범위 검색을 위한 CST+ 트리 인덱스 구조
(A CST+ Tree Index Structure for Range Search)

이 재 원 † 강 대 희 †† 이 상 구 †††
(Jae-won Lee) (Dae-hee Kang) (Sang-goo Lee)

요 약 최신 컴퓨터 시스템의 새로운 병목 현상이 메모리 접근에서 발생하고 있다. 메모리의 접근 속도를 줄이기 위해 캐시 메모리가 도입되었지만, 캐시 메모리는 원하는 데이타가 캐시에 옮겨져 있어야 메모리 접근 속도를 줄일 수 있다. 이를 해결하기 위해 기존의 T 트리를 개선한 CST 트리가 제안되었다. 하지만, CST 트리는 범위 검색 시, 불필요한 노드를 검색해야 한다는 단점이 있다. 본 논문은 캐시 효율적인 CST 트리의 장점을 가지며, 범위 검색이 가능하도록 하기 위해 연결 리스트로 각 노드를 연결한 CST+ 트리를 제안하였으며, CST 및 CSB+에 비해 4~10배의 성능 향상을 보였다.

또한, 메인 메모리 데이타베이스 시스템 장애 시, 빠른 데이타베이스 복구를 위해 인덱스의 빠른 재 구축은 전체 데이타 복구 성능에 있어 매우 중요한 부분이다. 이를 위해 본 논문은 병렬 삽입 기법을 제안하였다. 병렬 삽입은 노드 분할 오버헤드가 없으며, 데이타 복구 단계와 인덱스 구축 단계를 병렬로 수행할 수 있는 장점이 있다. 병렬 삽입은 순차 삽입 및 일괄 삽입에 비해 2~11배의 성능 향상을 보였다.

키워드 : 메인 메모리, 인덱스 구조, 캐시, CST+ 트리, 범위 검색

Abstract Recently, main memory access is a performance bottleneck for many computer applications. Cache memory is introduced in order to reduce memory access latency. However, it is possible for cache memory to reduce memory access latency, when desired data are located on cache. CST tree is proposed to solve this problem by improving T tree. However, when doing a range search, CST tree has to search unnecessary nodes. Therefore, this paper proposes CST+ tree which has the merit of CST tree and is possible to do a range search by linking data nodes with linked lists. By experiments, we show that CST+ is 4~10 times as fast as CST and CSB+.

In addition, rebuilding an index is an essential step for the database recovery from system failure. In this paper, we propose a fast tree index rebuilding algorithm called MaxPL. MaxPL has no node-split overhead and employs a parallelism for reading the data records and inserting the keys into the index. We show that MaxPL is 2~11 times as fast as sequential insert and batch insert.

Key words : main memory, index structure, cache, CST+-tree, range search, recovery

1. 서 론

일반적으로 데이타베이스 관리 시스템(DBMS)은 데이타를 디스크에 저장한다. 디스크의 속도는 메모리에 비해 느리지만 가격이 저렴하고 안정적이기 때문에 많이 사용되고 있다. 그러나 최근 기술의 발달로 컴퓨터에 사용되는 메모리의 크기가 대용량화 되고, 가격이 저렴해지면서 메인 메모리 데이타베이스의 활용도가 증가하게 되었다. 메인 메모리 데이타베이스의 활용도가 증가하게 된 또 다른 요인은 64비트 시스템의 출현으로 메인 메모리 공간 제약의 문제가 사라지고, 빠른 트랜잭션 처리를 통한 실시간 서비스에 대한 요구가 증가하였기 때문이다.

메인 메모리 데이타베이스 관리 시스템의 성능 향상(빠른 트랜잭션 처리 등)을 위한 많은 연구들이 이루지고 있으며, 그 중에서도 인덱스 구축에 관한 연구가 많이 이루

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

† 학생회원 : 서울대학교 컴퓨터공학부
lyonking@europa.snu.ac.kr
†† 비 회 원 : 서울대학교 컴퓨터공학부
kdh200@europa.snu.ackr
††† 종신회원 : 서울대학교 컴퓨터공학부
sglee@europa.snu.ac.kr

논문접수 : 2007년 6월 4일
심사완료 : 2007년 10월 16일

정보과학회논문지: 데이타베이스 제35권 제1호(2008.2)

어지고 있다. 이러한 연구들 중 하나로 메인 메모리 데이타베이스를 위한 인덱스 구조인 T 트리(T-Tree)가 고안되었다[1]. 실험을 통해 T 트리가 MMDB(Main Memory DataBase)에서 가장 적합한 인덱스 구조로 알려졌으나, 최근에는 B^+ 트리(B^+-Tree)의 성능이 T 트리보다 뛰어나다[2]. 디스크 기반의 B^+ 트리가 메모리 기반의 T 트리보다 빨라진 원인은 하드웨어의 발전 속도 차이에 있다. 지난 10년간 CPU의 발전 속도는 메모리의 발전 속도에 비하여 급속하게 발전하였다. 그 결과 CPU-메모리-디스크 사이에서 발생하던 디스크 입출력 병목 현상이 CPU-캐시-메모리 사이에서 메모리 접근 병목 현상으로 바뀌었다. 임의 접근과 포인터를 이용하는 T 트리보다 디스크 I/O를 최대한 줄이기 위해 고안된 B^+ 트리가 메인 메모리에서도 메모리 접근 수가 적은 것은 당연하며, CPU와 메모리 사이의 성능 격차가 커짐에 따라 두 개의 인덱스 구조 사이에 성능 역전 현상이 일어난 것이다[3].

느린 메모리 접근 속도를 극복하기 위해서는 캐시 메모리를 적절히 이용하여야 한다. 캐시 메모리는 CPU와 메모리 사이의 속도 격차를 극복하기 위해 만들어진 것이다. [4]는 현대 컴퓨터에서 CPU와 메모리의 성능차이에 의해 메모리가 데이타베이스의 새로운 병목이 됨을 보이고 캐시를 고려한 프로그램 설계로 메모리의 병목 현상을 완화 시킬 수 있음을 보였다. 캐시를 효율적으로 사용하기 위해 B^+ 트리를 개선한 CSB^+ 트리가 고안되었다[2]. 또한 T 트리를 개선한 CST(cache sensitive T tree)가 고안되었다[4]. CST 트리는 T 트리의 변형으로, 점 검색(point search)에서는 좋은 성능을 보이고 있다. 그러나 범위 검색(range search) 에서는 그 구조상 좋은 성능을 보이지 못하고 있다.

본 논문은 메인 메모리 데이타베이스 분야에서 가장 성능이 좋은 것으로 알려진 CST 트리를 변형한 CST^+ 트리를 제안하고자 한다. CST^+ 트리는 CST 트리와 유사한 점 검색 성능을 보이며 효율적으로 범위 검색이 가능하도록 구조를 변형하였다. 본 논문은 CST^+ 트리에서의 검색, 삽입, 삭제 알고리즘을 제시한다. 또한 시스템 장애 발생시, 빠른 복구를 위한 병렬 삽입(MaxPL insert) 알고리즘을 제시한다. 본 논문은 다음과 같이 구성되어 있다. 2장에서는 T 트리를 변형한 T^* 트리[5] 및 CST 트리 에 대하여 설명한다. 3장에서는 CST^+ 트리의 검색, 삽입, 삭제 알고리즘을 제시한다. 4장에서는 CST^+ 트리에서 사용되는 병렬 삽입 알고리즘을 제시한다. 5장은 실험을 통해 CST^+ 트리의 성능을 평가하며, 6장은 결론 및 향후 연구에 대해 기술한다.

2. 관련 연구

2.1 T^* 트리 인덱스

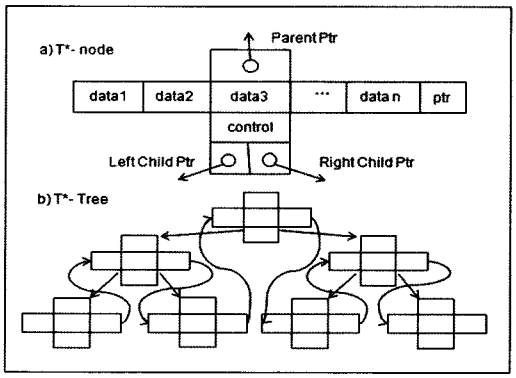

그림 1 T^* Tree Structure

T^* 트리는 T 트리를 개선한 인덱스 구조이다[5]. 특히 범위 검색에서 기존 T 트리 보다 좋은 성능을 보인다.

T^* 트리는 각 노드에 후위 포인터(next pointer)를 추가한 것이다. 이것은 질의어 처리를 위해 트리를 순회하는 기존의 인덱스 구조와 달리 연결 리스트를 이용함으로써 범위 검색을 보다 효율적으로 처리할 수 있는 인덱스 구조이다.

그림 1은 T^* 트리의 구조를 보여주고 있다.

T^* 트리의 검색 알고리즘은 T 트리와 유사하다. 검색 키 값이 노드의 최소 키 값보다 작다면 왼쪽 자식 포인터에 의해 왼쪽 서브 트리(sub tree)를 검색한다. 검색 키 값이 노드의 최대 키 값보다 크다면 오른쪽 자식 포인터에 의해 오른쪽 서브 트리를 검색한다. 만약 검색 키 값이 최소 키 값과 최대 키 값 사이의 값이면 현재 노드를 검색한다. T^* 트리에서 키 값은 정렬되어 있기 때문에 이진 검색을 이용한다.

만약 노드는 검색 되었으나 해당 키가 검색되지 않는다면 검색은 실패하게 된다. T^* 트리의 장점은 범위 검색을 할 때, 트리의 순회 비용이 T 트리에 비해 작다는 것이다. T 트리는 범위 검색 시, 항상 루트 노드(root node)부터 검색을 수행하므로 불필요한 노드를 순회한다. 하지만 T^* 트리는 후위 포인터를 이용하여 다음 노드를 바로 검색하므로 순회하는 동안 발생하는 비용을 줄일 수 있다.

예를 들어, $60 \le key \le 82$에 해당하는 키 값을 찾는 경우 T 트리는 그림 2처럼 N32-N21-N11-N22-N33 순으로 순회한다(최소값 60을 포함한 N32의 위치를 찾았다고 가정).

이때 N21, N22와 같이 찾고자 하는 키 값이 없는 노드까지 순회하게 된다. 하지만 T^* 트리는 후위 포인터를 사용하여 해당 키 값이 있는 노드만을 검색한다. 즉, N32-N11-N33을 순회한다. 그림 3은 이에 대한 예를 보여준 것이다.

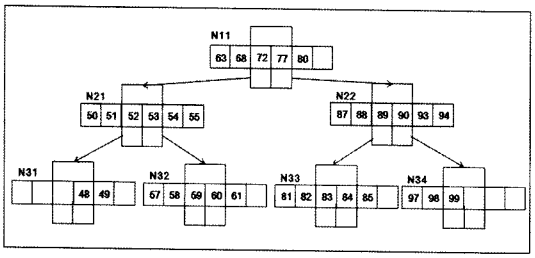

그림 2 Example of a Range Query on T Tree

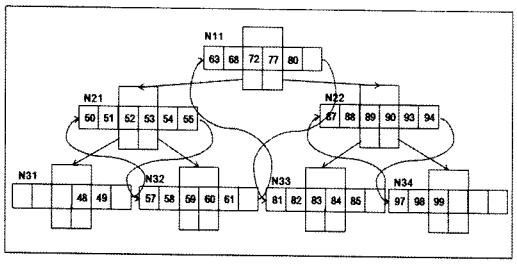

그림 3 The Example of a Range Query on T* Tree

T* 트리는 후위 포인터를 이용하여 효율적으로 범위 검색을 수행한다. 하지만 많은 포인터로 인해 삽입 및 삭제 시, 포인터 변경을 위한 비용이 크다는 단점이 있다.

2.2 CST 트리 인덱스

CST 트리[3]는 자식 노드들을 연속적인 메모리 공간에 할당함으로써 자식 노드를 가리키기 위한 포인터를 하나만 가지고 있는 구조로 T 트리를 변형한 것이다. 노드에 저장하는 포인터(left child ptr, right child ptr)가 줄어든 만큼 키 값을 더 저장할 수 있어 트리의 높이를 줄일 수 있다. 또한 포인터의 수를 줄임으로써, 포인터 접근 시 발생하는 캐시 미스 횟수를 줄인다. 그림 4는 CST 트리의 구조를 설명한다.

T 트리의 키 값 중에서 최소 또는 최대 키 값만을 이용하여 트리의 순회가 가능하므로, 이들로 이루어진

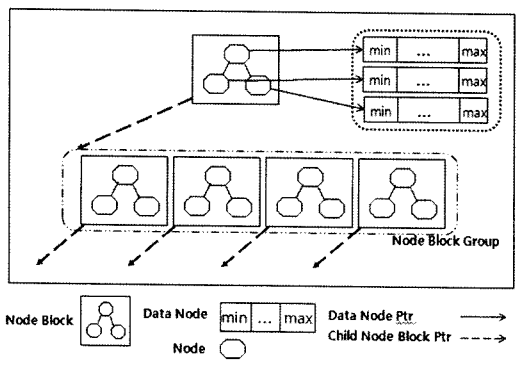

그림 4 Example of CST Tree

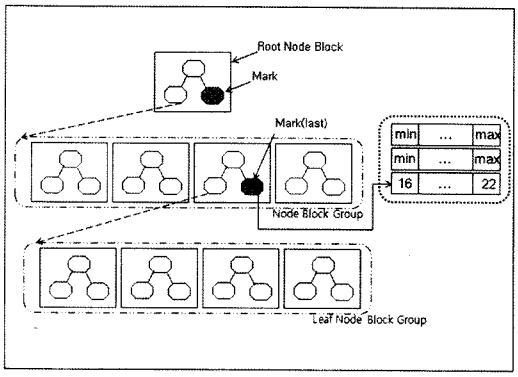

그림 5 Example of CST Tree Search

이진 트리를 구성한다. 이와 같은 이진 트리를 CST 에서는 노드 블록(node block)이라고 한다. 또한 노드의 필수 엔트리 (entry) 최소화를 위해, 포인터 제거를 통한 구조 압축을 한다. CST 트리의 자식 노드들을 연속된 메모리 공간에 할당하고, 하나의 자식 노드 포인터만을 노드 블록 그룹(node block group)에 저장한다.

CST 트리의 검색 알고리즘은 루트 노드 블록을 접근하여 검색 키가 현재 노드 블록 내의 키보다 작을 경우 비교한 위치를 그림 5처럼 마크(mark) 한다. 2번째 노드 블록에서 역시 검색 키가 노드 블록 내의 키보다 작은 경우에는 이전 마크 위치를 삭제하고 새 마크를 저장한다. 그리고 3번째 노드 블록(단말 노드 블록)에서 비교를 계속한다. 찾고자 하는 키 값이 단말 노드 블록에 존재하지 않는 경우, 가장 최근 마크에 대한 데이타 노드를 이진 검색한다. 데이타 노드에 찾은 키가 존재하면 검색을 성공한 것이고, 그렇지 않으면 해당 키가 존재하지 않는 것이다.

CST 트리는 점 검색에서는 다른 인덱스 구조에 비해 좋은 성능을 보였으나, 인덱스 구조상 범위 검색 시, 노드 블록간의 순회를 통해 검색을 해야 하므로 비용이 커질 것이라는 것은 명백하다. 즉 범위 검색 시, T* 트리에 비해 순회 비용이 크게 된다.

본 논문은 범위 검색을 하는데 있어서 CST 트리의 구조에서 범위 검색 비용을 최소화 할 수 있도록 데이타 노드간 전위/후위 포인터를 이용한 CST+ 트리를 제시한다.

2.3 인덱스 복구 알고리즘

인덱스를 복구하기 위한 방법으로 순차 삽입 (sequential insert)과 일괄 삽입(batch insert)이 있다[6]. 순차 삽입은 트리의 삽입 알고리즘에 따라 키를 하나씩 루트 노드부터 삽입하는 방법이다. 이 방법은 키를 삽입할 때마다 루트로부터 자신의 위치를 찾아 들어가야 한다. 또한 단말 노드가 꽉 차면 분할을 해야 하며, 이는

다시 상위 노드까지 전파되어 상당한 오버헤드(over-head)를 초래한다. 이 방법은 병렬처리가 가능한데, 디스크/백업 데이타로부터 키 값을 읽어 오는 단계(데이타 복구)와 트리 구조에 키 값을 삽입하는 단계(인덱스 구축)를 병렬 처리함으로써 성능을 향상 시키는 것이 가능하다.

일괄 삽입은 루트 노드부터 단말 노드까지의 높이가 일정한 경우 적용 가능하다. 순차 삽입 방식과 달리 단말 노드부터 키를 삽입하고, 그 상위 노드를 작성하는 바텀-업(bottom-up) 방식이다. 이는 노드 분할과 상위 노드로의 전파 과정이 필요 없기 때문에 좋은 성능을 보일 수 있다. 그러나 이 방법은 키 값을 데이타베이스로부터 모두 가져 온 후, 정렬을 해야 하므로 데이타 복구와 인덱스 구축을 병렬 처리하는 것이 어렵다 [8].

3. CST$^+$ 트리 인덱스

3.1 이중 연결 리스트를 이용한 인덱스 구조

노드 블록, 데이타 노드, 컨트롤 블록을 포함한 CST$^+$ 트리의 구조는 그림 6과 같다.

노드 블록은 최대 키 값들과 컨트롤 블록에 대한 포인터를 가지고, 컨트롤 블록은 노드 블록의 각 최대 키 값을 포함하고 있는 데이타 노드 포인터들과 부모 노드 블록 포인터, 자식 노드 블록 포인터를 갖는다. 데이타 노드는 키 값과 해당 레코드의 ID(RID)를 저장하는 노드로 각각 전위 포인터와 후위 포인터를 가진다. 이와 같이 데이타 노드를 포인터로 연결한 연결 리스트 구조는 기존 CST 트리에서 비효율적이던 범위 검색을 효율적으로 하기 위한 것이다.

[7]은 범위 검색을 위해 후위 포인터만을 이용한 단일 연결 리스트 구조를 제안하였다. 그러나 이 방법은 단방향 범위 검색만을 제공하므로 이전 데이타 노드로 접근하기 위한 방법이 없다. 즉, 키 값을 내림차순으로 검색하기 위해서는 CST 트리와 마찬가지로 루트 노드부터 키 값을 검색하여야 한다.

자료 구조의 차이를 바탕으로 CST 트리와 CST$^+$ 트리의 복잡도를 비교 분석하면 다음과 같다.

Def 1>

> n = 전체 키-레코드 쌍의 수
> s = 데이타 노드 내의 키-레코드 쌍 개수
> k = 노드 블록내의 노드 개수

전체 트리가 갖는 데이타 노드의 개수는 다음과 같다. CST 트리에서 n 개의 키를 저장하기 위해서 $\lceil \frac{n}{s} \rceil$ 개의 데이타 노드가 필요하나 CST$^+$ 트리에서는 하나의

```
CST+_NodeBlock
{
    Array of Maximal Keys;
    Pointer to a Control Block;
}
CST+_CtrlBlock
{
    K Pointers to Data Nodes;
    Pointer to the Parent;
    Pointer to the Child Node Groups;
}
CST+_DataNode
{
    Keys;
    Record IDs;
    Prev Pointer;
    Next Pointer;
}
```

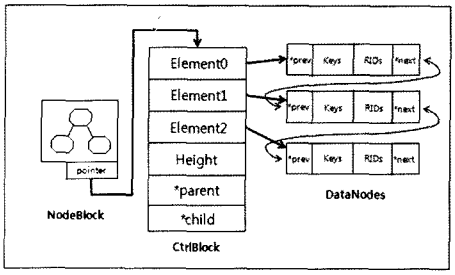

그림 6 The Data Structure of CST+ Tree

키-레코드 쌍 대신에 전위 포인터와 후위 포인터를 사용하므로 s-1 개의 키-레코드 쌍이 존재한다. 그러므로 $\lceil \frac{n}{(s-1)} \rceil$ 개의 데이타 노드가 필요하다. 하나의 노드에 s 개의 키가 저장될 경우, $\lceil \frac{n}{(s-1)} \rceil$ 개의 노드로 이진 트리를 만들면 트리의 높이는 $\log_2(\frac{n}{s-1})$ 이 된다. CST 트리와 CST$^+$ 트리의 각 노드 블록은 k개의 노드를 갖는 이진 트리이므로, 노드 블록의 높이는 $\log_2(k+1)$ 이다. 이러한 노드 블록으로 이루어진 CST$^+$ 트리의 전체 높이는 다음과 같다.

$$\log_2(\frac{n}{s-1}) \ / \ \log_2(k+1) \ = \ \log_{k+1}(\frac{n}{s-1})$$

표 1은 복잡도에 대한 비교를 하였다.

CST 트리와 CST$^+$ 트리는 검색 키를 찾기 위해 노드 블록을 검색한 후, 데이타 노드를 한번 더 검색한다. 데이타 노드를 검색하기 위해 포인터 접근을 수행하므로, 캐시 미스 수는 (트리의 높이+1)이다.

3.2 검색 알고리즘

CST$^+$ 트리는 기존의 CST 트리가 범위 검색에서 비

표 1 The Comparison of Complexities

구분	트리 높이	캐시 미스 수 (cache miss rate)
CSB+ 트리	$\log_k n$	$\log_k n$
CST 트리	$\log_{k+1}\frac{n}{s}$	$\log_{k+1}\frac{n}{s}+1$
CST+트리	$\log_{k+1}(\frac{n}{s-1})$	$\log_{k+1}(\frac{n}{s-1})_{+1}$

효율적이라는 점을 극복하기 위해 데이타 노드를 연결 리스트로 연결한 구조이다.

이와 같이 기본적인 구조가 유사하므로, CST+ 트리의 점 검색 알고리즘은 CST 트리와 유사하다. 하지만 범위 검색 시, 연결 리스트를 이용하기 때문에 검색 알고리즘이 다르게 된다. 그러므로 본 논문에서는 CST+ 트리의 범위 검색 알고리즘을 설명한다. CST+ 트리의 검색 알고리즘은 크게 2단계로 이루어진다.

1단계: 검색 범위에서의 최소 키 값을 찾기 위한 검색
2단계: 연결 리스트를 이용하여 1단계에서 찾은 키 값으로부터 범위 내의 키 값을 범위 검색

1단계 검색은 검색 키 값과 최대 키 값을 비교 한다. 검색 키 값이 최대 키 값보다 크면 오른쪽 서브 노드 블록에서 검색을 수행한다. 만약 최대 키 값보다 작으면 현재 노드에 검색 키 값이 존재하거나, 왼쪽 서브 노드 블록에 검색 키가 존재한다. 현재 왼쪽 서브 노드 블록을 검색 하기 이전에 현재 노드를 마크한다. 단말 노드까지 도달했는데도 찾지 못하는 경우, 마지막으로 마크한 노드의 데이타 노드에서 이진 검색을 수행한다.

2단계 검색은 1단계에서 찾은 키 값을 포함하고 있는 데이타 노드에서 주어진 범위 내의 키 값을 검색한다. 이때, 검색 범위의 최대 키 값과 비교하여 검색된 키 값이 같거나 크면 검색을 종료한다.

Example. 검색 키 범위가 $61 \le key \le 75$ 일 때의 검색 (데이타 노드는 키 값만을 가지고 있다고 가정)

CST 트리는 위와 같이 검색 범위가 주어졌을 때, 검색 알고리즘에 의해 각 노드를 마크하고 최종적으로 마크된 노드에서 최소 값(예제의 경우 61임)을 검색한 후, 범위 검색을 한다(그림 7). 즉, 65-60-70-95-90-80-75의 순서로 노드를 검색한다. 실제적으로 키 값이 들어있는 노드는 65-70-75이며, 불필요하게 검색한 노드는 60, 95, 90, 80이 된다. 노드 블록 안에 들어 있는 노드의 개수가 증가하고 트리의 높이가 높아질수록 이와 같은 오버헤드 비용은 커지게 된다. CST+ 트리의 경우, 첫 번째 키 값(예제의 경우 61 임)은 CST 트리와 같이 검색을 한다. 그러나 나머지 키 값은 데이타 노드를 실제

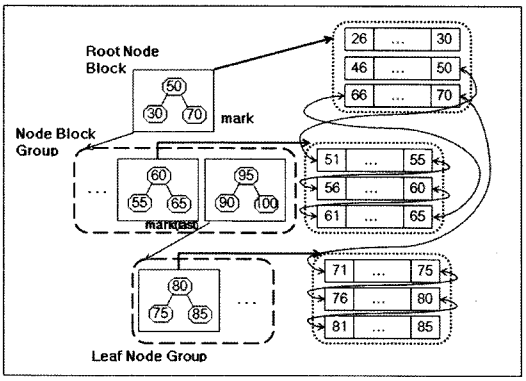

그림 7 Example of CST+ Tree Search

로 연결하고 있는 포인터들을 이용하여 검색하게 된다. 이 경우 검색 노드는 65-70-75로 줄어들게 된다.

CST+ 트리의 검색 알고리즘은 다음과 같다.

```
//traverse Node Groups
CST+ Search (min, max, tree)
 min : min key to find
 max : max key to find
 tree : CST+ Tree
 compareKey:key to compare from a tree
while (compareKey is not NULL)
  if( min <= compareKey )
    lastMarkNode = data node
          corresponding to current key;
    compareKey = the key of left sub tree;
  else
   compareKey = the key of right sub tree;
 endif
endwhile
// Binary Search in DataNode
if (lastMarkNode is not NULL)
    dataNode = data node from lastMarkNode;
    key = search key corresponding to min in
          the data node;
    if ( max <= key )
        if ( key is NULL )
          return Not Found;
        endif
     return key;
   else
      rangeSearch( key, max, tree )
   endif
endif
END
```

Algorithm1. CST+ Tree Search Phase 1

```
rangeSearch (key, max, tree)
valueStore = array of values searched;
keyPtr = pointer of key;
count = index of array;
```

```
for each elements of DataNode
  value = key value of (keyPtr + 1)th
  if( value <= max )
    *(valueStore+ count) = value;
    keyPtr = next keyPtr;
  else
    return *(valueStore);
  endif
endfor
END
```

Algorithm2. CST+ Tree Search Phase 2

3.3 삽입 알고리즘

삽입 알고리즘은 새로운 노드 블록과 데이타 노드가 생성될 경우, 데이타 노드를 포인터로 연결하는 것을 제외하고는 CST 트리의 삽입과 유사하다. 그러므로 본 논문은 새로운 노드 블록과 데이타 노드가 생성될 경우의 삽입 알고리즘을 설명하도록 한다. 삽입에 따른 트리 밸런싱(Tree Balancing) 알고리즘은 CST 트리의 알고리즘을 그대로 이용한다.

CST+ 트리의 삽입 알고리즘은 다음과 같다. 검색을 통해 삽입할 노드를 찾아 삽입 가능하면 삽입한다. 삽입할 노드를 찾았으나, 삽입할 공간이 없으면 데이타 노드의 최소 값을 삭제하고 해당 자리에 삽입 값을 넣는다. 삭제된 최소 값은 왼쪽 서브 노드 블록에 삽입한다. 이때, 왼쪽 서브 노드 블록이 없을 경우, 새로운 노드 블록을 생성하며 노드 밸런싱을 검사한다. 트리의 노드 블록이 생성되면, 현재 데이타 노드를 마크한 후, 새 노드 블록의 데이타 노드와 마크한 데이타 노드를 연결 리스트로 연결한다. 본 예제에서는 데이타 노드에 레코드 ID를 제외한 키 값만 존재하는 것으로 가정한다.

그림 8은 CST+ 트리에 키 값 97을 삽입하는 경우의 예시이다. 키 값 97을 삽입할 데이타 노드의 최소 키 값 96을 삭제하고 97을 삽입한다. 삭제된 키 값은 왼쪽의

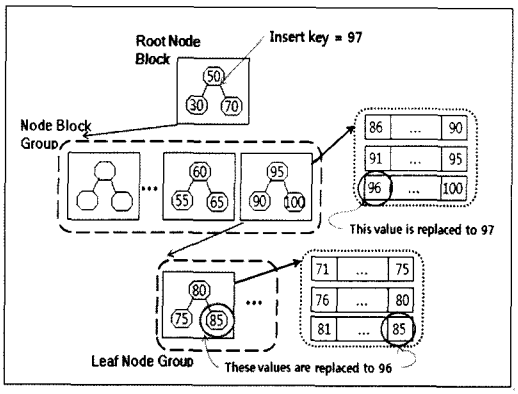

그림 8 CST+ Insert without Tree Creation

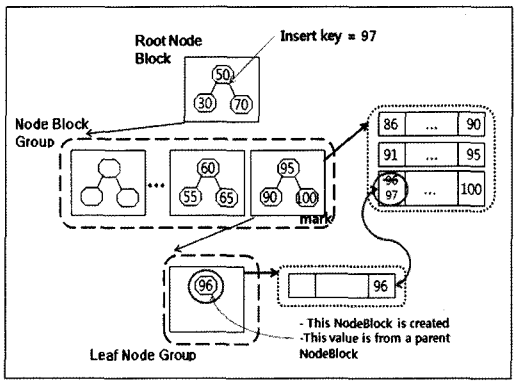

그림 9 CST+ Insert with Tree Creation

서브 노드 블록에 삽입할 공간이 있으므로 노드 블록의 최대 키 값 및 데이타 노드의 키 값 85를 치환한다.

그림 9는 서브 노드 블록에 공간이 없어서 새로운 노드 블록을 생성해야 할 경우, 키 값 97을 삽입하는 경우의 예시이다. 새로운 노드 블록을 생성한 후, 포인터로 데이타 노드를 연결해 주면 된다. 이때, 오버플로우가 발생한 노드를 마크한 후, 새로 생성된 노드를 마크한 노드의 데이타 노드로 연결한다.

```
CST+_insert ( key,tree)
dataNode=find a data node to insert key into tree;
If (dataNode != NULL )
      insert key to dataNode;
   if (key >= the maximal key of dataNode)
      modify the key of the current node group
      maximal key;
   end if
else //when dataNode is full
      delete the minimal key of dataNode & store
      it to tempKey;
      insert key to dataNode;
      markedNode = Nodeblock node of the
                   dataNode;
      leftSubtree=get the left subtree of
dataNode;
 if (leftSubtree is NULL )
  if (leftSubtree needs to create a new node
group)
   Add a new node group as left subtree of
   current dataNode;
   leftSubtree=add a data node to the new node
   group;
   replace markedDataNode's next pointer to
   leftSubtree's prev pointer;
```

```
    replace leftSubtree's prev pointer to
    markedDataNode's next pointer;
    RID = binary search in the dataNode;
    CST+_insert (tempKey,leftSubtree) ;
    CST+_BalancingTree (parent of new node
    group);
    else
     leftSubtree =add a data node to current node
     group;
    CST+_insert (tempKey,leftSubtree);
    balanceCheck();
    end if
  else CST+_insert (tempKey,leftSubtree);
  endif
endif
END
```

Algorithm3. CST+ Tree Insert

3.4 삭제 알고리즘

CST+ 트리의 삭제 알고리즘은 삭제할 키 값이 존재하는 데이타 노드를 검색한 후, 해당 데이타 노드에서 키 값과 해당 레코드 ID를 삭제한다. 이후 설명에서는 데이타 노드에 레코드 ID를 고려하지 않고 키 값만 존재하는 것으로 가정한다. 삭제로 인하여 데이타 노드에서 언더플로우(underflow)가 발생(키 수가 1/2 이하만 차 있는 경우)하면, 왼쪽 서브 노드의 최대 키 값을 현재 데이타 노드에 삽입한 후, 해당 왼쪽 서브 노드에서 최대 키 값을 삭제한다. 만약 왼쪽 서브 노드가 없을 경우에는 오른쪽 서브 노드의 최소 키 값을 현 데이타 노드에 삽입한 후, 오른쪽 서브 노드의 데이타 노드에서 해당 최소 키 값을 삭제한다. 이때, 삭제로 인하여 데이타 노드가 없어진다면 삭제되는 데이타 노드와 연결된 데이타 노드의 후위 포인터를 삭제되는 데이타 노드의 후위 포인터 값으로 치환한다. 삭제로 인하여 노드가 없어지면, 단말 노드 블록 내에서 트리 밸런싱을 검사하며, 노드 블록이 없어지는 경우, CST+ 트리 전체 밸런싱 체크를 하게 된다. 그림 10은 왼쪽 서브 노드 블록이 존재할 때, 키 값 96을 삭제하는 경우의 예시이다.

현재 데이타 노드에서 키 값 96을 삭제한 후, 왼쪽 서브 노드 블록의 최대 키 값인 85를 현재 데이타 노드에 삽입한 후, 해당 왼쪽 서브 노드 블록에서 최대 키 값을 삭제한다. 단말 노드 블록의 최대 키 값 및 데이타 노드의 최대 키 값은 84로 치환된다.

그림 11은 키 값 85를 삭제함으로써, 노드가 삭제되는 예시이다. 이 경우, 2번 데이타 노드가 삭제 됨에 따라 1번 데이타 노드의 후위 포인터는 3번 데이타 노드를 가리키게 되며, 3번 데이타 노드의 전위 포인터는 2번 데이타 노드가 아닌, 1번 데이타 노드를 가리키게 된다.

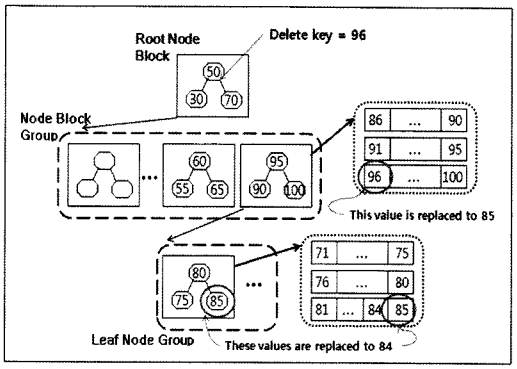

그림 10 CST+ Delete with Sub NodeBlock

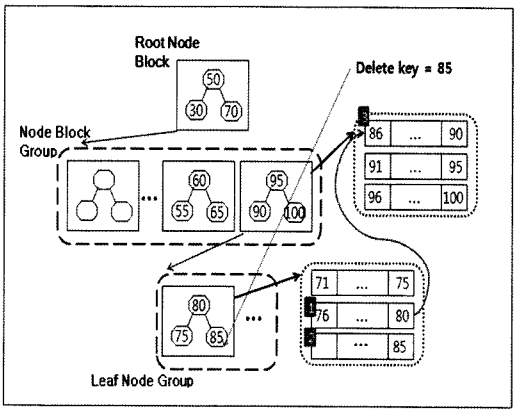

그림 11 CST+ Delete with Node Deletion

3.5 시간 복잡도

CST+ 트리의 데이타 노드 구조는 노드에 포인터를 가지고 다음 노드를 가리키므로 첫 번째 검색 키를 검색한 후에는 순차적인 검색이 가능하다. CST+ 트리를 하나의 이진 트리로 보면, 첫 번째 검색 키를 가진 노드를 찾기 위한 시간 복잡도는 $\log_2(\frac{n}{s-1})$ 이다. 데이타 노드를 검색하기 위해 한번의 캐시 미스가 더 발생하며, 데이타 노드 내에서 이진 검색을 하므로 이에 따른 검색의 시간 복잡도는 다음과 같다.

$$O(\log_2(\frac{n}{s-1}) + 1 + \log_2 s) = O(\log_2 n)$$

첫 번째 키를 찾은 이후, 연결 포인터를 이용하여 범위 검색 범위 내의 키를 검색하므로 시간 복잡도는 $O(m)$ 이다. 여기서 m은 검색 범위 내의 키의 개수이다. 그러므로 범위 검색을 위한 시간 복잡도는 다음과 같다.

$$O(\log_2 n) + O(m) = O(\log_2 n)$$

CST+ 트리의 삽입 연산은 삽입할 데이타 노드를 찾은 후에, 해당 데이타 노드에 키 값을 삽입한다.

이때 데이타 노드가 꽉 찬 경우(overflow), 현재 노드의 최소 키 값을 삭제하고 왼쪽 서브 트리에 삭제한 키 값을 삽입한다. 따라서, 삽입할 데이타 노드를 찾는데, 최대 $O(\log_2 n)$ 만큼의 비교가 필요하고, 데이타 노드가 꽉 찬 경우를 처리하기 위해 최악의 경우에 트리 높이 만큼 탐색하게 되므로 $O(\log_2 n)$ 만큼의 추가 연산이 필요하다. 포인터 연결을 위한 탐색 시간은 이중 연결 리스트이므로 O(2) 만큼의 추가 연산이 필요하다. 그러므로 CST+ 트리의 삽입 시간 복잡도는

$O(\log_2 n + \log_2 n + 2) = O(\log_2 n)$ 이다.

CST+ 트리의 삭제 알고리즘은 삽입과 유사하게 삭제할 키가 있는 데이타 노드를 찾고, 키 삭제 후에 언더플로우(underflow - 키 수가 1/2 이하만 차 있는 경우)를 처리하기 위한 추가 연산이 필요하다. 포인터 삭제 후 이를 연결하기 위해 검색 연산이 필요하므로, 시간 복잡도는 $O(\log_2 n + \log_2 n + 2) = O(\log_2 n)$ 이다.

```
CST+_Delete( key,tree)
dataNode=find a data node to delete key into tree;
delete key in dataNode
if (dataNode is underflow )
  leftSubtree = get the left subtree of dataNode;
  if (leftSubtree is NULL)
   rightSubtree=get the right subtree of dataNode;
    if (rightSubtree is NULL)
     if ( # of keys in dataNode == 0)
        find current dataNode's prev pointer dataNode;
        replace finded dataNode's next pointer to
        current dataNode's next pointer;
        replace linked dataNodes's prev pointer to
        finded dataNode's next pointer;
        drop current dataNode;
        modify the node group;
        balance node group;
        if (# of keys of the current node group == 0)
        find current dataNode's prev pointer dataNode;
        replace finded dataNode's next pointer to
        current dataNode's next pointer;
        replace linked dataNodes's prev pointer to
        finded dataNode's next pointer;
        drop current node group;
        CST+ _BalancingTree();
        endif
      else
        if (key was the maximal key in dataNode)
          modify the maximal key;
        end if
      end if
    else //rightSubtree is not NULL
     tempKey=get the minmal key in rightSubtree;
     insert tempKey to dataNode;
     CST+ _Delete(tempKey,rightSubtree);
    end if
  else //current node group has a left subtree
   tempKey=get the maximal key in leftSubtree;
   insert tempKey to dataNode;
   CST+ _Delete(tempKey,rightSubtree);
   end if
else //not underflow
  if (key is the maximal key in dataNode)
  modify the key of the node group corresponding to
  dataNode;
end if
END
```

Algorithm4. CST+ Tree Delete

4. CST+ 트리 인덱스 복구

본 논문은 키 값을 읽어오는 단계(데이타 복구)와 삽입하는 단계(인덱스 구축)를 병렬 처리함으로써, 순차 삽입과 일괄 삽입의 장점을 접목한다. [8,9]는 각각 CST 트리, B^+ 트리에서 병렬 삽입을 위한 알고리즘을 제시하였다. 본 논문은 노드 블록 내의 데이타 노드가 모두 채워지면 각 데이타 노드를 연결리스트로 연결하는 과정도 병렬 처리 한다.

4.1 인덱스 병렬 삽입 알고리즘

백업 시에 인덱스의 각 말단 노드 내의 최대 키 값들을 별도의 디스크 공간에 저장한다. 인덱스 재 구축 시에는 저장된 단말 노드의 최대 키 값을 메모리로부터 읽은 후, 트리의 단말 노드 블록을 구성한다. 단말 노드 블록이 구축되면 다음 상위 단계의 노드 블록을 구성하여 트리의 루트 노드 블록까지 구축한다. 데이타 노드에 키 값을 삽입하기 위해 디스크로부터 데이타를 읽은 후, CST+ 트리의 데이타 노드에 삽입한다. 이때 순차 삽입하는 것처럼 루트 노드부터 검색을 하여 삽입할 노드의 위치를 찾는다. 이미 노드 블록은 각 데이타 노드의 최대 키 값을 가지고 있으므로 해당 데이타 노드를 찾는 것은 점 검색 알고리즘을 이용한다. 데이타 노드에 키 값을 삽입할 때, 이진 검색을 통해 위치를 찾을 수도 있지만, 매번 위치를 검색하여 삽입하는 것은 비용이 크다. 그러므로 데이타 노드가 모두 채워지면 정렬한다. CST+ 트리 노드 블록 내의 데이타 노드가 모두 채워지면, CST+ 트리의 최소 키 값을 가진 왼쪽 단말 데이타 노드부터 순회하며 포인터 연결을 한다.

이와 같은 병렬 삽입 알고리즘은 순차 삽입과 달리 트리 분할에 의한 오버헤드가 발생하지 않으며, 일괄 삽입처럼 데이타를 정렬하기 위한 비용이 필요하지 않아

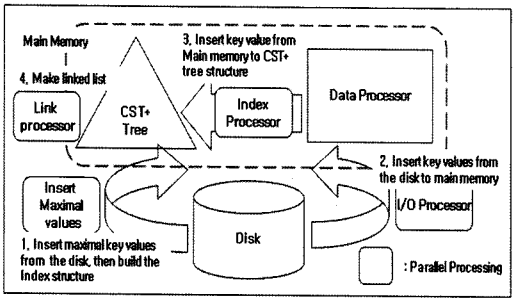

그림 12 Parallel Index Recovery

```
readingData
  find the total numRecs to be rebuild
  while(1)
    read a records and numRecs--;
    mutex_lock(mutex);
      put the record key to sharedMem;
    mutex_unlock(mutex);
    send a signal to the fillDataNode thead;
  end while
End
```

Algorithm5. CST+ Tree Recovery-ReadingData

```
fillDataNode
  while(1)
    mutex_lock(mutex);
    while( inserted Key is not NULL)
      thread_cond_wait(mutex);
    readKey = inserted Key;
    insertKey2CST+;
  end while
End
```

Algorithm6. CST+ Tree Recovery-filling Data to DataNode

효율적이다. 이를 도식화하면 그림 12와 같다.

알고리즘 5와 6은 병렬 삽입 알고리즘에서 키 값을 디스크로부터 읽어오는 알고리즘과 실제로 데이타 노드에 삽입하는 알고리즘의 pseudo-code이다.

4.2 복구 시간 복잡도

복구 과정에서 각 단계별 시간 복잡도를 살펴보고, 전체 시간 복잡도를 계산한다.

4.2 인덱스 구조 생성

저장된 단말 노드의 최대 키 값을 읽어서, 키 값마다 하나의 데이타 노드를 생성하므로, 시간복잡도는 $O(\frac{n}{s-1})$ 이다. 여기서 n은 레코드 수이며, s은 키-레코드 쌍의 개수이다. 분모가 (s-1)인 이유는 전위/후위 포인터 쌍을 제외하기 때문이다. 그러므로 시간 복잡도는 O(n)이다.

• 레코드 키의 CST+ 트리 삽입

레코드 키를 인덱스에 삽입하는 연산은 n개의 키를 인덱스에 삽입하는 것이고, 한 개의 키 삽입마다 인덱스 트리를 1회 탐색하므로 $n\log_2 n$ 번의 트리 탐색이 필요하다. 즉, $O(n\log_2 n)$ 의 시간 복잡도를 갖는다.

• 단말 노드 정렬

각 데이타 노드마다 $(s-1)\log_2(s-1)$ 의 시간 복잡도를 갖는 퀵정렬을 이용하므로 시간 복잡도는 다음과 같다.

$O(\frac{n}{s-1}*(s-1)\log_2(s-1)) = O(n)$ 이다.

• 포인터 연결

모든 데이타 노드가 채워지면, 트리를 순회하면서 전위/후위 포인터를 연결한다. 이때 포인터 연결을 위해 데이타 노드의 개수만큼 1회의 탐색이 이루어진다. 그러므로 시간 복잡도는 $O(\frac{n}{s-1}\log_2(\frac{n}{s-1})) = O(n\log_2 n)$ 이다.

전체 시간 복잡도는 위에서 구한 값을 이용하면 다음과 같다.

$$O(n)+O(n\log_2 n)+O(n)+O(n\log_2 n) = O(n\log_2 n)$$

5. 성능 평가

5.1 실험 환경

CST+ 트리는 C 언어로 구현되었으며, gcc 3.4.6을 이용하여 컴파일 하였다. 실험용 서버는 Sun Ultra Sparc III 1.2GHz CPU*2와 4GB 의 DRAM을 가지고 있다. 또한 512KB 의 L2 캐시 메모리를 가지고 있고, L2 캐시 블록 크기는 64byte이다. 실험에 이용된 CSB+ 트리[2], CST 트리[3]는 저자가 구현한 것을 이용하였다.

5.2 검색 성능 실험

트리가 각각 200,000, 300,000, 400,000, 500,000 개의 키 값을 가지고 있을 때, 200,000 건의 키 검색을 수행하였을 때의 성능을 비교한다. 각 실험 데이타는 10회씩 실험을 수행하여 나온 결과 값의 평균 값이다.

5.2.1 점 검색

점 검색은 200,000 건의 임의 키 값을 검색하여, 그에 대한 성능을 비교한다. 그림 13의 검색 성능은 CST, CST+, CSB+ 트리의 순으로 좋은 결과를 보이고 있다. 동일한 수의 키 값에 대하여 CSB+ 트리의 높이가 높기 때문이다. 하지만 CST+, CST 트리의 경우, 비슷한 검색 성능을 보이고 있는데, 이는 트리의 높이가 같기 때문이다. 키 값의 개수가 증가할수록 CST+ 트리가 더 느려질 수 있으나, 그 차이는 크지 않다.

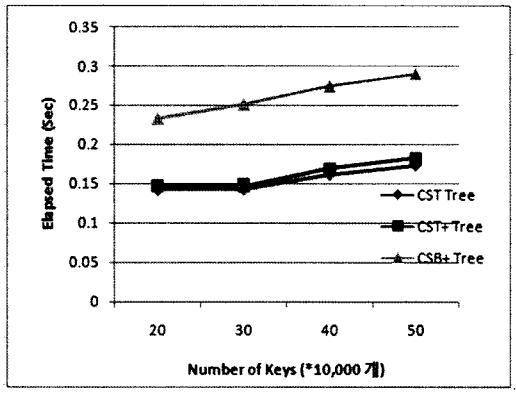

그림 13 Point Key Search with 200,000 Keys

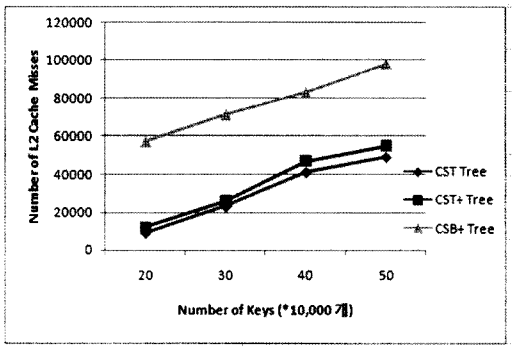

그림 14 Number of L2 Cache Misses

그림 14는 동일한 조건에서의 L2 캐시 미스 횟수를 측정한 것이다. CST$^+$ 트리의 경우, 데이타 노드에 포함되는 키 값과 레코드 ID의 개수를 줄이고 전위/후위 포인터를 이용하였으므로, 검색을 수행할 때 CST 트리보다 캐시 미스가 더 발생한다. 이러한 캐시 미스는 그림 13의 검색 시간에 영향을 주는 것을 볼 수 있다.

5.2.2 범위 검색

범위 검색은 200,000 건의 범위 키 값을 검색하여 그에 대한 성능을 비교한다. 그림 15는 범위 검색 시, CST$^+$, CSB$^+$, CST 트리의 순으로 좋은 검색 성능을 보이고 있다. CST 트리에서 범위 검색은 데이타 노드를 접근한 후, 데이타 노드의 크기만큼 연속된 키 값을 읽는다. 데이타 노드를 벗어나는 키 값은 다시 루트 노드부터 검색을 하여, 새로운 데이타 노드에 접근한다. 하지만 CST$^+$ 트리는 검색 범위의 최소 키 값을 찾기 위해 데이타 노드를 1회 검색하기 때문에 빠른 검색 결과를 보인다. CST$^+$ 트리가 CSB$^+$ 트리에 비해 빠른 이유는 트리의 높이 차이로 인해 검색 범위의 최소 키 값을 찾는데 시간이 덜 소요되기 때문이다.

그림 16은 L2 캐시 미스 횟수를 측정한 것이다. CST$^+$

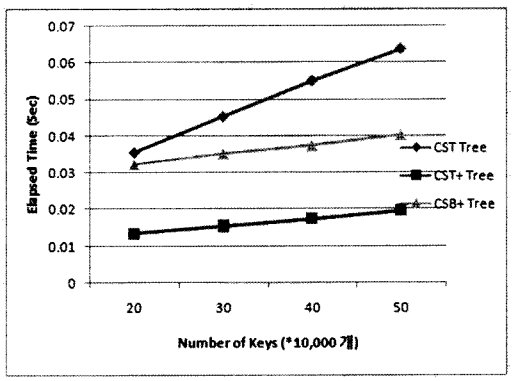

그림 15 Range Search with 200,000 Keys

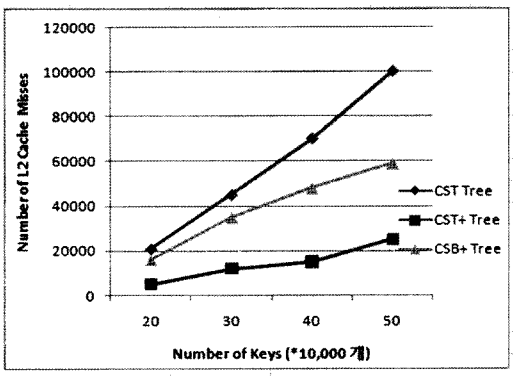

그림 16 Number of L2 Cache Misses

트리의 경우, 데이타 노드를 검색한 후, 연결 포인터를 이용하여 다음 데이타 노드를 검색한다. 이로 인해 발생하는 캐시 미스가 적으므로 CST 트리에 비해 빠르다.

5.3 삽입 성능 실험

트리가 각각 200,000, 300,000, 400,000, 500,000 개의 키 값을 가지고 있을 때, 200,000 건의 키 삽입을 수행하였을 때의 성능을 비교한다. CST$^+$ 트리가 CST 트리에 비해 느리지만, 거의 차이가 없다(그림 17). 점 검색

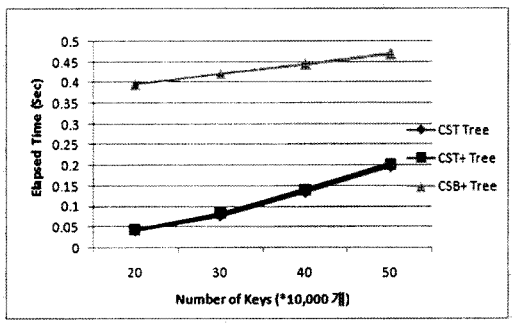

그림 17 Random Key Insertion with 200,000 Keys

과 마찬가지로 두 트리의 높이 차이가 거의 없기 때문이다. CST+ 트리가 CST 트리에 비해 느린 이유는 전위/후위 포인터 연결을 위한 시간이 더 걸리기 때문이지만, 실험 결과 포인터 연결을 위한 시간은 영향을 거의 미치지 않는 것을 확인할 수 있다.

5.4 삭제 성능 실험

트리가 각각 200,000, 300,000, 400,000, 500,000 개의 키 값을 가지고 있을 때, 200,000 건의 키 삭제를 수행하였을 때의 성능을 비교한다. 삽입 실험과 유사하게 CST+ 트리가 약간 느리지만 크게 차이는 없다(그림 18). CST+ 트리가 CST 트리에 비해 느린 이유는 키 삭제로 인하여 새로운 포인터 변경이 발생하기 때문이다. 이 역시, 포인터 변경으로 인한 비용이 크지 않음을 확인 할 수 있다. 삭제 키의 개수가 400,000 개 인 경우, 차이가 많이 나는데, 이는 임의의 키를 삭제하는 과정에서 노드 블록의 삭제로 인해 트리 밸런싱을 체크하는 비용이 발생하였기 때문이다.

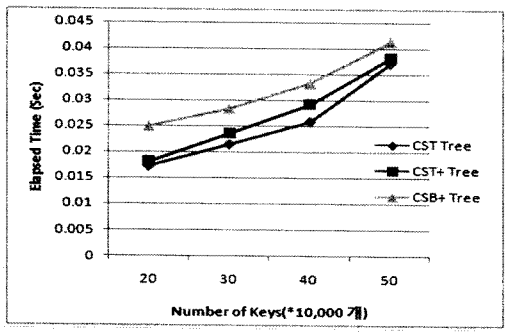

그림 18 Random Key Deletion with 20,000 Keys

5.5 복구 성능 실험

복구 성능을 테스트하기 위해 키 값을 100,000, 500,000, 1,000,000, 1,500,000, 2,000,000 개를 비어 있는 트리에 삽입한다. 순차 삽입(sequential insert), 일괄 삽입(batch insert), 병렬 삽입(MaxPL insert)을 비교한다.

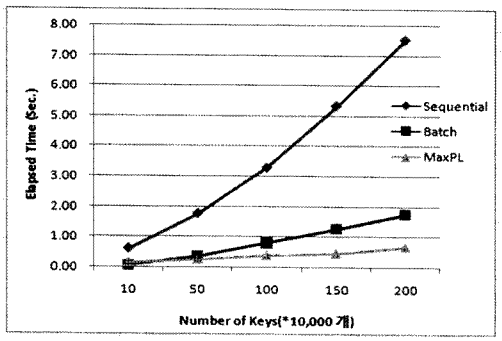

그림 19 Recovery

그림 19에서 순차 삽입은 키 값을 삽입할 때마다 밸런스 검사(balance check)를 하여야 하므로 삽입 시간이 오래 걸린다. 일괄 삽입은 단말 노드부터 구성 하므로, 밸런스 검사가 필요하지 않다. 병렬 삽입은 데이타를 읽어오는 단계와 삽입하는 단계를 병렬 처리함으로써 가장 좋은 성능을 보인다. 병렬 삽입은 순차 삽입에 비해 400~1,100%의 성능 향상을 보이며, 일괄 삽입에 비해 200~300%의 성능 향상을 보인다.

6. 결론 및 향후 과제

CST 트리는 캐시의 효율을 높임으로써 기존의 인덱스 구조인 T 트리, B+ 트리 그리고 CSB+ 트리 보다 좋은 성능을 보이며, 특히 검색에 있어서 150~300%의 성능 향상을 보인다. 하지만, CST 트리는 구조상 범위 검색에 있어서는 좋은 성능을 보이지 못한다.

본 논문은 CST 트리의 단점을 보완하기 위해 데이타 노드를 전/후위 포인터를 이용함으로써, 점 검색, 삽입, 삭제 시 성능은 비슷하지만 범위 검색 시 600~1000%의 성능 향상을 가짐을 보였다.

또한 시스템 장애 시 인덱스를 복구하기 위해 병렬 삽입 알고리즘을 제안하였다. 데이타를 읽어오는 단계와 인덱스에 삽입하는 단계를 병렬 처리 함으로써, 200~1,100%의 성능 향상을 보였다. 향후 과제로는 인덱스 구조와 유사하게 캐시를 고려한 레코드 저장 구조에 대하여 연구할 필요가 있다. 레코드 저장 블록의 크기를 캐시에 최적화시킴으로써, 성능 향상이 가능할 것으로 기대한다.

참 고 문 헌

[1] T. J. Lehman and M. J. Carey, "A Study of Index Structures for Main Memory Database Management Systems," In Proceedings of the 12th VLDB Conference, pages 294-303, 1986.

[2] J. Rao and K. A. Ross, "Making B+ Trees Cache Conscious in Main Memory," Proceedings of ACM SIGMOD 2000 Conference, pages 475-486, 2000.

[3] 이익훈, 김현철, 허재녕, 이상구, 심준호, 장준호, "캐시를 고려한 T 트리 인덱스 구조", 한국정보과학회 논문지, 제32권 제1호, pages 12-23, 2005.

[4] P. Boncz, S. Manegold, and M. Kersten, "Database Architecture Optimized for the new Bottleneck: Memory Access," Proceedings of the 19th VLDB Conference, pages 54-65, 1999.

[5] K. R. Choi, K. C. Kim, "T*-Tree: A Main Memory Database Index Structure for Real Time Applications," Proceedings of the 3rd International Workshop on RTCSA, pages 81-89, 1996.

[6] SangWook Kim, HeeSun Won, "Batch Construction of B+-Trees," Proceedings of ACM Symposium on

Applied Computing 2001, pages 231-235, 2001.
[7] 강대희, 이재원, 이상구, "CST-트리의 효과적인 범위 검색", 한국정보과학회 2006: 데이타베이스, pages 67-69, 2006.
[8] 이재원, 이익훈, 이상구, "최대 키 값을 이용한 CST-트리 인덱스의 빠른 재구축", 한국정보과학회 2005:데이타베이스, pages 85-87, 2005.
[9] Ig-hoon Lee, Junho Shim, Sang-goo Lee, "Fast Rebuilding B+ Trees for Index Recovery," IEICE Transactions on Information and Systems, vol. E89-D(7), pages 223-233, 2006.

이 재 원

2003년 2월 숭실대학교 정보통신공학과 학사. 2006년 2월 서울대학교 컴퓨터공학과 석사. ~현재 서울대학교 컴퓨터공학과 박사 과정. 관심분야는 Main Memory Database, Ontology Search, Personalization

강 대 희

1994년 2월 육군사관학교 토목공학 학사 2007년 2월 서울대학교 컴퓨터공학과 석사. ~현재 육군 영상정보 관리장교. 관심분야는 Semantic Web, Database, 영상정보 처리

이 상 구

1985년 서울대학교 계산 통계학과 학사 1987년 M.S. Computer Science, Northwestern University, Evanston, Illinois 1990년 Ph.D. Computer Science, Northwestern University, Evanston, Illinois ~현재 서울대 컴퓨터공학부 교수, e-Business 기술 연구 센터장. 관심분야는 e-Business Technology, Databases, Mobile Database