레벨 프리페칭 기법을 이용한 향상된 주기억장치 상주형 색인구조☆

lpCSB+ - tree : An Enhanced Main Memory Index Structure Employing the Level Prefetching Technique

홍 현 택* 강 태 호** 유 재 수***
Hyun-Taek Hong Tae-Ho Kang Jae-Soo Yoo

요 약

주기억장치 상주형 색인구조에서는 2차 캐쉬 실패가 성능에 매우 큰 영향을 미친다. 기존에 제안된 주기억장치 상주형 색인구조들은 2차 캐쉬 실패를 고려하긴 했지만 여전히 트리의 각 레벨을 접근할 때는 2차 캐쉬 실패가 발생한다. 본 논문에서는 이러한 문제점을 인식하고 트리 순회 시 각 레벨을 방문할 때도 캐쉬 실패가 발생하지 않는 주기억장치 색인구조를 제안한다. 제안하는 색인구조인 lpCSB+-트리는 다음 레벨에서 방문할 가능성이 있는 노드들을 프리페치하여 다음 레벨을 방문할 때도 캐쉬 실패가 발생하지 않도록 한다. 또한, 기본적인 구조는 노드그룹 개념을 이용하여 노드의 팬-아웃을 증가시키는 CSB+-트리에 기반하지만 CSB+-트리의 단점인 분할 비용의 증가문제를 해결하기 위한 방법을 제안한다. 성능평가를 통해 기존의 색인구조와 비교하여 제안하는 색인구조의 우수성을 보인다.

Abstract

In main-memory resident index structures, secondary cache misses considerably have an effect on the performance of index structures. Recently, several main-memory resident index structures that consider cache have been proposed to reduce the impact of secondary cache misses. However they still suffer from full secondary cache misses whenever visiting each level of a index tree. In this paper, we propose a new index structure that eliminates cache misses even when visiting each level of index tree. The proposed index structure prefetches the grandchildren of a current node. The basic structure of the proposed index structure is from CSB+-tree that uses the concepts of the node group to increase fan-out. However the insert algorithm of the proposed index structure reduces the cost of a split significantly. Also, we show the superiority of our algorithm through various performance evaluation.

⌐ Keyword : Cache, Prefetch ,Index Structure, MMDBMS

1. 서 론

현대 컴퓨터 구조에서는 중앙처리장치의 속도와 주기억장치의 접근속도의 차이가 점점 커지면서 이들 간에 병목현상이 발생한다. 최근의 중앙처리장치는 이 병목현상을 해소하기 위해 캐쉬기법을 도입하고 있으며 일반적으로 1차 및 2차 캐쉬를 가지는 계층적 기억장치 구조를 가지고 있다. 주기억장치의 가격이 하락하면서 주기억장치 데이터베이스 관리 시스템(Main Memory Database Management System : MMDBMS)이 널리 쓰이고 있는데 디스크 입출력이 발생하지 않는 MMDBMS에서는 주기억장치와 중앙처리장치간의 속도 차이가 전체 성능에 미치는 영향이 크다. 이에 관련한 기존연구의 보고에 의하면 특히 2차 데이터 캐쉬 실패가 가장 큰 성능저하 요인이다[1,2].

* 정 회 원 : LG전자 UMTS 단말연구소 연구원
hongry@netdb.chungbuk.ac.kr(제 1저자)
** 정 회 원 : 충북대학교 대학원 정보통신공학과 박사과정
segi21@netdb.chungbuk.ac.kr(공동저자)
*** 정 회 원 : 충북대학교 전기전자컴퓨터공학부 부교수
yjs@cbucc.chungbuk.ac.kr(공동저자)
☆ 본연구는한국과학재단 목적기초연구(R01-2003-000-10627-0)의 지원으로 수행되었음

특히, 2차 데이터 캐쉬 실패는 데이터를 순차적으로 접근하지 않는 색인 트리와 같은 데이터 구조에 더 큰 영향을 미친다. 최근 이를 극복하기 위한 연구가 활발히 진행중이며 그 결과 CSS-트리[3], CSB+-트리[4], pkB-트리[5], pB+-트리[6]와 같은 캐쉬를 고려하는 색인구조들이 제안되었다. 이 색인구조들은 여러 측면에서 색인구조에서 발생할 수 있는 캐쉬 실패를 줄였지만 여전히 트리의 각 레벨을 접근할 때는 캐쉬 실패가 발생하고 있으며, 이는 성능에 큰 영향을 미치고 있다.

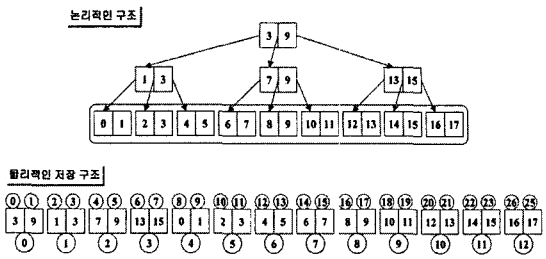
본 논문에서는 이러한 문제점에 착안하여 각 레벨에서도 캐쉬 접근 실패가 발생하지 않는 색인구조를 제안한다. 제안하는 색인구조인 lpCSB (level prefetching CSB)+-트리는 CSB+-트리에 기반하고 있으며, 프리페치 기법을 사용하여 현재 접근하는 노드의 자식과 손자 노드들을 미리 캐쉬에 올려놓음으로써 트리의 다음 레벨을 접근할 때 캐쉬 실패가 발생하지 않도록 한다.

본 논문의 구성은 다음과 같다. 2장에서 기존의 연구에 대하여 논의한 뒤 본 논문의 접근 방향을 서술하고, 3장에서 제안하는 lpCSB+-트리의 구조와 삽입, 탐색 방법을 서술하고 탐색 성능을 분석한다. 4장에서는 실제 구현을 통하여 제안하는 기법의 우수성을 증명하고 5장에서 결론을 맺는다.

2. 기존의 캐쉬 고려 색인 구조 및 접근 방향

이 장에서는 기존에 제시된 캐쉬를 고려한 색인 구조인 CSS-트리, CSB+-트리, pkB-트리, pB+-트리에 대하여 살펴보고 본 논문의 접근 방향에 대하여 서술한다.

CSS(Cache Sensitive Search)-트리는 OLAP 환경을 위해 고려된 색인구조로서 키들을 물리적으로 연속된 공간에 저장하여 포인터를 제거하였다. 트리에서 포인터는 다음에 접근할 위치를 알기 위하여 필요한 정보이므로, 이는 각 노드에서 모두 포함하고 있었다. 하지만, CSS-트리에서는 다음에



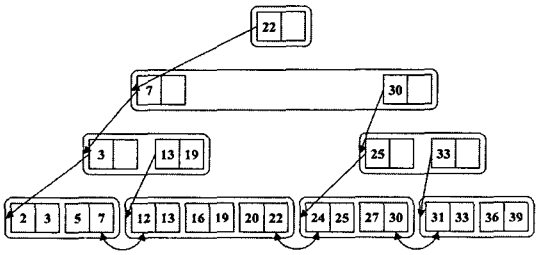
(그림 1) CSS-트리의 구조

접근할 위치를 알기 위한 방법으로 포인터를 사용하지 않고 k-ary 기법을 사용하여 계산하도록 하였다. 이로 인해 포인터가 차지하고 있던 공간에 더 많은 키를 포함할 수 있게 되었고, 따라서 한 노드가 포함할 수 있는 엔트리의 개수가 증가하였다.

이렇게 되면 동일한 개수의 엔트리로 트리를 구성하더라도 포인터를 포함했을 때보다 트리의 높이를 효과적으로 줄일 수 있다. 트리의 높이가 클 수록 캐쉬 실패는 많이 발생하고 CSS-트리에서는 이를 줄였으므로 보다 효과적으로 캐쉬 실패 횟수를 줄일 수 있다. 하지만 전체 트리를 물리적으로 연속된 공간에 저장하여야 하므로 변경이 발생할 경우 물리적인 연속성을 위하여 전체 트리를 재구성해야하는 단점이 있다. 그림 1은 CSS-트리의 한 구성 예를 보여준다.

CSB+(Cache Sensitive B+)-트리는 변경에 유연하게 대처하는 색인구조인 B+-트리를 캐쉬를 고려한 형태로 수정한 것으로서, OLTP 환경에 적합한 색인구조이다. 이는 한 노드의 자식 노드들을 물리적으로 연속된 공간인 노드 그룹에 저장하고, 부모 노드에는 자식 노드들을 갖고있는 노드 그룹을 향한 포인터 한 개만 저장함으로서 부모 노드의 포인터를 줄이는 기법을 사용하였다. 따라서 전체 트리를 연속된 공간에 유지하는 CSS-트리와 달리 변경에 대처할 수 있다.

그림 2는 CSB+-트리의 전체적인 구조를 보여준다. CSB+-트리는 균형 다원 탐색 트리이다. d차의 CSB+-트리에 있는 각 노드는 $m(d \leq m \leq 2d)$개의 키들을 포함한다. CSB+-트리의 노드는 자식 노드 그룹의 첫 번째 노드 포인터만을 저장



(그림 2) CSB+-트리의 구조

하여 공간 활용도를 높이게 되므로, CSB+-트리는 B+-트리에 비해서 한 노드 당 저장할 수 있는 키의 개수가 더 많아지게 된다. 예를 들어서 노드 크기가 64바이트이고 키와 포인터의 크기가 모두 4바이트라면 B+-트리는 노드 당 7개의 키를 포함할 수 있는 반면에 CSB+-트리는 14개의 키를 포함할 수 있다. 즉, CSS-트리와 유사하게 포인터의 개수를 줄임으로서 한 개의 노드가 여러 개의 엔트리를 포함할 수 있도록 하였고, 캐쉬 실패 횟수를 줄였다.

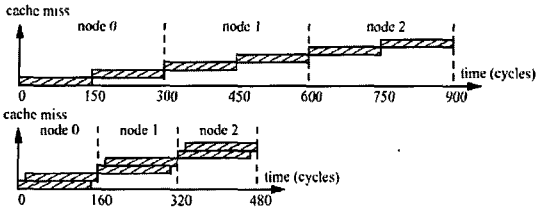
CSS-트리와 CSB+-트리의 경우는 물리적으로 연속된 공간을 사용하므로 삽입에 대해서는 좋은 성능을 보이지 않는다. CSB+-트리에 넘침이 발생했을 경우는 이전의 노드 그룹보다 큰 공간을 새로 할당받아 현재 노드 그룹의 내용을 복사하는 방법을 사용한다. 따라서 부담은 클 수밖에 없고, 이에 대한 대안으로 [4]에서는 노드 그룹 생성 시에 최대 크기만큼 노드 그룹을 할당하는 full CSB+-트리와, 노드 그룹을 2~3개의 세그먼트로 구성하는 segmented CSB+-트리를 제시하였다.

CSS-트리와 CSB+-트리가 포인터를 줄이는 것이 목적인 반면 pkB-트리는 키의 크기 및 비교 비용을 줄이는 것이 주 목적이다. 부분 키(partial key) 기법을 사용한 pkB-트리는 전체 키 중에서 고정된 일부분만을 비트로 인코딩 된 키로 이용하여 캐쉬 실패 횟수와 키 비교 비용을 줄인다. 이렇게 되면 저장공간을 효과적으로 줄일 수 있을 뿐만 아니라, 가변길이의 키나 크기가 큰 키의 경우에도 효과적으로 색인을 구성할 수 있다는 장점이 있다.

프리페치 기법을 B+-트리에 도입한 pB(prefetching B)+-트리는 기존의 방법에서 노드의 크기를 캐쉬라인의 크기로 제한했던 것과 달리 노드 크기를 캐쉬라인의 정수 배로 늘렸다. 노드의 크기를 여러 캐쉬라인으로 증가시킬 수 있었던 것은 최신 중앙처리장치에서 중앙처리장치의 연산과 병렬로 동작하면서 데이터를 가져올 수 있는 프리페치 명령어를 제공하기 때문이다. 즉, 중앙처리장치의 연산과는 별개로 데이터를 가져올 수 있기 때문에 다음에 접근할 데이터를 미리 캐쉬로 가져올 수 있다. 기존의 기법에서 노드의 크기를 캐쉬라인 하나로 제한하였던 이유는 노드가 여러 캐쉬라인이 된다면 한 노드 내에서도 캐쉬 실패가 발생하기 때문이다. 하지만, 노드의 첫번째 캐쉬라인을 접근하는 동안 프리페치 명령어를 사용하여 노드를 구성하고 있는 나머지 캐쉬라인을 가져온다면 캐쉬 실패는 발생하지 않는다.

프리페치 명령어에 대한 이해를 위하여 [6]에서 제시하고 있는 예가 그림 3에 나타나 있다. 이는 한 노드가 2개의 캐쉬 라인으로 구성되어 있을 경우 3개의 노드를 접근하는 예를 보여준다. 첫 번째 그림은 프리페치 명령어를 사용하지 않을 경우를 보여주고 있는데, 3개의 노드를 구성하고 있는 6개의 캐쉬라인 전체에서 캐쉬 실패가 발생하고 있음을 볼 수 있다. 그림 3의 두 번째 그림은 노드의 첫 번째 캐쉬라인을 접근하는 동안 나머지 캐쉬 라인을 프리페치 명령어를 사용하여 가져오는 경우를 보여준다. 그림을 보면 알 수 있듯이, 프리페치 명령어로 인해서 데이터를 병렬로 접근할 수 있게 되었다. 프리페치 명령어를 수행하는 시간이 소요되긴 하지만 이는 캐쉬 실패 시간에 비해 매우 적은 시간이다. 첫번째 경우에서는 전체 접근 시간이 900 사이클이 소요되었지만 프리페치 명령어를 사용한 두번째 경우는 480 사이클이 소요되어 데이터 접근에 상당한 이득이 있음을 알 수 있다.

또한 pB+-트리에서는 범위 검색을 위하여 Jump-pointer 기법을 제시하였는데, 이는 순차적으로 단



(그림 3) 노드의 크기가 캐쉬라인 2개일 경우 3개의 노드 접근 시간

말 노드들을 접근하는 범위 검색의 특성을 이용하여 다음에 접근하게 될 단말 노드를 미리 캐쉬에 올려놓는 방법을 사용한다. 하지만, pB+-트리에서도 여전히 트리의 높이에 비례하여 캐쉬 실패가 발생하게 된다.

이상에서 언급된 색인 구조들을 정리해보면 CSS-트리와 CSB+-트리는 트리의 포인터를 줄여서 캐쉬 실패 횟수를 줄였고, pkB-트리의 경우는 키의 크기를 줄여서 캐쉬 실패를 줄이는 기법을 사용하여 캐쉬 실패를 줄이는 이득을 얻었다. 이들의 공통적인 문제점은 각 레벨에서 캐쉬 실패가 발생한다는 것이다. 원하는 데이터가 캐쉬에 존재한다면 캐쉬 실패는 발생하지 않는다. 하지만, 트리의 특성상 다음 레벨에 접근할 노드는 현재 노드에서의 비교 연산이 끝난 후에 결정되게 된다. 즉, 이미 캐쉬에 데이터가 존재한다면 좋겠지만 다음에 접근할 노드를 알 수 없기 때문에 레벨에 따라 캐쉬 실패가 발생하게된다.

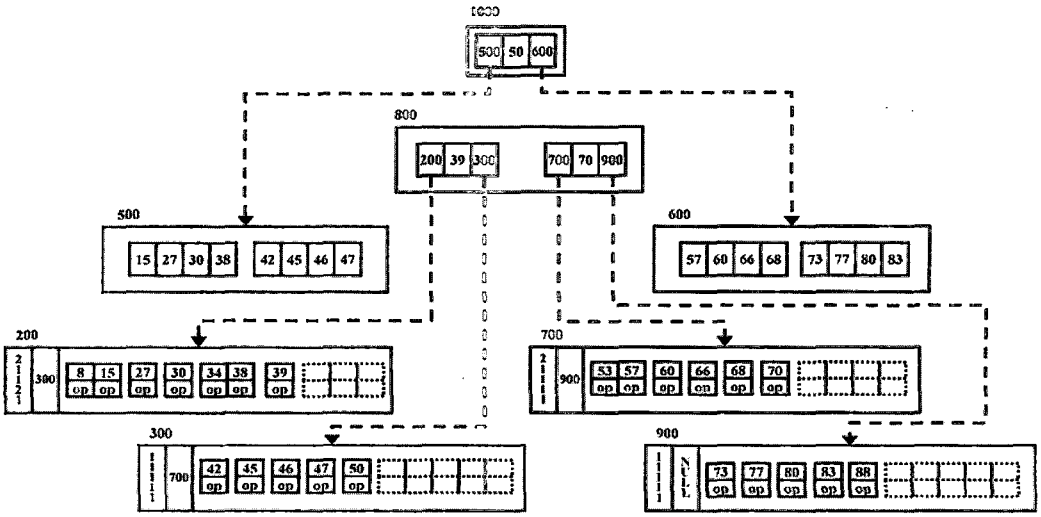
[6]에서는 이 문제를 인식하고 자식 노드나 손자 노드들을 병렬로 프리페치하는 등의 해결책과 그에 대한 문제점을 다음과 같이 지적하고 있다. 첫째로 자식 노드 포인터에 대한 데이터의 의존성, 둘째, 트리 노드들의 팬-아웃이 큰 점, 마지막으로 모든 자식노드들을 방문할 확률이 동일하다는 사실이다. 이들 중 가장 어려운 문제는 마지막 문제로 방문할 자손노드들을 예측할 수 없다는 것이다. 이미 서술하였다시피 트리의 각 레벨에서 캐쉬 실패가 발생하는 이유는 다음에 접근할 노드를 미리 알 수 없기 때문이다. 하지만, 접근할 노드는 현재 노드의 자식 중에 하나임은 분명하므로 현재 노드를 접근하기 이전에 현재 노드의 자식 노드들을 프리페치 해놓는다면 캐쉬 실패는 발생하지 않을 것이다. 그러나, 자식노드들을 프리페치하기 위해서는 먼저 그 노드들의 포인터들을 알아야 한다. 또한, 자식 노드들을 프리페치한다고 하더라도 높은 팬-아웃 때문에 동시에 프리페치해야 할 자식노드들의 크기가 너무 크다는 것이 문제가 된다.

본 논문에서는 이 문제점을 해결하기 위해 현재 접근중인 노드의 손자 노드들을 프리페치하는 접근 방법을 사용하며, 기본 구조로 CSB+-트리를 사용한다. CSB+-트리의 문제점으로 지적되는 것은 색인에 사용되는 키들을 정수로 한정했고 형제 노드들을 모두 연속된 공간에 저장하여 변경을 수행할 때 어렵다는 것이었다. 본 논문에서는 변경에 대한 비용을 줄이고 더불어 프리페치 기법을 적용하기 위해서 CSB+-트리의 구조를 수정한다. 또한, CSB+-트리의 변경 비용을 줄이기 위한 다른 방법으로 새로운 삽입 알고리즘을 제안한다.

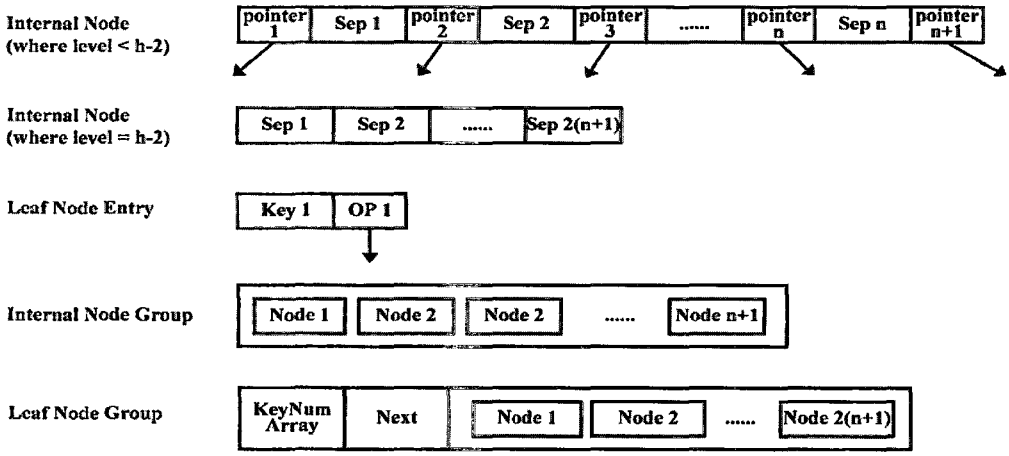
3. lpCSB+-트리

3.1 제안하는 색인 구조의 특징

본 논문에서는 색인 트리의 노드에서 자식 노드 그룹에 대한 포인터 하나만 유지하도록 하는 CSB+-트리의 구조를 프리페치 명령어를 이용하여 수정한다. 트리의 각 레벨을 접근하기 이전에 다음에 접근하게 될 가능성이 있는 노드들, 즉, 손자 노드들을 프리페치 명령어를 이용하여 미리 캐쉬에 올려놓는 기법을 사용한다. 프리페치 명령어는 중앙처리장치의 연산과 독립적으로 동작하면서 데이터를 캐쉬로 가져올 수 있는 명령이므로 현재 레벨에서 연산을 수행하면서 동시에 다음 레벨에서 접근하게 될 노드를 캐쉬로 가져올 수 있다. 현재 레벨에서 연산을 마친 뒤 다음 레벨로 접근할 때는 원하는 노드들을 이미 프리페



(그림 4) IpCSB+-트리의 구조



(그림 5) 노드와 노드 그룹의 구조

치 명령어를 통하여 가져왔으므로 캐쉬 실패가 발생하지 않게 된다.

3.2 구조

그림 4는 제안하는 색인 구조의 한 구성 예를 보여준다. CSB+-트리와 동일하게 한 노드의 자식 노드들은 물리적으로 연속된 공간인 노드 그룹에 저장되고, 각 노드 그룹에 대한 포인터가 한 개씩 트리에 유지된다. CSB+-트리에서 변경된 점은 이 포인터가 부모 노드에 존재하는 것이 아니고 할아버지 노드에서 유지된다는 점이다. 다시 말하면, 트리의 높이가 h이고 루트노드의 레벨을 0으로 할 때, 제안하는 색인 구조에서 레벨이 h-2 보다 적은 비 단말 노드들은 프리페치하게 될 손자 노드들의 노드 그룹에 대한 포인터들을 가지고 있다. 이 포인터는 다음 레벨에서 접근할 노드가 결정이 되면, 그 노드의 자식노드들을 프리페치하기 위하여 사용된다.

그림 5에서는 제안하는 색인 구조의 노드와 노

드 그룹 구조를 보여주고 있다. 레벨이 h-2보다 작은 비 단말 노드는 손자 노드 그룹들에 대한 포인터와 자식노드들에 대한 구분자로 구성되어 있고, 레벨이 h-2에 해당하는 비 단말 노드는 손자 노드들이 존재하지 않으므로 포인터를 가질 필요가 없으며 단지 구분자로만 구성되어 있다. 따라서, 전체 트리에 존재하는 포인터의 수는 CSB+-트리와 동일하다. 단말 노드의 경우는 노드의 크기에 제한이 없고 각 엔트리는 키값과 객체에 대한 포인터로 구성이 된다.

노드의 크기에 제한을 두지 않은 대신에 단말 노드 그룹에서는 각 노드들이 몇 개의 엔트리로 구성되어 있는 지에 관한 정보를 유지하고 있어야한다. 이 값들은 노드 그룹의 맨 앞에 배열의 형태로 갖고 있게 되며, 범위 탐색을 위한 다음 노드 그룹을 위한 포인터는 단말 노드 그룹에 한 개만 존재하면 된다. 이런 정보를 위한 추가적인 공간에 대한 부담은 하나의 노드 그룹에 포함되는 노드들의 개수가 작아서 4~8비트면 충분하므로 무시할 수 있다.

3.3 삽입 알고리즘

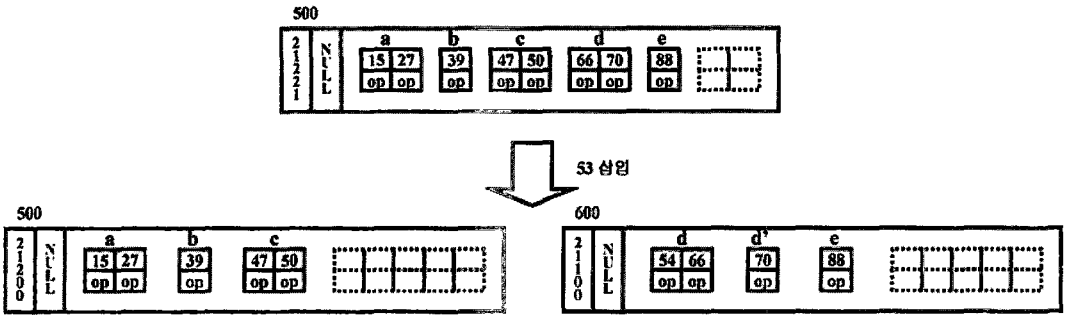
제안하는 색인구조의 삽입 연산은 두 단계로 수행된다. 첫 번째 단계는 새로운 엔트리를 삽입할 단말노드를 찾는 단계로서 맨 처음 루트 노드를 접근한다. 하지만 루트노드를 접근하기 전에 루트노드의 자식노드들을 병렬로 프리페치한다. 루트노드와 루트의 자식 노드그룹에 대한 포인터는 색인 설명자에 미리 기록이 되어 있다고 가정한다. 루트노드에서 삽입자가 다음 방문할 자식노드를 결정하면 그 자식노드의 자식들을 병렬로 프리페치한다. 즉, 두 레벨 다음에 접근하게 될 가능성이 있는 노드들을 캐쉬에 올려놓는 것이다. 이때 루트노드의 손자노드그룹에 대한 포인터는 루트노드에 저장된다. 다음에 삽입자는 결정된 자식노드를 방문한다. 삽입자는 이러한 과정을 다음 레벨에서도 반복해서 새롭게 삽입할 엔트리를 위한 단말 노드를 찾는다.

두 번째 단계에서 삽입자는 새로운 엔트리를 단말노드에 삽입한다. 이때 넘침이 발생하면 분할을 수행하여 넘침을 처리한다. 단말 노드에서의 분할 알고리즘은 CSB+-트리의 변경 비용을 줄이는 형태로 진행된다.

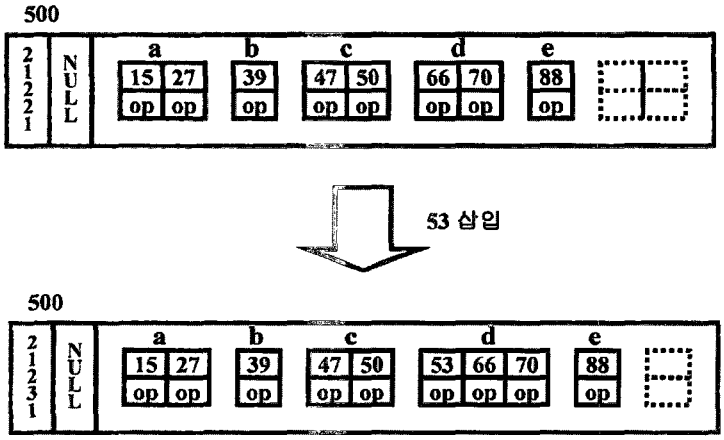
제안하는 색인 구조의 단말 노드들은 그 크기가 고정되어 있지 않으며, 단지 노드 그룹이 포함할 수 있는 단말 노드의 갯수만 고정되어 있다. 단말 노드 그룹의 크기는 최대만큼 이미 확보되어 있으며 그 공간을 활용하여 단말 노드의 크기를 증가시키는 형태로 삽입이 진행된다. 즉, 분할은 노드 그룹이 꽉 찼을때만 발생하게 된다.

그림 6과 같이 한 단말 노드 그룹이 메모리 주소 500번지에 저장되어있고, 단말 노드의 개수는 5개인 경우를 가정하자. 현재 네 번째 노드인 노드 d에 53이 삽입되어야 한다. 두가지 경우를 생각할 수 있는데 우선 CSB+-트리에서 처럼 단말 노드의 크기를 고정하는 경우인 그림 6을 살펴보자. 단말 노드가 포함할 수 있는 엔트리의 갯수를 2로 제한한다면 53이 삽입됨으로써 노드 d는 분할되어야 한다. 하지만 이미 현재 노드 그룹에는 노드 그룹이 포함할 수 있는 최대 노드 갯수만큼 꽉 차 있으며 따라서, 현재 노드 그룹을 분할해야 한다. 노드 그룹 분할을 위해서는 주소가 600인 새로운 단말 노드 그룹을 할당받고, 분할로 인해 생성된 노드까지 포함하여 6개의 노드를 두개의 노드 그룹에 3개씩 할당한다. 결과는 그림 6에서와 같으며 원래의 노드그룹에는 a, b, c 노드가 할당되었으며, 새로운 노드 그룹에는 d, d', e 노드가 할당되었다. 이는 CSB+-트리의 분할이다. 이와 같이 수행할 경우 이전의 노드그룹에 여유공간이 있음에도 불구하고 새로운 노드 그룹을 할당받아야한다는 부담이 있으며, 노드의 복사에 따른 부담도 크다.

본 논문에서 제안하는 분할 기법은 그림 6에서처럼 노드 d를 분할하지 않는 대신에 노드 그룹의 여유공간을 이용해 현재 노드 그룹을 재구성



(그림 6) 단말 노드의 크기를 고정했을 경우의 삽입



(그림 7) 단말 노드의 크기를 고정하지 않을 경우의 삽입

한다. 제안하는 트리에서는 노드를 캐쉬로 가져올 때 항상 노드 그룹단위로 가져온다. 따라서 노드 단위로 접근하던 기존의 방식과는 달리 노드그룹 전체에 엔트리들을 포함시키고 노드간의 경계를 배열로 유지할 수 있다. 그림 7에서 이를 보여주고 있는데, 삽입 이전에 노드 그룹에는 두개의 엔트리를 포함할 수 있는 여유공간이 있었다. 이 공간을 이용하여 53이 삽입될 노드 d의 크기를 늘리고 그 위치에 53을 삽입한다. 다음으로 각 노드들의 엔트리 개수를 포함하고 있는 노드 그룹 맨 앞단의 배열 값을 수정한다. 그림 7에서 삽입 이전에 네 번째 노드 d의 엔트리 개수가 2였었고, 삽입 이후에 3으로 변경된 것을 알 수 있다. 이렇게 분할을 수행하면 분할은 전체 노드 그룹이 꽉 찼을 때만 수행된다.

제안하는 분할 기법은 CSB+-트리의 기법보다 분할이 적게 발생할 뿐만 아니라 공간의 사용측면에서도 훨씬 효율적이다. 비 단말 노드의 분할은 CSB+-트리의 기법과 동일하다.

그림 8에 Insert 모듈의 의사코드가 나타나있다. Insert 모듈은 삽입되려는 엔트리, 현재 노드 그룹에서 노드의 위치, 현재 노드 그룹, 자식 노드 그룹을 매개변수로 받으며, 현재 노드는 세번째 매개변수인 curNodeGrp의 pos 위치에 해당하는 노드이다. 1행에서는 현재 노드에서 엔트리의 위치를 검색을 수행하며 검색 결과로 반환되는 loc 값은 노드 내에서 엔트리의 위치가 된다. 2행에서 두 레벨 다음에 접근하게 될 손자 노드들을 프리페치한다. 3행에서 현재 노드의 높이가 h-2인지를 확인하는데, 노드의 높이가 h-2인 경우는 포인터

```
Insert ( entry, pos, curNodeGrp, childNodeGrp )

 1 : loc = binarySearch(entry, 현재 노드);
 2 : 현재 노드의 loc 번째 손자 포인터에 해당하는 노드 그룹 프리페치;
 3 : if ( 현재 노드의 높이 != h-2 )
 4 :  newChildNodeGrp = Insert ( entry, loc, childNodeGrp, 현재 노드의 loc번째 손자 포인터 );
 5 : else
 6 : {
 7 :  newChildNodeGrp = InsertBottom ( entry, loc, childNodeGrp );
 8 : }
 9 : if ( newChildNodeGrp != NULL )
10 : {
11 :  if ( curNodeGrp 분할 발생 )
12 :  {
13 :          newNodeGrp = 새로운 노드 그룹 생성;
14 :          newNodeGrp에 노드 할당;
15 :          curNodeGrp에 노드 할당;
16 :          return newNodeGrp;
17 :  else
18 :  {
19 :          newCurNodeGrp = curNodeGrp 보다 노드 하나만큼 큰 노드 그룹 할당;
20 :          curNodeGrp 을 loc 위치에 새 노드를 고려하여 newCurNodeGrp으로 복사;
21 :          현재 노드에 childNodeGrp 으로부터 엔트리 재구성;
22 :          newCurNodeGrp의 loc 위치의 노드에 newChildNodeGrp 으로부터 엔트리 재구성;
23 :  }
24 : }
25 : return NULL;
```

(그림 8) Insert 모듈의 의사코드

를 하나도 포함하지 않는 최하단 비 단말 노드가 된다. h-2가 아닌 경우는 다시 Insert 모듈을 재귀 호출하고, h-2인 경우는 InsertBottom 모듈을 호출한다. 호출한 함수의 내부에서 새로운 노드 그룹이 생성되었을 경우에는 현재 노드 또한 분할되어야 한다. 현재 노드가 분할될 경우 현재 노드 그룹에 넘침이 발생하는 지 확인하여 넘침이 발생할 경우 13~15 행에서 새로운 노드 그룹을 할당한 뒤 노드들을 분배하고, 새로 생성된 노드 그룹의 포인터를 반환한다. 넘침이 발생하지 않는 경우는 19~22 행에서 현재 노드 그룹보다 노드 한 개만큼 더 큰 노드 그룹을 할당받아 현재 노드 그룹에 새로운 노드를 삽입한다.

그림 9에 리프 노드에 대한 삽입을 수행하는 InsertBottom 모듈에 대한 의사코드가 나타나있다. 이 모듈은 삽입하려는 엔트리와 현재 노드 그룹

```
InsertBottom (entry, pos, leafNodeGrp)

 1 : loc = binarySearch(entry, 현재 노드);
 2 : if ( leafNodeGrp 에 여유공간이 있을 경우 )
 3 : {
 4 :       현재 노드의 loc 번째 이후의 엔트리들 한칸씩
           뒤로 복사;
 5 :       현재 노드의 loc 번째에 entry 삽입;
 6 :       leafNodeGrp에서 현재 노드의 엔트리 개수 수정;
 7 :       return NULL;
 8 : }
 9 : else
10 : {
11 :       newLeafNodeGrp = 새로운 리프 노드 그룹
           할당;
12 :       newLeafNodeGrp의 엔트리 구성;
13 :       leafNodeGrp의 엔트리 구성;
14 :       return newLeafNodeGrp;
15 : }
```

(그림 9) InsertBottom 모듈의 의사코드

에서의 노드 위치, 현재 리프 노드 그룹을 매개변수로 받는다. 현재 리프 노드에서 엔트리의 삽입 위치를 검색한 뒤 현재 노드그룹의 여유공간 여부를 판단한다. 여유공간이 있을 경우는 4~6 행에서 현재 노드 그룹에 엔트리를 삽입을 수행하고 여유공간이 없을 경우는 11~13 행에서 새로운 노드 그룹을 할당받아 구성한다.

3.4 탐색 알고리즘

탐색 알고리즘은 삽입의 첫번째 단계와 유사하다. 루트 노드를 접근하기 이전에 색인 설명자에 기술된 포인터를 참조하여 루트 노드와 자식 노드들을 프리페치한다. 그리고 다음 순회할 자식 노드 A가 결정되면 루트노드에 포함된 포인터를 참조하여 A의 자식 노드들을 포함하고 있는 노드 그룹을 프리페치하고, 노드 A에 대한 탐색을 수행한다. 이러한 과정을 단말 노드에까지 진행한다. 이렇게 되면 탐색 수행 중에 각 레벨에서 발생하는 2차 캐쉬 실패 횟수를 줄일 수 있다.

그림 4에서 57을 탐색하는 경우의 예를 들어보자. 우선 색인 설명자에 루트 노드의 위치에 대한 1000과 루트 노드의 자식 노드 그룹에 대한 800이 기록되어 있고, 탐색을 수행하기 전에 1000번지와 800 번지의 노드들을 프리페치한다. 그림 4에서는 50, 39, 70의 구분자를 갖고 있는 노드 3개가 된다. 루트 노드에서 57을 탐색하여 오른쪽으로 분기해야 함이 결정되면 오른쪽 자식으로 분기하기 이전에 두 레벨 다음에 접근할 가능성이 있는 손자 노드들을 프리페치한다. 손자 노드 그룹의 주소는 루트 노드에 기록된 바에 의하면 600번지이고 이들에 대한 프리페치 명령어를 수행시킨 후 구분자로 70을 갖고 있는 오른쪽 자식 노드로 진행한다. 현재 노드에서 탐색을 수행하여 57을 검색하기 위해 왼쪽 자식으로 분기해야 함이 결정되면 위에서와 마찬가지로 손자 노드들이 포함되어 있는 700번지의 노드 그룹에 대한 프리페치 명령어를 수행시킨 뒤 다음으로 진행한다.

```
Search ( entry, curNode, childNodeGrp )

1 : loc = binarySearch( entry, curNode );
2 : if ( curNode의 높이 != h-2 )
3 : {
4 :         return Search ( entry, childNodeGrp의 loc
            번째 노드, curNode의 loc 번째 손자 포인터 );
5 : }
6 : else
7 : {
8 :         return childNodeGrp의 loc 번째 노드에서
            entry의 위치 탐색;
9 : }
```

(그림 10) Search 모듈의 의사코드

다음 노드는 포인터를 포함하지 않은 노드이며 600번지의 첫번째 노드이다. 이 노드는 이미 루트 노드에서 탐색을 수행한 다음 프리페치를 수행하였으므로 캐쉬에 존재하고 있으며, 캐쉬 실패는 발생하지 않는다. 탐색을 수행하면 다음 진행할 노드는 첫번째 자식임을 알 수 있고, 탐색은 700번지 리프 노드 그룹의 첫번째 노드에서 종료된다.

그림 10에 탐색을 수행하는 Search 모듈의 의사코드가 나타나있다. Search 모듈은 탐색하려는 엔트리와 탐색을 수행하려는 노드, 자식 노드 그룹을 매개변수로 받는다. 현재 노드에서 엔트리의 위치 loc을 1행에서 탐색한 뒤 현재 노드가 최하위 비 단말 노드인지를 2행에서 판별한다. 최하위 비 단말 노드가 아닐 경우 4행에서 자식 노드 그룹의 loc 번째 노드와, curNode의 loc 번째 포인터를 가지고 Search 모듈이 재귀 호출된다. 만약 최하위 비 단말 노드라면 자식 노드 그룹의 loc 번째 노드에서 entry의 위치를 탐색하여 탐색 결과를 반환한다.

3.5 탐색 성능 분석

제안하는 색인 구조의 탐색성능과 CSB+-트리의 노드 크기를 pB+-트리의 기법을 도입하여 확장시킨 pCSB+(prefetching CSB+)-트리의 탐색 성능을 캐쉬 실패 회수 관점에서 수학적으로 분석

한다. 높이가 h인 색인 트리의 탐색 연산의 총 수행시간(T_{total})를 다음의 수식 1로 표현할 수 있다.

$$T_{total} = T_1 + \sum_{l=1}^{h-1} \{\alpha T_1 + wT_p(tn - \alpha - pn)\}$$
$$\alpha = \begin{cases} pn = 0, & 1 \\ pn > 0, & 0 \end{cases}$$
$$T_1 = T_c + \{T_p(w-1)\}$$

(수식 1)

T_1은 w개의 캐쉬 라인으로 구성된 하나의 노드를 적재하는데 걸리는 평균 지연시간이다. 이 시간은 첫 번째 캐쉬라인을 적재하는데 걸리는 캐쉬 실패 지연 시간 T_C와 하나의 노드를 구성하는 나머지 캐쉬라인을 프리페치하는데 걸리는 지연시간으로 구성된다. T_P는 하나의 캐쉬라인을 프리페치하는데 걸리는 시간으로서 TC에 비해 매우 작은 값이다. T_{total}은 루트 노드를 접근하는 시간 T_1과 높이와 pn값과 tn값에 의존하는 부분으로 구성되어 있다.

pn은 하나의 노드그룹을 구성하는 노드 중 이미 프리페치된 노드들의 개수이다. pn이 0보다 크면 최소한 노드그룹의 첫 번째 노드는 이미 프리페치되어 있다는 것이다. 그렇지 않다면, 노드그룹의 첫 번째 노드를 로드하는데 완전 캐쉬 실패 지연 시간만큼 지연되게 된다. tn은 노드 그룹에서 실제로 접근할 자식노드의 위치이다. tn이 pn과 같거나 pn보다 작다면 원하는 데이터가 캐쉬에 이미 존재하고 있기 때문에 캐쉬 실패로 인한 지연은 발생하지 않으며 프리페치로 인한 지연도 발생하지 않는다. 만일 tn이 pn보다 크고 pn이 0보다 크다면 프리페치로 인한 지연이 발생한다. 이처럼 pn은 색인 트리의 성능에 심각한 영향을 끼친다. 만일 pn이 0보다 크다는 것을 확신할 수 있다면 매우 큰 성능향상을 기대할 수 있다.

pCSB+-트리의 T_{total}은 각 레벨에서 캐쉬실패를 경험해야 하므로 수식 2와 같다.

$$T_{total} = \sum_{l=0}^{h-1} T_1 \qquad \text{(수식 2)}$$

제안하는 색인구조의 탐색성능은 전적으로 pn와 h에 의존적이다. 이전 노드에서의 수행시간이 첫번째 자식 노드를 프리페치하는 시간보다 작다면 pn값이 항상 0이되고, pn이 0이면 각 레벨을 접근할 때마다 T_1 지연이 발생한다. 일반적으로 lpCSB+-트리의 h는 pCSB+-트리보다 높은데, 제안하는 방법에서는 손자노드들을 프리페치해야 하므로 트리의 노드 크기를 pCSB+-트리만큼 확장할 수 없기 때문이다. 만약 노드 크기를 확장한다면 프리페치해야 할 캐쉬라인의 수는 지수적으로 증가하게 되어, 성능에 악영향을 미친다. 탐색 성능에 큰 영향을 미치는 h와 pn이 성능에 큰 영향을 미치는 요인이기 때문에, 단순히 제안하는 색인 트리의 탐색 성능을 수식을 통해서 pCSB+-트리와 비교할 수는 없다.

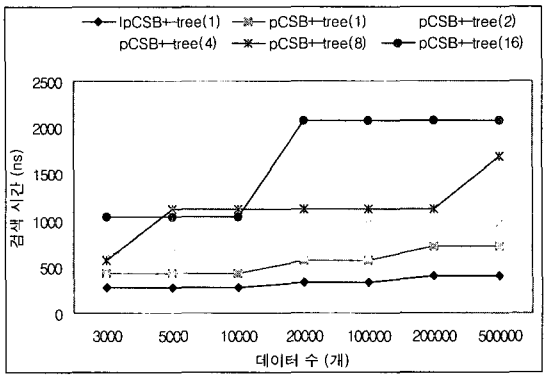
4. 구현 및 성능평가

구현에 사용된 시스템은 펜티엄-IV 1.7GHz 프로세서[7]에 256Mbyte의 메모리를 가지며, 운영체제는 RedHat 리눅스 7.1을 사용하였고, 컴파일러는 gcc 2.96이었다. 성능평가에 사용된 파라미터들이 표 1에 나타나있다.

성능평가에 사용된 데이터 집합은 균등 분포의 정수 500,000개로서 데이터 개수를 변경시키며 수

(표 1) 성능평가 파라미터

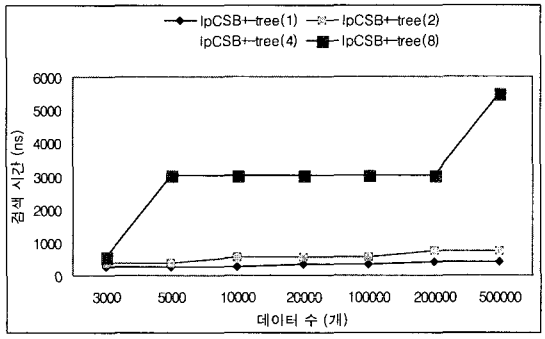
항목		값
시스템 파라미터	전체 L2 캐쉬 크기	256 KByte
	L2 캐쉬 라인 크기	64 Byte
	캐쉬 실패 비용	88.2 ns
	캐쉬 프리페치 비용	5.5 ns
	캐쉬라인 탐색 비용	54.2 ns
성능 파라미터	데이터 집합	균등분포 500,000 개
	노드 크기	1~16 캐쉬라인



(그림 11) 데이터 수와 노드 크기 변화에 따른 pCSB+-트리와 lpCSB+-트리의 검색시간

행하였고, 또한 색인 구성 시 한 노드의 크기는 1~16 캐쉬 라인수로 변경시키며 수행하였다. 성능평가에 사용된 펜티엄-IV 중앙처리장치의 총 2차 캐쉬의 크기는 256KByte이고, 하나의 2차 캐쉬 라인의 크기는 64Byte였다. 캐쉬 검색 시간과 관련하여 만약 검색하고자 하는 영역이 캐쉬에 없을 경우 발생하는 캐쉬 실패 지연시간은 88.2 ns였으며 하나의 캐쉬 라인을 프리페치 하는데 드는 지연시간은 5.5 ns이었다. 추가적으로 검색을 수행할 영역이 캐쉬 라인 상에 있을 때, 캐쉬 라인 상에서 원하는 키 값을 찾는데 소요되는 시간은 54.2 ns였다.

성능평가를 통해 제안하는 색인 기법을 pCSB+-트리와 검색 시간 관점에서 비교를 하였다. 먼저 그림 11은 데이터의 개수를 3000~500,000개까지 변경시키면서 lpCSB+-트리와 pCSB+-트리를 실험한 결과이다. 성능평가 시 lpCSB+-트리의 경우 노드의 크기를 하나의 캐쉬라인 크기로 하였으며, pCSB+-트리의 경우에는 노드의 크기를 1~16개의 캐쉬라인으로 변경하면서 수행하였다. 성능평가 결과 lpCSB+-트리가 pCSB+-트리보다 모든 경우에서 우수한 성능을 나타내었다. pCSB+-트리의 경우 한 노드의 크기를 하나의 캐쉬 라인으로 하였을 경우 가장 우수한 성능을 나타내었으며, 제안하는 lpCSB+-트리는 이보다 검색 시간측면에서 약 20% 향상된 결과를 나타내었다. 이는 상위노



(그림 12) 데이터수와 노드크기의 변화에 따른 lpCSB+-트리의 검색시간

드에서 자식 노드를 탐색하는 과정에서 원하는 자식노드를 프리페치 하였기 때문이다.

그림 12는 lpCSB+-트리의 노드 크기를 여러 개의 캐쉬 라인으로 하였을 경우에 대한 성능 평가이다. 성능평가 결과 lpCSB+-트리의 경우에는 하나의 캐쉬 라인 크기로 하였을 경우 가장 좋은 성능을 보이고 있다. 그 이유는 노드의 크기가 커질수록 프리페치 해야하는 손자 노드그룹내의 노드의 개수가 지수적으로 증가하여 노드그룹 적재 시간이 급격히 증가하기 때문이다.

5. 결 론

본 논문에서 제안한 lpCSB+-트리는 기존에 제안된 캐쉬를 고려한 주기억장치 상주형 색인구조들이 공통적으로 트리의 각 레벨을 접근할 때 캐쉬 실패가 발생하는 문제를 해결하였다. 제안하는 색인구조는 CSB+-트리의 구조에 기반하며 프리페칭 기법을 사용하여 현재 접근 중인 노드의 자식과 손자 노드들을 미리 캐쉬에 올려놓는 방법을 이용하였다. 그리고, CSB+-트리의 문제점인 분할 비용을 줄이기 위해 새로운 분할 알고리즘을 제안하였다. 성능평가 결과 제안하는 방법이 기존의 방법에 비해 검색측면에서 20% 정도 성능이 향상됨을 볼 수 있었다. 향후 연구에서는 실제 환경에서 다양한 실험을 통해 제안하는 색인구조의 성능을 분석한다.

참고문헌

[1] Anastassia Ailamaki, David J. DeWitt, Mark D Hill and David A. Wood, "DBMSs on a Modern Processor: Where Does Time Go?", In Proceedings of VLDB Conference, pp. 266~277, 1999.

[2] Stefan Manegold, Peter A. Boncz and Martin L. Kersten, "Optimizing database architecture for the new bottleneck: memory access", In VLDB Journal 9(3), pp. 231~246, 2000

[3] Jun Rao and Kenneth A. Ross, "Cache Conscious Indexing for Decision-Support in Main Memory", In proceedings of VLDB conference, pp. 78~79, 1999.

[4] Jun Rao and Kenneth A. Ross, "Making B+-Trees Cache Conscious in Main Memory", In proceedings of ACM SIGMOD Conference, pp. 475~486, 2000.

[5] Philip Bohannon, Peter McIlroy and Rajeev Rastogi, "Main-Memory Index Structures with Fixed-Size Partial Keys", In proceedings of ACM SIGMOD Conference, pp. 163~174, 2001.

[6] Shimin Chen, Phillip B. Gibbons and Todd C. Mowry, "Improving Index Performance through Prefetching", In proceedings of ACM SIGMOD Conference, pp. 235~246, 2001.

[7] Intel Corporation, "IA-32 Intel® Architecture Software Developer's Manual", Intel Corporation Order Number 245470-007, 2002.

◐ 저 자 소 개 ◑

홍 현 택
2001년 충북대학교 정보통신공학과 졸업(학사)
2003년 충북대학교 대학원 정보통신공학과 졸업(석사)
2003년~현재 : LG전자 UMTS 단말연구소 연구원
관심분야 : 데이터베이스 시스템, 이동통신 시스템, 자료저장 시스템, XML, etc.
E-mail : hongry@netdb.chungbuk.ac.kr

강 태 호
1999년 호원대학교 정보통신공학과 졸업(학사)
2002년 충북대학교 대학원 정보산업공학과 졸업(석사)
2003년~현재 : 충북대학교 정보통신공학과 박사과정
관심분야 : 데이터베이스 시스템, 웹 컨텐츠 관리시스템, 웹 마이닝, 자료저장 시스템, etc.
E-mail : segi21@netdb.chungbuk.ac.kr

유 재 수
1989년 전북대학교 컴퓨터공학과 졸업(학사)
1991년 한국과학기술원 전산학과 졸업(석사)
1995년 한국과학기술원 전산학과 졸업(박사)
1996년~현재 : 충북대학교 전기전자컴퓨터공학부 및 컴퓨터정보통신연구소 부교수
관심분야 : 데이터베이스 시스템, 정보검색, 멀티미디어 데이터베이스, 분산객체 컴퓨팅, etc.
E-mail : yjs@cbucc.chungbuk.ac.kr