부분키를 사용한 캐쉬 인식 B+ 트리†

이동민[O] 김원식 한욱신
경북대학교 컴퓨터공학과
{dmlee[O],wskim}@www-db.knu.ac.kr, wshan@knu.ac.kr

Cache Sensitive B+ Trees with Partial Keys

Dongmin Lee[O] Won-Sik Kim Wook-Shin Han
Dept. of Computer Engineering, Kyungpook National University

요 약

캐쉬 인식 트리는 주기억 장치의 느린 액세스 속도를 캐쉬를 활용함으로써 개선하려는 주기억 장치 데이터베이스 인덱스이다. 본 논문에서는 기존의 캐쉬 인식 트리에서 사용된 기법들을 살펴보고, 이를 통합, 개선하여 새로운 캐쉬 인식 트리를 제안한다. 기존의 캐쉬 인식 트리는 키 압축과 포인터 압축을 따로 고려하였기 때문에 각각 노드의 크기와 키의 길이 변화에 취약한 문제점이 있었다. 이에 반해 제안하는 부분키 캐쉬 인식 B+트리는 키와 포인터를 동시에 압축하여 이러한 문제점을 해결하고, 보다 캐쉬 활용도를 높였다. 또한 제안하는 트리의 벌크로드, 삽입, 삭제 알고리즘을 제시하고, 이론적인 분석을 통해 각 알고리즘이 올바르게 수행되고, 성능이 개선됨을 확인하였다.

1. 서론

주기억 장치로 사용되는 DRAM의 가격이 급속하게 하락함에 따라, 데이터베이스 인덱스를 주기억 장치에 상주할 수 있는 것이 가능하게 되었다. 그러나, CPU의 처리속도와 주기억 장치의 속도의 차이가 여전히 크고 점점 더 벌어지고 있어서, 주기억 장치 액세스 비용을 줄이기 위한 많은 연구가 진행 중이다. 즉, CPU와 주기억 장치 사이의 L1 및 L2 캐쉬를 활용할 수 있는 캐쉬 인식 트리에 대한 연구가 활발하게 진행 중이다[1, 2, 3].

캐쉬를 활용하는 캐쉬 인식 B-트리에 대한 연구는 압축 대상에 따라 1) 포인터 제거에 기반한 캐쉬 인식 트리와 2) 키 압축에 기반한 캐쉬 인식 트리로 나눌 수 있다. 포인터 제거에 기반한 캐쉬 인식 B-트리는 엔트리의 포인터를 제거하여 블록킹 인수(blocking factor)를 증가시킴으로써 캐쉬 미스를 감소시키는 트리로서, 대표적인 예로 CSB+-트리[2]가 있다. 키 압축에 기반한 캐쉬 인식 B-트리란 노드내의 키를 압축하여 블록킹 인수를 증가시킴으로써 캐쉬 미스를 감소시키는 트리로서, 대표적인 예로 부분키 B-트리(partial key B-tree)[3]가 있다.

포인터 제거에 기반한 CSB+-트리는 노드내에 저장되어 있는 포인터들의 일부 혹은 전체를 제거함으로써 하나의 노드에 더 많은 엔트리를 저장할 수 있도록 하였다. 즉, 노드 *n*이 가리키고 있는 자식 노드들을 연속된 공간에 함께 저장하고, *n*에는 첫번째 자식노드를 가리키는 포인터만을 저장함으로써 노드내의 저장이 필요한 포인터의 개수를 줄였다. 그러나 CSB+-트리에서는 노드에 저장되는 키는 고정길이의 정수형만을 다루고 있고, 가변길이의 경우를 고려하지 않은 문제점을 가진다. 특히 길이가 큰 가변길이 키의 경우에는 키 크기를 줄이는 것이 블록킹 인수에 큰 영향을 미치므로 이를 함께 고려하는 것이 필요하다.

키 압축에 기반한 부분키 B-트리는 연속된 키 사이에서 차이가 나는 부분 중 고정길이 일부만을 저장함으로써 키를 압축한다[3]. 따라서 길이가 큰 가변길이 키 대신에 크기가 작은 고정길이 키를 저장함으로써 캐쉬 미스의 발생 횟수를 줄일 수 있다. 그러나 키와 함께 엔트리를 구성하는 포인터의 압축은 고려하지 않은 문제점을 가진다. 특히, 64비트 머신의 경우 포인터의 크기가 8바이트나 되므로 포인터 제거도 함께 고려하는 것이 필요하다.

따라서 본 논문에서는 이러한 독자적인 두 가지 접근 방법을 통합하고 개선, 캐쉬 활용도가 높고 적용범위가 넓은 색인 구조인 부분키-CSB+ 트리를 제안한다. 본 논문의 공헌은 다음과 같이 요약된다.

- 키 압축과 포인터 제거를 함께 고려한 부분키-CSB+ 트리의 개념을 제안한다.
- 벌크로드, 삽입, 삭제 등의 완전한 알고리즘을 제시한다.
- 이론적인 분석을 통해 부분키-CSB+ 트리의 우수성을 증명한다.

† 이 논문은 2003년도 한국학술진흥재단의 지원에 의하여 연구되었음 (KRF-2003-003-D00347).

본 논문의 구성은 다음과 같다. 제 2절에서는 기존에 연구되었던 캐쉬 인식 트리들을 살펴보고 장단점을 논한다. 제 3절에서는 기존의 방법들을 통합, 개선한 새로운 캐쉬 인식 트리인 부분키-CSB+ 트리의 개념을 제안하고, 제 4절에서는 상세 알고리즘에 대해 설명한다. 제 5절에서는 부분키-CSB+ 트리를 포함하여 캐쉬 인식 트리들에 대한 이론적인 분석을 제시한다. 마지막으로 제 6절에서는 본 논문의 결론을 내린다.

2. 관련 연구

2.1 CSB+-트리

CSB+-트리[2]는 B+-트리의 효율적인 갱신 특성을 그대로 유지하면서 포인터 압축을 통해 검색 성능을 향상시킨 캐쉬 인식 트리이다[2]. CSB+-트리의 비단말 노드 *N*은 그 자식 노드들 모두에 대한 포인터들이 아닌 첫 번째 자식 노드에 대한 포인터 *p*만을 가진다. 단, 그 자식 노드들은 주기억 장치 내에 연속되게 배치되고, 이를 노드 그룹 이라고 부른다[2].

노드 *N*의 두번째 자식 노드에 대한 포인터는 *p*와 노드 크기의 합으로써 구할 수 있고, 다른 자식 노드들에 대한 포인터도 이와 같은 방법으로 얻을 수 있다. 노드에 저장되는 포인터의 개수를 줄여 CSB+-트리는 비단말 노드에 더 많은 키를 저장하도록 함으로써 노드의 블록킹 인수를 높여 검색 시에 발생하는 캐쉬 미스를 줄였다.

그러나 이러한 포인터 압축에 의한 캐쉬 미스 감소는 노드에 저장되는 키의 길이가 고정이고 짧은 경우에만 효과를 가진다. 즉, 가변길이 키의 경우 키 값을 주기억 장치의 다른 공간에 저장하고 이에 대한 정수형 색인을 키로 노드에 저장하도록 하고 있다[2]. 그러나 이는 키 액세스마다 추가적인 캐쉬 미스를 발생시키므로 다른 방법에 대한 고려가 필요하다.

2.2 부분키 B-트리

최근에 소개된 부분키 B-트리[3]는 가변길이키를 노드에 직접 저장하지 않고 연속된 키 사이에서 서로 다른 부분의 위치와 서로 다른 부분 중 일부만을 고정길이로 저장한다. 따라서 길이가 큰 가변길이 키 대신에 크기가 작은 고정길이 키를 저장함으로써 캐쉬 미스의 발생 횟수를 줄일 수 있다.

키 K_i에 대한 부분키는 (ref, d(K_i, $K_b(K_i)$), partkey)로 구성된다. 여기서 ref는 키를 포함하는 데이터 레코드에 대한 포인터를, d(K_i, $K_b(K_i)$)는 키 K_i와 그 베이스 키 $K_b(K_i)$의 차이 비트 오프셋을, partkey는 키의 비트들 중에서 d(K_i, $K_b(K_i)$) 위치 다음의 *l*개의 비트 (*l*>0) 들을 표현한다. 이때, K_i에 대한 베이스 키 $K_b(K_i)$는 K_i가 속한 노드와 같은 레벨 혹은 상위 레벨에 속한 노드의 키이면서 K_i를 제외하고 K_i보다 작거나 같은 키들 중에서 가장 큰 키로 정의된다[3]. 차이 비트 오프셋이란 K_i를 구성하는 비트들과 $K_b(K_i)$의 비트들을 자릿수가 큰 것에서부터 작은 것 순으로 비교할 때, 두 비트의 값이 다른 첫 번째 위치를 나타낸다. 따라서 ref, d(K_i, $K_b(K_i)$), partkey 모두 고정길이 이므로 부분키 역시 고정길이가

된다.

그림 1은 부분키의 예를 보여준다. $K_b(K_i)$를 베이스 키로 하는 K_i의 부분키는 K_i에 대한 포인터가 ref로 저장되고, $K_b(K_i)$와 K_i의 비트들을 자릿수가 큰 비트에서 작은 비트순으로 비교하여, 처음으로 다른 위치인 7이 $d(K_i, K_b(K_i))$로 저장되고, $d(K_i, K_b(K_i))$위치 이후의 4 비트를 partkey로 저장하고 있다.

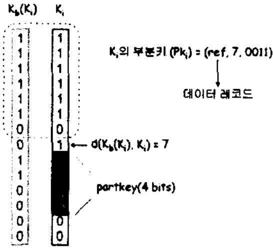

그림 1. 부분 키의 예.

부분키 B-트리는 부분키를 이용한 새로운 검색 알고리즘과 키압축을 통하여 캐쉬 미스 횟수를 줄였다. 그러나 키와 함께 엔트리를 구성하는 포인터의 압축은 고려하지 않은 문제점을 가진다. 키의 크기가 작은 경우에는 노드내에서 포인터가 차지하는 공간이 상대적으로 더 크게 될 수 있고, 더욱이 64비트 머신은 포인터의 크기가 8바이트나 되므로 포인터 제거도 함께 고려하는 것이 필요하다.

3. 개념

본 논문에서 제안하는 부분키 CSB^+-트리는 키 압축 방법과 포인터 압축 방법을 동시에 적용한 CSB^+-트리의 변형으로서, 정의 1에서 정형적으로 정의한다.

정의 1. 차수(order) k인 부분키-CSB^+ 트리는 다음과 같이 정의된다.

1. 모든 비단말 노드는 다음과 같이 구성된다.
   a. 오름차순으로 정렬된 n개의 키 K_i의 부분키 PK_i $(\frac{k}{2} \le n < 2)$
   b. 첫 번째 자식 노드에 대한 포인터
   c. n개의 부분키 PK_i $(1 \le i \le n)$가 저장된 비단말 노드의 j번째 자식 노드에 속한 모든 키 X의 범위는 다음과 같다.
      $K_{j-1} \le X < K_j$ $(1<j \le n)$; $X<K_j$ $(j=1)$; $K_{j-1}<X$ $(j=n+1)$
2. 모든 단말 노드는 다음과 같이 구성된다.[1]
   a. 오름차순으로 정렬된 n개의 키 K_i의 부분키 PK_i $(\frac{k}{2} \le n < 2)$
3. 모든 단말 노드는 같은 레벨에 존재한다.
4. n개의 부분키가 저장된 비단말 노드는 n+1개의 자식노드를 가지는데, 이 자식 노드들은 주기억 장치 내에 차례대로 연속하여 저장되며 하나의 노드 그룹을 형성한다.
5. 노드 N에 저장된 각각의 키의 베이스 키의 위치는 다음과 같다.
   a. N에 속한 키들 중에서 가장 왼쪽 키를 제외한 모든 키의 베이스 키는 각각 그 왼쪽 이웃 키가 된다.
   b. N에 속한 가장 왼쪽 키(leftmost key)의 베이스 키는 부모 노드에 속한, N의 가장 왼쪽 키(leftmost key)보다 작거나 같은 키들 중에서 가장 큰 키가 된다.
   c. 노드 N이 그 부모 노드의 첫번째 자식 노드인 경우, N의 가장 왼쪽키의 베이스 키는 그 부모 노드의 가장 왼쪽 키의 베이스 키와 동일하다.
   d. 트리의 각 레벨에서 가장 왼쪽 노드(leftmost node)의 가장 왼쪽 키의 베이스 키는 키의 정의역의 최소값으로 하며 이를 가상 베이스 키(virtual base key)라 부른다.

정의 1에서 부분키는 2.2장에서 살펴본 부분키 B-트리의 부분키와 동일한 구조를 가진다. 그러나 트리 각 레벨의 가장 왼쪽 노드의 가장 왼쪽 키는 트리내에 그 베이스 키가 존재하지 않게 되는 문제점을 가지며 논문[3]에서는 이에 대한 언급이 없다. 이를 해결하기 위해 이러한 키들은 트리마다 유일하게(unique) 정해진 공통의 베이스 키를 가지도록 하였고, 이 키를 가상 베이스 키라고 부른다. 가상 베이스 키는 저장할 필요 없이 키의 정의역의 최소값으로 해석한다.

그림 2는 부분키-CSB^+ 트리의 예를 나타내고 있다. 그림에서 실선 사각형은 노드를, 점선 사각형은 노드 그룹을 나타내고 있다. 그리고 실선 화살표는 주어진 키에 대한 베이스 키를 표현한다. 정의에서 설명하였듯이 각 레벨의 맨 왼쪽 노드의 맨 왼쪽 부분키의 베이스 키는 가상 베이스 키가 된다. 그리고 CSB^+-트리와 같이 임의 노드의 자식 노드들은 하나의 노드 그룹으로 연속적으로 주기억 장치 내에 할당되어 있다.

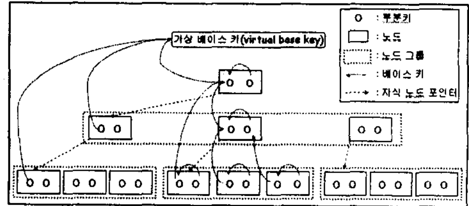

그림 2. 부분키-CSB^+ 트리의 예.

4. 알고리즘

4.1 벌크로드

벌크로드 알고리즘은 B^+-트리에서 소개된 벌크로드 알고리즘과 유사하나 부분키와 포인터 압축이 함께 고려 되어야 한다. 단말 노드들에 키들의 부분키들을 정렬된 순서대로 모두 삽입한 후, 그 상위 레벨의 노드들에는 하위 노드들의 맨 왼쪽 키들을 다시 삽입하는 방식으로, 단말 노드에서 최상위 루트 노드까지 트리의 각 레벨에 속한 노드들을 차례대로 구성하게 된다. 이때 노드에 저장되는 부분키를 생성하기 위해서는 저장하려는 키의 베이스 키를 알아야 하는데, 각 노드의 맨 왼쪽의 키는 상위노드에 그 베이스 키가 존재한다. 따라서 이에 대한 고려가 필요하다.

먼저, 정렬된 데이터를 단말 노드에 차례대로 삽입한다. 이때 노드에 저장되는 것은 부분키이며, 먼저 삽입된 키가 그 다음에 삽입될 키의 베이스 키가 된다. 그리고 각 단말 노드의 가장 왼쪽 키는 상위 비단말 노드에 다시 삽입되어, 그 단말 노드를 가리키는 포인터와 쌍으로 저장될 키이다.

단말 노드들의 구성이 끝나면, 포인터 pchild가 단말 노드의 첫 번째 노드 그룹의 가장 왼쪽 노드를 가리키도록 하고, 단말 노드들 각각의 가장 왼쪽 키들의 정렬된 목록을 A라 할 때, 한 레벨에 속한 노드의 수가 하나가(그것이 트리의 최상위 루트 노드가 된다) 될 때까지 다음과 같은 과정을 반복한다.

A에 속한 키들을 현재 레벨의 각 노드에 삽입한다. 이때 노드에 저장되는 것은 각 키들의 부분키이며, 그 베이스 키는 노드에 먼저 삽입된 키가 된다. 단, 빈 노드에 가장 먼저 삽입 시도 되는 키, 즉 첫 번째 자식 노드를 가리키는 키는 노드에 저장되는 포인터의 수가 키의 수보다 하나 더 많은 B^+-트리의 특성에 따라 현재 노드에 저장되지 않고 보다 상위 레벨의 노드에 저장되게 된다. 따라서 그 키는 그 다음에 노드에 삽입하는 키의 베이스 키로만 이용되고, 노드에는 저장되지 않는다. 각 노드에 첫 번째 키가 삽입될 때, pchild를 첫 번째 자식 노드로 현재 노드에 저장한 후, pchild를 노드 그룹의 크기만큼 증가시켜 다음 노드 그룹의 첫 번째 노드를 가리키도록 하고, 계속해서 노드에 정해진 수만큼 키를 삽입한다. A에 속한 키들을 모두 현재 레벨의 노드에 삽입하였으면, 앞의 과정에서 노드에 저장되지 않은 키들로 A를 재구성하고, pchild가 현재 레벨의 첫 번째 노드 그룹의 첫 번째 노드를 가리키도록 한다.

4.2 삽입

설명에 앞서 몇 가지 표기법을 표 1과 같이 정의한다.

임의의 노드 N의 베이스 키를 포함한 키 목록 $\{K_{upper}, K_{leftmost} \sim K_{rightmost}\}$에 속한 키들은 모두 연속적으로 존재하며, K_{upper}를 제외한 각 키의 베이스 키는 그 왼쪽에 위치한 키가 된다. 따라서 노드 N에 속한 키의 베이스 키의 변화는 이 키 목록의 변화에만 의존한다.

키의 삽입에 의한 노드나 노드 그룹의 분리(split)시에 부분키의 갱신은 보조정리 1.1~1.3을 따른다.

보조정리 1.1 부분키-CSB^+ 트리에서 키 삽입시에 노드를 분리하지 않는 경우, 최대 두 개의 키의 베이스 키가 변경되고, 최대 한번의 캐쉬 미스가 발생한다.

증명: 노드 N의 키 목록이 $\{K_{upper}, K_{leftmost} \sim K_{prev}, K_{next} \sim K_{rightmost}\}$와 같을 때 K_{insert}가 K_{prev}와 K_{next}사이에 삽입된다고 하면 변화된 키 목록은 $\{K_{upper}, K_{leftmost} \sim K_{prev}, K_{insert}, K_{next} \sim K_{rightmost}\}$ 이다. 따라서 K_{insert}와

[1] 부분키의 ref로 데이터 레코드에 대한 포인터가 저장되므로 단말 노드에서 다시 데이터 레코드에 대한 포인터를 저장할 필요가 없다.

K_{next}의 베이스 키가 변경된다. 이때 N이 단말 노드인 경우 K_{insert}는 검색의 결과로써 캐쉬에 존재하고 그 차이 비트 오프셋도 알고 있다. 따라서 K_{insert}의 부분키 생성은 어떠한 키의 참조도 발생시키지 않는다. N이 비단말 노드인 경우 K_{insert}의 차이 비트 오프셋은 K_{prev}와의 비교를 통해서만 알 수 있다. 따라서 K_{prev}의 키를 주기억 장치로부터 인출하여야 한다. K_{next}의 차이 비트 오프셋은 새로운 베이스 키인 K_{insert}와의 비교를 통해서만 알 수 있다. 따라서 K_{next}의 키를 주기억 장치로부터 인출하여야 한다.

표 1. 표기법.

표기	의미
K	부분키
$K_{leftmost}$	노드에 속한 가장 왼쪽 부분키
$K_{rightmost}$	노드에 속한 가장 오른쪽 부분키
K_{a-1}	노드에 속한 키 K_a의 왼쪽 이웃 부분키
K_{a+1}	노드에 속한 키 K_a의 오른쪽 이웃 부분키
$\{K_1, \cdots, K_n\}$	임의의 정렬된 부분키들의 목록
$\{K_i \sim K_n\}$	노드의 키 K_i과 K_n사이의 부분키의 정렬된 목록
K(N)	노드 N에 속한 부분키들의 정렬된 목록
$K(N)_{upper}$	노드 N에 속한 가장 왼쪽 키의 베이스 키

보조정리 1.2 부분키-CSB^+ 트리에서 키 삽입시에 노드는 분리(split)되나 노드 그룹이 분리되지 않는 경우, 노드 분리에 의해 최대 한 개의 키의 베이스 키가 변경되며, 이때 최대 한번의 캐쉬 미스가 발생한다.
증명 생략.
노드 분리 이후 키의 삽입은 보조정리1.1을 따른다.

보조정리 1.3 부분키-CSB^+ 트리에서 키 삽입시에 노드와 노드 그룹이 다 분리(split)되는 경우, 노드 그룹 분리시에는 노드의 이동만 있고 키의 이동이 없으므로, 어떠한 키의 베이스 키도 변경되지 않는다.
증명 생략.
삽입 알고리즘은 보조정리 1.1 ~ 1.3을 따르며, 지면상 생략한다.

4.3 삭제

키의 삭제에 의한 노드나 노드 그룹의 병합(merge)및 노드나 키의 재분배(redistribute)에 의한 부분키 갱신은 보조정리 2.1~2.3을 따른다.

보조정리 2.1 부분키-CSB^+ 트리에서 키 삭제시에 노드의 병합 및 키의 재분배가 없는 경우, 최대 한 개의 키의 베이스 키가 변경되고, 최대 한번의 캐쉬 미스가 발생한다.
증명 생략.

보조정리 2.2 부분키-CSB^+ 트리에서 키 삭제시에 노드의 병합 혹은 키의 재분배가 있는 경우, 두 노드의 병합에 의해서 최대 2개의 키의 베이스 키가 변경되고 최대 2번의 캐쉬 미스가 발생하며, 두 노드의 키의 재분배에 의해서는 최대 2개의 키의 베이스 키가 변경되고 최대 3번의 캐쉬 미스가 발생한다.
증명 생략.
노드의 병합 및 키의 재분배 이전에 노드에서 키의 삭제는 보조정리 2.1을 따른다.

보조정리 2.3 부분키-CSB^+ 트리에서 키 삭제시에 노드 그룹의 병합 혹은 노드들의 재분배가 있는 경우, 노드 그룹이 병합될 때나 노드가 재분배될 때 노드의 이동에 의한 베이스 키의 변경은 없다.
증명 생략.
노드 그룹의 병합 및 노드의 재분배 이전에 노드에서 키의 삭제 그리고 노드의 병합 및 키의 재분배는 각각 보조정리 2.1, 2.2를 따른다.

삭제 알고리즘은 보조정리 2.1 ~ 2.3을 따르며, 지면상 생략한다.

5. 성능분석

기존 캐쉬 인식 트리와 제안하는 부분키-CSB^+ 트리의 블록킹 인수는 표 2, 3과 같다.

표 2. 고정길이 키를 저장하는 경우에 트리들의 블록킹 인수.

	B_L	B_I
B^+-트리	N/(K+4)	N/(K+4)
CSB^+-트리(직접저장)	N/(K+4)	(N-4)/K
CSB^+-트리(간접저장)	N/(4+4)	(N-4)/4
부분키 B^+-트리	N/K_P	$N/(K_P+4)$
부분키-CSB^+ 트리	N/K_P	$(N-4)/K_P$

표 3. 가변길이 키를 저장하는 경우에 트리들의 블록킹 인수.

	B_L	B_I
B^+-트리	$(N-1)/(K_V+6)$	$(N-1)/(K_V+6)$
CSB^+-트리(직접저장)	$(N-1)/(K_V+6)$	$(N-5)/(K_V+2)$
CSB^+-트리(간접저장)	N/(4+4)	(N-4)/4
부분키 B^+-트리	N/K_P	$N/(K_P+4)$
부분키-CSB^+ 트리	N/K_P	$(N-4)/K_P$

이때, N은 노드 크기, B_L은 단말 노드의 블록킹 인수, B_I는 비단말 노드의 블록킹 인수, K는 고정길이 키의 길이, K_V 는 가변길이 키의 평균 길이, K_P는 부분키의 길이를 각각 나타낸다.

트리마다 노드의 블록킹 인수는 노드에 저장되는 키의 크기와 포인터의 수에 따라 변화하게 되고, 블록킹 인수에 따라 트리의 높이가 변하게 된다. 결과적으로 트리의 높이에 따라 트리를 순회할 때 발생하는 캐쉬 미스의 횟수가 변하게 된다. 트리의 높이 h는 다음과 같다.

$$h = \left\lceil \log_{(M+1)} \frac{n}{B_l} \right\rceil + 1$$

B^+-트리와 CSB^+-트리는 노드내에서 키의 검색에 이진 검색을 사용하므로 캐쉬 라인 크기가 #일 때, 노드당 발생하는 캐쉬 미스의 최대 횟수 Mn은 다음과 같다.

$$Mn = \lceil \log_2(N/\#) \rceil$$

부분키 B-트리와 부분키-CSB^+ 트리는 노드 내에서 키의 검색에 순차 검색을 사용하여 노드당 발생하는 캐쉬 미스의 최대 횟수 Mn은 다음과 같다.

$$Mn = \lceil N/\# \rceil$$

트리를 검색할 때 각 레벨당 하나의 노드가 액세스 되므로, 트리 검색시에 발생하는 전체 캐쉬 미스 횟수 M은 식 4와 같다.

$$M = M_n \times h$$

위의 식에 의해 키의 길이가 8바이트 이상인 경우에 부분키-CSB^+ 트리가 기존의 캐쉬 인식 트리를 보다 나은 성능을 보이고, 키의 길이가 16바이트 이상인 경우에는 그 성능이 크게 차이가 남을 확인할 수 있었다.

6. 결론

본 논문에서는 기존의 캐쉬 인식 트리에서 적용된 기술들을 살펴보고, 이를 바탕으로 보다 캐쉬 활용도를 높인 부분키-CSB^+ 트리를 제안하였다. 부분키-CSB^+ 트리는 CSB^+-트리의 포인터 제거 기법과 부분키B-트리의 키압축 기법을 함께 적용한 캐쉬 인식 트리로서 노드의 블록킹 인수를 더욱 향상시켰고, 가변길이 키를 지원함으로써 그 사용범위를 넓혔다. 또한 이론적인 분석에 의해 기존 캐쉬 인식 트리보다 더 나은 성능을 확인 할 수 있었다.

또한 부분키 B-트리에서 자세히 논의되지 않았던 벌크로드, 삽입, 삭제 알고리즘을 완전하게 제시하고, 캐쉬 미스의 발생 횟수를 최소로 줄이고, 각 알고리즘이 올바르게 수행됨을 확인하였다.

참고문헌

[1] J.Rao and K.A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB* 99.

[2] J. Rao and K.A. Ross. Making B^+-trees cache conscious in main memory. In *ACM SIGMOD* 2000

[3] P. Bohannon, P. Mcilroy, R. Rastogi. Main-Memory Index Structures with Fixed-Size Partial Keys. In *ACM SIGMOD 2001*.