Journal of Korea Multimedia Society Vol. 12, No. 10, October 2009(pp. 1374-1385)

범위질의 검색을 위한 캐시적응 T-트리 주기억장치 색인구조

최상준[†], 이종학[††]

요 약

최근 CPU의 속도는 메모리의 속도에 비해 훨씬 빠르게 향상되었다. 따라서 주기억 장치의 접근이 주기억 장치 데이터베이스 시스템의 성능에서 병목현상으로 나타나고 있다. 기억장치 접근 속도를 줄이기 위해 캐시메모리를 이용하지만, 캐시메모리는 요구되는 데이터가 캐시에서 찾을 수 있는 경우에만 기억장치 접근 속도를 줄일 수 있다. 본 논문에서는 CST*-트리라는 범위질의를 위한 새로운 캐시 적응 T-트리 색인구조를 제안한다. CST*-트리는 색인 엔트리를 저장하지 않는 축소된 내부노드들을 캐시메모리에 올려 사용함으로써 캐시메모리의 활용도를 높인다. 그리고 인접한 단말노드들과 내부 색인노드들을 링크포인터를 통해 서로 연결함으로써 색인 엔트리들의 순차적 접근을 가능하도록 한다. 본 논문에서는 성능평가를 위한 비용 모델을 개발하고, 이를 이용하여 캐시미스 발생 횟수를 평가하였다. 그 결과 단일키 값 검색에서는 기존의 캐시만을 고려한 CST-트리에 비해 약 20~30%의 캐시미스 발생 횟수가 감소하였고, 범위질의에서는 기존의 범위질의만을 고려한 색인구조인 T*-트리에 비해 약 10~20%의 캐시미스 발생 횟수가 감소하였다.

Cache Sensitive T-tree Main Memory Index for Range Query Search

Sang-Jun Choi[†], Jong-Hak Lee[††]

ABSTRACT

Recently, advances in speed of the CPU have for out-paced advances in memory speed. Main-memory access is increasingly a performance bottleneck for main-memory database systems. To reduce memory access speed, cache memory have incorporated in the memory subsystem. However cache memories can reduce the memory speed only when the requested data is found in the cache. We propose a new cache sensitive T-tree index structure called as CST*-tree for range query search. The CST*-tree reduces the number of cache miss occurrences by loading the reduced internal nodes that do not have index entries. And it supports the sequential access of index entries for range query by connecting adjacent terminal nodes and internal index nodes. For performance evaluation, we have developed a cost model, and compared our CST*-tree with existing CST-tree, that is the conventional cache sensitive T-tree, and T*-tree, that is conventional the range query search T-tree, by using the cost model. The results indicate that cache miss occurrence of CST*-tree is decreased by 20~30% over that of CST-tree in a single value search, and it is decreased by 10~20% over that of T*-tree in a range query search.

Key words: T-tree(T-트리), Main-Memory Database(주기억장치 데이터베이스), Range Query(영역질의), Index Structure(색인구조)

※ 교신저자(Corresponding Author) : 이종학, 주소 : 경북 경산시 하양읍 금락1리 330(712-702), 전화 : (053) 850-2746, FAX : (053)850-2750, E-mail : jhlee11@cu.ac.kr
접수일 : 2009년 3월 5일, 수정일 : 2009년 7월 15일
완료일 : 2009년 7월 22일
[†] 준회원, 대구가톨릭대학교 컴퓨터정보통신공학과(공학석사)
(E-mail : database@cu.ac.kr)
[††] 정회원, 대구가톨릭대학교 컴퓨터정보통신공학부 교수

1. 서 론

데이터베이스 관리 시스템은 일반적으로 데이터를 디스크에 저장하여 관리하고 있으며, 이를 디스크 기반 데이터베이스(DRDB : Disk Resident Database)라 한다. 하지만 최근에는 메모리의 크기가 대용량화되고 가격도 저렴해짐에 따라, 디스크를 사용하지 않고 데이터를 메인메모리 내에서만 저장하여 관리하는 메인메모리 데이터베이스(MMDB : Main Memory Database)의 활용이 증가하고 있다. 메인메모리 데이터베이스는 디스크 기반 데이터베이스에 비해 속도면에서 우수한 성능을 보이므로 실시간 서비스에 대한 요구에 적합하다.

데이터베이스 관리시스템에서는 데이터의 빠른 접근을 위하여 색인구조를 사용한다. 대표적인 색인구조로 디스크 기반 데이터베이스에서는 B^+-트리[1-3]를 사용하고, 메인메모리 데이터베이스에서는 T-트리[4]를 사용한다. 메인메모리 데이터베이스에서 B^+-트리보다 T-트리가 유용한 이유는 메인메모리의 용량의 제한으로 적은 용량의 색인구조가 필요하기 때문이다. 만약, 같은 n개의 색인엔트리를 저장함에 있어서 B^+-트리는 한 노드에 n+1개의 자식노드 포인터를 가지지만 T-트리는 3개의 포인터만을 가지므로 한 노드의 크기가 B^+-트리에 비해 T-트리가 적다. 하지만 T-트리는 범위질의에 대한 지원을 고려하지 않고 있다. 이를 보완하기 위해 T^+-트리[4,6]와 T^*-트리[5,6]가 제안되었다.

T^+-트리는 T-트리의 내부노드를 AVL-트리[7]처럼 최소값 하나만을 가지도록 구성하고 단말노드에 모든 색인엔트리를 저장하여 그 단말노드를 서로 연결함으로써 범위질의를 효율적으로 지원한다. 하지만 T^+-트리는 T-트리에 비해 트리의 높이가 높아지고, 단말노드에 모든 색인엔트리를 저장하기 때문에 특정 색인엔트리를 검색하기 위해서는 항상 단말노드까지 검색을 해야만 한다. T^*-트리는 기존 T-트리의 각 노드에 후위노드 포인터를 사용하여 노드를 순차적으로 연결한 것으로 범위질의를 효율적으로 지원한다. 하지만 T^*-트리는 캐시메모리를 효율적으로 이용하지 못하고 있다.

메인메모리 데이터베이스에서는 메모리의 접근 횟수를 줄이기 위해 캐시메모리를 사용하고 있다[8,9]. 최근에는 색인구조에서도 이러한 캐시메모리의 효율적인 사용을 위한 노력이 시도되고 있다. B^+-트리와 같은 경우 CSB^+-트리[10]를 통해 캐시메모리를 효율적으로 활용하고 있으며 T-트리 역시 CST-트리[11-13]를 통해 캐시메모리를 효율적으로 활용하고 있다. CST-트리는 T-트리에서 각 노드의 최대값을 추출하여 별도의 이진트리를 구축하여 캐시메모리를 효율적으로 사용하고 있지만, 범위질의의 처리를 고려하지 못하고 있다. 따라서 기존의 메인메모리 데이터베이스 색인구조인 T-트리에서 캐시메모리의 사용을 고려함과 동시에 범위질의를 효율적으로 처리하기 위한 새로운 색인구조가 필요하다.

본 논문에서는 캐시메모리를 효율적으로 사용하면서 동시에 범위질의의 처리를 잘 지원하는 새로운 메인메모리 데이터베이스 색인구조를 제안하고 이를 CST^*-트리라 한다. CST^*-트리는 캐시메모리를 효율적으로 사용하기 위해 기존 T-트리에서 내부노드에는 최소/최대 색인값과 이에 따른 두 개의 자식노드 포인터만을 유지하고 색인엔트리는 별도의 노드로 구성함으로써 내부노드를 축소한다. 이처럼 축소된 내부노드들의 블록을 캐시메모리에 올려 사용함으로써 캐시미스 발생 횟수를 줄인다. 그리고 T-트리의 모든 단말노드들과 내부노드에 있던 색인엔트리들을 저장하여 구성한 내부 색인노드들을 링크 포인터를 통해 순차적으로 연결함으로써 범위질의를 효율적으로 처리한다.

본 논문의 구성은 다음과 같다. 제 2절에서는 관련 연구로 T^*-트리와 CST-트리에 대해 설명한다. 제 3절에서는 본 논문에서 제안하는 CST^*-트리의 구조와 검색, 삽입 그리고 삭제 알고리즘을 제시한다. 제 4절에서는 본 논문에서 제안하는 CST^*-트리의 성능평가를 위한 비용모델을 개발하고, 이를 통한 성능평가 결과를 기술한다. 마지막으로 제 5절에서는 결론을 기술한다.

2. 관련 연구

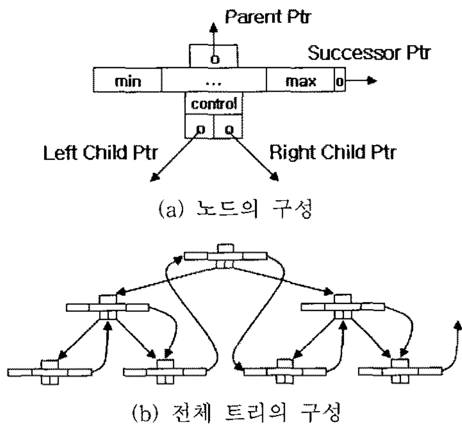
이 절에서는 본 논문에서 제안하는 색인구조의 기반이 되는 T-트리가 가지는 범위질의 문제점을 보완하기 위해 제안된 T^*-트리와 캐시 메모리의 사용을 보완한 CST-트리에 대한 내용을 기술한다.

2.1 T*-트리

T*-트리[5]는 T-트리의 범위질의 처리를 보완하기 위해 제안된 색인구조이다. 그림 1은 T*-트리 노드의 구조와 전체 트리의 구성을 보여준다. T*-트리의 한 노드는 그림 1의 (a)와 같이 T-트리에서 하나의 색인 엔트리 대신 각 노드를 순차적으로 연결하는 후위 포인터가 추가되며, 이 후위포인터를 통해 범위질의를 효율적으로 처리한다. T*-트리는 T-트리와 마찬가지로 내부노드에도 색인엔트리가 존재하기 때문에 단말노드에 모든 색인 엔트리를 저장하는 T+-트리처럼 항상 단말노드까지 검색할 필요가 없다. 전체 T*-트리의 구성은 그림 1의 (b)와 같다.

T*-트리의 검색, 삽입 그리고 삭제는 T-트리와 유사하지만 회전이 일어난 후 후위포인터를 재구성하는 비용이 추가된다. 범위질의의 경우 후위포인터를 통해 노드를 순차적으로 접근할 수 있기 때문에 T-트리에서 발생하였던 불필요한 노드의 순회가 발생하지 않는다. 그림 2는 T*-트리의 예를 나타내고 있다. 그림 2에서 범위질의의 예로 60 ~ 84사이의 모든 키 값을 검색하라는 질의가 주어졌을 경우, 우선 질의 범위 내 최소값인 60을 T-트리의 단일키 값 검색을 통해 검색하여 N3을 검색한다. 그 후 노드의 후위포인터를 통해 N4와 N5에 순차적으로 접근하여 범위질의 내 키 값들을 검색한다.

이처럼 T*-트리는 불필요한 노드 순회를 줄여 범위질의에 있어서 T-트리에 비해 좋은 성능을 보이지만, T*-트리는 T-트리와 마찬가지로 노드의 크기로 인한 캐시메모리의 활용이 효율적이지 못하다.



(a) 노드의 구성

(b) 전체 트리의 구성

그림 1. T*-트리 노드의 구성 및 전체 트리 구성

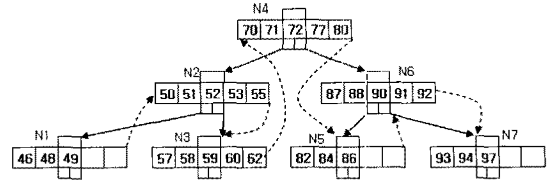


그림 2. T*-트리의 예

2.2 CST-트리

CST-트리[11]는 T-트리 노드의 크기로 인해 발생하는 캐시미스의 횟수를 줄이기 위해 고안된 색인구조이다. 그림 3은 CST-트리의 논리적인 구조를 보여준다. CST-트리는 그림 3과 같이 기존 T-트리 각 노드들을 데이터 노드로 하고 각 데이터 노드의 최대키 값을 이용하여 이진트리를 새로이 구성한다. 그렇게 구성된 이진트리를 캐시메모리의 블록 크기만큼 노드블록으로 설정하여 캐시메모리로 올려 사용함으로써 캐시미스 발생 횟수를 줄인다. 즉, T-트리는 하나의 노드를 캐시메모리로 불러올 때마다 캐시미스가 발생하는 반면 CST-트리는 최대값으로 구성된 이진트리의 노드 블록을 캐시메모리에 올려 검색을 실시하므로 캐시미스 발생 횟수가 줄어든다.

그러나, CST-트리는 최대키 값만으로 구성된 이진트리를 이용하여 검색을 진행하기 때문에 내부노드에 존재하는 키 값을 검색하고자 하는 경우에도 반드시 단말노드까지 검색을 해야만 하는 비효율성을 가진다. CST-트리의 검색은 이진트리의 검색과 유사하다. 검색하고자 하는 키 값과 노드 블록내의 키 값을 비교하여 검색하고자 하는 키 값이 노드 블록내의 키 값보다 크면 오른쪽 자식 포인터를 따라 오른쪽 서브트리를 검색하고, 작으면 현재의 노드에 마킹(marking)을 한 뒤 왼쪽 자식 포인터를 따라 왼쪽 서브트리를 검색한다.

마킹을 하는 이유는 검색하고자 하는 키 값이 노

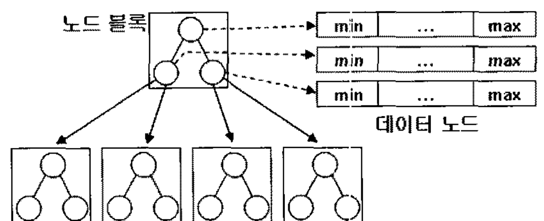


그림 3. CST-트리의 논리적 구조

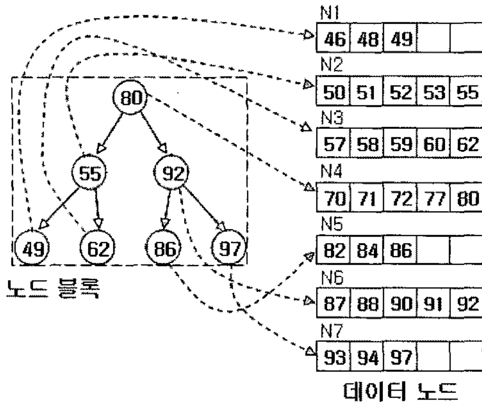


그림 4. CST-트리의 예

드 블록내의 키 값보다 작을 경우 현재 노드와 연결된 데이터 노드에 키 값이 존재할 수도 있기 때문에, 왼쪽 서브트리에 존재하는 모든 노드들에 연결된 데이터 노드들에 키 값이 존재하지 않을 경우 현재 노드와 연결된 데이터 노드를 검색하기 위해서이다. 이와 같이 현재 노드와 연결된 데이터 노드에 검색하고자 하는 키 값이 존재하는 경우에도 단말노드까지 검색하기 전에는 이를 알 수 없다. 따라서 단말노드까지 검색을 진행한 후 검색하고자 하는 키 값을 발견하지 못하면 마지막으로 마킹된 노드와 연결된 데이터 노드에서 검색을 실시한다.

그림 4는 CST-트리의 예를 나타내고 있다. 그림 4에서 검색질의의 예로 70의 키 값을 검색하고자 하는 경우 70이 노드블록내의 최대값 80보다 작기 때문에 현재 노드에 마킹을 하고 왼쪽 서브트리로 이동한다. 70이 55보다는 크므로 오른쪽 서브트리로이동하고, 70이 62보다는 크지만 더 이상의 서브트리가 존재하지 않기 때문에 마지막에 마킹된 노드인 80과 연결된 데이터노드에서 검색을 진행한다. 이와 같이 내부노드에 존재하는 키 값을 검색하기 위해서는 반드시 단말노드까지 검색을 해야만 하는 CST-트리 검색의 비효율을 알 수 있다.

그리고 CST-트리는 T-트리에서처럼 범위질의의 경우 검색 범위내의 키 값들을 가진 노드들을 루트노드로부터 매번 검색해야 하는 비효율성이 존재한다.

3. 범위질의 및 캐시를 고려한 CST*-트리

이 절에서는 본 논문에서 제안하는 범위질의와 캐시를 동시에 고려한 새로운 색인구조인 CST*-트리를 제안한다. 제 3.1절에서는 CST*-트리의 구조를 제안하고, 제 3.2절에서는 단일키 값 검색 및 범위질의 알고리즘을 제안한다. 제 3.3절에서는 삽입 알고리즘을 제안하고, 마지막으로 제 3.4절에서는 삭제 알고리즘을 제안한다.

3.1 CST*-트리의 구조

CST*-트리는 범위질의를 고려한 T*-트리가 캐시 메모리를 효율적으로 활용하지 못하는 점을 보완하고, T+-트리가 트리의 높이가 크고 트리내에 존재하는 키 값을 찾기 위해서 반드시 단말노드까지 검색을 해야만 하는 점을 보완한 색인구조이다. 또 캐시에 적응적인 색인구조인 CST-트리가 내부노드에 존재하는 키 값을 검색하기 위해서도 반드시 단말노드까지 검색해야만 하고, 이진트리를 새로이 구축해야만 하는 비효율성을 보완한 색인구조이다.

그림 5는 CST*-트리의 내부노드, 단말노드와 내부 색인노드 그리고 전체 트리의 구조를 보여주고 있다. CST*-트리는 그림 5의 (a)와 같이 T-트리의 내부노드를 최소/최대키 값과 균형제어값만을 남기고, 내부 색인엔트리들로는 별도의 내부 색인노드를 구성하여 이에 대한 포인터만을 유지함으로써 내부노드를 축소한다. 이와 같이 축소된 내부노드들을 캐시블록의 크기만큼 노드블록으로 구성하여 캐시메모리에 올려 사용함으로써 캐시미스 발생 횟수를 줄인다. 단말노드와 내부 색인노드는 그림 5의 (b)와

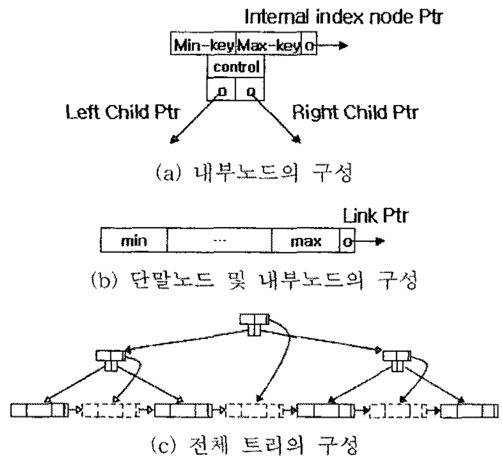


그림 5. CST*-트리의 구조

같이 색인엔트리들을 가지고 있으며 범위질의를 효율적으로 처리하기 위해 노드들을 순차적으로 연결하는 링크포인터를 가진다. 그리고 CST*-트리의 전체 구성은 그림 5의 (c)와 같다.

3.2 CST*-트리의 검색

먼저, CST*-트리의 단일키 값 검색 알고리즘은 그림 6과 같다. 즉,

① 루트노드로부터 검색할 키 값과 내부노드의 키 값을 비교한다.

- 검색하고자 하는 키 값이 내부노드의 최소키 값 보다 작으면 왼쪽 서브트리를 검색한다.
- 검색하고자 하는 키 값이 내부노드의 최대키 값 보다 크면 오른쪽 서브트리를 검색한다.
- 검색하고자 하는 키 값이 내부노드의 최소/최대키 값 사이이면 내부 색인노드 포인터와 연결된 내부 색인노드를 검색한다.

② ①의 방법으로 단말노드 또는 내부 색인노드까지 검색한 후 키 값이 존재하면 그 값을 반환하고 종료한다. 반대로 키 값이 존재하지 않으면 실패를 반환하고 종료한다.

그리고 CST*-트리의 범위질의 알고리즘은 그림 7과 같다. 즉,

```
CST* Search (Inode, key)
//Inode : internal node
//Imin : min key of internal node
//Imax : max key of internal node
//key : key to find
    while (Inode is not NULL)
        if (key < Imin)
            Inode = Internal node of left sub tree;
        else if (key > Imax)
            Inode = Internal node of right sub tree;
        else (Imin <= key <= Imax)
            Tnode_IInode ();
        endif
    endwhile
    Tnode_IInode ()
        if (key is NULL)
            return Not Found;
        else
            return key;
        endif
    END
END
```

그림 6. CST*-트리 단일키 값 검색 알고리즘

```
CST* RangeSearch (min, max)
//min : min key to find
//max : max key to find
//value : value of searched key
//valueStore : array of searched value
//keyPtr : pointer of key
//count : index of array
    CST* Search (Inode, min)
        valueStore = value of min;
    for each element of Tnode_IInode
        value = key value of (keyPtr + 1)th
        if (value <= max)
            *(valueStore + count) = value;
            keyPtr = next keyPtr;
        else
            return *(valueStore);
        endif
    endfor
END
```

그림 7. CST*-트리 범위질의 알고리즘

① 검색할 키 값들 중 최소값을 단일키 값 검색을 통해 검색한다.

② ①의 방법으로 검색하여 검색된 최소값에서 최대값 사이의 값들을 단말노드와 내부 색인노드의 링크포인터를 이용하여 순차적으로 검색한 후 검색된 값을 반환하고 종료한다.

그림 8은 CST*-트리의 예로써 그림 4의 CST-트리를 재구성한 것이다. 단일키 값 검색에서 내부 색인노드에 존재하는 키 값을 검색하는 예로 그림 8에서 71의 값을 검색하라는 경우 71이 내부노드의 최소키 값인 70과 최대키 값인 80 사이의 값이므로 내부노드의 내부 색인노드 포인터와 연결된 내부 색인노드 N4를 한 번에 검색하여 원하는 키 값인 71을 찾는다. 즉, 여기서는 CST-트리에서처럼 단말노드까지 검색 할 필요가 없다.

범위질의의 예로 그림 8에서 60 ~ 84사이의 모든 키 값을 검색하라는 경우 우선 검색 범위내 최소값인 60을 단일키 값 검색을 통해 검색하여 N3을 찾고, 그 후 단말노드와 내부 색인노드들의 링크포인터를

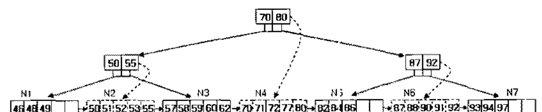


그림 8. CST*-트리의 예

이용하여 순차적으로 검색을 진행한다.

3.3 CST*-트리의 삽입

CST*-트리의 삽입 알고리즘은 그림 9와 같다. 즉,

① 제 3.2절의 단일키 값 검색을 통해 키 값을 삽입할 노드를 검색한다.

② 단말노드 및 내부 색인노드에 키 값을 삽입할 자리가 있으면 삽입하고 종료한다.

③ 단말노드에 자리가 없는 경우 이를 내부 색인노드로 하고 최소/최대키 값을 이용하여 내부노드를 구성한다.

- 삽입할 키 값이 내부노드의 최소키 값보다 작은 경우 왼쪽 서브트리에 단말노드 생성하고 키 값을 삽입한다.
- 삽입할 키 값이 내부노드의 최대키 값보다 큰 경우 오른쪽 서브트리에 단말노드 생성하고 키 값을 삽입한다.
- 삽입할 키 값이 내부노드의 최소/최대키 값 사이인 경우 왼쪽 서브트리에 단말노드 생성하고 내부 색인노드의 최소값을 삽입한다. 삽입할 키 값은 내부 색인노드에 삽입하고, 그에 따른 내부노드의 최소키 값을 수정한다.

④ 내부 색인노드에 자리가 없는 경우

- 연결된 내부노드의 왼쪽 서브트리가 존재하지 않으면 왼쪽 서브트리에 단말노드 생성하고 내부 색인노드의 최소값을 삽입한다. 삽입할 키 값은 내부 색인노드에 삽입하고, 그에 따른 내부노드의 최소키 값을 수정한다.
- 연결된 내부노드의 왼쪽 서브트리가 존재하고 자리가 있는 경우 왼쪽 서브트리에 내부 색인노드의 최소값을 삽입하고, 삽입할 키 값은 내부 색인노드에 삽입한다. 그리고 그에 따른 내부노드의 최소키 값을 수정한다.
- 연결된 내부노드의 왼쪽 서브트리가 존재하고 자리가 없는 경우 ③을 수행한다.

⑤ 내부노드의 균형제어 값에 따라 트리의 균형을 조절한다.

그림 10은 CST*-트리 삽입에서 발생하는 각 경우를 예로 나타낸 것이다. 그림 10에서 ⓐ는 단말노드에 키 값을 삽입할 자리가 있는 경우를 나타낸 것으로 7을 삽입할 자리가 있기 때문에 삽입하고 종료한다. ⓑ는 단말노드에 키 값을 삽입할 자리가 없고 삽

```
CST* Insert(key, Tnode, IInode, Pnode)
//key : key to insert
//Tnode : terminal node
//IInode : internal index node
//Pnode : parents node
    if (Tnode is not FULL || IInode is not FULL)
        insert key;
    else if (Tnode is FULL)
        create Pnode;
        IInode = Tnode;
        Pnode.interPtr = IInode's address;
        Pnode.Imin = IInode's min;
        Pnode.Imax = IInode's max;
        if (key > Pnode.Imax && Pnode.rightPtr
           is NULL)
            create Tnode;
            Pnode.rightPtr = Tnode's address;
            insert key in Tnode;
            IInode.linkPtr = Tnode's address;
        else if (key > Pnode.Imax &&
                 Pnode.rightPtr is not NULL)
            CST* Insert (key, Tnode, IInode,
            Pnode);
        else if (key < Pnode.Imin && Pnode.liftPtr
               is NULL)
            create Tnode;
            Pnode.leftPtr = Tnode's address;
            insert key in Tnode;
            Tnode.linkPtr = IInode's address;
        esle if (key < Pnode.Imin && Pnode.liftPtr
              is not NULL)
            CST* Insert (key, Tnode, IInode,
            Pnode);
        else if (Pnode.Imin <= key <= Pnode.Imax
               && Pnode.leftPtr is NULL)
            create Tnode;
            Pnode.leftPtr = Tnode's address;
            Tnode.linkPtr = IInode's address;
            insert IInode's min key in Tnode;
            insert key in IInode;
        else if (Pnode.Imin <= key <= Pnode.
        Imax && Pnode.leftPtr is not NULL)
            CST* Insert (key, Tnode, IInode,
                         Pnode);
            insert key in IInode;
        endif
    else (IInode is Full)
        move to left Tnode;
        CST* Insert (IInode's min key, Tnode,
                     IInode, Pnode);
        insert key in IInode;
    endif
END
```

그림 9. CST*-트리 삽입 알고리즘

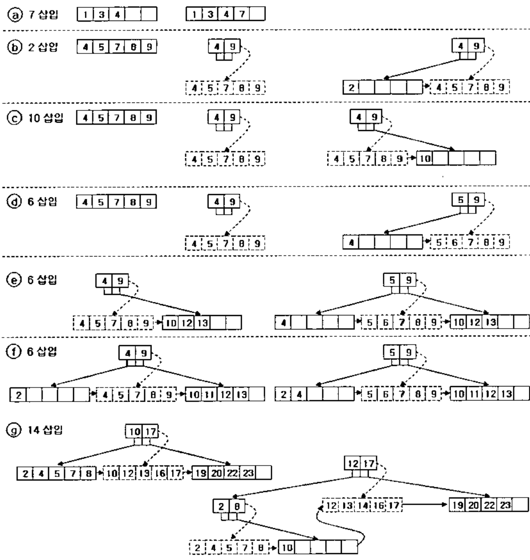


그림 10. CST*-트리 삽입 예

입하고자 하는 키 값이 단말노드의 최소값보다 작은 경우를 나타낸 것이다. 이런 경우 현재의 단말노드를 내부 색인노드로 하고 최소/최대키 값인 4와 9를 이용하여 내부노드를 구성한다. 삽입하고자 하는 키 값 2는 구성된 내부노드의 왼쪽 서브트리에 단말노드를 생성하여 삽입하고 종료한다.

ⓒ는 단말노드에 키 값을 삽입할 자리가 없고 삽입하고자 하는 키 값이 단말노드의 최대값보다 큰 경우를 나타낸 것이다. 이런 경우 현재의 단말노드를 내부 색인노드로 하고 최소/최대키 값인 4와 9를 이용하여 내부노드를 구성한다. 삽입하고자 하는 키 값 10은 구성된 내부노드의 오른쪽 서브트리에 단말노드를 생성하여 삽입하고 종료한다.

ⓓ는 단말노드에 키 값을 삽입할 자리가 없고 삽입하고자 하는 키 값이 단말노드의 최소/최대값 사이에 존재하는 경우를 나타낸 것이다. 이런 경우 현재의 단말노드를 내부 색인노드로 하고 최소/최대키 값인 4와 9를 이용하여 내부노드를 구성한다. 구성된 내부노드의 왼쪽 서브트리에 단말노드를 생성하여 기존 내부 색인노드의 최소값인 4를 삽입하고 삽입하고자 하는 키 값인 6은 내부 색인노드에 삽입한다. 삽입 후 내부 색인노드의 최소값이 4에서 5로 변경되었기 때문에 연결된 내부노드의 최소키 값을 수정한다.

ⓔ는 내부 색인노드에 자리가 없고 연결된 내부노드의 왼쪽 서브트리가 없는 경우를 나타낸 것이다. 이런 경우 내부노드의 왼쪽 서브트리에 단말노드를 생성하여 기존 내부 색인노드의 최소값인 4를 삽입하고 삽입하고자 하는 키 값인 6은 내부 색인노드에 삽입한다. 삽입 후 내부 색인노드의 최소값이 4에서 5로 변경되었기 때문에 연결된 내부노드의 최소키 값을 수정한다.

ⓕ는 내부 색인노드에 자리가 없고 연결된 내부노드의 왼쪽 서브트리가 있으며 그 왼쪽 서브트리에 자리가 있는 경우를 나타낸 것이다. 이런 경우 내부노드의 왼쪽 서브트리에 기존 내부 색인노드의 최소값인 4를 삽입하고 삽입하고자 하는 키 값인 6은 내부 색인노드에 삽입한다. 삽입 후 내부 색인노드의 최소값이 4에서 5로 변경되었기 때문에 연결된 내부노드의 최소키 값을 수정한다.

ⓖ는 내부 색인노드에 자리가 없고 연결된 내부노드의 왼쪽 서브트리가 존재하며 그 왼쪽 서브트리에도 자리가 없는 경우를 나타낸 것이다. 이런 경우 내부노드의 왼쪽 단말노드를 내부 색인노드로 하고 최소/최대키 값인 2와 8을 이용하여 내부노드를 구성한다. 구성된 내부노드의 오른쪽 서브트리에 단말노드를 생성하여 자리가 없던 내부 색인노드의 최소값인 10을 삽입하고 삽입하고자 하는 키 값인 14는 기존의 내부 색인노드에 삽입한다. 삽입 후 기존 내부 색인노드의 최소값이 10에서 12로 변경되었기 때문에 연결된 내부노드의 최소키 값을 수정한다.

3.4 CST*-트리의 삭제

CST*-트리의 삭제 알고리즘은 그림 11과 같다. 즉,

① 제 3.2절의 단일키 값 검색을 통해 삭제할 키 값을 검색한다.

② 삭제할 키 값이 존재하지 않는 경우 실패를 반환하고 종료한다.

③ 삭제할 키 값이 존재하면 삭제하고 노드의 삭제가 발생하지 않는 경우 종료한다.

④ 삭제할 키 값을 삭제한 후 노드의 삭제가 발생하는 경우

- 삭제된 노드가 단말노드이면 노드를 삭제한다.
- 삭제된 노드가 내부 색인노드이면 왼쪽 단말노드를 내부 색인노드로 하고 내부노드의 최소/최대키 값을 수정한다.
- 노드의 삭제 후 내부노드와 연결된 노드가 하

나만 존재하는 경우 내부노드를 삭제하고 내부 색인노드를 단말노드로 구성하여 내부노드와 연결한다.

⑤ 내부노드의 균형제어 값에 따라 트리의 균형을 조절한다.

그림 12는 CST*-트리 삭제에서 발생하는 각 경우를 예로 나타낸 것이다. 그림 12에서 ⓐ는 삭제할 키 값이 삭제된 후 노드의 삭제가 발생하지 않는 경우를 나타낸 것이다. 이런 경우 7의 키 값을 삭제하더라도 노드의 삭제가 발생하지 않기 때문에 삭제하고 종료

```
CST* Delete (key, Tnode, IInode, Pnode)
//key : key to delete
//Tnode : terminal node
//IInode : internal index node
//Pnode : parents node
    CST* Search (Inode, min)
    if (key is NULL)
        return Not Found;
    else if (key is not NULL)
        delete key;
        if (Tnode is empty)
            delete Tnode;
                if (Pnode.leftPtr = NULL &&
                    Pnode.rightPtr = NULL)
                     Tnode = IInode;
                     Tnode connecting to Pnode's
                     Pnode;
                     Tnode.linkPtr = Pnode's
                     Pnode.interPtr;
                     delete Inode;
                endif
        else if (IInode is empty && Pnode.leftPtr
                is not NULL)
            Pnode.interPtr = Pnode.lftPtr;
            IInode = Pnode.leftTnode;
            Pnode.leftPtr = NULL;
            Pnode.Imin = IInode's min key;
            Pnode.Imax = IInode's max key;
            IInode.linkPtr = Pnode.rightTnode's
                                address;
        else (IInode is empty && Pnode.leftPtr is
              NULL)
            Tnode = Pnode.rightTnode;
            Tnode connecting to Pnode's Pnode;
            Tnode.linkPtr = Pnode's Pnode.interPtr;
            delete Inode;
        endif
    endif
END
```

그림 11. CST*-트리 삭제 알고리즘

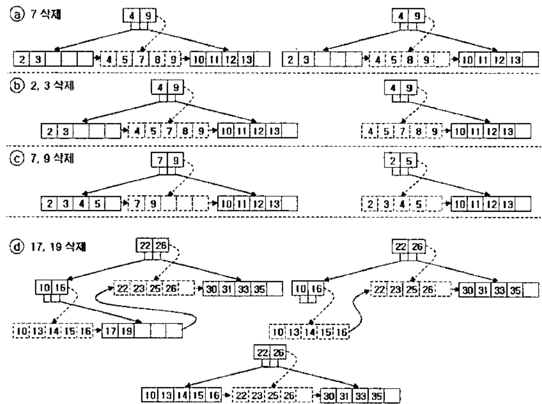


그림 12. CST*-트리 삭제 예

한다. ⓑ는 삭제할 키 값이 삭제된 후 단말노드의 삭제가 발생하는 경우를 나타낸 것이다. 이런 경우 2와 3의 키 값이 삭제된 후 단말노드의 삭제가 발생하기 때문에 단말노드를 삭제하고 종료한다. ⓒ는 삭제할 키 값이 삭제된 후 내부 색인노드의 삭제가 발생하는 경우를 나타낸 것이다. 이런 경우 7과 9의 키 값이 삭제된 후 내부 색인노드의 삭제가 발생하기 때문에 내부 색인노드를 삭제하고 왼쪽 단말노드를 내부 색인노드로 구성한다. 그리고 내부 색인노드의 최소/최대키 값인 2와 5를 이용하여 내부노드의 최소/최대키 값을 수정하고 종료한다. ⓓ는 삭제할 키 값이 삭제된 후 노드의 삭제가 발생하고 내부노드와 연결된 단말노드 또는 내부 색인노드가 하나만 존재하는 경우를 나타낸 것이다. 이런 경우 17과 19의 키 값이 삭제된 후 오른쪽 단말노드가 삭제되어 내부노드와 연결된 노드가 내부 색인노드 하나만 존재하기 때문에 내부노드를 삭제한다. 그리고 내부 색인노드를 단말노드로 구성하고 내부노드와 연결한 뒤 종료한다.

4. 성능 평가

이 절에서는 성능평가를 위해 지금까지 제안된 기존의 메인메모리 데이터베이스 색인구조들과 함께 본 논문에서 제안하는 CST*-트리의 성능평가 비용모델을 개발하고, 이를 통한 성능평가 결과를 도출한다. 비교되는 기존의 색인구조로는 앞서 설명했던 T-트리, T+-트리, T*-트리 그리고 CST-트리이다.

표 1은 지금까지 제안된 메인메모리 데이터베이스 색인구조들에 대해서 검색시 발생하는 캐시미스

표 1. 각 색인구조의 검색시 캐시미스 발생 횟수

	단일키 값 검색시 캐시미스 발생 횟수(SC)	범위질의시 캐시미스 발생 횟수
T-트리	$\sum_{i=1}^{\frac{n}{s}} \lvert \log_2(i+1) \rvert / \frac{n}{s}$	$SC_T \times \frac{l}{s}$
T^+-트리	$\left\lvert \log_2(\frac{n}{s}+1) \right\rvert + 1$	$SC_{T^+} + \frac{l}{s}$
T^*-트리	$\sum_{i=1}^{\frac{n}{s-1}} \lvert \log_2(i+1) \rvert / \frac{n}{s-1}$	$SC_{T^*} + \frac{l}{s-1}$
CST-트리	$\left\lvert \log_k(\frac{n}{s}(k-1)+1) \right\rvert + 1$	$SC_{CST} \times \frac{l}{s}$
CST^*-트리	$(\sum_{i=1}^{\frac{n}{s-1}} \left\lvert \log_k(\frac{i}{2}(k-1)+1) \right\rvert + 1) / \frac{n}{s-1}$	$SC_{CST^*} + \frac{l}{s-1}$

의 횟수를 수식으로 정리한 것이다. 여기서 n은 색인엔트리의 총 개수이고, s는 T-트리 한 노드가 가지는 색인엔트리의 개수이며, k는 CST-트리와 CST^*-트리의 한 노드블록이 가지는 자식 노드 포인터의 개수이다. 그리고 SC는 각 색인구조의 단일키 값 검색시 캐시미스 발생 횟수고, l은 범위질의 시 범위 내 키의 개수이다.

표 1의 T-트리, T^+-트리, T^*-트리 그리고 CST-트리의 수식은 각 논문의 내용을 바탕으로 하였다. 본 논문에서 제안하는 CST^*-트리는 하나의 노드에 s개의 색인엔트리가 저장될 경우 하나의 색인엔트리 대신 링크포인터를 사용하여 s-1개의 색인엔트리를 가지게 되고, 따라서 $\left\lvert \frac{n}{s-1} \right\rvert$개의 단말노드가 필요하다. 내부노드의 개수는 $\left\lvert \frac{n}{s-1}/2 \right\rvert$로 단말노드 개수의 반이고, 이러한 CST^*-트리의 내부노드들을 캐시메모리의 블록 크기만큼 노드블록으로 구성하여 캐시메모리에 올려 사용함으로써 캐시미스 발생 횟수를 줄인다. 이때 각 노드블록이 k개의 자식 노드 포인터를 가진다면 내부노드의 전체 높이는 $\left\lvert \log_k(\frac{n}{s-1}/2)(k-1)+1 \right\rvert$이고, 단말노드까지 한번의 높이를 더한 전체 트리의 높이는 $(\left\lvert \log_k(\frac{n}{s-1}/2)(k-1)+1 \right\rvert + 1)$이다.

위 식에서 CST^*-트리의 k값은 CST-트리의 k값보다 작다. 이유는 CST-트리의 내부노드들은 하나의 키 값(최대값)만을 가지고 있지만 CST^*-트리의 내부노드는 두개의 키 값(최소값, 최대값)을 가지기 때문이다. k의 값이 작다는 의미는 캐시메모리의 블록 사이즈에 맞게 내부노드를 노드블록으로 설정할 때 CST^*-트리가 CST-트리에 비해 노드블록 내에 포함되는 키의 개수가 작다는 의미이다. 따라서 노드블록을 캐시메모리로 가져올 때 CST-트리가 CST^*-트리보다 더 많은 노드를 가져올 수 있다. 그에 따라 단말노드까지의 검색이 이루어지는 경우 CST^*-트리가 CST-트리에 비해 많은 캐시미스가 발생한다는 단점이 있다. 하지만 내부노드에 존재하는 키 값의 검색이 이루어지는 경우에는 최소/최대값을 이용하여 검색을 하기 때문에 CST-트리처럼 반드시 단말노드까지 검색할 필요가 없는 점으로 이를 보완한다.

캐시미스는 노드블록의 높이만큼 발생하므로 CST^*-트리 단말노드까지의 캐시미스 발생 횟수는 $(\left\lvert \log_k(\frac{n}{s-1}/2)(k-1)+1 \right\rvert + 1)$이지만 CST^*-트리는 내부노드에 존재하는 키 값을 검색하는 경우 내부 색인노드 포인터를 통해 한번에 검색 할 수 있기 때문에 CST-트리처럼 반드시 단말노드까지 검색하지 않는다. 그러므로 CST^*-트리의 평균적인 단일키 값 검색시 캐시미스 발생 횟수는 $(\sum_{i=1}^{\frac{n}{s-1}} \left\lvert \log_k(\frac{i}{2}(k-1)+1) \right\rvert + 1) / \frac{n}{s-1}$이다.

범위질의를 고려하지 않은 T-트리와 CST-트리는 범위질의 내 키를 포함하는 노드의 개수만큼 단일키 값 검색으로 노드들을 검색해야만 한다. 범위질의 내 키의 개수가 l개이면 검색해야할 노드의 개수는 $\frac{l}{s}$이 되고, 각 색인구조의 단일키 값 검색 시 캐시미스 발생횟수를 SC라 했을때 T-트리와 CST-트리는 범위질의 시 $\frac{l}{s}$개 만큼의 SC가 발생한다. 따라서 T-트리와 CST-트리의 범위질의 시 캐시미스 발생 횟수는 $SC \times \frac{l}{s}$이다.

T^+-트리, T^*-트리 그리고 CST^*-트리의 경우 첫 번째 키 값을 검색한 후 노드의 후위포인터나 링크포인터를 통해 노드의 순차적인 접근이 가능하다. 그러므로 T^+-트리, T^*-트리 그리고 CST^*-트리의 범위질의 시 캐시미스 발생 횟수는 첫 번째 키 값을 검색한 후 검색해야할 노드의 개수를 더한 것이 된다. 범위질의 내 키의 개수가 l개일때 T^+-트리는 한 노드에 T-트리와 같은 개수의 색인엔트리를 포함하므로 검색해야할 노드의 개수는 $\frac{l}{s}$이다. 그러므로 T^+-트리의 범위질의 시 캐시미스 발생 횟수는 $SC + \frac{l}{s}$이다. T^*-트리와 CST^*-트리의 경우에는 하나의 색인엔트리 대신 후위포인터나 링크포인터를 사용하기 때문에 한 노드에 s-1개의 색인엔트리를 가지며 그에 따라 검색해야할 노드의 개수는 $\frac{l}{s-1}$이다. 그러므로

T^*-트리와 CST^*-트리의 범위질의 시 캐시미스 발생 횟수는 $SC+\frac{l}{s-1}$이다.

그림 13은 단일키 값 검색시 각 색인구조의 캐시미스 발생 횟수를 그래프로 나타낸 것이다. 여기서 캐시블록의 크기는 128바이트, s의 크기를 15로 고정하고 n의 크기를 200,000 ~ 500,000으로 변경하여 캐시미스 발생 횟수를 측정하였다. 그 결과 CST^*-트리, CST-트리, T-트리 순으로 캐시미스 발생 횟수가 적었다. 이는 표 1의 수식에서 유추할 수 있는 결과이고, CST-트리가 CST^*-트리에 비해 캐시미스 발생 횟수가 많은 이유는 CST-트리 내부노드에 존재하는 키 값을 찾기 위해서도 반드시 단말노드까지 검색을 해야만 하기 때문이다. T-트리와 T^*-트리가 CST^*-트리에 비해 캐시미스 발생 횟수가 많은 이유는 한 노드의 크기가 커서 노드에 한번 접근할 때마다 캐시미스가 발생하기 때문이다. T-트리와 T^*-트리의 캐시미스 발생 횟수를 비교해보면 거의 유사하지만 T^*-트리 캐시미스 발생 횟수가 좀 더 많은 것을 알 수 있다. 그것은 T^*-트리가 하나의 색인엔트리 대신 후위포인터를 사용하여 T-트리에 비해 한 노드가 가지는 색인엔트리의 개수가 적기 때문이다. T^+-트리의 캐시미스 발생 횟수가 가장 많은 이유는 다른 색인구조들에 비해 트리의 높이가 높고, 모든 색인엔트리가 단말노드에 존재하기 때문에 T^+-트리의 검색은 항상 단말노드까지 이루어지기 때문이다. 따라서 그림 13과 같은 그래프가 나타났다.

그림 14는 범위질의 시 각 색인구조의 캐시미스 발생 횟수를 그래프로 나타낸 것이다. 여기서 캐시블록의 크기는 128바이트, s의 크기를 15, 그리고 n의 크기를 200,000으로 고정하고 l의 값을 40 ~ 100으로 변경하여 캐시미스 발생 횟수를 측정하였다. 그 결과

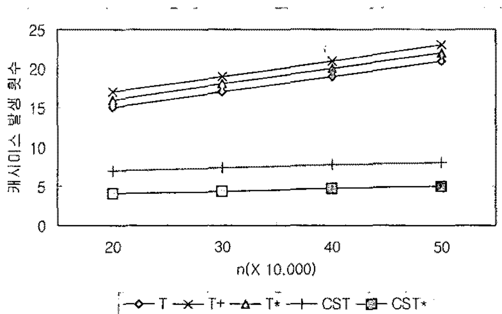


그림 13. 각 색인구조의 단일키 값 검색시 캐시미스 발생 횟수

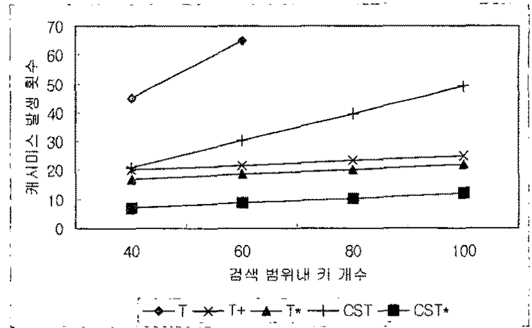


그림 14. 각 색인구조의 범위질의시 캐시미스 발생 횟수

CST^*-트리, T^*-트리, T^+-트리의 순으로 캐시미스 발생 횟수가 적었다. 이는 표 1의 수식에서 유추할 수 있는 결과이고, T^*-트리와 T^+-트리가 CST^*-트리에 비해 캐시미스 발생 횟수가 많은 이유는 첫 번째 키 값을 찾는데 더 많은 캐시미스가 발생하기 때문이다. T-트리와 CST-트리가 CST^*-트리에 비해 캐시미스 발생 횟수가 많은 이유는 첫 번째 키 값을 찾는데 더 많은 캐시미스가 발생할 뿐만 아니라, 범위질의를 고려하지 않았기 때문에 범위 내에 존재하는 키 값을 검색하기위해서 범위 내 키 값을 포함하는 모든 노드들을 루트노드로부터 다시 검색해야만 하기 때문이다. 따라서 그림 14와 같은 그래프가 나타났다.

5. 결 론

본 논문에서는 캐시메모리에 적응적이면서 동시에 범위질의에도 효율적인 새로운 메인메모리 데이터베이스 색인구조인 CST^*-트리를 제안한다. CST^*-트리는 캐시메모리에 적응적이기 위해 T-트리의 내부노드에 있던 색인엔트리들로 별도의 내부 색인노드를 구성하여 이를 가리키는 포인터만을 내부노드에 둠으로써 내부노드의 크기를 대폭 축소시켰다. 또 범위질의를 효율적으로 처리하기 위해 T-트리의 내부노드에서 떼어낸 내부 색인노드들과 단말노드들을 링크포인터를 통해 서로 연결하였다.

또한 본 논문에서 성능평가를 위해 지금까지 제안된 기존의 메인메모리 데이터베이스 색인구조들과 함께 CST^*-트리의 성능평가 비용 모델을 개발하고, 이를 통한 성능평가 결과를 도출하였다. 그 결과 CST^*-트리는 기존의 캐시만을 고려한 색인구조인

CST-트리에 비해 단일키 값 검색의 경우 캐시미스 발생 횟수가 20 ~ 30% 감소하였고, 범위질의의 경우에도 캐시미스 발생 횟수가 매우 적었다. 이는 단일키 값 검색의 경우 CST-트리가 내부노드에 존재하는 키 값을 찾기 위해서도 반드시 단말노드까지 검색을 필요로 하기 때문이고 범위질의의 경우 CST-트리는 범위질의를 고려하지 않았기 때문이다. 그리고 CST*-트리는 기존의 범위질의만을 고려한 T*-트리에 비해 단일키 값 검색질의의 경우 캐시미스 발생 횟수가 매우 적었으며 범위질의의 경우에도 캐시미스 발생 횟수가 10 ~ 20% 감소하였다. 이는 단일키 값 검색의 경우 T*-트리가 캐시메모리를 고려하지 않았기 때문이며 범위질의의 경우 첫 번째 키 값을 찾는데 CST*-트리에 비해 많은 캐시미스가 발생하기 때문이다.

참 고 문 헌

[1] D. Comer, "The Ubiquitous B-Tree," ACM Computing survey, No. 11, pp. 121-138, June 1979.

[2] I. H. Lee, J. H. Shim and S. G. Lee, "Fast Rebuilding B+-Trees for Index Recovery," IEICE Transactions on Information and Systems, pp. 2223-2233, 2006.

[3] S. W. Kim, H. S. Won, "Batch Construction of B+-Trees," Proc. of ACM Symposium on Applied Compution 2001, pp. 231-235, 2001.

[4] T. J. Lehman, M. J. Carey, "A Study of Index Structures for Main Memory Database Management Systems," Proc. of the 12th VLDB Conf., Aug. 1986.

[5] K. R. Choi, K. C. Kim, "T*-tree: A Main Memory Database Index Structure for Real Time Application," Proc. of the 3rd International Workshop on RTCSA, pp. 81-88, Nov. 1996.

[6] 최공림, 김기룡, 김경창, "T*-트리: 주기억 데이터베이스에서의 효율적인 색인기법," 한국통신학회논문지, 제21권 제10호, pp. 2597-2604, 1996년 6월.

[7] A. V. Aho, J. E. Hopcroft and J. D. Ullman, "The Design and Analysis of Computer Algorithms," Addison-Wesley Publishing Company, June 1974.

[8] A. J. Smith, "Cache memories," ACM Computing survey, No. 14, pp. 473-530, Sep. 1982.

[9] J. Rao, K. A. Ross, "Cache Conscious Indexing for Decision-Support in Main Memory," Proc. of the 25th VLDB Conf., pp. 78-89, 1999.

[10] J. Rao, K. A. Ross, "Making B+-Tree Cache Conscious in Main Memory," Proc. of ACM SIGMOD 2000, pp. 475-486, May 2000.

[11] 강대희, 이재원, 이상구, "CST-트리의 효과적인 범위 검색," 한국컴퓨터종합학술대회 논문집, 제33권 제1호, pp. 67-69, 2006년 6월.

[12] 이익훈 외 5명, "캐시를 고려한 T-트리 인덱스 구조," 한국정보과학회 논문지, 제32권 제1호, pp. 12-23, 2005년 2월.

[13] 이재원, 이익훈, 이상구, "최대키 값을 이용한 CST-트리 인덱스의 빠른 재구축," 한국컴퓨터종합학술대회 논문집, 제32권 제1호(B), pp. 85-87, 2005년 7월.

최 상 준

2007년 대구가톨릭대학교 컴퓨터공학과 졸업(학사)

2009년 대구가톨릭대학교 컴퓨터정보통신공학과(공학석사)

관심분야 : 색인구조, 데이터베이스 설계 및 관리, 데이터웨어하우스, XML 데이터베이스

이 종 학

1982년 경북대학교 전자공학과 (전자계산 전공) 졸업 (학사)
1984년 한국과학기술원 전산학과 졸업(공학석사)
1997년 한국과학기술원 전산학과 졸업(공학박사)
1991년 정보처리기술사
1984년~1987년 금성통신(주) 부설연구소 주임연구원
1987년~1998년 한국통신 연구개발본부 선임연구원
1998년~현재 대구가톨릭대학교 컴퓨터정보통신공학부 교수
관심분야 : 색인구조, 다차원 파일구조, 데이터베이스 설계, XML 데이터베이스, 데이터 웨어하우스, 생물정보학 등