렬처리를 함으로써 가장 좋은 성능을 보이고 있다.

실험2) 캐시 라인의 크기에 따른 재구축 성능을 비교한다. 이를 위해 삽입하는 키 값의 개수를 100만으로 고정시키고, 캐시 라인의 크기를 변경한다.



[그림4] 캐시 라인 크기에 따른 성능 비교

[그림4]에서 캐시 라인 크기가 커질수록 트리의 높이가 낮아지므로 키 값을 삽입하기 위한 트리 탐색 시간이 줄어든다. 또한 실험에 사용된 서버의 경우 프리패치(pre-fetch)를 사용하기 때문에 캐시 라인 크기가 커질수록 좋은 성능을 보이고 있다.

5. 결론

지금까지 메인 메모리 데이터베이스 시스템에서 시스템 장애 시 빠른 복구를 위해 사용될 수 있는 CST-트리 인덱스의 재구축 알고리즘을 보았다. 병렬 삽입 알고리즘의 경우 순차 삽입에 비해 평균 400%~1100% 의 성능 향상을 보이고 있으며, 일괄 삽입에 비해 평균 200~300 % 의 성능 향상을 보였다(성능 향상 비 = 기존기법 / MAXPL). 또한 캐시 라인의 크기가 커질수록 같은 개수의 키를 삽입하기 위한 트리의 높이가 낮아진다. 이는 다시 트리 탐색 시간을 줄이게 되므로 좋은 성능을 보인다.

6. 참고 문헌

[1] D. Comer, The Ubiquitous B-Tree, Computing Surveys 11, 2, 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] 김태훈, 이익훈, 이상구, 메인 메모리 데이터베이스에서 B+-트리 인덱스의 빠른 복구, KDBC 2004 (pp. 101-108),

[5] 이익훈, 캐시를 고려한 T-트리 인덱스 구조, 정보과학회 논문지:데이터베이스, 제 32권,2005