CC-GiST: 임의의 캐시 인식 검색 트리를 효율적으로 구현하기 위한 일반화된 프레임워크

노 웅 기[†] · 김 원 식[††] · 한 욱 신[†††]

요 약

최근 메인 메모리 가격이 하락하고 용량이 크게 증가함에 따라 메인 메모리 데이터베이스에 기반한 응용이 급격히 증가하고 있다. 캐시 미스(cache miss)는 CPU에서 액세스하고자 하는 데이터가 캐시에 존재하지 않아 메모리로부터 읽어 들이는 과정이며, 메인 메모리 데이터베이스의 성능 감소의 중요한 원인이다. 메인 메모리 데이터베이스에서의 캐시 미스를 줄이고 캐시를 최대한 활용하기 위하여 여러 가지 캐시 인식 트리들(cache conscious trees)이 제안되었다. 이러한 캐시 인식 트리들은 각각 특성이 다르므로 하나의 응용에서 둘 이상의 캐시 인식 트리들이 동시에 관리될 수 있다. 또한, 만약 기존의 캐시 인식 트리가 응용에서의 요구를 만족시키지 못하면 새로운 캐시 인식 트리를 구현하여야 한다.

본 논문에서는 캐시 인식하는 일반화된 검색 트리(Cache-Conscious Generalized Search Tree, CC-GiST)를 제안한다. CC-GiST는 디스크 기반의 일반화된 검색 트리(Generalized Search Tree, GiST) [HNP95]를 캐시 인식하도록 확장한 것이며, 포인터 압축(pointer compression)과 키 압축(key compression) 기법을 비롯하여 임의의 캐시 인식 트리의 공통적인 기능 및 알고리즘들을 동시에 제공한다. CC-GiST를 기반으로 특정 캐시 인식 트리를 구현하려면 그 트리에 해당된 기능만을 구현하면 된다. 본 논문에서는 CC-GiST를 기반으로 기존의 대표적인 캐시 인식 트리인 CSB+-트리, pkB-트리, CR-트리를 구현하는 방법을 기술한다. CC-GiST를 이용함에 따라 메인 메모리 데이터베이스 응용에서 여러 개의 캐시 인식 트리를 관리하는 번거로움에서 벗어날 수 있고, 응용의 요구에 따른 새로운 캐시 인식 트리를 최소한의 노력으로 효율적으로 구현할 수 있다.

키워드 : 캐시 인식 트리, 일반화된 검색 트리, 포인터 압축, 키 압축

CC-GiST: A Generalized Framework for Efficiently Implementing Arbitrary Cache-Conscious Search Trees

Woong-Kee Loh[†] · Won-Sik Kim[††] · Wook-Shin Han[†††]

ABSTRACT

According to recent rapid price drop and capacity growth of main memory, the number of applications on main memory databases is dramatically increasing. Cache miss, which means a phenomenon that the data required by CPU is not resident in cache and is accessed from main memory, is one of the major causes of performance degradation of main memory databases. Several cache-conscious trees have been proposed for reducing cache miss and making the most use of cache in main memory databases. Since each cache-conscious tree has its own unique features, more than one cache-conscious tree can be used in a single application depending on the application's requirement. Moreover, if there is no existing cache-conscious tree that satisfies the application's requirement, we should implement a new cache-conscious tree only for the application's sake.

In this paper, we propose the cache-conscious generalized search tree (CC-GiST). The CC-GiST is an extension of the disk-based generalized search tree (GiST) [HNP95] to be cache-conscious, and provides the entire common features and algorithms in the existing cache-conscious trees including pointer compression and key compression techniques. For implementing a cache-conscious tree based on the CC-GiST proposed in this paper, one should implement only a few functions specific to the cache-conscious tree. We show how to implement the most representative cache-conscious trees such as the CSB+-tree, the pkB-tree, and the CR-tree based on the CC-GiST. The CC-GiST eliminates the troublesomeness caused by managing more than one cache-conscious tree in an application, and provides a framework for efficiently implementing arbitrary cache-conscious trees with new features.

Key Words : Cache Conscious Tree, Generalized Search Tree, Pointer Compression, Key Compression

1. 서 론

급속한 기술 발전에 따라 메인 메모리 가격이 크게 하락하고 그 용량이 급격하게 증가되고 있다. 이러한 흐름에 의해 최근 'Asilomar Report' 등에서는 10년 이내에 테라 바이트 크기의 데이터베이스가 메인 메모리에 상주할 수 있을 것으로 예상하고 있다 [Bern98, RR00]. 동시에, CPU의 처리 속도(증가 속도 연간 60%)와 메인 메모리의 액세스 속도(증가 속도 연간 10%)의 차이가 매년 커지고 있다 [CLH98, RR00]. 따라서, CPU와 메인 메모리 사이에 존재하는 캐시의 활용이 중요하게 인식되어 이에 대한 많은 연구들이 진

※ 이 논문은 2006년도 학술진행재단의 지원에 의하여 연구되었음. (KRF-2003-003-D00347)
† 정 회 원 : 미국 University of Minnesota 방문연구원
†† 준 회 원 : ㈜다음 연구원
††† 정 회 원 : 경북대학교 컴퓨터공학과 조교수(교신저자)
논문접수 : 2005년 8월 3일, 심사완료 : 2006년 11월 29일

행 중이다 [ADHW99, Bonc99, CKJA98, CLH98, CLH00, Lehm86, SKN94]. 이러한 연구의 하나로 메인 메모리 인덱스의 액세스 비용을 줄여 주는 캐시 인식 트리(cache conscious tree)들이 많이 연구되어 오고 있다 [BMR01, RR00, KCK01]. 캐시 인식 트리는 인덱스 내의 데이터를 액세스할 때 메인 메모리와 CPU 사이에 존재하는 캐시의 특성을 고려하여 캐시 미스(cache miss)가 가능한 한 적게 발생하도록 설계한 트리이다. 대표적인 캐시 인식 트리로는 CSB^+-트리 [RR00], pkB-트리 [BMR01], CR-트리 [KCK01] 등이 있다.

하지만, 기존의 캐시 인식 트리들은 각각 장단점들이 있으며 이들을 개별적으로 구현하는 것은 많은 비용이 들고 에러가 발생하기 쉬우므로, 새로운 접근 방식이 필요하다. 이와 유사한 배경에서 디스크 기반의 인덱스를 쉽게 개발하기 위한 프레임워크인 디스크 기반 일반화된 검색 트리(Generalized Search Tree, GiST)가 제안되었다 [HNP95]. 즉, GiST는 디스크 기반의 모든 균형 검색 트리(balanced search tree)들의 공통적인 함수들을 제공하므로, GiST를 이용하여 디스크 기반의 인덱스를 구현할 때 공통적인 함수들을 제외하고 각 인덱스에 한정되는 함수만을 구현하면 된다. 하지만, 이러한 GiST는 디스크 기반으로 구현되었기 때문에 메인 메모리 인덱스로 그대로 사용하면 캐시를 제대로 활용할 수 없는 문제점을 가진다.

본 논문에서는 디스크 기반의 GiST를 캐시 인식하도록 확장한 일반화된 검색 트리(Cache Conscious-Generalized Search Tree, CC-GiST)를 제안한다. 본 논문의 공헌은 다음과 같다:

(1) 기존의 캐시 인식 트리들의 기법을 분석함으로써, 캐시 인식 트리에 적용할 수 있는 일반적인 기법들을 도출한다. 구체적으로, 포인터 압축(pointer compression)과 키 압축(key compression) 개념을 일반화한다.

(2) 일반화된 포인터 압축과 키 압축 기법을 지원하도록 디스크 기반의 GiST를 확장하여 CC-GiST를 정형적으로 제시한다. 포인터 압축을 위해 세그먼트(segment) 개념을, 키 압축을 위해 베이스 키(base key) 개념을 CC-GiST에 도입한다.

(3) 디스크 기반의 GiST에 정의되어 있는 메소드들을 캐시 인식하도록 완전하게 수정한다. 그러한 메소드들 중에는 임의의 객체를 검색(search), 삽입(insert), 삭제(delete)를 위한 일반화된 메소드들이 포함된다.

(4) 제안된 CC-GiST를 이용하여 기존의 대표적인 캐시 인식 트리인 CSB^+-트리, pkB-트리, CR-트리를 구현하는 방법을 기술한다.

본 논문의 구성은 다음과 같다. 제 2 장에서는 기존의 캐시 인식 트리와 디스크 기반의 GiST에 대해 간략히 설명한다. 제 3 장에서는 일반화된 포인터 압축과 키 압축 기법을 적용하여 디스크 기반의 GiST를 캐시 인식하도록 확장한 CC-GiST를 제안한다. 제 4 장에서는 CC-GiST를 이용하여 CSB^+-트리, pkB-트리, CR-트리의 구현 방법을 설명한다. 제 5 장에서는 CC-GiST와 기존의 캐시 인식 트리의 성능을 분석적으로 비교, 평가한다. 마지막으로, 제 6 장에서는 결론을 내린다.

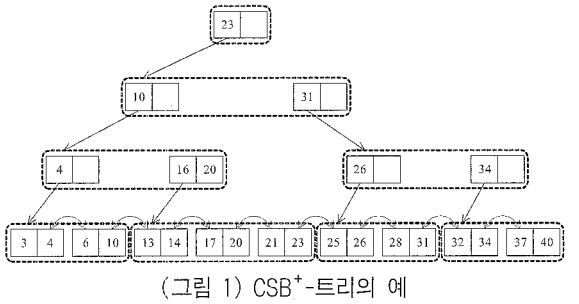
2. 관련 연구

본 장에서는 먼저 기존의 캐시 인식 트리들의 기법을 분석하여 포인터 압축과 키 압축 기법을 일반화한다. 그리고, 디스크 기반의 GiST에 대해 간략하게 설명한다.

2.1 캐시 인식 트리

캐시 인식 트리는 캐시 사용 효율을 높이기 위하여 인덱스가 사용하는 메모리의 양을 줄이는 방법을 사용한다. 본 논문에서는 이러한 방법을 통칭하여 압축(compression)이라고 부른다. 일반적으로, 같은 데이터를 표현하기 위하여 적은 양의 메모리를 사용하게 되면, 하나의 캐시라인(cache-line) 내에 관련 있는 데이터가 함께 포함될 가능성이 커지므로 캐시 활용도를 높이고 캐시 미스를 줄일 수 있다[1]. 캐시 인식 트리는 압축 대상에 따라 (1) 포인터 압축에 기반한 캐시 인식 트리와 (2) 키 압축에 기반한 캐시 인식 트리로 구분할 수 있다. 포인터 압축에 기반한 캐시 인식 트리는 비단말 노드 내에서 자식 노드를 가리키는 포인터들 중 일부를 제거하여 블로킹 인수(blocking factor)를 증가시킴으로써 캐시 활용을 증가시키는 인덱스로서, 대표적인 예로는 CSB^+-트리 [RR00]와 세그먼트 CSB^+-트리 [RR00] 등이 있다. 키 압축에 기반한 캐시 인식 트리는 노드 내의 키의 길이를 짧게 하여 블로킹 인수를 증가시킴으로써 캐시 활용을 증가시키는 인덱스로서, 대표적인 예로 pkB-트리 [BMR01]와 CR-트리 [KCK01] 등이 있다.

(그림 1)은 CSB^+-트리의 예를 보이고 있다. 비단말 노드의 자식 노드들을 점선 사각형으로 표시된 연속된 공간, 즉 노드 그룹(node group)에 저장하고, 부모 노드는 노드 그룹에 저장된 첫번째 자식 노드를 가리키는 포인터(실선 화살표로 표시)만을 저장한다. 노드 그룹 내의 모든 자식 노드들은 물리적으로 연속된 공간에 저장되므로, 하나의 포인터만으로도 세그먼트 내의 모든 노드에 대한 위치를 구할 수 있다. 이렇게 함으로써 각 비단말 노드에서 필요로 하는 포인터의 개수를 줄이고 블로킹 인수를 증가시켜 캐시 활용도를 향상시킨다.

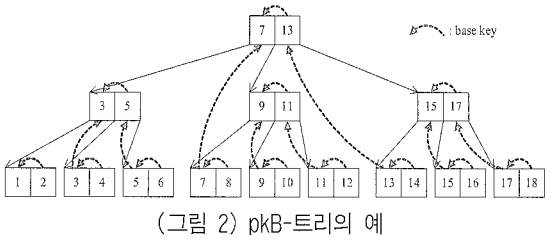


(그림 1) CSB^+-트리의 예

1) 디스크 상의 데이터가 섹터(sector) 단위로 메모리에 올라가는 것과 마찬가지로, 메모리 상의 데이터는 캐시라인 단위로 캐시에 올라간다. 캐시라인의 크기는 대체로 64~128 바이트이며, 캐시 인식 트리의 한 노드의 크기는 캐시라인 크기의 정수 배이다.

CSB^+-트리는 갱신 비용이 크다는 문제점을 가진다. 즉, CSB^+-트리의 노드에서 분할(split) 또는 병합(merge)이 일어났을 때, 분할 또는 병합 후에도 노드 그룹 내의 모든 노드들이 물리적으로 연속되게 저장하기 위하여 이전 노드 그룹에 있는 모든 노드들을 복사해야 하기 때문에 비용이 크다. 세그먼트 CSB^+-트리는 이러한 문제를 해결하기 위한 변형(variant)이며, 하나의 노드 그룹을 여러 개의 세그먼트로 분할함으로써 복사 비용을 줄인다. 자식 노드들은 세그먼트 내에서만 연속적으로 저장되며, 부모 노드는 각 세그먼트 내의 첫번째 자식 노드를 가리키는 포인터만을 저장한다. (그림 1)의 CSB^+-트리는 모든 노드 그룹이 한 개의 세그먼트로 구성된 세그먼트 CSB^+-트리로 생각할 수 있다.

키 압축에 기반한 pkB-트리는 연속된 키 사이에서 차이가 나는 부분 중 고정 길이의 일부만을 저장함으로써 키를 압축한다[BMR01]. 따라서, 길이가 큰 가변 길이 키 대신에 길이가 작은 고정 길이 키를 저장함으로써 캐시 활용도를 높일 수 있다. (그림 2)는 pkB-트리의 예를 나타내고 있다. 실선 화살표는 자식 노드를 가리키는 포인터이고 점선 화살표는 베이스 키를 나타낸다. 베이스 키와 현재 키 사이에서 차이가 나는 부분만을 추출함으로써 키를 압축하며, 압축과정반대의 과정을 통하여 압축된 키로부터 원래의 키를 복원한다.



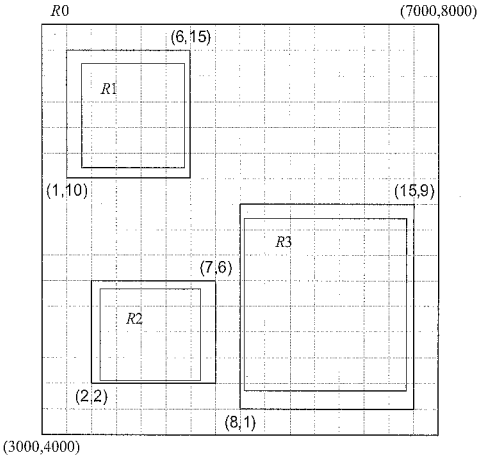
(그림 2) pkB-트리의 예

키 압축에 기반한 또 다른 트리인 CR-트리는 키의 MBR 좌표 값으로 절대 좌표 값을 저장하는 대신, 부모 노드 키의 MBR 좌표에 대한 상대적 좌표화 및 양자화를 거친 MBR, 즉 QRMBR(Quantized Relative Representation of MBR)을 저장함으로써 키를 압축한다 [KCK01]. MBR의 절대 좌표는 길이가 긴 반면 QRMBR의 상대 좌표는 길이가 짧으므로, 하나의 노드에 더 많은 엔트리를 저장함으로써 캐시 활용도를 높인다.

(그림 3)은 CR-트리의 QRMBR의 예를 보인 것이다. 바깥쪽 사각형 $R0$는 부모 노드 키의 MBR을 나타내고, 사각형 $R1$, $R2$, $R3$는 자식 노드의 MBR를 나타낸다. $R1$, $R2$, $R3$를 둘러싼 굵은 사각형은 각각의 QRMBR을 나타내며, $R0$를 기준으로 상대 좌표로 표현된다. (그림 3)의 QRMBR의 상대 좌표는 16 등분하여 양자화되므로 4 비트를 이용하여 표현할 수 있는 반면, MBR의 절대 좌표는 4 ~ 8 바이트를 이용하여 표현하여야 한다. 데이터의 차원이 높아짐에 따라 QRMBR 좌표를 표현하는 저장 공간을 더욱 줄일 수 있다.

2.2 디스크 기반의 GiST

GiST [HNP95]는 디스크 기반의 모든 균형 검색 트리를 구성할 수 있는 프레임워크이다. 균형 검색 트리의 공통적인 기능인 검색, 삽입, 삭제 등은 트리 메소드로서 GiST에서 제공한다. 특정 균형 검색 트리마다 다른 특성들만을 키 메소드로서 개별적으로 구현해 준다. 따라서, 적은 비용으로 디스크 기반의 모든 균형 검색 트리를 구현할 수 있다. 그러나, GiST는 디스크 기반으로 구현되었기 때문에 메인 메모리 인덱스로 그대로 사용하면 캐시를 제대로 활용할 수 없는 문제점을 가진다.



(그림 3) CR-트리의 QRMBR 예

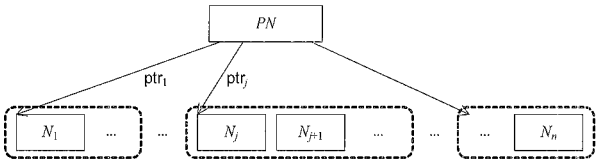
3. 캐시 인식하는 CC-GiST

본 장에서는 먼저 CC-GiST에서 도입한 개념과 CC-GiST의 구조 및 특성을 설명한다. 그리고, CC-GiST의 포인터 메소드(pointer method), 키 메소드(key method), 트리 메소드(tree method)에 대해 차례대로 설명한다. CC-GiST의 세 가지 메소드들 중에서 캐시 인식 트리마다 사용자가 별도로 구현해 주어야 하는 것은 키 메소드 뿐이며, 포인터 메소드와 트리 메소드는 모든 캐시 인식 트리에 공통된 기능을 위하여 CC-GiST가 제공하는 것들이다. CC-GiST가 인덱싱하는 데이터는 객체 지향 DBMS(Object-Oriented DBMS) 또는 객체 관계형 DBMS(Object-Relational DBMS)의 객체(object)이며, 디스크 기반의 GiST와 같이 임의의 데이터 타입의 객체들을 인덱싱할 수 있다.

3.1 CC-GiST 개념

본 논문에서는 기존의 캐시 인식 트리들을 분석하여 다음과 같이 포인터 압축과 키 압축 개념을 정형적으로 정의한다.

[정의 1] 캐시 인식 트리에서 포인터 압축이란 비단말 노드 내의 n 개의 자식 포인터 ptr_i $(1 \le i \le n)$ 중에서 일부를 제거하고, 제거되지 않은 포인터 ptr_j $(1 \le j \le n)$가 가리키는 자식 노드와 ptr_j와 인접한 제거된 포인터들 ptr_{j+1},



(그림 4) 포인터 압축 개념

〈표 1〉 표기법 정리

표기법	설명
N	하나의 노드
L	단말 노드
R	루트 노드
p	키를 나타내는 프레디키트(predicate)
q	질의 프레디키트(query predicate)
P	키 프레디키트 p의 집합
l	트리 내의 노드의 레벨 (≥ 0)
NodePointer	노드 포인터
SegmentPointer	세그먼트 포인터
Key	키
BaseKey	베이스 키

ptr_{j+2}, …가 가리키는 자식 노드들을 함께 하나의 세그먼트에 저장함으로써 자식 포인터의 수를 줄이는 방법이다. □

(그림 4)는 포인터 압축 개념을 도식적으로 보인 것이다. 그림에서 자식 노드 N_{j+1}, N_{j+2}, ...를 가리키는 포인터 ptr_{j+1}, ptr_{j+2}, …를 제거하고, 대신에 부모 노드 PN에서는 자식 노드 N_j를 가리키는 포인터 ptr_j만을 유지한다. 이는 자식 노드 N_j와 함께 N_{j+1}, N_{j+2}, …를 아래 그림에서 점선 사각형으로 표시된 하나의 세그먼트에 물리적으로 연속되게 저장함으로써 가능하다.

[정의 2] 캐시 인식 트리에서 키 압축이란 $Len(Compress(\text{BaseKey}_i, \text{Key}_i)) \leq Len(\text{Key}_i)$ 관계를 만족시키도록 키의 크기를 줄이는 방법이다. 여기서 Key_i는 임의의 노드의 i번째 키이고 BaseKey_i는 Key_i의 베이스 키를 나타낸다. *Compress* 함수는 BaseKey_i를 이용하여 Key_i를 압축한 키를 반환하는 함수이며, *Len* 함수는 키의 길이를 구하는 함수이다. □

본 논문에서 제안하는 CC-GiST는 디스크 기반의 GiST에 포인터 압축과 키 압축 개념을 적용하여 기존의 모든 균형 캐시 인식 트리들을 쉽게 구현할 수 있도록 한 프레임워크이다. 다음 절부터 CC-GiST의 구조와 특성, 포인터 메소드, 키 메소드, 트리 메소드를 차례로 설명한다. <표 1>은 본 논문에서 사용하는 표기법을 정리한 것이다.

3.2 구조 및 특성

CC-GiST는 kM과 M 사이의 차수(degree)를 가지는 균형 트리이다. 여기에서, k는 최소 채움 인수(minimum fill factor), M은 블로킹 인수(blocking factor)를 나타내며, k는 $2/M$와 1/2 사이의 값을 가진다. 루트 노드를 제외한 CC-GiST의 모든 노드는 2와 M 사이의 차수를 가진다.

CC-GiST에서는 포인터 압축 개념을 적용하기 위하여 비단말, 단말 노드들을 세그먼트에 저장하고 비단말 노드는 자식 노드가 저장된 세그먼트들을 가리키는 포인터들만을 가진다. 세그먼트는 물리적으로 연속되게 저장되어 있는 하나 이상의 노드들을 포함한다. 하나의 비단말 노드에서 가리키는 세그먼트의 수를 그 노드의 세그먼트 차수라 하고, 전체 트리 내의 노드의 최대 세그먼트 차수를 그 트리의 세그먼트 차수라 한다. CC-GiST는 고정 길이 세그먼트와 가변 길이 세그먼트를 지원한다. 고정 길이 세그먼트는 세그먼트에 저장된 노드의 개수가 일정한 반면에 가변 길이 세그먼트는 저장된 노드 개수가 가변적이다. IsFixedSizeSegment 플래그는 CC-GiST가 고정 길이 또는 가변 길이 세그먼트의 사용 여부를 나타낸다. CC-GiST는 세그먼트 크기와 관련하여 하나의 세그먼트 내의 자식 노드의 최대 개수 MaxNumOfNodesInSegment를 유지한다.

CC-GiST의 비단말 노드에서는 자식 노드를 포함하는 세그먼트들에 대한 포인터의 리스트 ListOfSegmentPointers를 관리한다. 또한, 비단말 노드에서는 각 세그먼트 내에 포함된 자식 노드들의 개수를 나타내는 ListOfNumOfChildNodesInSegment를 관리한다. 비단말 노드에서 자식 노드를 가리키는 포인터 NodePointer = (SegmentPointer, NodeOrderInSegment)는 그 자식 노드가 포함되어 있는 세그먼트에 대한 포인터와 그 세그먼트 내에서의 자식 노드의 순번으로 구성된다 (NodeOrderInSegment $\geq$ 0).

CC-GiST에서는 키 압축 개념을 적용하기 위해 StoreBaseKeyInNode라는 플래그를 관리하고, 이 플래그에 TRUE 값을 설정하면 CR-트리처럼 CC-GiST 내의 모든 노드 내에 베이스 키를 저장할 수 있는 데이터 구조를 포함한다. 그리고, CC-GiST는 키를 압축, 복원할 때 사용하는 조상 노드의 베이스 키를 저장하기 위해 AncestorKeyStack이라는 객체를 제공한다. AncestorKeyStack은 검색 경로 상의 조상 노드들의 키들을 저장한다. CC-GiST 노드의 키는 GiST [HNP95]에서와 같이 프레디키트(predicate)로 표현된다. 키 프레디키트는 하나 이상의 자유 변수(free variable)을 포함하며, 그 키와 연관된 포인터를 따라 도달 가능한 모든 객체의 애트리뷰트 값을 대입했을 때 항상 TRUE를 반환한다. CC-GiST의 노드는 키 프레디키트의 리스트를 관리한다.

CC-GiST의 특성을 정리하면 다음과 같다:

(1) 루트 노드를 제외한 모든 노드의 차수는 kM과 M 사이이다.
(2) 단말 노드 내의 한 엔트리의 키 프레디키트 p의 자유 변수에 그 엔트리 포인터가 가리키는 객체의 애트리뷰트 값을 대입할 때 항상 TRUE이다.
(3) 비단말 노드 내의 한 엔트리의 키 프레디키트 p의 자유 변수에 그 엔트리 포인터가 가리키는 서브트리 내의 모든 객체의 애트리뷰트 값을 대입할 때 항상 TRUE이다.
(4) 루트 노드가 단말 노드가 아니면 적어도 2 개의 자식 노드를 가진다.
(5) 모든 단말 노드는 같은 레벨에 존재한다.

3.3 포인터 메소드

포인터 메소드는 포인트 압축 기법을 구현하기 위한 함수들의 집합이다. 각 캐시 인식 트리마다 포인트 압축 기법이 다르지만, CC-GiST를 이용하여 각 캐시 인식 트리를 구현하기 위하여 해주어야 할 일은 IsFixedSizeSegment 플래그와 MaxNumOfNodesInSegment 변수 값을 적절하게 설정해 주는 것뿐이다. CC-GiST에서는 이 값들에 따라 자동적으로 포인터 압축을 수행한다. 포인터 메소드는 다음과 같은 두 가지가 있다:

- **PointerCompress**(NodePointer): 입력 파라메터로 노드를 가리키는 NodePointer = (SegmentPointer, NodeOrderInSegment)를 받는다. 만약 입력된 노드가 SegmentPointer가 가리키는 세그먼트 내의 첫번째 노드라면 SegmentPointer를, 그렇지 않으면 NULL을 반환한다.
- **PointerDecompress**(ChildNodeOrder, ListOfSegmentPointers, ListOfNumOfChildNodesInSegment): 입력 파라메터로 부모 노드 내에서의 자식 노드의 순번을 나타내는 ChildNodeOrder, 자식 노드들이 저장된 세그먼트를 가리키는 포인터의 리스트 ListOfSegmentPointers, 각 세그먼트 내에 저장된 자식 노드들의 개수의 리스트 ListOfNumOfChildNodesInSegment를 받는다. 만약 포인터 압축을 수행한다면 다음과 같이 자식 노드를 가리키는 포인터를 복원하여 반환한다. ChildNodeOrder와 ListOfNumOfChildNodesInSegment를 이용하여 ChildNodeOrder 번째 자식 노드가 저장된 세그먼트의 순번과 그 세그먼트 내에서 자식 노드의 순번 NodeOrderInSegment을 구한다. 세그먼트의 순번과 ListOfSegmentPointers를 통해서 자식 노드가 저장된 세그먼트 포인터 SegmentPointer를 구한다. 최종적으로, NodePointer = (SegmentPointer, NodeOrderInSegment)을 반환한다. 만약 포인터 압축을 수행하지 않는다면 NodePointer = (ListOfSegmentPointers 내의 ChildNodeOrder 번째 세그먼트 포인터, 0)을 반환한다.

3.4 키 메소드

키 메소드는 키 압축 기법을 포함한 키 처리 알고리즘을 구현하기 위한 함수들의 집합이다. 각 캐시 인식 트리마다 키를 처리하는 알고리즘이 다르므로, 구현하고자 하는 캐시 인식 트리의 특성을 적절하게 반영하도록 키 메소드를 구현해 주어야 한다. 아래의 키 메소드 중에서 *Compress*, *Decompress*, *ConstructBaseKey*, *ConvertToKey*, *ConvertToPredicate* 함수는 키 압축 기법을 구현하기 위한 함수이다.

- **GetConsistentEntries**(N, q): 입력 파라메터로 노드 N과 질의 프레디키트 q를 받는다. 이 함수는 노드 N에서 질의 q를 만족시키는 키를 포함하는 엔트리들의 리스트를 반환한다. 이 함수는 검색 과정 중 어느 자식 노드를 검색할지 결정하기 위하여 호출된다.
- **Union**(P): 입력 파라메터로 노드 내의 모든 키 프레디키트의 집합 $P = \{p_1, p_2, \cdots, p_n\}$이 주어지고, 모든 프레디키트 p_i $(1 \leq i \leq n)$로부터 만족되는 프레디키트 r을 생성해서 반환한다. 즉, $(p_1 \vee p_2 \vee \cdots \vee p_n) \Rightarrow r$을 만족하는 프레디키트 r을 반환한다.
- **GetMinPenaltyEntry**(N, p): 입력 파라메터로 노드 N과 새로 삽입할 키 프레디키트 p를 받는다. 이 함수는 노드 N의 엔트리 중에서 그 엔트리가 가리키는 서브트리에 p를 삽입했을 때 도메인 특정(domain-specific)의 최소 삽입 비용을 발생시키는 하나의 엔트리를 반환한다. 이 함수는 삽입 과정 중 어느 서브트리에 새로운 객체를 저장할지 결정하기 위하여 호출된다.
- **PickSplit**(E): 입력 파라메터로 엔트리의 집합 E가 주어지고, E를 두 개의 집합 $E1$, $E2$으로 나누어서 반환한다.
- **ConstructBaseKey**(NodeKeyList, AncestorKeyStack): 입력 파라메터로 노드 안에서 압축 또는 복원하려는 키보다 앞선 키들의 리스트 NodeKeyList와 검색 경로 상의 조상 노드들의 키를 저장한 스택 AncestorKeyStack을 받는다. 이 함수는 NodeKeyList와 AncestorKeyStack에 저장된 키 중에서 하나를 베이스 키로 선택해서 반환한다. 이 함수는 키를 압축 또는 복원할 때 사용되므로, *Compress* 또는 *Decompress* 함수가 호출되기 직전에 호출된다.
- **Compress**(BaseKey, Key): 입력 파라메터로 압축하고자 하는 키 Key와 이 키의 베이스 키 BaseKey를 받는다. 베이스 키를 이용하여 압축된 키를 반환한다.
- **Decompress**(BaseKey, Key): 입력 파라메터로 복원하고자 하는 키 Key와 이 키의 베이스 키 BaseKey를 받는다. 베이스 키를 사용하여 복원된 키를 반환한다.
- **ConvertToKey**(p): 입력으로 받는 프레디키트 p를 키로 변환하여 반환한다.
- **ConvertToPredicate**(Key): 입력으로 받은 키 Key를 프레디키트 p로 변환하여 반환한다.

3.5 트리 메소드

트리 메소드는 모든 캐시 인식 트리의 공통된 알고리즘인 검색, 삽입, 삭제 등의 알고리즘을 구현한 메소드다. CC-GiST를 이용하여 캐시 인식 트리를 구현할 때 각 트리에 특정한 키 메소드는 별도로 구현해 주어야 하나, 포인터 메소드와 트리 메소드는 공통적으로 사용할 수 있다. CC-GiST가 이러한 공통된 알고리즘을 트리 메소드로 제공함으로써 임의의 캐시 인식 트리를 구현하는 것이 매우 능률적이 된다.

트리 메소드는 매우 복잡한 연산을 수행하지만, 실질적으로 기존의 모든 캐시 인식 트리에 공통적으로 포함된 알고리즘을 하나로 묶은 것이다. 따라서, 각 트리 메소드 알고리즘의 수행 과정 및 결과에 대해서는 기존의 캐시 인식 트리 논문들을 함께 참조하여도 무방하다. CC-GiST의 트리 메소드는 GiST [HNP95]에서 정의된 트리 메소드들의 알고리즘을 캐시 인식하도록 완전히 수정하였다. 기존의 GiST로부터 첨가, 수정된 알고리즘 단계들은 맨 앞에 § 기호로 나타내었

다. 예를 들어, 3.5.1 절의 *FindMin* 알고리즘의 FM1과 FM2 단계는 첨가, 수정된 단계이며, FM3 단계는 그렇지 않다.

3.5.1 검색

Search 함수는 사용자 또는 응용 프로그램에서 주어진 질의를 만족시키는 객체들을 찾아 반환한다. 포인터 압축과 관련하여, 이 함수에서는 자식 노드에 대한 복원된 포인터를 얻기 위해 *PointerDecompress* 함수를 호출한다. 키 압축과 관련하여, 이 함수에서는 검색 경로 상의 베이스 키를 생성하여 AncestorKeyStack에 저장하고, 압축된 키들 중에서 질의를 만족하는 엔트리들의 집합을 얻기 위하여 *GetConsistentEntries* 함수를 호출한다.

Algorithm Search(*R*, *q*)

입력: CC-GiST의 루트 노드 *R*, 질의 프레디키트 *q*
출력: 질의를 만족시키는 모든 객체
요약: 질의를 만족하는 키의 자식 노드를 재귀적으로 검색하면서 트리를 내려간다.
§S1: AncestorKeyStack을 비운다.
§S2: [비단말 노드를 탐색한다] 만약 *R*이 비단말 노드라면, *GetConsistentEntries*(*R*, *q*)를 호출하고 반환된 모든 엔트리의 키에 대하여 다음의 S2.1부터 S2.4까지의 단계를 수행한다.
§S2.1: [자식 노드를 가리키는 포인터를 얻는다] 질의를 만족하는 키의 순번을 입력으로 *PointerDecompress* 함수를 호출하여 자식 노드를 가리키는 포인터를 얻는다.
§S2.2: AncestorKeyStack에 키를 푸시(push)한다.
§S2.3: [자식 노드에 대해서 재귀적으로 검색을 수행한다] 자식 노드를 루트로 하는 서브트리에 대해서 재귀적으로 *Search* 함수를 호출한다.
§S2.4: AncestorKeyStack로부터 키를 팝(pop)한다.
§S3: [단말 노드를 검색한다] 만약 *R*이 단말 노드이면 *GetConsistentEntries*(*R*, *q*)을 호출하여 반환되는 모든 엔트리가 가리키는 객체들을 반환한다. □

인덱싱한 도메인에 선형 순서(linear order)가 존재한다면 동등 검색(equality search)과 함께 영역 검색(range search)을 수행할 수 있다. 다음의 *FindMin* 함수를 한번 호출한 후 반복적으로 *Next* 함수를 호출함으로써 영역 검색의 전체 결과를 얻을 수 있다.

Algorithm FindMin(*R*, *q*)

입력: CC-GiST의 루트 노드 *R*, 질의 프레디키트 *q*
출력: 질의 프레디키트 *q*를 만족시키는 객체들 중 가장 작은 키 값을 가진 객체
요약: 비단말 노드에서 질의를 만족하는 키를 가진 가장 왼쪽 노드를 따라 내려간다. 단말 노드에서 질의를 만족하는 가장 작은 키 값을 가진 객체를 반환한다.
§FM1: AncestorKeyStack을 비운다.
§FM2: [비단말 노드를 검색한다] 만약 *R*이 비단말 노드이면, *GetConsistentEntries*(*R*, *q*)를 호출하고 반환된 엔트리의 키들 중에서 *q*를 만족하는 가장 작은 키 *p*를 찾는다. 만약 그러한 *p*가 발견되지 않으면, NULL을 반환한다. *p*를 입력으로 *PointerDecompress* 함수를 호출하여 자식 노드를 가리키는 포인터를 얻는다. *p*를 AncestorKeyStack에 푸시한다. 자식 노드를 루트로 하는 서브트리에 대해서 *FindMin* 함수를 재귀적으로 호출한다.
FM3: [단말 노드를 검색한다] 만약 *R*이 단말 노드이면, *q*를 만족시키는 가장 작은 키 값을 가진 객체를 반환한다. 만약 그러한 객체를 발견하지 못하면 NULL을 반환한다. □

Next 함수는 질의 프레디키트 *q*와 그를 만족하는 키를 포함하는 엔트리 *E*를 입력으로 주며, *q*를 만족하는 모든 객체들을 찾을 때까지 반복적으로 호출된다. 만약 *q*를 만족하는 객체가 없다면 *Next* 함수는 NULL을 반환하고 더 이상 호출되지 않는다.

Algorithm Next(*R*, *q*, *E*)

입력: CC-GiST의 루트 노드 *R*, 질의 프레디키트 *q*, 현재 엔트리 *E*
출력: 질의를 만족시키는 다음 객체
요약: 단말 노드에서 주어진 엔트리의 바로 다음에 오며 *q*를 만족하는 객체를 반환한다. 만약 그러한 객체가 없다면 NULL을 반환한다.
N1: [같은 노드에서 다음 객체를 검색] 만약 *E*가 단말 노드에서 가장 오른쪽 엔트리가 아니라면, *E*의 바로 다음에 오고 *q*를 만족하는 객체를 반환한다. 만약 그러한 객체가 없다면 NULL을 반환한다.
N2: [다음 단말 노드에서 다음 객체를 검색] 만약 *E*가 단말 노드에서 가장 오른쪽 엔트리라면, 현재 노드의 다음 단말 노드 *L*을 검색한다. 만약 *L* 내의 가장 왼쪽 객체의 키가 *q*를 만족하면 그 객체를 반환한다. 만약 그러한 객체가 없다면 NULL을 반환한다. □

3.5.2 삽입

Insert 함수는 CC-GiST 내의 주어진 레벨에 하나의 엔트리를 삽입한다. CC-GiST에서는 GiST [HNP95]에서와 같이 단말 노드의 레벨을 0으로 하고 상위 노드로 올라가면서 레벨이 1씩 증가한다. 포인터 압축과 관련하여, 이 함수는 단말 노드에서 *PointerDecompress* 함수를 호출한다. 키 압축과 관련하여, 이 함수는 *Compress* 함수를 호출하고 검색 경로 상의 베이스 키를 생성하여 AncestorKeyStack에 저장한다.

Algorithm Insert(*R*, *E*, *l*)

입력: CC-GiST의 루트 노드 *R*, 엔트리 *E* = (*p*, ptr), 삽입할 노드 레벨 *l*. 여기에서, *p*는 키 프레디키트, ptr은 포인터이다.
출력: 레벨 *l*에 *E*를 삽입한 CC-GiST
요약: 최소 삽입 비용으로 *E*가 삽입될 노드를 찾아서 *E*를 삽입한다. 만약 충분한 공간이 없다면 *Split* 함수를 호출한다.

§I1: AncestorKeyStack을 비운다.
I2: [E를 삽입할 노드를 찾는다] *ChooseSubtree*(R, E, l)을 호출하고 반환된 노드를 L이라 하자.
I3: 노드 L에 E을 삽입할 충분한 공간이 있는가에 따라 I3.1 또는 I3.2을 수행한다.
I3.1: 만약 노드 L에 충분한 공간이 있으면 다음을 수행한다.
§I3.1.1: [키 p를 저장한다]
§I3.1.1.1: [노드 내에 엔트리를 삽입할 위치를 찾는다] 만약 IsOrdered 플래그가 TRUE이면, L 내에서 키의 순서에 따라 *ConvertToKey*(p)의 결과를 삽입할 올바른 위치를 찾는다. 만약 IsOrdered가 FALSE이면 L 내의 가장 오른쪽에 삽입한다.
§I3.1.1.2: [키를 노드에 삽입한다] 노드 L에서 *ConvertToKey*(p)보다 앞선 키들의 리스트를 LeftKeyList라고 하자. *Compress*(*ConstructBaseKey*(LeftKeyList, AncestorKeyStack), *ConvertToKey*(p))를 호출하여 압축된 키를 얻어 노드 L에 저장한다.
§I3.1.1.3: [*ConvertToKey*(p) 뒤에 오는 압축된 키를 재생성한다] 노드 L에서 *ConvertToKey*(p)의 뒤에 오는 키들의 리스트를 RIghtKeyList라고 하자. 만약 IsOrdered 플래그가 TRUE이면, RIghtKeyList 내의 키들에 대한 베이스 키가 갱신되었으므로, RIghtKeyList 내의 압축된 키들을 재생성한다.
§I3.1.2: [포인터 ptr을 저장한다]
§I3.1.2.1: 만약 L이 비단말 노드라면, 포인터 ptr을 입력으로 *PointerCompress* 함수를 호출하여 반환된 SegmentPointer를 L에 저장한다.
§I3.1.2.2: 만약 L이 단말 노드라면, 포인터 ptr을 L에 저장한다.
I3.2: 만약 노드 L에 충분한 공간이 없으면 *Split*(R, L, E)를 호출한다.
I4: [삽입으로 인한 변경을 상위 노드로 전파시킨다] *AdjustKeys*(R, L)을 호출한다. □

ChooseSubtree 함수는 CC-GiST 내의 지정된 레벨 상에 엔트리를 삽입할 노드를 선택한다. 포인터 압축 관련하여, 이 함수는 *PointerDecompress* 함수를 호출한다. 키 압축 관련하여, 이 함수는 *GetMinPenaltyEntry* 함수를 호출하여 최소 삽입 비용을 갖는 엔트리를 얻는다.

Algorithm ChooseSubTree(R, E, l)
입력: CC-GiST의 루트 노드 R, 엔트리 E = (p, ptr), 노드 레벨 l
출력: 레벨 l에서 E를 저장하기 위한 노드
요약: 레벨 l까지 *GetMinPenaltyEntry* 함수의 결과로 반환된 노드를 재귀적으로 내려간다.
CS1: 노드 R의 레벨이 l과 같다면 R을 반환한다.
§CS2: *GetMinPenaltyEntry*(R, $E.p$)를 호출하여 최소 비용으로 E를 삽입할 엔트리 E'을 찾고, *ConvertToKey*($E'.p$)의 결과를 AncestorKeyStack에 저장한다. *PointerDecompress* 함수를 호출하여 자식 노드에 대한 포인터 E'.ptr을 얻는다. 재귀적으로 *ChooseSubtree*(E'.ptr, E, l)을 호출한다. □

Split 함수는 오버플로우(overflow) 노드를 처리하며, 다음과 같이 포인터 압축과 키 압축을 수행한다. 노드 내에 엔트리들을 분배하기 전에 *PointerDecompress*와 *Decompress* 함수들을 호출하여 노드 내의 모든 포인터와 키를 복원한다. 모든 엔트리들은 오버플로우 노드와 새로 생성된 노드에 분할 저장된다. 엔트리들의 분배 후에 *PointerCompress*와 *Compress* 함수를 호출하여 모든 포인터와 키를 다시 압축한다.

Algorithm Split(R, N, E)
입력: CC-GiST의 루트 노드 R, 분할할 노드 N, 오버플로우를 일으킨 엔트리 E
출력: N을 분할하고 E가 삽입된 CC-GiST
요약: N 내의 모든 엔트리와 E를 입력으로 *PickSplit* 함수를 호출하여 두 개의 그룹으로 분할한다. 이 중 하나는 새로이 생성된 노드 N'에 저장하고 부모 노드에는 N'에 대한 새로운 엔트리 E'을 삽입한다.
SP1: [N 내의 모든 엔트리와 E를 두 그룹으로 분할하고 노드에 저장한다]
§SP1.1: [N 내의 키와 포인터를 복원한다] N 내의 모든 엔트리의 집합을 SE라고 하자. SE 내의 각 엔트리의 프레디키트는 *Decompress*와 *ConvertToPredicate* 함수를 호출함으로써 구한다. SE 내의 각 엔트리의 포인터는 *PointerDecompress* 함수를 호출함으로써 구한다.
SP1.2: [*PickSplit* 함수를 호출] N 내의 모든 엔트리와 E를 입력으로 *PickSplit* 함수를 호출하여 두 개의 그룹 $G1$과 $G2$로 분할한다.
§SP1.3: 새로운 노드 N'을 할당한다.
§SP1.4: [자식 노드를 N'의 세그먼트로 이동한다]
§SP1.4.1: $G2$ 내의 엔트리들이 가리키는 자식 노드들을 N'의 세그먼트로 이동한다.
§SP1.4.2: $G2$ 내의 엔트리들의 포인터를 이동된 자식 노드를 가리키도록 변경한다.
§SP1.5: [N와 N' 내의 키와 포인터를 압축한다] G1과 G2 내의 엔트리들을 각각 N와 N'에 저장한다. 이때, *ConvertToKey*, *Compress*, *PointerCompress* 함수를 호출하여 키와 포인터를 압축한다.
SP2: [새로운 루트 노드를 할당한다] 만약 N이 루트 노드 R과 같다면, 새로운 루트 노드 R'을 할당한다. 노드 N과 N'에 대한 엔트리를 각각 E_N = (p_N, ptr_N), $E_{N'}$ = ($p_{N'}$, $\text{ptr}_{N'}$)이라고 하자. 여기에서, p_N은 N을 루트로 하는 트리 내의 모든 키 프레디키트를 입력으로 *Union* 함수를 호출하여 얻어진 키 프레디키트이며, ptr_N은 N을 가리키는 포인터이다. $p_{N'}$과 $\text{ptr}_{N'}$도 유사하게 구해진다. *Insert*(R', E_N, *Level*(R'))과 *Insert*(R', $E_{N'}$, *Level*(R'))을 호출한다.
SP3: [N'에 대한 새로운 엔트리를 부모 노드에 삽입한다] 노드

N'에 대한 엔트리를 SP2 단계에서와 같이 $E_{N'}$ = ($p_{N'}$, $ptr_{N'}$)이라고 하자.

SP3.1: 만약 부모 노드에 $E_{N'}$을 위한 충분한 공간이 있다면 다음을 수행한다.

§SP3.1.1: 만약 IsFixedSizeSegment 플래그가 TRUE이면, 다음을 수행한다.

§SP3.1.1.1: 만약 IsOrdered 플래그가 TRUE이면, 다음을 수행한다. 노드 N을 포함하는 세그먼트를 S라 하자. 부모 노드의 모든 세그먼트 내에서 N 다음에 오는 노드들을 오른쪽으로 시프트(shift)한다. 세그먼트 S 내에서 N 바로 다음에 N'을 저장한다.

§SP3.1.1.2: 만약 IsOrdered 플래그가 FALSE이면, 다음을 수행한다. 부모 노드의 가장 오른쪽 노드를 포함하는 세그먼트를 S라 하자. 노드 N'을 S 내의 가장 오른쪽 노드로 저장한다.

§SP3.1.2: 만약 IsFixedSizeSegment 플래그가 FALSE이면, 다음을 수행한다. 만약 IsOrdered 플래그가 TRUE이면 노드 N을 포함하는 세그먼트를 S라 하고, 만약 FALSE이면 부모 노드의 가장 오른쪽 노드를 포함하는 세그먼트를 S라 하자.

§SP3.1.2.1: 새로운 세그먼트 S'을 할당한다. S 내의 모든 노드와 N'을 S'으로 이동한다.

§SP3.1.2.2: 만약 IsOrdered 플래그가 TRUE이면, 세그먼트 내에서 N 바로 다음에 N'을 저장한다.

§SP3.1.2.3: 만약 IsOrdered 플래그가 FALSE이면, 부모 노드의 가장 오른쪽 노드로 N'을 저장한다.

§SP3.1.2.4: S를 가리키던 SegmentPointer를 S'을 가리키도록 변경하고 S를 제거한다.

§SP3.1.3: *Insert*(*Parent*(N), $E_{N'}$, *Level*(*Parent*(N))을 호출한다.

SP3.2: [부모 노드를 분할한다] 만약 부모 노드에 $E_{N'}$을 위한 충분한 공간이 없다면, *Split*(R, N, $E_{N'}$)을 호출한다.

§SP4: [부모 노드 내의 키를 보정하고, 자식 노드 내의 키를 재생성한다] 부모 노드 내에서 자식 노드 N에 대한 키 프레디키트를 p라고 하자. p를 p = *Union*(N 내의 모든 키 프레디키트)가 되도록 보정한다. p는 N 내의 모든 압축된 키에 대한 새로운 베이스 키이므로, N 내의 모든 압축 키를 재생성한다. 만약 StoreBaseKeyInNode 플래그가 TRUE이면, N과 N' 내의 베이스 키로 *ConvertToKey*(*Union*(N 내의 모든 프레디키트))와 *ConvertToKey*(*Union*(N' 내의 모든 프레디키트))의 결과를 각각 저장한다. □

AdjustKeys 함수는 상위 노드의 키 프레디키트를 자식 노드의 모든 키 프레디키트로부터 유추될 수 있도록 보정한다. 상위 노드의 보정된 키는 자식 노드의 압축된 키들에 대한 베이스 키이므로, 자식 노드 내의 모든 키를 재생성하여야 한다.

Algorithm AdjustKeys(R, N)

입력: CC-GiST의 루트 노드 R, 하나의 노드 N

출력: 삽입/삭제 경로 상의 키 프레디키트가 모든 자식 노드 내의 키 프레디키트로부터 유추될 수 있도록 보정된 CC-GiST

요약: 삽입/삭제 경로 상의 조상 노드로 올라가면서 부모 노드의 키 프레디키트가 자식 노드의 모든 키 프레디키트로부터 유추될 수 있도록 보정한다. 만약 루트 노드에 도달하거나 부모 노드의 키 프레디키트에 변경이 없다면, 실행을 종료한다.

AK1: 만약 노드 N이 루트 노드라면 종료한다. 만약 자식 노드 N에 대한 부모 노드 PN 내의 키 프레디키트 p가 p = *Union*(N 내의 모든 키 프레디키트)를 만족하면 종료한다.

§AK2: p를 p = *Union*(N의 모든 키 프레디키트)가 되도록 보정한다. p가 N 내의 모든 압축된 키에 대한 베이스 키이므로, N 내의 압축된 키들을 재생성한다. 만약 StoreBaseKeyInNode 플래그가 TRUE이면, *ConvertToKey*(*Union*(N 내의 키 프레디키트))의 결과를 베이스 키로 저장한다.

AK3: 재귀적으로 *AdjustKeys*(R, PN)을 호출한다. □

3.5.3 삭제

Delete 함수는 CC-GiST의 단말 노드 내의 하나의 엔트리 E를 삭제한다. 삭제된 키 $E.p$는 단말 노드 내에서 $E.p$의 다음에 오는 모든 압축된 키들의 베이스 키이므로, $E.p$ 다음의 모든 키를 재생성하여야 한다.

Algorithm Delete(R, E)

입력: CC-GiST의 루트 노드 R, 단말 노드 엔트리 E = (p, ptr)

출력: E가 삭제된 CC-GiST

요약: 단말 노드에서 E를 삭제한다. 삭제로 인한 언더플로우(underflow)를 처리하기 위해 *CondenseTree* 함수를 호출한다.

D1: [삭제할 엔트리를 저장한 단말 노드를 찾는다] *Search*(R, $E.p$)를 호출하여 E를 저장하고 있는 단말 노드 L을 찾는다. 만약 그러한 L을 찾지 못하면 종료한다.

D2: [엔트리를 삭제한다] 노드 L에서 엔트리 E를 삭제한다.

§D3: [압축된 키를 재생성한다.] 만약 IsOrdered 플래그가 TRUE이면, L 내에서 $E.p$보다 큰 모든 키의 베이스 키가 변경되었으므로 $E.p$보다 큰 모든 압축 키를 재생성한다.

D4: [삭제로 인한 변경을 전파시킨다] *CondenseTree*(R, L) 함수를 호출한다.

D5: *CondenseTree* 함수 호출 후 만약 루트 노드가 하나의 자식 노드만을 가지면 그 자식 노드를 새로운 루트 노드로 만들고 이전 루트 노드를 제거한다. □

언더플로우 노드는 *CondenseTree* 함수에서 처리한다. 재분배 또는 병합의 경우에 관계 없이, 인접 노드로 이동한 엔트리들의 포인터와 키는 *Split* 함수에서처럼 다시 압축되어야 한다. 또한, 이동한 엔트리의 모든 자식 노드들도 인접 노드의 세그먼트로 이동하여야 한다.

Algorithm CondenseTree(*R*, *L*)

입력: CC-GiST의 루트 노드 *R*, 단말 노드 *L*

출력: 3.2 절에서 설명한 특성을 유지하는 CC-GiST

요약: 만약 단말 노드 *L*이 *kM*보다 적은 개수의 엔트리를 가진다면 아래의 두 가지 중 하나를 수행한다. (1) 노드 *L*을 제거하고 *L*에 포함되어 있던 엔트리들을 트리에 다시 삽입한다. (2) 다른 단말 노드로부터 엔트리를 추출해 *L*에 삽입한다. 노드의 삭제로 인한 변경을 트리의 상위 노드로 전파시킨다. 노드 *L*에서 루트 노드까지의 삭제 경로상의 키를 보정한다.

CT1: [초기화] $N = L$로 대입한다. 제거되는 노드 내의 키들을 저장할 Q를 비운다.

CT2: 만약 N이 루트 노드라면 CT6을 수행한다. 그렇지 않으면 $PN = Parent(N)$이라 하고, N을 가리키는 PN 내의 엔트리를 E_N이라 하자.

§CT3: [언더플로우 노드를 처리한다] 만약 N이 kM보다 적은 개수의 엔트리를 가진다면 다음을 수행한다. N이 저장된 세그먼트를 S라고 하자.

§CT3.1: [만약 IsOrdered 플래그가 FALSE이면 N을 제거한다] 만약 IsOrdered 플래그가 FALSE이면, N 내의 모든 키들을 Q에 저장하고, PN 내의 엔트리 E_N을 제거하고 *AdjustKeys*(R, PN)을 호출한다. N을 제거한다.

§CT3.2: [만약 IsOrdered 플래그가 TRUE이면 엔트리를 재분배하거나 인접 노드와 병합한다] 만약 IsOrdered 플래그가 TRUE이면 다음을 수행한다. N'을 N에 인접한 노드라고 하자. 재분배 또는 병합의 모든 경우에, 이동한 엔트리의 포인터와 키는 다시 압축되어야 한다.

§CT3.2.1: [만약 N과 N' 내의 엔트리의 개수의 합 ≥ $2kM$이면, N과 N' 내의 엔트리와 자식 노드를 재분배한다.] 만약 N과 N' 내의 엔트리의 개수의 합 ≥ $2kM$이면, N과 N' 내의 엔트리 개수가 같도록 재분배한다. 재분배된 엔트리가 가리키는 N의 자식노드들을 N'의 세그먼트로 이동한다.

§CT3.2.2: [그렇지 않으면, 노드 및 세그먼트를 병합한다.] 만약 N과 N' 내의 엔트리의 개수의 합 < $2kM$이면, N 내의 모든 엔트리를 N'으로 이동하고 N을 제거한다.

§CT3.2.3: [이동한 엔트리들이 가리키는 자식 노드들을 N'의 세그먼트로 이동한다] N과 인접한 노드가 N의 오른쪽 노드라고 하자. 그리고, S를 N'의 가장 왼쪽 자식 노드를 포함하는 세그먼트라고 하자.

§CT3.2.3.1: 만약 IsFixedSizeSegment 플래그가 TRUE이면, N'의 모든 자식 노드들을 오른쪽으로 시프트한다. 이동한 엔트리들이 가리키는 자식 노드들을 S 내의 왼쪽에 저장한다.

§CT3.2.3.2: 만약 IsFixedSizeSegment 플래그가 FALSE이면, 다음을 수행한다. 새로운 세그먼트 S'을 할당한다. 이동한 엔트리들이 가리키는 자식 노드들과 S에 포함되어 있던 자식 노드들을 S'에 이동한다. N의 부모 노드에서 S를 가리키던 SegmentPointer를 S'을 가리키도록 수정하고, S를 제거한다.

§CT3.2.4: *AdjustKeys*(R, N')과 *AdjustKeys*(R, PN)을 호출한다.

§CT3.3: [세그먼트 S 내의 노드들을 시프트한다] 만약 노드 N이 제거되었다면, 세그먼트 S 내에서 N 다음에 오는 노드들을 왼쪽으로 시프트한다.

§CT3.3.1: 만약 IsFixedSizeSegment 플래그가 TRUE이면, 세그먼트 S 내에서 N 다음에 오는 노드들을 왼쪽으로 시프트한다.

§CT3.3.2: 만약 IsFixedSizeSegment 플래그가 FALSE이면, 다음을 수행한다. 새로운 세그먼트 S'을 할당하고, S 내의 모든 노드를 S'으로 이동한다. N의 부모 노드에서 S를 가리키던 SegmentPointer를 S'을 가리키도록 수정하고, S를 제거한다.

CT4: [상위 노드의 키를 보정한다] 만약 E_N이 PN에서 삭제되지 않았다면, *AdjustKeys*(R, N)을 호출한다.

CT5: [삭제로 인한 변경을 전파시킨다] 만약 PN에서 E_N이 삭제되었다면, $N = PN$을 대입하고 CT2를 수행한다.

CT6: [고아 엔트리(orphaned entry)를 재삽입한다] 만약 Q가 비어 있지 않으면 Q 내의 각 엔트리 E에 대해서 *Insert*(R, E, *Level*(E))를 호출한다. □

4. CC-GiST를 이용한 캐시 인식 트리의 구현

본 장에서는 CC-GiST를 이용하여 기존의 대표적인 캐시 인식 트리인 CSB$^+$-트리 [RR00], pkB-트리 [BMR01], CR-트리 [KCK01]를 구현하는 방법에 대해 설명한다. 구체적으로, 각 트리에 대한 메소드의 구현과 플래그들의 설정 방법을 기술한다.

4.1 CSB$^+$-트리의 구현

CSB$^+$-트리의 특성은 표 2와 같이 정리할 수 있다. 이러한 특성을 반영하여 메소드를 구현하고 플래그를 설정할 수 있다.

〈표 2〉 CSB$^+$-트리의 특성

특성	설정
포인터 압축	TRUE
키 압축	FALSE
세그먼트 크기	가변 크기
세그먼트 내의 최대 노드 개수	블로킹 인수와 같음
비단말 노드에서 가리키는 세그먼트의 개수	1
노드 내의 엔트리 간의 순서 설정	TRUE

CC-GiST를 이용하여 구현한 CSB$^+$-트리에서 키 프레디키트 p는 $p = Contains([x_p, y_p), v)$로 표현된다. 여기에서, $y_p = x_{p+1}$이므로, 키 p는 하나의 정수 x_p 만으로도 줄여서 표현 가능하다. 키 메소드의 구현은 아래와 같다:

- **GetConsistentEntries**(N, q): 만약 $q = Equal(x_q, v)$이면 $x_p \le x_q$를 만족하는 키를 가지는 가장 오른쪽 엔트

리를 반환한다. 만약 q = *Contains*($[x_q, y_q)$, v)이면 $x_q < x_p < y_q$를 만족하는 키를 가지는 엔트리들을 반환한다.

- **Union**(P = {$[x_1, y_1)$, ⋯ , $[x_n, y_n)$}): 이 함수는 [min(x_1, ⋯, x_n), max(y_1, ⋯, y_n))을 반환한다.
- **GetMinPenaltyEntry**(N, p): 노드 N의 엔트리 중에서 p보다 작거나 같은 키 중에서 가장 큰 키 값을 가지는 엔트리를 반환한다.
- **PickSplit**(E): 앞쪽의 $\left\lfloor \frac{|E|}{2} \right\rfloor$ 개의 엔트리들을 첫번째 집합 $E1$, 나머지 엔트리들을 두번째 집합 $E2$에 분배한다.
- **Compress**(BaseKey, Key): 항상 Key를 반환한다.
- **Decompress**(BaseKey, Key): 항상 Key를 반환한다.
- **ConstructBaseKey**(NodeKeyList, AncestorKeyStack): 항상 NULL를 반환한다.
- **ConvertToKey**($[x_p, y_p)$): x_p를 반환한다.
- **ConvertToPredicate**(Key): 프레디키트 $[x, y)$를 반환한다. 키 Key를 포함하는 엔트리를 E라고 하고 바로 다음 엔트리를 E'이라고 하자. 비단말 노드에서는 x = Key, y = $E'.p$이며, 단말 노드에서는 x = Key, y = x + 1이다. 만약 E가 비단말 노드의 가장 왼쪽 엔트리라면 x = $-\infty$이며, 만약 E가 비단말 노드의 가장 오른쪽 엔트리라면 y = ∞이다.

CSB$^+$-트리에 대한 변수는 아래와 같이 설정한다:

- IsFixedSizeSegment = FALSE: 가변 길이 세그먼트를 사용한다.
- IsOrdered = TRUE: 노드 내의 엔트리들 간에 순서가 존재한다.
- MaxNumOfNodeInSegment = 블로킹 인수: 하나의 세그먼트 안에 저장될 수 있는 최대 노드 개수는 자식 노드의 개수와 같다.
- StoreBaseKeyInNode = FALSE: 노드 내에 베이스 키를 저장하지 않는다.

4.2 pkB-트리의 구현

pkB-트리의 특성은 표 3과 같이 정리할 수 있다. 이러한 특성을 반영하여 메소드를 구현하고 플래그를 적절히 설정할 수 있다.

〈표 3〉 pkB-트리의 특성

특성	설정
포인터 압축	FALSE
키 압축	TRUE
세그먼트 크기	고정 길이
세그먼트 내의 최대 노드 개수	1
비단말 노드에서 가리키는 세그먼트의 개수	블로킹 인수와 같음
노드 내의 엔트리 간의 순서 설정	TRUE

CC-GiST를 이용하여 구현한 pkB-트리에서 키 프레디키트 p는 CSB$^+$-트리에서와 같은 방식으로 표현된다. 키 메소드의 구현은 아래와 같다:

- **GetConsistentEntries**(N, q): 노드 N에서 질의 키 q보다 작거나 같은 키 중에서 가장 큰 키를 가지는 엔트리를 반환한다.
- **Union**(P = {$[x_1, y_1)$, ⋯ , $[x_n, y_n)$}): CSB$^+$-트리와 동일하다.
- **GetMinPenaltyEntry**(N, p): CSB$^+$-트리와 동일하다.
- **PickSplit**(E): CSB$^+$-트리와 동일하다.
- **Compress**(BaseKey, Key): BaseKey를 이용해서 부분 키(partial key)을 생성하여 반환한다.
- **Decompress**(BaseKey, Key): BaseKey를 이용하여 부분 키 Key로부터 전체 키를 복원하여 반환한다.
- **ConstructBaseKey**(NodeKeyList, AncestorKeyStack): 만약 NodeKeyList 내에 하나 이상의 키가 포함되어 있으면 가장 마지막 키를 반환한다. 만약 NodeKeyList 내에 키가 존재하지 않으면 AncestorKeyStack의 탑(top)의 키를 반환한다.
- **ConvertToKey**($[x_p, y_p)$): CSB$^+$-트리와 동일하다.
- **ConvertToPredicate**(Key): CSB$^+$-트리와 동일하다.

pkB-트리에 대한 변수는 아래와 같이 설정한다:

- IsFixedSizeSegment = TRUE: 고정 길이 세그먼트 사용한다.
- IsOrdered = TRUE: 노드 내의 엔트리들 간에 순서가 존재한다.
- MaxNumOfNodeInSegment = 1: 하나의 세그먼트는 최대 하나의 자식 노드를 포함한다.
- StoreBaseKeyInNode = FALSE: 노드 내에 베이스 키를 저장하지 않는다.

4.3 CR-트리의 구현

CR-트리의 특성은 <표 4>와 같이 정리할 수 있다. 이러한 특성을 반영하여 메소드를 구현하고 플래그를 적절히 설정할 수 있다.

〈표 4〉 CR-트리의 특성

특성	설정
포인터 압축	FALSE
키 압축	TRUE
세그먼트 크기	고정 길이
세그먼트 내의 최대 노드 개수	1
비단말 노드에서 가리키는 세그먼트의 개수	블로킹 인수와 같음
노드 내의 엔트리 간의 순서 설정	FALSE

CC-GiST를 이용하여 구현한 CR-트리에서 키 프레디키트 p는 p = *Contains*($(x^{ul}, y^{ul}, x^{lr}, y^{lr})$, v)로 표현되며, (x^{ul}, y^{ul})과 (x^{lr}, y^{lr})은 각각 QRMBR의 왼쪽 위와 오른쪽 아래의

좌표이다. 키 p는 줄여서 (x^{ul}, y^{ul}, x^{lr}, y^{lr})만으로도 표현 가능하다. 키 메소드 구현은 아래와 같다:

- **GetConsistentEntries**(N, q): 이 함수는 노드 N의 엔트리들 중에서 질의 q를 입력으로 *Compress* 함수를 호출하여 반환된 QRMBR R과 겹치는 QRMBR R'을 가지는 모든 엔트리들을 반환한다.
- **Union**(P = {(x_1^{ul}, y_1^{ul}, x_1^{lr}, y_1^{lr}), ⋯ , (x_n^{ul}, y_n^{ul}, x_n^{lr}, y_n^{lr})}): (min(x_1^{ul}, ⋯ , x_n^{ul}), max(y_1^{ul}, ⋯ , y_n^{ul}), max(x_1^{lr}, ⋯ , x_n^{lr}), min(y_1^{lr}, ⋯ , y_n^{lr})) 좌표를 갖는 QRMBR을 반환한다.
- **GetMinPenaltyEntry**(N, E): 이 함수는 노드 N의 엔트리들 중에서 E의 QRMBR R과 병합했을 때 크기가 가장 적게 증가하는 QRMBR R'을 가지는 엔트리 E'를 반환한다.
- **PickSplit**(E): R-트리 [Gutt84]에서 사용된 선형 비용 분할(linear-cost split) 알고리즘을 이용하여 두 개의 그룹으로 분배하여 반환한다.
- **Compress**(BaseKey, Key): 키 Key에 대하여 CR-트리에서와 같이 •상대 좌표 및 양자화를 이용한 QRMBR을 생성하여 반환한다.
- **Decompress**(BaseKey, Key): 키 Key의 좌표에 베이스 키 BaseKey의 상대 좌표를 더하고 역양자화(de-quantization)을 수행한다. 이 과정을 AncestorKeyStack이 빌 때까지 반복한다. 최종적으로, 키 Key의 실제 MBR의 좌표를 반환한다.
- **ConstructBaseKey**(NodeKeyList, AncestorKeyStack): AncestorKeyStack의 탑의 키를 반환한다.
- **ConvertToKey**(p): p를 반환한다.
- **ConvertToPredicate**(Key): Key를 반환한다.

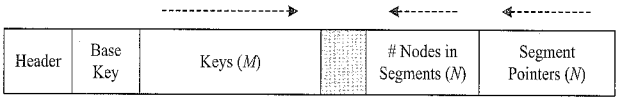
CR-트리에 대한 변수는 아래와 같이 설정한다:

- IsFixedSizeSegment = TRUE: 고정 길이 세그먼트를 사용한다.
- IsOrdered = FALSE: 노드 내의 엔트리들 간에 순서가 존재하지 않는다.
- MaxNumOfNodeInSegment = 1: 하나의 세그먼트는 최대 하나의 자식 노드를 포함한다.
- StoreBaseKeyInNode = TRUE: 노드 내에 베이스 키를 저장한다.

5. 성능 분석

본 절에서는 본 논문에서 제안한 CC-GiST의 성능을 수식적으로 분석한다. 캐시 인식 트리의 성능은 노드 내의 평균적인 엔트리 개수에 대체로 반비례한다. 노드 내의 엔트리 개수가 감소하면 캐시 미스가 발생할 가능성이 높아지며 메인 메모리에 대한 액세스가 증가하게 되어 캐시 인식 트리의 성능을 저하시킨다. 따라서, 기존의 캐시 인식 트리들은 포인터 압축 또는 키 압축을 통하여 노드 내의 엔트리 개수를 증가시키려 하였다. 본 절에서는 기존의 캐시 인식 트리를 CC-GiST를 이용하여 구현하였을 때에 비단말 노드 내의 엔트리 개수가 어떻게 변화하는지 검토한다.

(그림 5)는 CC-GiST의 비단말 노드의 구조를 보인 것이다. 비단말 노드는 헤더, 베이스 키(base key), 키 리스트, 세그먼트 내의 자식 노드의 개수 리스트, 세그먼트 포인터 리스트로 구성된다. 헤더는 키의 개수(M), 세그먼트의 개수(N), 플래그 등으로 구성되며, 베이스 키는 CC-GiST로 구현하는 캐시 인식 트리에 따라 노드 내에 저장하지 않을 수도 있다. 키 리스트는 M 개의 키로 구성되고, 세그먼트 내의 자식 노드의 개수 리스트, 세그먼트 포인터 리스트는 각각 N 개의 요소로 구성된다. 점선 화살표는 새로운 키 또는 세그먼트가 삽입되는 방향을 나타낸다.



(그림 5) CC-GiST 비단말 노드의 구조

CC-GiST의 비단말 노드 내의 엔트리 개수는 포인터 압축을 수행하는 경우와 수행하지 않는 경우로 구분하여 계산할 수 있다. 포인터 압축을 수행하는 경우에는 $M > N$이며, 엔트리 개수는 다음과 같다:

$$\frac{NodeSize-(HeaderSize+BaseKeySize+SegNodesListSize+SegPtrListSize)}{KeySize} \quad (1)$$

여기에서, *SegNodesListSize*는 세그먼트 내의 자식 노드의 개수 리스트의 크기이며, *SegPtrListSize*는 세그먼트 포인터 리스트의 크기이다. 포인터 압축을 수행하지 않는 경우에는, 모든 키에 대하여 하나의 세그먼트 포인터를 할당하므로 (M = N), 엔트리 개수는 다음과 같다:

$$\frac{NodeSize-(HeaderSize+BaseKeySize)}{KeySize+SegNodesSize+SegPtrSize} \quad (2)$$

여기에서, *SegNodesSize*는 하나의 세그먼트 내의 자식 노드의 개수를 표현하기 위한 크기이며, *SegPtrSize*는 하나의 세그먼트 포인터의 크기이다. 공식 (1)의 SegNodesListSize와 SegPtrListSize에 대하여 각각 $SegNodesListSize = SegNodesSize \times N$, $SegPtrListSize = SegPtrSize \times N$이 성립한다.

아래의 보조 정리 1, 2, 3은 CC-GiST의 비단말 노드 내의 엔트리 개수를 각각 CSB$^+$-트리, pkB-트리, CR-트리의 엔트리 개수로 나눈 비율을 보인 것이다. 보조 정리에서 CC-GiST 노드의 헤더의 크기는 4 바이트, 세그먼트 포인터의 크기는 4 바이트, 세그먼트 내의 자식 노드의 개수를 표현하기 위한 크기는 1 바이트로 가정한다.

[보조 정리 1] 임의 크기의 키에 대하여, 기존의 CSB$^+$-트리의 비단말 노드 내의 엔트리 개수와 CC-GiST로 구현한 CSB$^+$-트리의 엔트리 개수 간의 평균적인 비율은 다음과 같다:

$$1+\frac{3}{NodeSize-9} \tag{3}$$

[증명] 기존의 CSB$^+$-트리의 비단말 노드는 헤더, 하나의 세그먼트 포인터, M 개의 키로 구성된다. 헤더는 키의 개수 및 플래그로 구성되고, 2 바이트로 표현된다고 가정한다. CSB$^+$-트리의 비단말 노드 내의 엔트리 개수는 다음과 같다:

$$\frac{NodeSize-(HeaderSize+SegPtrSize)}{KeySize}=\frac{NodeSize-6}{KeySize} \tag{4}$$

CC-GiST로 구현한 CSB$^+$-트리의 비단말 노드는 베이스 키를 저장하기 위한 공간이 필요하지 않으며, 포인터 압축을 수행하여 하나의 세그먼트에 대한 데이터만을 가지므로 ($M > N = 1$), 엔트리 개수는 공식 (1)을 이용하여 다음과 같이 구할 수 있다:

$$\frac{NodeSize-(4+0+1+4)}{KeySize}=\frac{NodeSize-9}{KeySize} \tag{5}$$

공식 (4)를 공식 (5)로 나누면 공식 (3)을 얻는다. □

[보조 정리 2] 기존의 pkB-트리의 비단말 노드 내의 엔트리 개수와 CC-GiST로 구현한 pkB-트리의 엔트리 개수 간의 평균적인 비율은 다음과 같다:

$$\left(1-\frac{2}{NodeSize-4}\right)\cdot\frac{14}{13} \tag{6}$$

여기에서, 기존의 pkB-트리와 CC-GiST로 구현한 pkB-트리의 비단말 노드에 저장되는 키는 부분 키(partial key)이며, 부분 키는 공통적으로 9 바이트(베이스 키를 찾기 위한 포인터 4 바이트, 오프셋 1 바이트, 고정길이 partkey 4 바이트 [BMR01])로 구성된다고 가정한다.

[증명] 기존의 pkB-트리의 비단말 노드는 헤더, 맨 처음 자식 노드에 대한 포인터, M 개의 엔트리로 구성된다. 각 엔트리는 부분 키와 자식 노드에 대한 포인터로 구성된다. 헤더는 엔트리의 개수 및 플래그로 구성되고, 2 바이트로 표현된다고 가정한다. pkB-트리의 비단말 노드 내의 엔트리 개수는 다음과 같다:

$$\frac{NodeSize-(HeaderSize+SegPtrSize)}{EntrySize}=\frac{NodeSize-6}{13} \tag{7}$$

CC-GiST로 구현한 pkB-트리의 비단말 노드는 헤더 바로 다음에 베이스 키를 저장하기 위한 공간이 필요하지 않으며, 포인터 압축을 수행하지 않으므로 ($M = N$), 엔트리 개수는 공식 (2)를 이용하여 다음과 같이 구할 수 있다:

$$\frac{NodeSize-(4+0)}{9+1+4}=\frac{NodeSize-4}{14} \tag{8}$$

공식 (7)을 공식 (8)로 나누면 공식 (6)을 얻는다. □

[보조 정리 3] 기존의 CR-트리의 비단말 노드 내의 엔트리 개수와 CC-GiST로 구현한 CR-트리의 엔트리 개수 간의 평균적인 비율은 다음과 같다:

$$\left(1+\frac{2}{NodeSize-20}\right)\cdot\frac{7}{6} \tag{9}$$

여기에서, MBR 및 QRMBR은 4.3 절에서 보인 것과 같이 네 개의 좌표 값으로 표현한다. 부모 노드에 대한 MBR은 네 개의 4 바이트 = 16 바이트로 표현하며, 자식 노드에 대한 QRMBR은 2.1 절에서와 같이 네 개의 4 비트 = 2 바이트로 표현한다고 가정한다.

[증명] 기존의 CR-트리의 비단말 노드는 헤더, 부모 노드에 대한 MBR, M 개의 엔트리로 구성된다. 각 엔트리는 자식 노드에 대한 QRMBR 및 포인터로 구성된다. 헤더는 엔트리의 개수 및 플래그로 구성되고, 2 바이트로 표현된다고 가정한다. CR-트리의 비단말 노드 내의 엔트리 개수는 다음과 같다:

$$\frac{NodeSize-(HeaderSize+ParentMbrSize)}{EntrySize}=\frac{NodeSize-18}{6} \tag{10}$$

여기에서, *ParentMbrSize*는 부모 노드에 대한 MBR을 표현하기 위한 크기(16 바이트)를 의미한다.

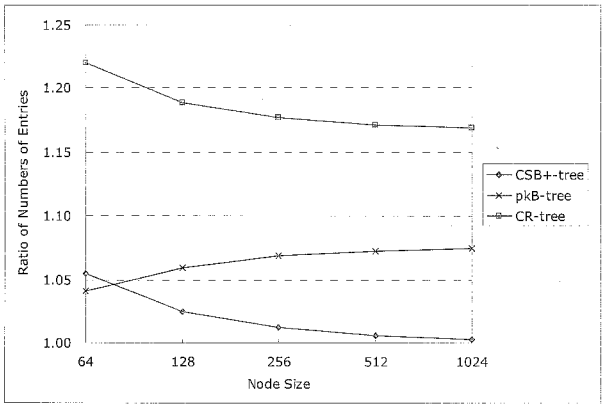
CC-GiST로 구현한 CR-트리의 비단말 노드는 부모 노드에 대한 MBR을 베이스 키로 저장하고, 포인터 압축을 수행하지 않으므로 ($M = N$), 엔트리 개수는 공식 (2)를 이용하여 다음과 같이 구할 수 있다:

$$\frac{NodeSize-(4+16)}{2+1+4}=\frac{NodeSize-20}{7} \tag{11}$$

공식 (10)을 공식 (11)로 나누면 공식 (9)를 얻는다. □

(그림 6)은 노드 크기를 변화시키며 보조 정리 1, 2, 3에서 주어진 엔트리 개수 비율을 비교한 것이다. 노드의 크기는 캐시라인의 최소 크기인 64 바이트의 배수로 정하였다.

(그림 6)에서 보는 바와 같이, CC-GiST로 구현한 캐시 인식 트리들의 비단말 노드 내의 엔트리 개수는 기존의 캐시 인식 트리들에 비하여 크게 증가하지 않는다. 특히, CC-GiST로 구현한 CSB$^+$-트리와 pkB-트리는 최대 7% 이내로 증가하였으며, 노드의 크기가 증가하더라도 비율이 더 이상 증가하지 않는다. CC-GiST로 구현한 CR-트리의 경우 다른 트리들에 비하여 큰 값이 나왔으나, 세그먼트 내에 포함된 자식 노드의 개수(공식 (2)의 *SegNodesSize*)를 유지



(그림 6) CC-GiST와 기존 캐시 인식 트리들 간의 엔트리 개수 비교

하지 않으면 엔트리 개수 비율이 크게 떨어질 것이다.

결론적으로, CC-GiST는 기존의 모든 캐시 인식 트리를 하나의 프레임워크 상에서 지원함에도, 성능에는 별다른 차이가 없음을 알 수 있다. 본 절에서는 비단말 노드에 대한 분석 결과만을 제시하였으나, 단말 노드는 비단말 노드에 비하여 구조가 단순하므로, 단말 노드에 대한 분석 결과도 비단말 노드에 대한 분석 결과와 거의 유사하리라 판단된다.

6. 결 론

본 논문에서는 기존의 디스크 기반의 GiST [HNP95]를 캐시 인식하도록 확장한 CC-GiST를 제안하였다. 본 논문에서는 먼저 기존의 캐시 인식 트리들의 기법을 분석함으로써 캐시 인식 트리에 일반적으로 적용할 수 있는 포인터 압축 및 키 압축 기법들을 도출하였다. 이러한 포인터 압축과 키 압축 기법을 지원하도록 GiST를 확장하여 CC-GiST를 정형적으로 제시하였다. CC-GiST에서는 포인터 압축을 위해 세그먼트의 개념을, 키 압축을 위해 베이스 키의 개념을 도입하였고, GiST에 정의되어 있는 메소드들을 캐시 인식하도록 완전하게 수정하였다. 다음에, 제안된 CC-GiST를 이용하여 기존의 대표적인 캐시 인식 트리인 CSB^+-트리 [RR00], pkB-트리 [BMR01], CR-트리 [KCK01]를 구현하는 방법을 기술하였다. 마지막으로, 기존의 캐시 인식 트리와 CC-GiST 간의 성능을 분석적으로 비교, 평가하였다.

성능 분석 결과, CC-GiST는 기존의 모든 캐시 인식 트리를 하나의 프레임워크 상에서 지원함에도, 성능에는 별다른 차이가 없음을 알 수 있었다. 본 논문에서 제안된 CC-GiST를 이용하여 기존의 캐시 인식 트리를 각각 효율적으로 구현할 수 있을 뿐만 아니라, 기존의 트리들의 장점들을 통합한 하나의 트리를 구현할 수도 있다.

참 고 문 헌

[ADHW99] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "DBMS on a Modern Processor: Where Does Time Go?" In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pp. 54-65, Sept. 1999.

[Bern98] P. Bernstein et al., "The Asilomar Report on Database Research," *SIGMOD Record*, Vol. 27, No. 4, pp. 74-80, Dec. 1998.

[BMR01] P. Bohannon, P. Mcilroy, and R. Rastogi, "Main-Memory Index Structures with Fixed-Size Partial Keys," In *Proc. ACM Int'l Conf. on Management of Data*, ACM SIGMOD, pp. 163-174, May 2001.

[Bonc99] P. A. Boncz et al. "Database Architecture Optimized for the New Bottleneck: Memory Access," In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pp. 54-65, Sept. 1999.

[CKJA98] B. Calder, C. Krintz, S. John, and T. Austin, "Cache-Conscious Data-Placement," In *Proc. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 139-149, Oct. 1998.

[CLH98] T. M. Chilimbi, J. R. Larus, and M. D. Hill, Improving Pointer Based Codes through Cache-Conscious Data Placement, Technical Report, University of Wisconsin-Madison, Computer Science Department, 1998.

[CLH00] T. M. Chilimbi, J. R. Larus, and M. D. Hill, "Making Pointer Based Data Structures Cache Conscious," *IEEE Computer*, Vol. 33, No. 12, pp. 67-74, Dec. 2000.

[Gutt84] A. Guttman, "R-Tree: A Dynamic Index Structure For Spatial Searching," In *Proc. ACM Int'l Conf. Management of Data*, ACM SIGMOD, pp. 47-57, June 1984.

[HNP95] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer, "Generalized Search Trees for Database Systems," In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pp. 562-573, Sept. 1995.

[KCK01] K. Kim, S. K. Cha, and K. Kwon, "Optimizing Multidimensional Index Trees for Main Memory Access," In *Proc. ACM Int'l Conf. on Management of Data*, ACM SIGMOD, pp. 139-150, May 2001.

[Lehm86] T. J. Lehman et al., "A Study of Index Structures for Main Memory Database Management Systems," In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pp. 294-303, Aug. 1986.

[RR99] J. Rao and K. A. Ross, "Cache conscious indexing for decision-support in main memory," In *Proc. of the International Conference on Very Large Data Bases (VLDB)*, pp. 7-10, Sept. 1999.

[RR00] J. Rao and K.A. Ross, "Making B^+-trees Cache Conscious in Main Memory," In *Proc. ACM Int'l Conf. on Management of Data*, ACM SIGMOD, pp. 475-486, May 2000.

[SKN94] A. Shatdal, C. Kant, and J. Naughton, "Cache Conscious Algorithms for Relational Query Processing," In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pp. 510-512, Sept. 1994.

노 웅 기

e-mail : woong@mozart.kaist.ac.kr
1991년 2월 한국과학기술원 전산학과(학사)
1993년 2월 한국과학기술원 전산학과 멀티미디어 전공(석사)
2001년 2월 한국과학기술원 전산학과 데이터 마이닝 전공(박사)
2001년 2월~2003년 9월 ㈜티맥스소프트 책임연구원 (미들웨어 개발)
2003년 10월~2005년 3월 ㈜티맥스데이터 수석연구원 (DBMS 개발)
2005년 4월~2006년 5월 한국과학기술원 전산학과 초빙교수
2006년 6월~현재 미국 University of Minnesota 방문연구원
관심분야 : 데이터 마이닝/데이터 웨어하우징, 멀티미디어 시스템, 멀티미디어 내용기반 검색, 정보 검색, 공간 데이터베이스, 임베디드 데이터베이스, 데이터베이스 시스템

김 원 식

e-mail : wskim@www-db.knu.ac.kr
2003년 2월 계명대학교 컴퓨터공학과(학사)
2005년 8월 경북대학교 컴퓨터공학과(석사)
2006년 2월~현재 ㈜다음 연구원
관심분야 : 임베디드 데이터베이스, 대용량 메일 시스템

한 욱 신

e-mail : wshan@knu.ac.kr
1994년 2월 경북대학교 컴퓨터공학과(학사)
1993년 2월 한국과학기술원 전산학과 데이터베이스 전공(석사)
2001년 8월 한국과학기술원 전산학과 데이터베이스 전공(박사)
2001년 9월~2002년 8월 한국과학기술원 첨단정보기술연구센터 포스트닥
2002년 9월~2003년 2월 한국과학기술원 첨단정보기술연구센터 연구교수
2005년 8월~2006년 8월 IBM Almaden Research Center 포스트닥
2003년 3월~현재 경북대학교 컴퓨터공학과 조교수
관심분야 : 질의 최적화, 유사 검색, 임베디드 데이터베이스, 웹 데이터베이스