# CC-GiST: 캐쉬 인식하는 일반화된 검색 트리†

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

# CC-GiST: Cache Conscious-Generalized Search Trees

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

요 약

주기억 장치 DBMS 성능에 캐쉬 미스가 중요한 요소이다. 그래서 캐쉬 미스를 줄여주는 캐쉬 인식 트리(cache conscious trees)들이 개발되어 왔다. 캐쉬 인식 트리에서 사용한 기법들은 포인터 압축, 키 압축 개념으로 일반화 할 수 있다. 포인터 압축은 CSB$^+$-트리처럼 노드에 각 자식 노드를 가리키는 포인터를 제거하고 대신 세그먼트에 저장된 자식 노드들 중 첫번째 자식 노드를 가리키는 포인터를 저장하는 개념이다. 키 압축은 pkB-트리, CR-트리처럼 키 길이를 줄이는 개념이다.

본 논문에서는 키 압축 개념과 포인터 압축 개념을 동시에 지원하고, 디스크 기반의 GiST를 캐쉬 인식하도록 확장한 CC-GiST를 제안한다. 본 논문의 공헌은 다음과 같이 요약된다. 1) 기존의 캐쉬 인식 트리들의 기법을 분류하고 분석함으로써, 캐쉬 인식 트리에 적용할 수 있는 일반적인 방법을 도출하였다. 2) 포인터 압축을 위해 세그먼트의 개념을, 키 압축을 위하여 베이스 키의 개념을 CC-GiST에 도입하였다. 3) 디스크 기반의 GiST를 위해 기정의된 메소드들을 캐쉬 인식하도록 완전하게 수정하였다. 4) 제안한 CC-GiST를 이용하여 기존의 대표적인 캐쉬 인식 트리인 CSB+-트리와 CR-트리를 구현하는 방법을 기술하였다.

## 1. 서 론

RAM 가격의 하락으로 인하여 주기억 장치의 용량이 증가함에 따라 대부분의 테이블과 인덱스들이 주기억 장치에 상주할 수 있게 되었다[1]. 그러나 CPU와 주기억장치 사이에 존재하는 캐쉬 메모리의 활용 여부가 성능에 직접적인 영향을 미치므로, 주기억 장치 색인을 구현할 때 캐쉬 활용을 고려하는 것이 필요하다[2,3,4]. 대표적인 캐쉬 인식 주기억 장치 색인으로는 CSB$^+$-트리, pkB-트리, CR-트리 등이 있다.

그러나 기 개발된 디스크 기반의 색인들을 하나씩 수작업으로 구현하는 것은 많은 비용이 들고 에러가 나기 쉬우므로, 다른 접근 방법이 필요하다. 이와 유사한 맥락에서 디스크 기반의 색인을 쉽게 개발하기 위한 프레임워크인 디스크 기반 일반화된 검색 트리(Generalized Search Tree: GiST)가 개발되어 사용되고 있다[5]. 즉, 디스크 기반의 색인들을 하나씩 수작업으로 만드는 것이 아니라 GiST를 이용하여 쉽게 새로운 색인을 구현할 수 있게 되었다. 그러나, GiST는 디스크 기반으로 구현되었기 때문에 주기억장치 색인으로 그대로 사용하면 캐쉬를 제대로 활용할 수 없는 문제점을 가진다.

본 논문에서는 디스크 기반의 GiST를 캐쉬 인식하도록 확장한 CC-GiST를 제안한다. 본 논문의 공헌은 아래와 같이 요약된다.

- 기존의 캐쉬 인식 트리들의 기법을 분류하고 분석함으로써, 캐쉬 인식 트리에 적용할 수 있는 일반적인 방법을 도출하였다. 세부적으로는 포인터 압축과 키 압축의 개념을 일반화하였다.
- CC-GiST의 개념을 정형적으로 제안하였다. 세부적으로는 일반적인 포인터 압축과 키 압축을 디스크 기반의 GiST에 적용하였다.
- 포인터 압축을 위해 세그먼트의 개념을, 키 압축을 위하여 베이스 키의 개념을 CC-GiST에 도입하였다.
- 디스크 기반의 GiST를 위해 기정의된 메소드들을 캐쉬 인식하도록 완전하게 수정하였다.
- 제안한 CC-GiST를 이용하여 기존의 대표적인 캐쉬 인식 트리인 CSB$^+$-트리와 CR-트리를 구현하는 방법을 기술하였다.

이후 본 논문의 구성은 다음과 같다. 제 2절에서는 기존에 연구되었던 캐쉬 인식 트리와 디스크 기반의 GiST에 대해 설명한다. 제 3절에서는 일반화된 포인터 압축과 키 압축을 적용하여 디스크 기반의 GiST가 캐쉬 인식하도록 확장한 캐쉬 인식 GiST를 제안한다. 제 4절에서는 CC-GiST를 이용하여 CSB$^+$-트리와 CR-트리의 구현 방법을 설명한다.

## 2. 관련 연구

기존에 연구되었던 캐쉬 인식 트리들의 기법을 분류하고 분석하여 키 압축, 포인터 압축의 개념을 일반화 한다. 그리고 디스크 기반의 GiST에 대해 간략하게 설명한다.

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

### 2.1 캐쉬 인식 트리(Cache Conscious Trees)

캐쉬를 활용하는 캐쉬 인식 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 트리는 엔트리의 포인터를 제거하여 블록킹 인수를 증가시킴으로써 캐쉬 미시를 감소시키는 트리로서, 대표적인 예로는 CSB$^+$-트리[2]와 세그먼트 CSB$^+$-트리[2] 등이 있다. 키 압축에 기반한 캐쉬 인식 트리는 노드내의 키를 압축하여 블록킹 인수를 증가시킴으로써 캐쉬 미스를 감소 시키는 트리로서, 대표적인 예로 pkB-트리[3], CR-트리[4] 등이 있다.

그림1은 CSB$^+$-트리의 예를 나타내고 있다. 노드가 가리키고 있는 모든 자식 노드들을 실선 사각형 모양의 연속된 공간, 즉 노드 그룹(*node group*)에 저장하고 노드는 노드 그룹에 저장된 첫번째 자식 노드를 가리키는 실선 화살표 모양의 포인터만을 저장한다. 이렇게 함으로써 각 노드에 저장이 필요한 포인터 갯수를 줄이고 블록킹 인수를 증가시켜 검색 성능을 향상 시킨다.

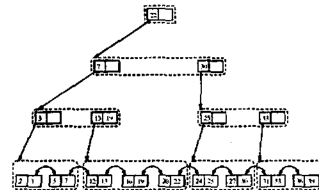

그림 1. CSB$^+$-트리의 예.

그러나 변경 비용이 증가 되는 문제점을 가진다. 즉, CSB$^+$-트리의 노드에서 분할(split)이 발생했을 때, 분할 전 노드 그룹보다 하나의 노드가 더 많은 새로운 노드 그룹을 생성하고 분할 전 노드 그룹에 있는 모든 노드들을 복사해야 하기 때문에 분할 비용이 크다. 이러한 문제를 해결하기 위해서 세그먼트 CSB$^+$-트리는 연속적으로 저장되어 있는 노드의 시퀀스인 세그먼트에 자식 노드들을 나누어서 저장한다. 그리고 부모 노드는 세그먼트 내의 첫번째 자식 노드를 가리키는 포인터를 저장함으로써 분할 비용을 줄인다. 즉, 하나의 노드 그룹을 여러 개의 세그먼트로 확장함으로써 분할시 복사 비용을 줄인다. 세그먼트 개념을 노드 그룹으로 확장할 수 있다. 그림1의 CSB$^+$-트리의 노드 그룹을 1개의 세그먼트라고 생각할 수 있다.

키 압축에 기반한 pkB-트리는 연속된 키 사이에서 차이가 나는 부분 중 고정길이 일부만을 저장함으로써 키를 압축한다[3]. 따라서 길이가 큰 가변길이 키 대신에 길이가 작은 고정길이 키를 저장함으로써 캐쉬 미스의 발생 횟수를 줄일 수 있다.

그림2는 pkB-트리의 예를 나타내고 있다. 점선 화살표는 자식 노드를 가리키는 포인터이고 실선화살표는 베이스 키를 나타내고 있다. 여기서 베이스 키와 압축하고자 하는 키 사이에서 차이가 나는 부분 중 고정 길이 일부만을 저장해서 키를 압축한다.

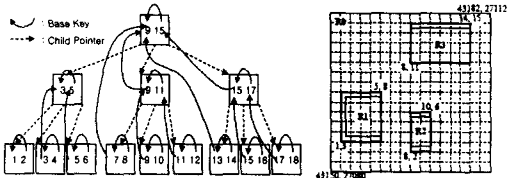

그림 2. pkB-트리의 예.

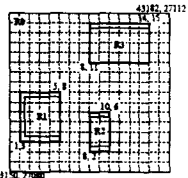

그림 3. CR-트리의 QRMBR 예.

또 다른 키 압축에 기반한 CR-트리는 키의 MBR 좌표값을 부모 노드의 키의 MBR 좌표에 대하여 상대좌표화와 양자화한 MBR, 즉 QRMBR(*quantized relative representation of MBR*)을 저장함으로써 키를 압축한다[4]. 따라서 키 길이가 큰 절대 좌표 MBR 대신 길이가 작은 상대좌표 MBR을 저장함으로서 캐쉬 미스 발생 횟수를 줄일 수 있다.

그림3은 CR-트리의 QRMBR의 예를 나타낸다. 가장 바깥쪽 사각형은 부모 노드의 MBR인 R0를 나타내고, 사각형 R1, R2, R3는 자식 노드의 QRMBR를 나타낸다. R1은 R0을 기준으로 상대좌표로 표현된다. 그리고 그 상대좌표를 16등분하여 양자화 시킴으로써 QRMBR을 표현하는 저장공간을 줄일 수 있다. R2와 R3도 이러한 과정으로 표현한다.

### 2.2 일반화된 검색 트리 [5]

**일반화된 검색 트리**(*Generalized Search Tree: GiST*)는 모든 디스크 기반 균형 검색 트리를 만들 수 있는 프레임워크이다. 균형 검색 트리의 공통적인 알고리즘인 search, insert, delete등은 트리 메소드로서 GiST가 제공해 준다. 각 균형 검색 트리마다 다른 특징들은 키 메소드로서 사용자가 구현해 주어야 한다. 그러므로 사용자는 적은 비용으로 모든 디스크 기반 균형 검색 트리를 만들 수 있다. 그러나, GiST는 디스크 기반으로 구현되었기 때문에 주기억장치 색인으로 그대로 사용하면 캐쉬를 제대로 활용할 수 없는 문제점을 가진다.

## 3. 캐쉬 인식하는 일반화된 검색 트리 (Cache Conscious Generalized Search Trees)

기존 캐쉬 인식 트리들을 분석한 결과 다음과 같이 포인터 압축과 키 압축의 개념을 정형적으로 정의할 수 있다.

**정의 1.** 캐쉬 인식 트리에서 **포인터 압축**(*pointer compression*)이란 각 노드에 있는 n개 포인터 $P_i(1\le i\le n)$중에서 k개$(k\le m)$를 제거하고 각 연속해서 제거된 포인터들 $P_m, P_{m+1}, \cdots, P_l(m\le l<n)$에 의해 가리켜지고 있는 자식 노드들을 포인터 $P_{m-1}$에 의해 가리켜지고 있는 노드와 함께 세그먼트에 저장하는 방법이다.

**정의 2.** 캐쉬 인식 트리에서 **키 압축**(*key compression*)이란 베이스 키와 키간의 관계를 이용하여 키 길이를 줄이는 방법이다.

포인트 압축 개념과 키 압축 개념을 디스크 기반의 GiST에 적용한다면 적은 비용으로 모든 캐쉬 인식 트리를 만들 수 있을 것이다. **캐쉬 인식하는 일반화된 검색 트리**(*Cache Conscious Generalized Search Trees: CC-GiST*)는 디스크 기반의 GiST에 포인트 압축, 키 압축 개념을 적용하여 기존의 모든 균형 캐쉬 인식 트리들을 쉽게 만들 수 있는 프레임워크이다.

본 절에서는 CC-GiST의 구조와 프로퍼티, 포인터 메소드, 키 메소드, 트리 메소드를 설명한다. 본격적인 설명에 앞서 4절에서 사용하는 파라메타에 대한 설명은 표1과 같다.

표 1. 파라메타 설명.

| 파라메타 | 설명 |
|---|---|
| nodePointer | 노드를 가리키는 포인터 |
| segmentPointer | 세그먼트를 가리키는 포인터 |
| childNodeIndex | 노드안에서 엔트리가 가리키는 자식 노드의 인덱스 |
| R | 루트 노드 |
| N | 노드 |
| L | 단말 노드 |
| E | 엔트리 |
| P | 엔트리들의 집합 |
| q | 쿼리 |
| l | 노드의 레벨 |
| baseKey | 베이스 키 |
| key | 키 |
| keyIndex | 노드안에서 엔트리의 키에 대한 인덱스 |

### 3.1 구조

CC-GiST에서 포인트 압축 개념을 적용하기 위해 비단말, 단말 노드들을 세그먼트에 저장하고 비단말노드는 자식 노드가 저장된 세그먼트들을 가리키는 포인터들을 가진다. **세그먼트**(*segment*)는 연속적으로 저장되어 있는 노드들의 시퀀스이다. 1개의 노드가 가지는 세그먼트를 가리키는 포인트의 수를 노드의 세그먼트 차수라고 하자. 그리고 트리의 세그먼트 차수는 트리에 있는 노드의 최대 세그먼트 차수라고 하자.

CC-GiST에서 키 압축 개념을 적용하기 위해 StoreBaseKeyInNode라는 옵션에 true값을 주면 CR-트리처럼 노드에 베이스 키를 저장할 수 있는 노드 구조를 제공한다. 그리고 CC-GiST는 키를 압축, 복원할때 사용하는 조상 노드의 베이스 키를 저장하기 위해 AnscestorKeyStack이라는 객체를 제공한다. AnscesotKeyStack은 검색경로상의 조상 노드들의 키들을 저장할 때 사용된다.

CC-GiST는 고정 길이 세그먼트, 가변 길이 세그먼트를 지원한다. 고정 길이 세그먼트는 세그먼트의 크기가 고정되어 있는 반면에 가변 길이 세그먼트는 세그먼트의 크기가 가변적이다. FixedSizeSegment 옵션에 true값을 주면 CC-GiST가 고정 길이 세그먼트를 지원해 준다.

노드를 가리키는 nodePointer 자료형은 다음과 같이 정의할 수 있다. 노드가 세그먼트에 저장되므로 세그먼트를 가리키는 포인터 segmentPointer와 세그먼트 안에서 노드의 인덱스nodeIndexInSegment로 구성된다.

nodePointer = ( segmentPointer, nodeIndexInSegment)

트리의 차수가 M이고 트리의 세그먼트 차수가 S인 CC-GiST의 정의는 기존 GiST의 정의에 아래의 항목을 변경하고 추가함으로써 정의된다.

1. 비단말 노드는 검색 키로 사용되는 프레디킷 p의 리스트와 자식 노드가 저장된 세그먼트를 가리키는 S개의 포인터의 리스트와 S개의 세그먼트에 저장된 노드 갯수의 리스트를 가진다.
2. StoreBaseKeyInNode라는 옵션이 true이면 단말, 비단말 노드는 베이스 키를 저장할 수 있는 저장공간을 가진다.
3. AnscesotKeyStack은 검색경로상의 조상 노드들의 키들을 저장하는 스택이다.
4. FixedSizeSegment라는 옵션이 true이면 CC-GiST가 고정 길이 세그먼트를 사용한다. 그렇지 않으면 CC-GiST가 가변길이 세그 먼트을 지원해 준다.

### 3.2 프로퍼티(properties)

변경된 프로퍼티는 다음과 같다.

- 비단말 노드의 각 인덱스 엔트리에 대해서, 엔트리의 프레디킷 p가 엔트리가 가리키는 서브 트리의 모든 튜플의 키값을 만족할 때 p는 true 이다.

### 3.3 포인터 메소드

**포인터 메소드**(*Pointer methods*)는 포인트 압축 개념을 표현한 함수들의 집합이다. 그리고 각 캐쉬 인식 트리마다 포인트 압축하는 방식이 다르므로 사용자가 포인터 메소드를 구현해 주어야 한다.

- **PointerCompress**(N, nodePointer, childNodeIndex): N에 childNodeIndex번째 자식 노드가 저장된 세그먼트 포인터 nodePointer.segmentPointer를 저장한다.
- **PointerDecompress**(N, childNodeIndex): N의 childNodeIndex번째 자식 노드를 가리키는 포인터를 리턴한다.

### 3.4 키 메소드

**키 메소드**(*key methods*)의 특징을 반영해 주는 함수로서 사용자에 의해서 구현되어야 한다. Compress, Decompress, GetBaseKey 함수는 키 압축 개념을 표현한 함수이다

- **GetConsistentEntries**(N, q): N에서 q를 만족시키는 엔트리들의 리스트를 리턴한다.
- **GetMinPenaltyEntry**(N, E): N의 각 엔트리들이 가리키는 서브 트리에 E을 삽입했을때 최소 삽입 비용을 발생시키는 엔트리를 리턴한다.
- **Union**(P): P = $\{(p_1, ptr_1), \cdots, (p_n, ptr_n)\}$이면 $ptr_1, \cdots, ptr_n$가 가리 키는 모든 튜플의 키값을 만족시켜주는 프레디킷 r을 생성해 리턴한다. 즉 $(p_1 \vee \cdots \vee p_n) \rightarrow r$을 리턴한다.
- **PickSplit**(P): 입력 파라메타로서 엔트리 집합 P가 주어지면 P를 두개의 집합 P1, P2으로 나누어서 리턴한다.
- **Compress**(baseKey, key): baseKey와 key를 이용하여 key를 압축하여 리턴한다.
- **Decompress**(baseKey, key): baseKey와 key를 이용하여서 압축된 키를 복원해서 리턴한다.
- **GetBaseKey**(N, keyIndex): N에 저장된 베이스 키, N의 keyIndex-1번째 키, AnscesorKeyStack에 저장된 키 중 1개를 선 택해서 리턴한다.

### 3.5 트리 메소드

모든 균형 검색 트리는 공통된 search, insert, delete 등의 알고리즘을 사용한다. **트리 메소드**(*tree methods*)는 모든 균형 검색 트리의 공통된 search, insert, delete 등의 알고리즘을 표현한 메소드로서 CC-GiST가 제공해 주는 메소드이다. CC-GiST의 트리 메소드는 GiST에서 기 정의된 메소드들의 알고리즘을 캐쉬 인식하도록 수정하였다. 자세한 트리 메소드의 알고리즘을 아래와 같다.

- **Search**(R, q): q를 만족하는 모든 튜플을 리턴한다. *GetConsistentEntries*(R, q)을 호출하여 쿼리를 만족시키는 엔트리 집합을 얻는다. 엔트리 집합의 각 엔트리에 대해서 다음과 같이 수행한다. 만약 R이 비단말 노드이면 PointerDecompress(R, childNodeIndex)를 수행하여 자식 노드 CN를 가리키는 포인터를 얻은후 자식 노드에 대해서 재귀적으로 search(CN, q)를 수행한다. 만약 R이 단말 노드이면 childNodePointer가 가리키는 튜플을 읽어 리턴한다.
- **Insert**(R, E, I): 레벨 I인 노드에 엔트리 E를 삽입한다. ChooseSubTree(R, E, I)를 호출하여 E가 삽입될 노드 N를 찾는다. 노드에서 엔트리 E가 삽입될 위치 keyIndex를 찾는다. baseKey = GetBaseKey(R, keyIndex), Compress(baseKey, E.p)을 호출하여 E의 키를 압축하여 N에 저장한다. 만약 N이 비단말 노드라면 PointerCompress(N, E.ptr, childNodeIndex)를 호출하여서 자식 노드를 가리키는 포인터를 포인터압축한다. 만약 N이 단말 노드라면 삽입한 엔트리의 튜플 ID을 E.ptr에 저장한다. 만약 노드에 키값을 저장할 공간이 없으면 Split(R, N, E)를 호출한다. 삽입으로 인한 변화를 부모 노드로 전파하기 위해 AdjustKey(R, N)를 호출한다.
- **ChooseSubTree**(R, E, I): E을 삽입하기 적합한 레벨이 I인 노드를 찾는다. 만약 R의 레벨이 I과 같으면 R를 리턴한다. GetMinPenalyEntry(N, E)를 호출하여서 삽입 비용인 최소인 엔트리가 가리키는 자식 노드 CN를 얻어서 ChooseSubTree(CN, E, I)를 재귀적으로 호출한다.
- **Split**(R, N, E): PickSplit( N ∪ {E} )의 리턴값에 따라서 N의 엔트리들과 엔트리 E를 두개의 집합 LS, RS으로 나눈다. LS을 기존 노드 N에 저장하고 RS을 새로운 노드 N′에 저장한다. 그리고 부모 노드에 N′을 가리키는 엔트리를 삽입한다.
- **AdjustKey**(R, N): 부모 노드로 올라가면서 부모 노드의 키의 프레디킷이 자식 노드의 모든 키의 프레디킷을 만족시키도록 부모 노드의 프레디킷을 수정한다.
- **Delete**(R, E): 단말노드에서 E를 삭제한다. E가 저장된 단말노드를 찾은 뒤 E를 삭제한다. 삭제로 인한 변화를 부모 노드로 전파시키기 위해 CondenseTree(R, L)를 호출한다.
- ***CondenseTree***(R, L): 만약 L에 언더플로우(underflow)가 발생하면 L를 삭제하고 L의 모든 엔트리들을 트리에 재삽입시키거나 형제 노드에 재분배 또는 합병 시킨다. 그리고 L의 삭제로 인한 변화를 부모 노드로 전파시킨다. 엔트리를 재삽입, 재분배, 합병할때 압축된 키를 복원해서 다른 노드에 그 키를 압축해서 재삽입, 재분배, 합병해야 한다. 재삽입, 재분배, 합병을 하기 전에 baseKey = GetBaseKey( L, keyIndex), key = Decompress ( baseKey, E.p)를 호출하여 키를 복원한다. 그리고 재삽입, 재분배, 합병할때 baseKey′ = GetBaseKey( L, keyIndex′), Compress( baseKey′, key)을 호출하여 압축된 키를 저장한다.

## 4. 캐쉬 인식 트리의 구현에 대한 CC-GiST

본 장에서는 CC-GiST에서 $CSB^+$-트리, CR-트리를 구현하기위해 사용자가 포인터 메소드, 키 메소드, 옵션을 구현해 주는 것을 설명한다.

### 4.1 $CSB^+$-트리의 구현을 위한 CC-GiST

$CSB^+$-트리에서 키는 정수의 쌍이다. 키 a는 프레디킷 Contains( [a, b], v )으로 표현 할 수 있다. b는 오른쪽 편 엔트리의 키 이고 v는 free variable이다. 만약 a의 오른쪽 편 엔트리가 없으면 b는 ∞ 이다.

$CSB^+$-트리에서 지원해 주는 쿼리 프레디킷은 아래와 같다.

- **Contains**( [x, y], v): 만약 $x \le v < y$이면 true를 리턴한다. 그렇지 않으면 false를 리턴한다.
- **Equal**(x, v): 만약 x = v이면 true를 리턴한다. 그렇지 않으면 false를 리턴한다.

포인터 메소드의 구현은 아래와 같다.

- **PointerCompress**(N, nodePointer, childNodeIndex): childNodeIndex가 0이면 *Nodepointer.sementpointer*를 노드의 첫번째 세그먼트 포인터 값에 저장하고 true를 리턴한다. 그렇지 않으면 false를 리턴한다.
- ***PointerDecompress***(N, childNodeIndex): nodePointer = (N의 첫번째 세그먼트를 가리키는 포인터 값, childNodeIndex)을 리턴한다.

키 메소드의 구현은 아래와 같다.

- **GetConsistentEntries**(N, q): 만약 q= Equal($x_q$, v)이면 $p \le x_q$을 만족하는 가장 오른쪽 엔트리(right most entry)를 결과 집합에 추가한다. 만약 q=Contains( [$x_q$, $y_q$), v )이면 $x_q<p<y_q$을 만족하는 entry들을 결과 집합에 추가한다.
- **GetMinPenaltyEntry**(N, *E*): *E.p*보다 같거나 작은 가장 오른쪽 엔트리의 key를 가진 entry를 리턴한다.
- ***Union***( $E_1$=( $x_1$, $ptr_1$), ⋯ , $E_n$=( $x_n$, $ptr_n$) ): *min*($x_1$, ⋯, $x_n$) 값을 리턴한다.
- **PickSplit**(P): P 엔트리들을 반을 왼쪽 노드 집합에 나머지 반을 오른쪽 노드 집합에 배분한다.
- **Compress**(baseKey, key): key를 리턴한다.
- **Decompress**(baseKey, key): key를 리턴한다.
- **GetBaseKey** (N, keyIndex): null를 리턴한다.

$CSB^+$-트리를 위한 옵션은 아래와 같다.

- FixedSizeSegment = false
- StoreBaseKeyInNode = false
- AnscetorKeyStack 객체

### 4.2 CR-트리 구현을 위한 CC-GiST

CR-트리에서 키는 4개의 실수이다. 키 ($x_{ul}$, $y_{ul}$, $x_{lr}$, $y_{lr}$)는 Contains( ($x_{ul}$, $y_{ul}$, $x_{lr}$, $y_{lr}$), v) 프레디킷을 나타낸다. ($x_{ul}$, $y_{ul}$)는 MBR의 좌상점을 ($x_{lr}$, $y_{lr}$)은 MBR의 우하점을 나타낸다.

CR-트리가 지원해 주는 쿼리 프레디킷은 아래와 같다.

- ***Contains***( ($x^1_{ul}$, $y^1_{ul}$, $x^1_{lr}$, $y^1_{lr}$), ($x^2_{ul}$, $y^2_{ul}$, $x^2_{lr}$, $y^2_{lr}$) ) 만약 $(x^1_{lr} \ge x^2_{lr}) \wedge (x^1_{ul} \le x^2_{ul}) \wedge (y^1_{lr} \le y^2_{lr}) \wedge (y^1_{ul} \ge y^2_{ul})$ 이면 true 리턴한다. 그렇지 않으면 false를 리턴한다.
- **Overlap**( ($x^1_{ul}$, $y^1_{ul}$, $x^1_{lr}$, $y^1_{lr}$), ($x^2_{ul}$, $y^2_{ul}$, $x^2_{lr}$, $y^2_{lr}$) ) 만약 $(x^1_{ul} \le x^2_{lr}) \wedge (x^2_{ul} \le x^1_{lr}) \wedge (y^1_{lr} \le y^2_{ul}) \wedge ((y^2_{lr} \le y^1_{ul})$ 이면 true 리턴한다. 그렇지 않으면 false를 리턴한다.
- **Equal**( ($x^1_{ul}$, $y^1_{ul}$, $x^1_{lr}$, $y^1_{lr}$), ($x^2_{ul}$, $y^2_{ul}$, $x^2_{lr}$, $y^2_{lr}$) ) 만약 $(x^1_{ul} = x^2_{ul}) \wedge (y^1_{ul} = y^2_{ul}) \wedge (x^1_{lr} = x^2_{lr}) \wedge ((y^1_{lr} = y^2_{lr})$ 이면 true 리턴한다. 그렇지 않으면 false를 리턴한다.

포인터 메소드의 구현은 아래와 같다.

- **PointerCompress**(N, nodePointer, childNodeIndex): N의 childNodeIndex번째 세그먼트 포인터 값에 *nodePointer.segmentPointer*를 저장한다.
- ***PointerDecompress***(N, *childNodeIndex*): *nodePointer* = (N의 childNodeIndex번째 세그먼트 포인터값, 0)를 리턴한다.

키 메소드의 구현은 아래와 같다.

- **GetConsistentEntries**(N, q): GetBaseKey, Compress을 통해서 q의 MBR을 압축시킨다. 압축된 q의 MBR과 N의 모든 엔트리의 MBR간의 overlap를 체크해서 overlap하는 엔트리들의 리스트를 리턴한다.
- **GetMinPenaltyEntry**(N, E): GetBaseKey, Compress을 통해서 E의 MBR을 압축시킨다. 압축된 E의 MBR을 N의 모든 엔트리가 가르키는 서브트리에 삽입할 때 넓이 확장이 최소인 엔트리를 리턴한다.
- **Union**( $E_1$=($x^1_{ul}$, $y^1_{ul}$, $x^1_{lr}$, $y^1_{lr}$), ⋯ , $E_n$=($x^n_{ul}$, $y^n_{ul}$, $x^n_{lr}$, $y^n_{lr}$) ) ( MIN($x^1_{ul}$, ⋯ , $x^n_{ul}$), MAX($y^1_{ul}$, ⋯ , $y^n_{ul}$), MAX($x^1_{lr}$, ⋯ , $x^n_{lr}$), MIN($y^1_{lr}$, ⋯ , $y^n_{lr}$) )인 QRMBR을 리턴한다.
- **PickSplit**(P): R-트리의 linear-cost split 알고리즘을 사용한다.
- **Compress**(*baseKey, key*): *baseKey*와 *Key*을 이용해서 CR-트리에서 사용한 상대좌표와 양자화를 이용한 QRMBR을 생성하여 리턴한다.
- **Decompress**(baseKey, key): key의 QRMBR을 역양자화하고 baseKey의 MBR의 절대좌표를 더해서 절대좌표를 사용하는 MBR을 리턴한다.
- **GetBaseKey** (N, keyIndex): AncestorKeyStack의 top의 원소를 리턴한다.

CR-트리를 위한 옵션은 아래와 같다.

- FixedSizeSegment = false
- StoreBaseKeyInNode = true
- AnscetorKeyStack 객체

## 5. 결론

본 논문에서는 기존의 디스크 기반의 GiST을 캐쉬 인식하도록 확장한 CC-GiST를 제안하였다. 먼저 기존의 캐쉬 인식 트리에 적용되었던 기법들을 분류하고 분석하여 포인터 압축과 키 압축의 개념을 도출하였다. 추가적으로 포인터 압축을 위해 세그먼트 개념을, 키 압축을 위해 베이스 키 개념을 도출하였다. 포인터 압축과 키 압축의 개념을 기존의 디스크 기반의 GiST에 적용하여 캐쉬 인식하도록 확장한 CC-GiST의 개념들을 정형적인 제안하였다.

또한 디스크 기반의 GiST를 위해 기 정의된 메소드들을 캐쉬 인식하도록 알고리즘을 완전하게 수정하였고 CC-GiST를 이용하여 기존의 대표적인 캐쉬 인식 트리인 $CSB^+$-트리와 CR-트리의 구현방법도 기술하였다.

## 참고 문헌


[1] Phil Bernstein, et al.," The *Asilomar report on database research,*" *SIGMOD Record*, Volume 27, Number 4, pp. 74-80, 1998.

[2] J. Rao, K. A. Ross, " Making $B^+$-trees Cache Conscious in Main Memory," In *Proc. Int' l Conf. on management of Data, ACM SIGMOD*, Volume 29, Issue 2, pp. 475-486, 2000.

[3] P. Bohannon, P. Mcilroy, R. Rastogi, " Main-Memory Index Structures with Fixed-Size Partial Keys," In *Proc. Int' l Conf. on management of Data, of ACM SIGMOD*, Volume 30, Issue 2, pp. 163  174, 2001.

[4] Kihong Kim, Sang K. Cha, Keunjoo Kwon, " Optimizing Multidimensional Index Trees for Main Memory Access," In *Proc. Int' l Conf. on management of Data, of ACM SIGMOD*, Volume 30, Issue 2, pp. 139 - 150, 2001.

[5] Joseph M. Hellerstein, Jeffrey F. Naughton, Avi Pfeffer, " Generalized Search Trees for Database Systems," In *Proc. of VLDB Conference* VLDB, 562-573, 1995.