CST-트리의 효과적인 범위 검색

강대희° 이재원 이상구
서울대학교 대학원 전기·컴퓨터 공학부
{kdh200°, lyonking, sglee}@europa.snu.ac.kr

Efficient range search of CST-tree

Dae-hee Kang°, Jae-won Lee, Sang-goo Lee
School of Computer Science & Engineering, Seoul National University

요 약

기술의 발달로 CPU의 속도는 메모리의 속도에 비해 급속한 속도로 발전하였다. 그 결과 데이터베이스 시스템을 포함한 다른 컴퓨터 응용분야에서 메모리의 접근속도가 병목현상을 일으키게 되었다. 그래서 메모리의 접근 속도를 줄이기 위해 캐시 메모리가 도입되었고, 이를 활용하여 CPU 캐시를 효율적으로 활용하기 위한 많은 연구들이 있었고, 그 중 하나가 CST(Cache Sensitive T-tree)이다. 이 인덱스 구조는 점 검색(Point search)에서는 좋은 성능을 보이지만 범위 검색(range search)에서는 그렇지 못하다. 본 논문에서는 범위 검색(range search)을 위한 CST-tree에 대한 구축 기법을 제안한다.

1. 서 론

최근 인덱스 구조 연구에 있어서 중요한 이슈는 최신 정보처리시스템에서의 새로운 성능 병목 현상이다. 메모리의 용량은 빠르게 커져서 점점 더 많은 데이터를 메모리에 저장할 수 있게 되었다. 반면에, CPU의 속도는 무어의 법칙에 따라 18개월마다 2배 속도로 빨라지고 있지만 메모리속도는 그만큼 빨라지지 않고 있다. 즉, CPU와 메모리의 속도 차이로 인해 메모리 접근 병목현상이 발생하고 있다.[1,2,3] 이러한 CPU와 메모리 속도 차이는 점점 더 커질 것이다. 그래서 이를 극복하기 위해서는 CPU와 메모리 사이의 속도 격차를 극복하기 위해 만들어진 캐시 메모리를 적절하게 이용하여야 한다. 이를 활용한 메인메모리 인덱스 구조에 대한 연구로는 T-트리, B+-트리, CSS-트리,CSB+-트리와 본 연구실에서 연구했던 CST((Cache Sensitive T-tree)가 있다. 이러한 연구들은 CPU 캐시를 효율적으로 활용하기 위해 캐시 미스 수를 줄이기 위한 방법들이다.

본 논문에서는 이러한 방법들 중 보다 좋은 성능을 보인 CST트리[4]에서 효율적인 범위 검색(range search)에 대한 구축 기법을 제안하고자 한다.

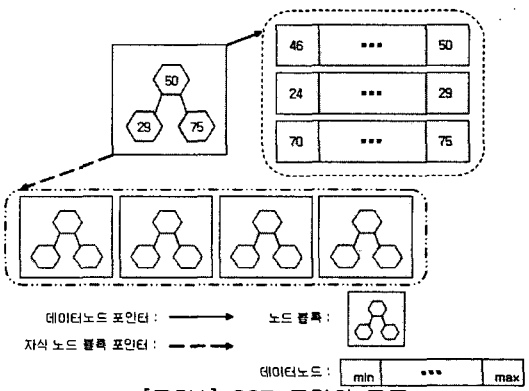
본 논문은 다음과 같이 구성되어 있다. 2장에서 CST에 대한 설명을 하고, 3장에서는 CST의 범위 검색에 대한 알고리즘을 설명하며, 마지막으로 4장에서는 결론 및 향후 과제를 제시한다.

2. 관련 연구

CST-트리(Cache Sensitive T-tree)[2]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.



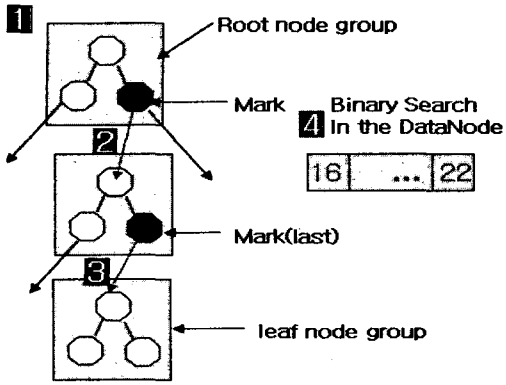
[그림1] CST-트리의 구조

[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.

CST-트리는 검색하는 동안 루트 노드 그룹으로부터 단말 노드그룹에 도착 할 때 까지, 노드그룹에서 이진 탐색 시에는 캐시 미스가 발생하지 않고(노드그룹의 크기가 캐시라인의 크기이므로), 다음 자식 노드그룹을 접근할 때와 데이터 노드를 접근 할 때만 캐시 미스가 발생

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

한다. 캐시 미스 회수는 "CST-트리의 높이 +1"만큼 발생한다.[그림2]



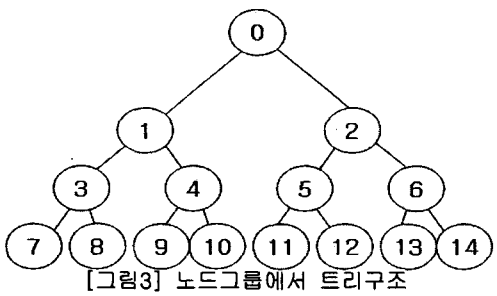
[그림2] CST-트리 검색 예

하지만 범위 검색(range search)시 현재 검색 알고리즘에서는 모두 루트 노드그룹의 0번째 노드에서부터 검색하므로 예를 들어 [그림2]에서 16에서부터 18까지 찾는다고 하면 총 캐시미스가 최대 (CST-트리의 높이(3)+1)*3=12까지 발생 할 수가 있다. 하지만 범위검색(range search)알고리즘에서는 tree의 순환을 통해 찾고자 하는 데이터가 같은 데이터 노드에 있을 경우 최소 (CST-트리의 높이(3)+1)*1=4까지 줄 일수 있다.

여기서 CST-트리는 주어진 캐시라인 크기 내에서 최대한 많은 엔트리를 포함하여 메모리 접근 횟수를 줄이기 위해 ,즉 노드의 필수 엔트리 최소화를 위해 포인터를 제거 하였다. 본 논문은 앞서 말한 포인터 제거에 따른 노드 그룹 내에서의 트리의 순환(traverse)을 위한 방법과 이를 가지고 범위 검색(range query)에 대한 알고리즘을 제시한다.

3. 제안 기법

3.1 노드그룹내에서의 트리의 순환



[그림3] 노드그룹에서 트리구조

캐시블록의 크기가 64byte인 경우에 노드그룹은[그림3]과 같은 노드그룹과 노드그룹사이를 연결하는 포인터 한 개로 구성된다.

데이터 저장 순서는 다음과 같다.

order[15] = { 7, 3, 8, 1, 9, 4, 10, 0, 11, 5, 12, 2, 13, 6, 14}

CST-트리는 포인터를 제거 했으므로 포인터 없이 해당 노드의 상위노드(현 노드 다음으로 큰 값이 저장된 노드)를 찾기 위한 방법은 다음과 같다.[그림4]

```
OrderNum orderSearch(nodeNum,k)
    //nodeNum: 노드그룹의 노드수
    //k:현재 노드의 위치
    h(height)=⌈ log 2(nodeNum)⌉;
    d(depth)=⌊ log 2(k+1)⌋;

    o r d e r N u m =
(2^(h-1-d)-1)+(k-(2^d-1))*2^(h-d);
    return OrderNum
END
```

[그림4] Ordering 알고리즘

[그림4]처럼 노드그룹의 노드수(nodeNum)와 현위치의 노드 인덱스 값(k)을 알면 order 배열에서 현 위치 노드에 대한 order 인덱스 값을 알 수 있다.

$$orderNum=\underbrace{(2^{(h-d)}-1)}_{A}+\underbrace{(k-(2^{d}-1))}_{B}*\underbrace{2^{((h+1)-d}}_{C}$$

여기서 A는 Order배열의 첫 번째 시작값을 구하는 식이고, B는 현 위치의 노드 인덱스(k)와 좌측값(0,1,3,7)과의 차이를 나타내는 것이고, C는 증가율(수)를 나타낸다.

예를 들어 현재 노드의 인덱스 값이 OrderNum이라면 다음 노드값은 order[OrderNum+1]이 되는 것이다.

3.2 CST-트리의 범위 검색(range search) 알고리즘

최소키 또는 최대키값만을 사용하여 검색하는 T-트리 검색 알고리즘[6]과는 달리 CST-트리 알고리즘은 최대값만을 사용하고, 인덱스가 노드 그룹과 데이터 노드로 이루어져 있기 때문에 일반적인 T-트리와 약간의 차이가 있다.

~~(1단계) 먼저 기존의 CST_Search알고리즘[7]에서 범위~~ 검색의 첫 번째 값을 찾는다.[그림5]

```
RID CST_Search ( key,tnode,order,max)
//key,max: a key to find, tree: a CST-tree
//RID: a record ID to be found
compareKey = get first key to compare in the root node group of tree
// 1st step: traverse node groups
while ( compareKey != NULL )
    if ( key <= compareKey )
     lastMarkedNode = data node corresponding to current key;
     compareKey = get the key of left subtree;
    else
     compareKey = get the key of right subtree;
     end if
end while

// 2nd step: binary search in a data node
if ( lastMarkedNode != NULL)
   dataNode = get the data node from lastMarkedNode;
   RID = binary search in the dataNode;
   return RID;
else
   return NOTFOUND;
end if
```

End

[그림5]CST_Search알고리즘

(2단계) CST-Search 알고리즘[그림5]에서 찾은 범위검색(range search)의 첫 번째 값의 위치를 기점으로 찾고자하는 다음 값을 CST_RangeSearch 알고리즘[그림6]을 통해서 찾는다. 이때 CST-트리는 포인터가 없으므로 다음 데이터 노드를 순환하기 위해서는 앞에서 제시한 ordering 알고리즘[그림4]를 통해 orderNum를 구한 후 order[orderNum+1]배열을 가지고 다음 노드를 찾아서 다음 값을 검색하면 된다.

이때 노드 그룹에서 다음 노드를 검색시 현재 노드의 위치가 리프노드(leaf node)인지 여부를 확인한 다음 리프노드가 아니면 ordering 알고리즘을 통해 검색하고

```
CST_RangeSearch (tnode,min,max,k)
//tnode: 현재 노드그룹의 위치
//min, max: range 검색 범위
//k: 현재 노드 인덱스 값
 key=min;
 num=min-max
 while(count<=num)
    if(key<=(tnode->elements[k])
      value=binsearch(key)
      vcount++;
      *(valueStore+Vcount)=value;
      key++;
      count++;
    else
      if(k<NUM_NODEGRP_ELEMENT/2)
        orderNum=orderSearch(order,k);
        orderNum=order[orderNum+1];
        if(orderNum>=NUM_NODEGRP_ELEMENT/2)
          if((FCP(td)+left)->elements[0]!=NILDATA)
          CST_Search(key,td,order,max);
        else
        CST_RangeSearch(td,key,max,orderNum,traverseCount);
        end if
      else
          if((FCP(td)+right)->elements[0]!=NILDATA)
          CST_Search(key,td,order,max);
          else
          orderNum=2*(k-(NUM_NODEGRP_ELEMENT/2));
          orderNum=order[orderNum+1];
          CST_RangeSearch(td,key,max,orderNum,traverseCount);
          end if
       end if
 end while
return valueStore
END
```

[그림6]CST_RangeSearch 알고리즘

반대로 노드그룹의 리프노드(leaf node)를 만났을 때는 CST-Search 알고리즘[그림5]에서 다시 찾고자 하는 값을 검색한 다음 그 찾은 값을 기점으로 해서 다시 CST_RangeSearch 알고리즘[그림6]검색을 되풀이 하면 되겠다.

4.결론 및 향후 과제

지금까지 메인 메모리 데이터베이스 시스템에서 CPU 캐시를 효율적으로 활용하기 위해 캐시 미스 수를 줄이기 위한 방법들 중 CST트리[4]에서 효율적인 범위 검색(range search)에 대한 구축 기법으로 엔트리 최소화를 위해 포인터가 제거된 노드 그룹내에서의 트리의 순환(traverse)을 위한 방법으로ordering 알고리즘[그림4]를 제시 하였고 이를 바탕으로 범위검색(range query) 알고리즘을 제시하였다.

하지만 보다 효율적인 범위검색을 위해서는 노드그룹간의 불필요한 검색을 배제 한다면 보다 나은 성능을 보일 것이라 생각되어진다. 이는 향후과제로 남긴다.

5. 참고 문헌

[1] S. Manegold, P. A. Boncz and M. L. Kersten, "Optimizing database architecture for the new bottleneck: memory access", VLDB Journal, Vol.9,No.3, pp.231-246, 2000

[2]J. Rao and K. A. Ross, "Making B+ Trees Cache Conscious in MainMemory", Proceedings of ACM SIGMOD 2000 Conference, 2000.

[3]S. P. Vanderwiel and D. J. Lilja, "Data Prefetch Mechanisms", ACMComputing Surveys, Vol.32, No.2, pp.174-199, 2000.

[4] 이익훈, "캐시를 고려한 T-트리 인덱스 구조" 정보 과학회 논문지:데이터베이스, 제 32권,2005

[5] Jun Rao, K. A. Ross, Making B+-Trees Cache Concious in Main Memory, ACM, 2000

[6]T. J. Lehman and M. J. Carey, "A Study of Index Structures for Main Memory Database Management System", Proceedings of the 12th VLDBConference, 1986.

[7] 이익훈,"index Structures and Recovery for Main Memory Database Systems".Ph.D paper 2005.8