Prefetch R-tree: 디스크와 CPU 캐시에 최적화된 다차원 색인 구조

박 명 선†

요 약

R-tree는 일반적으로 트리 노드의 크기를 디스크 페이지의 크기와 같게 함으로써 I/O 성능이 최적화 되도록 구현한다. 최근에는 주메모리 환경에서 CPU 캐시 성능을 최적화하는 R-tree의 변형이 개발되었다. 이는 노드의 크기를 캐시 라인 크기의 수 배로 하고 MBR에 저장되는 키를 압축하여 노드 하나에 더 많은 엔트리를 저장함으로써 성능을 높였다. 그러나, 디스크 최적 R-tree와 캐시 최적 R-tree의 노드 크기 사이에는 수십-수백 바이트와 수-수십 킬로바이트라는 큰 차이가 있으므로, I/O 최적 R-tree는 캐시 성능이 나쁘고 캐시 최적 R-tree는 디스크 I/O 성능이 나쁜 문제점을 가지고 있다. 이 논문에서는 CPU 캐시와 디스크 I/O에 모두 최적인 R-tree, PR-tree를 제안한다. 캐시 성능을 위해 PR-tree 노드의 크기를 캐시 라인 크기보다 크게 만든 다음 CPU의 선반입(prefetch) 명령어를 이용하여 캐시 실패 횟수를 줄이고, 트리 노드를 디스크 페이지에 낭비가 적도록 배치함으로써 디스크 I/O 성능도 향상시킨다. 또한, 이 논문에서는 PR-tree에서 검색 연산을 수행하는데 드는 캐시 실패 비용을 계산하는 분석 방법을 제시하고, 최적의 캐시와 I/O 성능을 보이는 PR-tree를 구성하기 위해, 가능한 크기의 내부 단말 노드, 중간 노드를 갖는 PR-tree 생성하여 성능을 비교하였다. PR-tree는 디스크 최적 R-tree보다 삽입 연산은 3.5에서 15.1배, 삭제 연산은 6.5에서 15.1배, 범위 질의는 1.3에서 1.9배, k-최근접 질의는 2.7에서 9.7배의 캐시 성능 향상이 있었다. 모든 실험에서 매우 작은 I/O 성능 저하만을 보였다.

키워드 : 최적화, 캐시 성능, I/O 성능, R-tree, 선반입

Prefetch R-tree: A Disk and Cache Optimized Multidimensional Index Structure

Myungsun Park†

ABSTRACT

R-trees have been traditionally optimized for the I/O performance with the disk page as the tree node. Recently, researchers have proposed cache-conscious variations of R-trees optimized for the CPU cache performance in main memory environments, where the node size is several cache lines wide and more entries are packed in a node by compressing MBR keys. However, because there is a big difference between the node sizes of two types of R-trees, disk-optimized R-trees show poor cache performance while cache-optimized R-trees exhibit poor disk performance. In this paper, we propose a cache and disk optimized R-tree, called the PR-tree (Prefetching R-tree). For the cache performance, the node size of the PR-tree is wider than a cache line, and the prefetch instruction is used to reduce the number of cache misses. For the I/O performance, the nodes of the PR-tree are fitted into one disk page. We represent the detailed analysis of cache misses for range queries, and enumerate all the reasonable in-page leaf and nonleaf node sizes, and heights of in-page trees to figure out tree parameters for best cache and I/O performance. The PR-tree that we propose achieves better cache performance than the disk-optimized R-tree: a factor of 3.5-15.1 improvement for one-by-one insertions, 6.5-15.1 improvement for deletions, 1.3-1.9 improvement for range queries, and 2.7-9.7 improvement for k-nearest neighbor queries. All experimental results do not show notable declines of the I/O performance.

Key Words : Optimize, Cache Performance, I/O Performance, R-tree, Prefetch

1. 서 론

프로세서의 속도 증가와 DRAM의 처리 속도 증가의 차이가 점점 커짐에 따라 데이타베이스 시스템의 성능 향상을 위해 CPU 캐시의 효율적인 이용이 새로운 문제로 떠오르고 있다[1]. 최근에 기존의 인덱스 구조를 캐시에 효율적으로 개선한 연구 결과가 발표되고 있다. B-tree에서는 한 노드의 크기를 캐시 라인의 크기와 같게 하여 캐시 실패(miss) 횟수를 줄이는 연구가 있다[2, 3]. 또한, B-tree의 노드 크기

※ 본 연구는 두뇌한국21 사업과 정보통신부의 대학 IT연구센터(ITRC) 지원을 받아 수행되었습니다.
† 정 회 원 : KT BcN본부 제어플랫폼담당 유무선통합인증개발부 전임연구원
논문접수 : 2006년 3월 15일, 심사완료 : 2006년 6월 8일

를 캐시 라인의 정수 배로 만들고 최신 CPU가 지원하는 캐시 선반입(prefetch) 명령을 이용하여 트리의 높이를 낮추고 범위 스캔 연산의 성능을 향상시킨 연구가 있다[4]. 여기에 노드를 디스크 페이지에 적절히 배치하여 디스크 I/O 성능도 향상시킨 연구가 있다[5]. R-tree에서는 MBR에 저장되는 키를 압축함으로써 한 노드에 더 많은 엔트리를 저장하여 캐시 성능을 높이는 연구가 있다[6].

디스크에 최적화된 R-tree는 캐시 성능이 나쁘고, 캐시에 최적화된 R-tree는 디스크 I/O 성능이 나쁘다는 문제점이 있다. 이는 두 가지 R-tree의 노드 크기가 크게 다르기 때문이다. 디스크 페이지는 보통 4KB에서 32KB이고, 캐시 라인의 크기는 시스템에 따라 다르지만 32B에서 128B이다. 디스크 최적 R-tree는 노드 크기가 페이지 크기와 같고 캐시 최적 R-tree의 노드 크기는 캐시 라인의 수 배 정도이다.

이 논문은 캐시와 디스크 모두에 효율적인 R-tree, PR-tree (Prefetching R-tree)를 제안한다. PR-tree는 캐시 성능을 향상시키기 위해 노드의 크기를 캐시 라인 크기보다 크게 만들고 최근의 프로세서에서 지원하는 캐시 선반입 명령어를 이용하여 메모리에 있는 노드 데이타를 CPU 캐시에 미리 읽어 들임으로써 캐시 실패 비용을 최적화한다. 그리고 노드를 디스크 페이지에 공간의 낭비 없이 배치함으로써 디스크 입출력 성능도 향상시킨다. 또한, 이 논문은 범위 질의를 수행하는데 발생하는 캐시 실패 비용을 수식으로 표현하여 내부 중간 노드와 단말 노드의 크기, 내부 트리의 높이를 정하는 방법을 제시하고, 가능한 페이지 크기와 노드 크기에 대해 그 수식의 값을 최소화함으로써 최적의 PR-tree를 구성하는 방법을 보인다. PR-tree는 디스크 최적 R-tree보다 삽입 연산은 3.5에서 15.1배, 삭제 연산은 6.5에서 15.1배, 범위 질의는 1.3에서 1.9배, k-최근접 질의는 2.7에서 9.7배의 성능 향상이 있었다. PR-tree는 모든 실험에서 매우 작은 I/O 성능 저하만을 보였다.

본 논문의 의의는 다음과 같다. 첫째, 본 논문은 캐시와 디스크에 모두 효율적인 PR-tree를 제안하였다. R-tree를 개선한 새로운 인덱스인 PR-tree를 제시하고, 대량적재, 삽입, 삭제, 범위 질의, k-최근접 질의 등을 효율적으로 처리하는 새로운 알고리즘을 상세한 예와 함께 보였다. 대량적재와 재구성 알고리즘은 PR-tree의 페이지와 그 안의 노드라는 이중 구조에 맞게 페이지 단위로 객체를 나눈 후, 각 페이지 단위로 다시 엔트리를 정렬하여 노드와 페이지를 구성함으로써 PR-tree의 페이지와 노드 공간을 최적으로 이용하도록 한다. 삽입 알고리즘은 삽입에 의해 발생하는 트리 구조의 변화에 대해 페이지 분할을 최소화함으로써 페이지와 노드 공간을 최적으로 이용한다. 둘째, PR-tree에 대한 범위 질의를 수행하는데 소요되는 캐시와 I/O 비용을 분석하고, 비용을 계산하는 방법과 수식을 제시하였다. 셋째, 다양한 페이지 크기와 페이지 내부 중간, 단말 노드 크기를 갖는 PR-tree를 구현하고 삽입, 삭제, 범위 질의, k-최근접 질의에 대해 실험함으로써 기존의 디스크 기반 R-tree보다 더 나은 성능을 가짐을 보였다.

본 논문은 다음과 같이 구성되어 있다. 먼저 2절에서 관련 연구에 대해 기술하고 3절에서 PR-tree의 구조와 알고리즘에 대해 설명한다. 4절에서는 PR-tree의 캐시와 I/O 성능을 분석하고, 5절에서 실험 결과를 보이며, 6절에서 결론을 맺는다.

2. 관련 연구

최근에 주메모리 인덱스의 성능을 향상시키기 위해, 인덱스 구조의 노드 크기를 조절하고 특정한 CPU 명령어를 사용하여 성능을 높이는 연구 결과가 발표되고 있다. 인덱스를 사용할 때 캐시 실패 비용을 줄이기 위해 CSS-tree와 CSB^+-tree는 노드 크기를 CPU의 캐시 라인 크기와 같게 만든다[2, 3]. 여기에서 캐시 라인은 CPU 캐시와 주 메모리 사이의 데이타 전송 단위이다. pB^+-tree는 노드 크기를 캐시 라인의 수 배로 만들고 선반입(prefetch) 명령어를 사용하여 캐시 성능을 높인다. 여기에서는 노드 크기가 커짐에 따라 B-Tree의 높이가 낮아지고 따라서 검색할 때 캐시 실패 비용이 감소하게 되어 성능을 향상시킬 수 있다. 이 연구에서는 범위 스캔의 성능을 높이기 위해 점프 포인터 배열(jump-pointer array) 구조를 이용하였다[4]. fpB^+-tree는 pB^+-tree를 디스크 I/O에 최적화한 인덱스 구조이다. fpB^+-tree는 여러 노드를 하나의 페이지에 디스크 낭비가 적도록 배치하여 디스크 성능을 높이고, 캐시 라인보다 큰 노드를 선반입 명령어로 미리 CPU 캐시에 적재하여 캐시 성능을 최적화한다[5]. 본 논문은 fpB^+-tree의 기본 구조를 R-tree에 적용하지만, R-tree의 특성에 맞도록 노드 구조를 개선하고, 캐시와 I/O 성능을 최적화하기 위한 대량적재, 삽입, 재구성 알고리즘을 제안하며, 본 논문에서 제안한 PR-tree가 범위 질의를 수행하는데 소요되는 캐시와 I/O 비용을 분석하고 최적의 트리 구성 방법을 제시한다.

CR-tree는 주 메모리 환경에서 R-tree를 캐시에 최적화한 인덱스 구조이다. 하나의 노드에 더 많은 엔트리를 넣기 위해, CR-tree는 QRMBR이라는 기법을 이용하여 키를 압축한다. QRMBR 기법은 자식 노드의 MBR이 부모 노드의 MBR에 포함되는 R-tree의 특성을 이용하여 자식 노드의 MBR을 부모 노드의 MBR에 상대적인 좌표로 바꾸고 일정한 데이타 크기를 갖도록 양자화하는 것이다. 이 연구는 노드 크기에 따라 다양한 종류의 R-tree 변형에 대해 성능을 실험하고 있다[6]. CR-tree는 주메모리 환경에서 사용할 수 있는 인덱스 구조이므로 본 논문이 가정한 환경인 디스크 기반 환경에서 사용할 수 없는 한계가 있다.

3. PR-tree

PR-tree는 디스크에 우선으로 최적화된 인덱스 구조이다. 이는 fpB^+-tree에서 디스크 우선 최적화가 일반적인 경우 더 효율적이라 하였기 때문이다[5]. 또한, 이 논문에서는

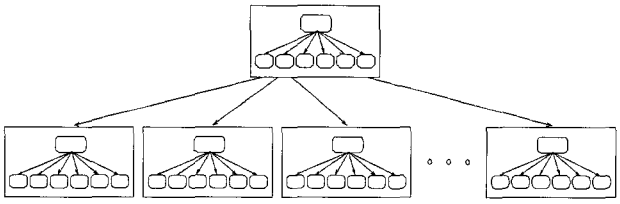
CR-tree에서 제안한 MBR 키 압축은 고려하지 않았는데, PR-tree에 MBR 압축을 쉽게 적용할 수 있기 때문이다.

3.1 PR-tree의 구조

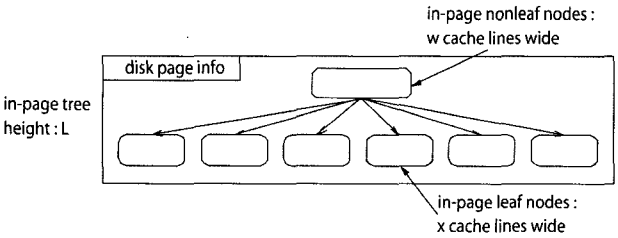
PR-tree는 R-tree와 같은 높이 균형(height-balanced) 트리이다. (그림 1)은 PR-tree의 구조를 나타내고 있다. 그림에서 큰 사각형은 페이지를 나타내고 모서리가 둥근 작은 사각형은 노드를 나타낸다. PR-tree는 그림에서 보듯이 여러 개의 페이지로 구성된 외부 트리를 가지고 있고, 각 페이지는 하나의 내부 트리를 가진다. 내부 트리에는 하나의 루트 노드와 한개 이상의 페이지 내부 중간 노드(페이지 내부 루트 노드 포함), 두개 이상의 페이지 내부 단말 노드가 존재하며 역시 높이 균형 트리를 이룬다.

각 노드는 엔트리의 리스트를 가지고 있다. 단말 노드의 엔트리는 인덱스할 객체를 가리키는 포인터와 객체의 MBR을 저장하고 있다. 중간 노드의 엔트리는 자식 노드에 대한 포인터와 자식 노드의 MBR을 가지고 있다. 중간 노드는 페이지 내부 중간 노드와 페이지 내부 단말 노드로 구분할 수 있는데, 페이지 내부 중간 노드의 엔트리는 페이지 헤더부터 상대적인 거리인 옵셋을 자식 노드의 포인터로 가지고 있고, 페이지 내부 단말 노드의 엔트리는 자식 노드의 포인터가 자식 노드가 속해 있는 페이지 ID이다. 그 밖의 트리 속성은 R-tree의 속성과 같다.

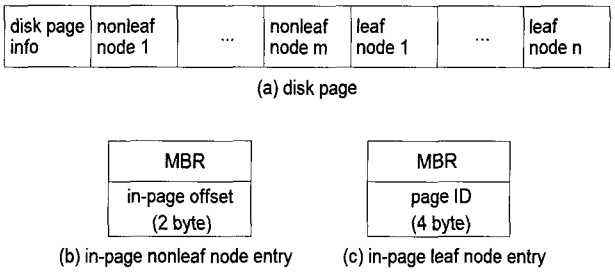
(그림 2)는 PR-tree의 페이지 구조를 보인다. 이 그림은 내부 트리의 높이가 2인 디스크 페이지를 보이고 있다. 페이지 내부 중간 노드와 단말 노드는 크기가 다른데, 이는 노드들을 디스크 페이지 하나에 공간의 낭비가 최소가 되도록 채워 넣기 위해서이다. 디스크 최적 R-tree의 노드는 크기가 너무 커서 검색 연산을 수행하는데 캐시 실패를 많이 발생시킨다. 반면에, PR-tree의 노드 크기는 디스크 최적 R-tree보다 작으므로 선반입이 효율적으로 작동하며, 따라서 캐시 실패가 줄어든다. 공간의 낭비가 전혀 없게 내부 트리를 디스크 페이지 하나에 저장할 수 없기 때문에 페이지 공간 이용률은 R-tree보다 작다. 하지만, 캐시 실패의 횟수가 줄어들기 때문에 I/O 성능에서 약간 손해를 보더라도 캐시 성능을 크게 향상 시킬 수 있다.



(그림 1) PR-tree의 구조



(그림 2) PR-tree의 페이지 구조



(그림 3) 페이지의 데이타 구조

(그림 3)은 PR-tree 페이지의 데이타 구조이다. (그림 3)(a)는 디스크 페이지에 단말과 중간 노드를 저장하는 구조를 나타낸다. 페이지의 첫 부분은 페이지 내부 루트 노드의 오프셋, 사용 중인 페이지 내부 중간 노드와 단말 노드의 수 등의 페이지 정보를 저장하고 있다. 모든 노드는 페이지 내부 중간 노드와 단말 노드 순으로 저장된다. (그림 3)(b)와 3(c)는 각각 페이지 내부 중간 노드와 단말 노드의 엔트리인데, 중간 노드 엔트리는 같은 페이지 내의 다른 노드를 가리키므로 2바이트 오프셋의 자식 포인터를 가지며, 단말 노드 엔트리는 다른 페이지 내의 루트 노드를 가리키기 때문에 4바이트의 페이지 ID를 자식 포인터로 가지고 있다.

3.2 알고리즘

PR-tree는 각 페이지에 내부 트리를 가지고 있으므로 삽입, 삭제, 검색 등의 트리 연산은 페이지와 노드의 두 단계를 갖는다. 모든 노드는 디스크 페이지에 저장되어 있다. 접근할 노드가 메모리에 없으면 그 노드가 속한 페이지를 디스크에서 읽어서 메모리 내의 버퍼 풀에 보관한다. 같은 페이지 내의 다른 노드를 요청하는 경우 디스크를 읽지 않고 페이지 오프셋을 이용하여 접근할 수 있고, 다른 페이지 내의 노드를 방문하는 경우(현재 노드가 내부 단말 노드인 경우) 방문할 노드가 메모리에 없으면 그 페이지를 읽어 온 다음 노드를 접근하며, 그렇지 않으면 바로 노드를 접근한다. 노드에 대해 트리 연산을 하기 전에 그 노드를 구성하는 모든 캐시 라인에 대해 선반입 연산을 수행한다. 다음에 대량적재(bulkload), 삽입, 삭제, 검색 등의 알고리즘을 설명한다.

3.2.1 대량적재(bulkload)

PR-tree의 대량적재는 페이지와 노드의 두 단계로 구성되어 있다. 페이지 수준에서는 모든 내부 단말 엔트리에 대해 일반적인 R-tree 대량적재 알고리즘을 수행한다. 각 페이지는 내부 중간, 단말 노드의 크기에 의해 정해지는 최대 크기만큼의 엔트리를 저장한다. 노드 수준에서는, 각 페이지로 분할된 엔트리를 동일한 대량적재 알고리즘을 이용하여 페이지 내부 트리로 구성한다. 본 논문에서 제안하는 대량

```
Algorithm Bulkload
Data : set of entries
Result: PR-tree
Q ← entries;
repeat
        sort entries in Q by x coordinate;
        partition them into vertical slices;
        Gs ← vertical slices, Q ← empty, and Gp ← empty;
        foreach vertical slice s of set Gs do
                sort entries in s by y coordinate;
                partition them into pages, and add the pages to Gp;
        end
        foreach page p of set Gp do
                Qp ← all entries in page p;
                for l ← 1 to in-page tree height do
                        if number of entries in Qp <= b then
                                generate one node for entries in Qp;
                                exit for loop;
                        end
                        sort entries in Qp by x coordinate;
                        partition them into vertical slices;
                        G's ← vertical slices, and Qp ← empty;
                        foreach vertical slice s of set G's do
                                sort entries in s by y coordinate;
                                partition them into in-page nodes, and store in-page nodes;
                                add (MBR, node-offset) of this in-page node to Qp;
                        end
                end
                add (MBR, page-ID) of the in-page root node to Q;
        end
        if Gp has only one page then exit repeat loop;
until no entries in Q;
```

(그림 4) 대량적재 알고리즘

적재 알고리즘은 STR 알고리즘 기반으로, 객체를 PR-tree에 대량적재할 때 공간을 효율적으로 이용하도록 개선하였다[7]. (그림 4)에서 대량적재 알고리즘을 보인다.

r 개의 엔트리로 이루어진 2차원 점 객체 집합이 있다고 가정하자. 페이지 내부 단말 노드 하나가 b 개의 엔트리를 저장할 수 있고, 페이지 하나가 P 개의 내부 단말 노드를 가질 수 있다고 하면, 페이지 하나에 $r_p = b \cdot P$개의 엔트리를 저장할 수 있고, r 개의 엔트리를 저장하는데 $P_p = \lceil r/r_p \rceil$개의 페이지가 필요하다. 먼저, 단말 엔트리를 x축 기준으로 정렬한 다음 $S_p = \lceil \sqrt{P_p} \rceil$개의 수직 슬라이스로 분할한다. 이렇게 하면, 각 수직 슬라이스는 약 $\sqrt{r/r_p}$ 개의 엔트리를 가지게 된다. 그 다음, 각 슬라이스의 단말 엔트리를 y축을 기준으로 정렬한 후 r_p 크기의 단위로 묶는다. 이 경우 마지막 슬라이스는 r_p보다 적은 수의 엔트리를 가질 수도 있다.

엔트리를 페이지 단위로 묶은 후, 루트 노드가 생성될 때까지 각 페이지의 엔트리에 대해 앞에서 설명한 알고리즘과 같은 작업을 수행한다. 다시 설명하면, 먼저 각 페이지에 속한 r_p개의 엔트리를 x축으로 정렬한 후, $S = \lceil \sqrt{P} \rceil$개의 수직 슬라이스로 분할한다. 각 슬라이스는 약 $\sqrt{r_p/b}$개의 엔트리를 갖는다. 그 다음 각 슬라이스의 엔트리를 y축으로 정렬한 후 b 개씩 묶는다. 이 경우에도 역시 마지막 슬라이스는 b보다 적은 수의 엔트리를 가질 수도 있다. 각 페이지에서 모든 노드를 적재한 후 (MBR, node-offset) 쌍을 저장해 둔다. node-offset은 바로 위 트리 레벨의 노드에서 자식 포인터로 사용한다. 현재 페이지에 대해 하나의 페이지 내부 루트 노드가 생성될 때까지 저장한 MBR들을 엔트리로 간주하여 대량적재하면 상위 레벨의 중간 노드를 생성할 수 있다.

앞에서 수행한 알고리즘을 통해 같은 높이를 갖는 페이지에 대한 대량적재가 완료되면, 각 페이지를 가리키는 엔트리를 가지고 상위 레벨의 페이지에 대량적재하는 방법은 다음과 같다. 먼저, 대량적재가 완료된 각 페이지를 가리키는 (MBR, page-ID) 쌍들을 가져온 다음 MBR 들을 상위 레벨의 페이지로 대량적재한다. page-ID는 상위 레벨 페이지의 내부 단말 노드에 저장되는 자식 포인터로 사용한다. 생성된 MBR의 수가 r_p 보다 작거나 같으면 MBR을 적재하는데 페이지 하나만 있으면 되므로 페이지 하나를 생성한 후 대량적재가 종료된다. 데이타의 차원이 2보다 크면, 대량적재 알고리즘은 STR에서 설명한 것과 같은 방법으로 고차원에 대해 일반화할 수 있다[7].

3.2.2 삽입

엔트리 E를 PR-tree에 삽입하는데 MBR을 가장 작게 확장하는 단말 노드 N을 선택한다. 여기에서 단말 노드를 선

```
Algorithm Insert
Data : entry E to insert, PR-tree
Result: updated PR-tree
Select a leaf node N that needs least enlargement to include E from root to leaf;
invoke InsertEntry with E and N to insert E into N;
invoke AdjustTree with N and N';
if root is split then
        create a new root;
        if new root is in-page leaf node then
                create a new page;
                store new root into the new page;
        end
end
```

(그림 5) 삽입 알고리즘

```
Algorithm InsertEntry
Data : entry E to insert, node N where to insert E
Result: node N, resulting second node N' (if N is split)
P ← page enclosing node N;
if there is any empty slot in N or any empty node slot in P then
        if there is any empty slot in N then
                insert E to N;
        else
                split N, and generate N';
                store N' in P;
        end
else
        switch node type of N do
                case in-page leaf node
                        if there is any empty slot in any in-page leaf node of P then
                                if P is already reorganized then
                                        S ← vertical slice enclosing N;
                                        while there is no empty slot in S do
                                                add neighboring S' to S;
                                        end
                                        invoke Reorganize with all nodes in S and E;
                                else
                                        invoke Reorganize with all in-page leaf nodes in P and E;
                                end
                        else
                                split N, and generate N';
                                split P, and generate P';
                                store N' to P';
                        end
                case in-page nonleaf node
                        split N, and generate N';
                        store N' in new page (already created) P';
                        if N is in-page root node then
                                move descendant nodes of N' in P to P';
                                move descendant nodes of N in P' to P;
                        end
                end
        end
end
```

(그림 6) InsertEntry 알고리즘

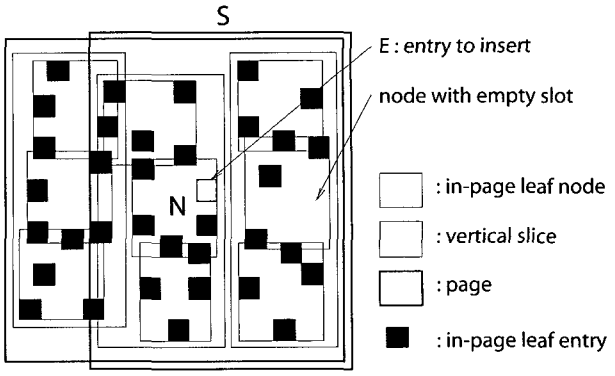
택하는 방법은 R-tree의 ChooseLeaf 알고리즘과 같다[8]. 그리고, InsertEntry 알고리즘을 호출하여 엔트리 E를 노드 N에 삽입한 다음, 삽입의 결과로 변경된 노드 N과 노드 분할이 발생한 경우 새로 생성된 노드 N'을 매개 변수로 AdjustTree 알고리즘을 호출한다. AdjustTree 알고리즘은 자식 노드의 변경을 상위 노드로 전파하는 역할을 한다. 만약 루트 노드가 분할된다면, 새 루트를 만들고 새로 생성된 루트가 페이지 내부 단말 노드이면 페이지를 새로 할당하여 저장한다. (그림 5)는 삽입 알고리즘을 보이고 있다.

InsertEntry 알고리즘은 엔트리 E를 노드 N에 삽입하는

작업을 수행한다. (그림 6)은 InsertEntry 알고리즘을 보여주고 있다. 노드 분할이 발생한 경우 새 노드를 가리키는 엔트리를 부모 노드에 추가하기 위해 AdjustTree 알고리즘에서도 InsertEntry 알고리즘을 호출한다. InsertEntry 알고리즘은 다음과 같이 동작한다. 먼저, 노드 N에 빈 슬롯이 있으면 엔트리 E를 N의 엔트리 리스트 끝에 삽입한다. 노드 N에 빈 슬롯이 없고 N이 속해 있는 페이지 P에 빈 노드 슬롯이 있으면 N을 분할하고 새로 생성된 노드 N′을 P에 저장한다. 노드 N과 페이지 P에 비어있는 엔트리 슬롯이나 노드 슬롯이 없고, 그 대신 P에 포함된 다른 노드 중에 엔트리 슬롯이 있으면, 재구성 알고리즘을 호출하여 P에 대한 '페이지 재구성'을 하고, 페이지의 어느 노드에도 엔트리 슬롯이 없으면 N을 분할한 다음 P도 분할한다. 이 경우 노드의 종류 - 내부 단말 노드, 중간 노드 - 에 따라 수행하는 연산이 달라진다.

N이 페이지 내부 단말 노드인 경우. N이 페이지 내부 단말 노드이면 재구성 또는 노드와 페이지를 분할하는 두 가지 경우가 있다. 페이지 P의 내부 단말 노드 중 하나라도 빈 엔트리 슬롯을 가지고 있으면, P를 재구성하여 엔트리 E를 삽입한다. P가 이미 재구성되어 있다면 N이 속한 슬라이스의 최소 신장(minimum span) 슬라이스에 속한 내부 단말 엔트리와 E를 함께 재구성하고, P가 재구성되어 있지 않다면 P에 속한 모든 내부 단말 엔트리와 E를 함께 재구성한다. 최소 신장 슬라이스란 N이 포함된 슬라이스와 인접한 슬라이스 중 빈 엔트리 슬롯을 포함하는 최소한의 슬라이스 집합을 말한다.

(그림 7)은 재구성의 예를 보이고 있다. 이 그림은 PR-tree의 한 페이지를 나타내고 있는데, 노드 한 개당 최대 4개의 엔트리를 가질 수 있고, 각 수직 슬라이스는 최대 3개의 노드를 가질 수 있다. 그림에서 엔트리 E를 노드 N에 삽입하려고 할 때 페이지 재구성이 일어나는 상황을 보여주고 있다. N에 E를 삽입할 수 없지만, N이 포함된 페이지 안의 다른 노드에 빈 슬롯이 있다. 따라서 노드 N과 빈 슬롯이 있는 노드를 포함하는 수직 슬라이스 S를 구하고, S와 삽입할 엔트리 E에 대해 재구성 알고리즘을 호출한다.



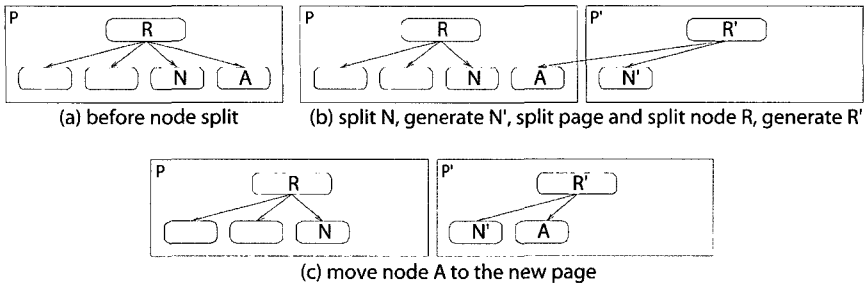
(그림 7) 재구성의 예

(그림 8)의 재구성 알고리즘에서는 S에 포함된 엔트리들과 E를 정렬하여 페이지 분할 없이 E를 S에 삽입하게 된다. 먼저, S와 E의 모든 엔트리를 x축으로 정렬하여 원래 S에 포함된 개수만큼의 수직 슬라이스로 엔트리를 묶는다. 이렇게 생성된 수직 슬라이스는 포함된 엔트리를 각각 y축으로 정렬한 후 페이지 내부 노드 단위로 엔트리를 묶어 노드를 생성한다. 페이지 내부 루트 노드 레벨까지 이런 단계를 거치고 나면 재구성 알고리즘에 주어진 인자에 포함된 엔트리는 S에 포함된 페이지 내부 단말 노드에 삽입되고 페이지 분할은 발생하지 않는다.

만약, 페이지 재구성을 하지 않으면 최악의 경우 페이지 내부의 노드에 저장할 수 있는 엔트리 공간 중 절반의 엔트리만 사용하는 경우도 발생할 수 있으므로 공간의 낭비가 크게 된다. 페이지 재구성을 하는 경우 추가 I/O를 발생하는 페이지 분할 작업을 하지 않으므로 분할할 때 필요한 I/O 작업을 줄이고, 트리 구조에 소요되는 페이지 수를 줄일 수 있으므로 페이지 재구성 후에 다른 트리 연산을 수행하는 데도 I/O 비용을 줄일 수 있다. 재구성을 하게 되면 하나의 노드 내에 엔트리가 더 많이 저장되므로 캐시 성능도 역시

```
Algorithm Reorganize
Data : in-page leaf nodes Nl and entry E to insert
Result: reorganized PR-tree page
Q ← all entries of Nl, and E;
for l ← level of in-page leaf node to level of in-page root node do
        if Q consists of two or more vertical slices then
                sort entries in Q by x coordinate;
        end
        partition them into vertical slices;
        Gs ← vertical slices;
        Q ← empty;
        foreach vertical slice s of the set Gs do
                sort entries in s by y coordinate;
                partition them into in-page nodes;
                add (MBR, node-offset) of this in-page node to Q;
        end
end
```

(그림 8) 재구성 알고리즘



(a) before node split (b) split N, generate N', split page and split node R, generate R'

(c) move node A to the new page

(그림 9) 페이지 분할

```
Algorithm AdjustTree
Data : node N, resulting second node N'(if N was split previously)
Result: updated PR-tree
while N is not root do
        F ← parent node of N;
        En ← N's entry in F;
        adjust MBR of En to tightly enclose all entry rectangles in N;
        if N' is exist then
                create a new entry En';
                child-pointer of En' points to N';
                MBR of En' encloses all rectangles in N';
                invoke InsertEntry with En' and F;
        end
        N ← F;
        N' ← F' if a split occurred;
end
```

(그림 10) AdjustTree 알고리즘

좋아진다.

페이지 P에 빈 노드 슬롯이 없고 P의 어떤 내부 단말 노드에도 비어있는 엔트리 슬롯이 없으면 노드 N을 분할하고 새 노드 N'을 생성한다. P에 새 노드 N'을 삽입할 슬롯이 없으므로 P도 분할하여 새 페이지 P'을 생성하고 N'을 새 페이지 P'에 저장한다. P에 저장되어 있는 노드 중에서 P'의 페이지 루트 노드의 자손들은, P의 페이지 루트 노드를 분할할 때 P'으로 이동하게 된다.

N이 페이지 내부 중간 노드인 경우. N이 페이지 내부 중간 노드이고, 엔트리 E를 저장할 빈 슬롯이 없으면 이미 자손인 내부 단말 노드에서 노드와 페이지 분할이 일어나 N'이 생성되고 새 페이지 P'에 N'이 저장되어 있는 경우이다. N이 페이지 내부 루트 노드이면 PR-tree의 구조를 만족시키기 위해서 P와 P'에 있는 노드를 재배치할 필요가 있다. 새로 생성된 내부 단말 노드와 중간 노드는 P'에 있기 때문에 N'의 자손 노드 중 페이지 P에 있는 것들은 P'으로 옮기고, N의 자손 노드 중 페이지 P'에 있는 것들은 P로 옮긴다.

(그림 9)는 페이지가 삽입 연산에 의해 분할되는 예를 보이고 있다. (그림 9)(a)는 내부 루트 노드 하나와 네 개의 단말 노드를 갖는 페이지를 보여준다. 엔트리의 삽입으로 N이 분할되어 노드 N'이 생성되고, 페이지 P에 빈 노드 슬롯이 없기 때문에 새 페이지 P'을 생성한 후 N'을 P'에 저장한 모습을 (그림 9)(b)에서 보이고 있다. 내부 루트 노드인 R도 분할되어 R'이 생성되고 P에 빈 노드 슬롯이 없으므로 R' 역시 P'에 저장된다. 페이지 내부 트리의 모든 노드는 그의 루트 노드와 같은 페이지에 저장되어야 하므로, 노드 A가 R'의 자식이라고 하면, 분할의 마지막 과정으로 A는 P'으로 이동한다.

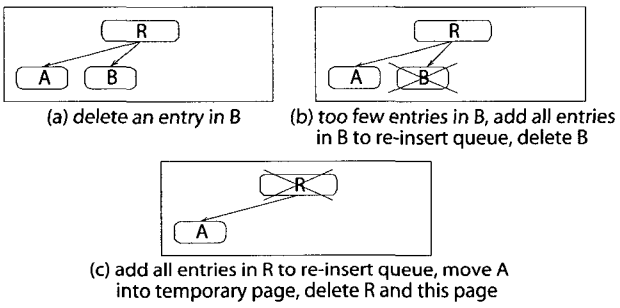
AdjustTree 알고리즘은 삽입 연산으로 인하여 PR-tree 노드에 생긴 변경을 루트 노드까지 전달하는 일을 한다. 먼저, 이 알고리즘은 엔트리가 삽입된 노드의 부모 엔트리 MBR을 갱신한다. 그 다음 노드 분할로 인하여 새 노드 N'이 생성되는 경우 InsertEntry 알고리즘을 호출하여 노드 N'의 부모 엔트리를 부모 노드에 추가한다. 그리고 루트 노드에 도달할 때까지 이 작업을 계속한다. (그림 10)은 AdjustTree 알고리즘을 보이고 있다.

3.2.3 삭제

PR-tree의 삭제 알고리즘은 (그림 11)에서 보이고 있으며, 다음과 같이 동작한다. 먼저, 삭제할 엔트리 E와 겹치는 모든 단말 노드를 주어진 PR-tree에서 검색한 다음, 삭제할 엔트리 E를 포함하고 있는 노드 N을 찾는다. 그 다음, 노드 N에서 엔트리 E를 삭제한 후 N이 노드의 최소 엔트리보다 적은 수의 엔트리를 갖는다면, 노드 N의 부모 노드 엔트리 중 노드 N에 해당하는 엔트리 En을 삭제한 후 노드 N의 모든 엔트리를 재삽입 집합 Qr에 추가하고 N을 삭제한다. 이 때, N이 페이지 내부 루트 노드라면 페이지에 남은 자손

```
Algorithm Delete
Data : entry E to delete
Result: updated PR-tree
Find all leaf nodes N_L that overlaps E and find E in N_L;
delete E from N;
Qr ← empty;
while N is not root do
        F ← parent of N and En ← N's entry in F;
        if number of entries in N < minimum number of entries then
                add all entries in N to Qr;
                delete En from F, and delete N;
                if N is in-page root node then
                        move all descendant nodes of N in P into temporary page;
                        delete P;
                end
        else
                adjust En's MBR to tightly contain all entries in N;
        end
        N ← F;
end
reinsert all entries in Qr into nodes of corresponding levels;
if root has one child then
        make the child the new root;
end
```

(그림 11) 삭제 알고리즘



(그림 12) 페이지 삭제

노드들을 임시 저장 장소로 옮겨 재삽입을 준비한 후 페이지를 삭제한다. 삭제 연산이 모두 끝난 후 Qr에 엔트리가 남아 있으면, 앞에서 서술한 삽입 알고리즘을 이용하여 Qr에 있는 모든 엔트리를 해당하는 높이의 노드에 하나씩 재삽입한다.

(그림 12)는 페이지 내부 루트의 자식 노드가 삭제됨에 따라 페이지가 삭제되는 예를 보이고 있다. (그림 12)(a)에서 삭제 연산에 의해 노드 B에 있는 엔트리 하나가 삭제된다. 삭제 연산의 결과로 B에 최소 엔트리 수보다 작은 수의 엔트리가 존재하게 되면, 삭제 알고리즘에 따라 B의 모든 엔트리를 재삽입 큐에 추가하고 B를 삭제하게 된다. 이 때, (그림 12)(b)에서 보듯이 페이지 내부 루트 노드 R의 자식 노드는 A 밖에 남지 않게 되고 이 경우 하나의 페이지로 유지할 수 없게 된다. 따라서 (그림 12)(c)와 같이 R에 있는 A의 부모 엔트리를 재삽입 큐에 추가하여 다시 PR-tree에 삽입하도록 하고 A를 임시 페이지에 옮긴다. 그리고 페이지 내부 루트 노드 R을 삭제하고, 페이지도 삭제한다.

3.2.4 검색

PR-tree의 범위 질의와 k-최근접 질의 처리 알고리즘은 자식 노드를 접근할 때 접근 할 노드가 현재 노드와 같은 페이지에 있는 경우와 다른 페이지에 있는 경우에 따라 자식 노드의 주소를 계산하는 방법만 R-tree의 검색 알고리즘과 다르다. 먼저, 루트 노드부터 각 자식 노드를 검색 대상에 포함시킬지 여부를 질의 종류에 따라 결정한다. 범위 질의이면 자식 노드의 MBR이 질의와 겹치는 경우, k-최근접 질의이면 거리에 의해 가지치기되지 않는 경우 자식 노드에 대해 검색 알고리즘을 재귀 호출한다. k-최근접 질의에서 가지치기하는 방법은 [9]를 참조한다. 알고리즘에서 특정한 자식 노드를 접근할 때, 그 노드가 현재 노드와 같은 페이지에 있으면 엔트리에 있는 자식 노드의 주소는 페이지 옵셋이므로, 메모리에 있는 현재 페이지의 처음 위치부터 상대적인 위치를 계산하여 자식 노드를 접근한다. 그렇지 않고 접근할 노드가 다른 페이지에 있으면, 엔트리에 있는 주소는 페이지 ID이므로, 페이지 ID에 해당하는 페이지를 메모리에 적재한 다음 페이지 내부 루트 노드를 접근한다. 현재 노드가 단말 노드인 경우에는 질의 종류에 따라 특정 조건을 만족하면 검색 질의에 대한 결과로 엔트리에 포함된 객체를 리턴한다. 즉, 범위 질의이면 객체가 질의와 겹치는 경우, k-최근접 질의이면 주어진 질의와 거리를 계산하여 k개 안에 포함되는 경우가 이에 해당한다. k-최근접 질의에서 주어진 질의와 거리를 계산하는 방법과 k개의 객체를 유지하는 방법은 [9]를 참조한다.

이 논문에서는 k-최근접 질의를 처리하기 위해 branch and bound 알고리즘을 사용한다[9]. 실험에서 점 데이타를

```
Algorithm Search
Data : query parameters, node N to be searched
Result: set of entries
if N is not a leaf then
        foreach entry E in N do
                Ep ← child-pointer of E;
                if E overlaps the query then
                        if Ep is an in-page offset then
                                invoke Search with node accessed by in-page offset Ep;
                        else
                                invoke Search with in-page root node of a page accessed by page ID
                        Ep;
                        end
                end
        end
else
        foreach entry E in N do
                if E overlaps the query then
                        report E as a search result;
                end
        end
end
```

(그림 13) 범위 질의 알고리즘

사용하였으므로 branch and bound 알고리즘은 충분히 효율적으로 동작한다. (그림 13)은 범위 질의 처리 알고리즘을 보이고 있다. k-최근접 질의 처리 알고리즘은 [9]를 참조한다.

4. 성능 분석

PR-tree에서 검색 연산은 각 레벨에서 하나 이상의 노드를 검색하므로, 검색 비용을 계산하기 위해 범위 질의가 접근하는 노드 수에 노드 당 캐시 실패 비용을 곱한 값을 트리의 높이 별로 모두 더해서 캐시 실패 비용을 구한다. 또한, 버퍼를 고려한 디스크 I/O 비용을 계산하기 위해 범위 질의가 접근하는 페이지 수를 구한다.

각 데이타 객체가 각 차원에 대해 [0, 1]인 도메인에 균등분포(uniform distribution)되어 있는 점이라고 가정하면 각 질의 MBR은 hyper cube가 된다. 그리고 같은 높이의 각 노드는 대략 같은 크기를 갖는다고 가정한다.

4.1 캐시 실패 비용

h를 노드의 높이 또는 레벨이라 하고 단말 노드의 높이는 1이라고 가정한다. n을 디스크 페이지의 높이라 하고, H_D는 루트 디스크 페이지의 높이이다. 예를 들어, L이 2이고 h가 3 또는 4이면 n은 2가 된다. 그러면, $h = nL, nL+1, \ldots, nL+(L-1), n = 1, 2, \ldots, H_D$, $n = \lfloor (h-1)/L \rfloor + 1$이 된다. 이 절에서 사용할 용어는 <표 1>과 같다.

N을 트리에 저장된 데이타 객체의 수라 하고, M_h를 높이 h인 노드의 수라고 하면 위의 가정에 의해

〈표 1〉 용 어

h	노드의 높이
n	페이지의 높이
w/x	페이지 내부 중간/단말 노드의 크기(캐시 라인 단위)
f_w/f_x	페이지 내부 중간/단말 노드의 fan-out
L	페이지 내부 트리의 높이
N_w/N_x	질의가 접근하는 중간/단말 노드의 수
T_I	캐시 실패 지연 시간
T_{next}	파이프 라인(선반입) 캐시 실패 지연 시간
M_h	높이가 h인 노드의 수
a_h	높이가 h인 노드 하나의 평균 면적($= 1/M_h$)
s	범위 질의의 크기
d	차원
H_D	루트 노드를 가지고 있는 루트 페이지의 높이

$$M_h = \left\lceil \frac{N}{f_x^n f_w^{h-n}} \right\rceil$$

이 된다.

a_h를 높이 h인 노드 하나가 차지하는 평균 면적이라 하면, $a_h = 1/M_h$이 된다. Minkowski sum 기법을 이용하면 크기가 s인 질의 사각형과 높이 h인 노드 하나가 겹칠 확률은 $(\sqrt[d]{s} + \sqrt[d]{a_h})^d$이 된다[10]. 따라서 질의 사각형과 겹치는 높이가 h인 노드의 수는 $M_h(\sqrt[d]{s} + \sqrt[d]{a_h})^d$ 또는

$$\left(1+\sqrt[d]{\left\lceil \frac{N}{f_x^{\,n} f_w^{\,h-n}} \right\rceil \cdot s}\right)^d$$

가 된다.

T_1을 캐시 실패의 지연 시간, T_{next}를 파이프 라인(선반입) 캐시 실패 지연 시간이라 하면, L 레벨의 페이지 내부 트리로 구성된 PR-tree에서 주어진 범위 질의를 수행하는데 필요한 캐시 실패 비용은 다음과 같다.

$$C_{search} = N_w[T_1 + (w-1)T_{next}] + N_x[T_1 + (x-1)T_{next}] \qquad (1)$$

여기에서 N_w와 N_x는 각각 질의가 접근하는 페이지 내부 트리의 중간 노드와 단말 노드의 수이다. 디스크 페이지의 높이 $H_D = \lceil \log_{f_x f_w^{L-1}} N \rceil$이 되므로, 높이가 H_D인 디스크 페이지의 페이지 내부 단말 노드의 수는

$$M_{L(H_D-1)+1} = \left\lceil \frac{N}{f_x^{H_D-1} f_w^{(H_D-1)(L-1)+1}} \right\rceil$$

이 된다.

루트 레벨의 높이는 $H = L(H_D-1)+1+\lceil \log_{f_w} M_{L(H_D-1)+1} \rceil$이므로, N_w와 N_x는 다음과 같이 계산할 수 있다.

$$N_w = \sum_{h=1,n=1}^{h \neq nL+1, h \leq H, n \leq H_D} \left(1+\sqrt[d]{\left\lceil \frac{N}{f_x^{\,n} f_w^{\,h-n}} \right\rceil \cdot s}\right)^d$$

$$N_x = \sum_{n=1}^{h=nL+1, n \leq H_D} \left(1+\sqrt[d]{\left\lceil \frac{N}{f_x^{\,n} f_w^{\,h-n}} \right\rceil \cdot s}\right)^d$$

4.2 디스크 I/O 비용

PR-tree의 디스크 I/O 비용을 분석하기 위해 여기에서 Leutenegger와 Lopez가 제안한 버퍼 모델을 사용한다. 이 모델의 목적은 R-tree를 이용하여 범위 질의를 수행할 때 발생하는 디스크 접근 횟수를 분석하는 것이다[11]. M_D를 PR-tree가 차지하고 있는 총 페이지 수, Q를 지금까지 수행한 질의의 수, P_h를 높이 h인 노드 하나가 주어진 질의 사각형과 겹칠 확률이라 하자. Q^*는 버퍼를 채울 때까지 필요한 질의의 수의 기대값이라 하자. Q^*의 값은 이진 탐색에 의해 구할 수 있다[11].

Q개의 질의에 의해 접근되는 서로 다른 페이지 수의 기대값 N_Q는 다음과 같다.

$$N_Q = M_D - \sum_{n=1,\ldots,H_D}^{h=nL} M_h (1-P_h)^Q$$

안정 상태(steady state)에서 PR-tree에 주어진 범위 질의가 접근하는 디스크 페이지 수의 기대값 N_{Q^*}는 다음과 같다.

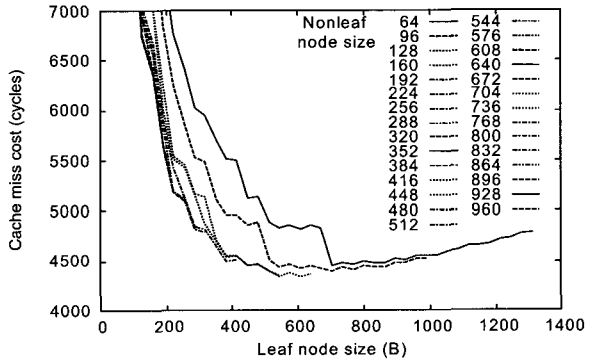
$$\begin{aligned} N_{Q^*} &= 1 + \sum_{n=1,\ldots,H_D-1}^{h=nL} M_h P_h (1-P_h)^{Q^*} \\ &= 1 + \sum_{n=1,\ldots,H_D-1}^{h=nL} M_h (\sqrt[d]{s}+\sqrt[d]{a_h})^d \left\{1-(\sqrt[d]{s}+\sqrt[d]{a_h})^d\right\}^{Q^*} \end{aligned} \qquad (2)$$

4.3 최적의 내부 노드 크기 결정

여기에서 우리의 목표는 캐시와 디스크 I/O 성능을 최적화하는 것이다. 캐시 성능을 최적화하기 위해 수식 1의 캐시 실패 비용을 최소화 하면 되고, 디스크 I/O 성능을 최적화 하려면 수식 2의 디스크 I/O 비용을 최소화 하면 된다.

하지만, 이와 같은 두 개의 목표는 동시에 달성할 수 없으므로, 수식 1에서 계산한 캐시 실패 비용이 최적의 5% 안에 들어오는 내부 단말, 중간 노드 크기 조합에 대해 최대의 디스크 공간 이용률을 보이는 것을 선택한다. 이를 위해, 디스크 페이지 크기가 4KB, 8KB, 16KB, 32KB인 경우, $L \geq 2, w \geq 1, x \geq 1$이고, 한 페이지 내의 공간을 가장 많이 이용하는 조건을 만족하는 가능한 모든 L, w, x의 조합에 대해 캐시 실패 비용을 계산하였다. (그림 14)는 4KB 페이지에 대해 캐시 실패 비용 분석 결과를 그래프로 표시하였다. 이 그림에서 각 그래프는 페이지 내부 중간 노드, 가로축은 페이지 내부 단말 노드의 크기를 나타내며 세로축은 캐시 실패 비용을 나타낸다. 그림의 모든 그래프는 급격히 감소했다가 천천히 증가한다. 이는 어느 한도까지는 큰 노드가 캐시 실패 횟수를 줄인다는 것을 보인다.

<표 2>는 각 페이지 별로 위에서 계산한 캐시 실패 비용이 최적의 5% 안에 들어오는 노드 크기 조합에 대해 최대의 페이지 이용률을 갖는 것을 선택하였다. 여기에서 각 PR-tree는 2차원, 균등 분포인 점 데이타 100만개를 가지고 있으며, MBR은 16바이트, $s = 0.0001$, $T_1 = 150$, $T_{next} = 10$이다.



(그림 14) 캐시 실패 비용 분석 결과

〈표 2〉 최적의 노드 크기 선택

페이지 크기	중간 노드 크기	단말 노드 크기	비용/최적	페이지 이용률
4KB	96B	992B	1.0434	99.22%
8KB	224B	704B	1	97.27%
16KB	384B	832B	1.0314	99.83%
32KB	544B	1184B	1.0449	99.22%

5 실험 결과

이 절에서는 PR-tree의 캐시와 I/O 성능을 측정하기 위해 PR-tree와 디스크 기반 R-tree을 구현하고 비교하였다. 주 메모리 R-tree가 아닌 디스크 기반 R-tree와 비교하는 것은 PR-tree가 디스크 기반 인덱스 구조이기 때문이다. R-tree는 Berchtold가 구현한 R*-tree를 수정하여 구현하였다[12]. 페이지 교체 전략으로는 현재 페이지와 루트 페이지에 이르는 페이지에 잠금을 하는 기능을 추가한 LRU 알고리즘을 사용하였다. MBR의 크기는 2차원 점 객체에 대해 16바이트이고, 내부 단말 노드 엔트리는 4바이트의 페이지 ID를 자식 포인터로 사용하고, 내부 중간 노드 엔트리는 2바이트의 페이지 내부 옵셋을 사용하였다. 실험에 사용된 CPU는 Intel Pentium III 1GHz이고 L1과 L2 캐시 라인 크기는 모두 32바이트이다. 하드디스크는 5400 rpm 속도의 40GB, 주 메모리는 768MB이고 Redhat Linux가 설치되어 있다. 시간 측정에는 wall-clock 시간을 사용하였다. 페이지 크기 4KB, 8KB, 16KB, 32KB에 대해 <표 3>에 표시된 내부 중간, 단말 노드 크기를 갖는 PR-tree를 구현하여 실험하였다. 선반입 루틴은 각 노드 크기에 대해 매크로 코드로 작성하였다. 구현된 트리들은 GCC 3.2.1로 최적화 없이 컴파일하였다. 실험에 사용한 데이타는 [0,1]의 도메인을 갖는 2차원 점 객체이며 균등 분포(uniform distribution)를 갖는다.

CPU 캐시 성능을 측정하기 위해서, 검색이나 갱신 연산을 수행하기 전에 버퍼 풀에 트리의 모든 페이지를 적재하였다. I/O 성능을 측정하기 위해서는, 첫 질의를 수행하기 전에 모든 버퍼 풀을 비우고 버퍼 풀에 대해 실패하는 I/O 페이지 읽기 횟수를 기록하였다.

5.1 노드 크기 선택

이 절에서는 범위 질의 성능에 내부 중간, 단말 노드의 크기가 미치는 영향에 대해 분석하겠다. 이를 위해 4.3절에서 제시한 방법에 의해 설정된 노드 크기를 갖는 PR-tree의 범위 질의 성능을 비교하였다. 실험은 0.1의 크기를 갖는 1000개의 범위 질의를 균등하게 생성하여, 4KB, 8KB, 16KB, 32KB의 페이지 크기에 대해 각각 10만개의 2차원 점 객체를 대량 적재한 PR-tree에 질의하였다.

<표 3>은 각 페이지 크기에 대해 위 실험을 수행한 후 최적의 노드 크기를 선택한 것을 보이고 있다. 노드 크기는 4.3절에서 설명한 대로 선택하였다. 즉, 각 페이지에 대해 질의 수행 시간이 최고의 5% 안에 들어오는 노드 크기 중에서 가장 좋은 페이지 이용률을 보이는 것을 선택하였다. 표 3에서 구한 최적의 노드 크기는 캐시 실패 비용 분석에서 얻는 <표 2>와 매우 유사한 값을 보이는데, 이것은 4.3절에서 구한 수식이 실제 시스템을 매우 잘 반영한다는 것을 보여준다. <표 3>에서 정한 노드 크기는 다음의 모든 실험에서 사용하고 있다.

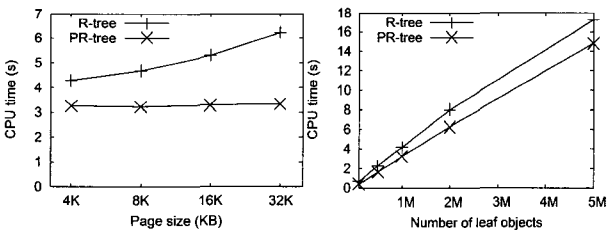
〈표 3〉 실험으로 정한 최적의 노드 크기

페이지 크기	중간 노드 크기	단말 노드 크기	질의 수행 시간	페이지 이용률
4KB	96B	992B	3.37s	99.22%
8KB	192B	864B	3.59s	97.27%
16KB	384B	832B	3.42s	98.83%
32KB	544B	1184B	3.62s	99.22%

5.2 검색 성능

5.2.1 범위 질의 성능

PR-tree와 R-tree의 범위 질의 성능을 비교하기 위해 실험을 하였다. 실험은 페이지 크기와 객체의 수를 변화시켜 가면서 수행하였다. (그림 15)의 모든 실험은 점 객체를 대량적재한 트리에 질의 크기가 0.01인 범위 질의 1000개를 준 후 수행 시간을 측정하였다. 첫 검색 전에 버퍼 풀에 트리의 모든 페이지를 적재하였고, 각 질의는 연속적으로 수행하였다. PR-tree의 노드 크기는 <표 3>에 페이지별로 지정된 것을 사용하였다. (그림 15)(a)는 페이지 크기가 각각 4KB, 8KB, 16KB, 32KB이고, 100만 개의 점 객체를 가지고 있는 트리의 성능을 비교한 것이다. 그림에서 PR-tree는 R-tree보다 1.32에서 1.87배 정도 좋은 성능을 보인다. 페이지 크기가 커짐에 따라 두 트리의 성능 차이가 커지는데 이는 R-tree의 페이지 크기가 더 크면 캐시 실패가 증가하기 때문으로 보인다. (그림 15)(b)는 4KB 페이지를 갖는 트리에 적재할 객체를 10만개에서 500만개로 변화시키면서 실행 시간을 측정하였다. 그림에서 두 그래프는 객체 수가 커짐에 따라 비슷한 성능차이를 보이고 있다.



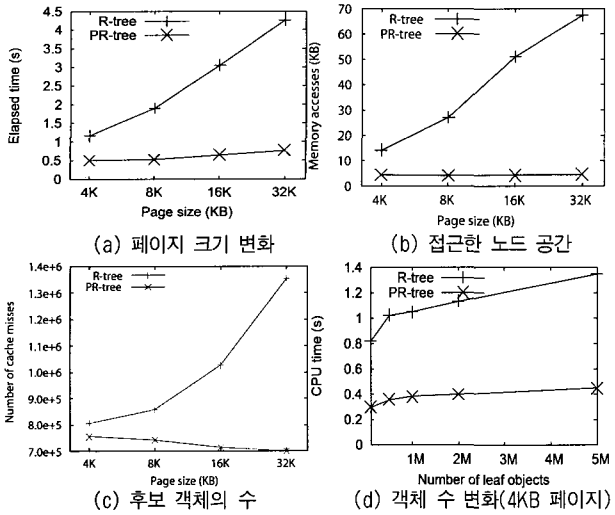
(a) 페이지 크기 변화 (b) 객체 수 변화(4KB 페이지)

(그림 15) 범위 질의 성능

5.2.2 k-최근접 질의 성능

이 절에서는 PR-tree와 R-tree의 페이지 크기와 객체 수를 변화시켜 가면서 k-최근접 질의의 성능을 비교하였다. 모든 실험은 k 값이 5인 k-최근접 질의 1000개를 수행할 때 소요되는 시간을 측정하였다. 첫 검색 전에 버퍼 풀에 트리의 모든 페이지를 적재하였고, 각 질의는 연속적으로 수행하였다.

(그림 16)(a), (b), (c)는 페이지의 크기를 4KB, 8KB, 16K, 32KB로 변화시켜 가면서, 각각 k-최근접 질의의 수행 시간, 질의가 접근하는 노드 공간의 양, 질의 알고리즘이 접근하는 후보 객체의 수를 나타내고 있다. (그림 16)(a)에서 PR-tree는 R-tree보다 2.74에서 9.67배의 성능 향상을 보인다. 페이지 크기가 커지면 성능 차이가 더 커지는데 이것은 (그림 16)(b)와 (그림 16)(c)에서 보듯이, PR-tree의 경우 노드의 크기가 R-tree보다 작으므로 질의가 접근하는 노드 공간의 양과 후보 객체의 수도 작아지기 때문이다.



(a) 페이지 크기 변화 (b) 접근한 노드 공간

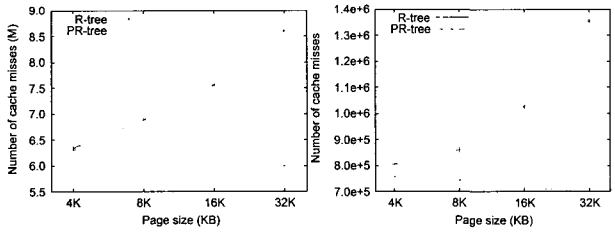
(c) 후보 객체의 수 (d) 객체 수 변화(4KB 페이지)

(그림 16) k-최근접 질의 성능

(그림 16)(d)는 객체의 수를 변화시켜가면서 PR-tree와 R-tree의 k-최근접 질의 성능을 비교한 것이다. 그림에서 PR-tree는 R-tree보다 더 좋은 성능을 보이며, 성능 차이는 객체의 수와 비례한다. 이는 많은 수의 객체를 색인한 후 k-최근접 질의를 처리할 때 PR-tree가 더 우수하다는 것을 보여준다.

5.2.3 캐시 성능

여기에서는 PR-tree와 R-tree의 범위 질의와 k-최근접 질의에 대한 캐시 성능을 비교한다. (그림 17)은 페이지 크기를 변화시켜가면서 100만개의 객체를 대량적재한 PR-tree와 R-tree에서 질의 크기가 0.01인 범위 질의 1000개와 k 값이 5인 k-최근접 질의 1000개를 수행할 때 발생한 캐시 실패 횟수를 보여준다. 캐시 실패 횟수는 Performance API[13]를 이용하여 L2 캐시 실패 횟수를 측정하였다. (그림 17)(a)와 (그림 17)(b)에서 페이지 크기가 증가함에 따라 R-tree의 캐시 실패 횟수는 같이 증가하고 있으나 PR-tree의 캐시 실패 횟수는 반대로 감소하고 있다. (그림 17)의 그래프는 (그림 15)(a)의 범위 질의 수행 시간 그래프, (그림 17)(a)의 k-최근접 질의 수행 시간 그래프와 같은 경향을 보이고 있으므로, 캐시 성능은 질의 수행 성능에 영향을 미치는 주요한 인자임을 알 수 있다.



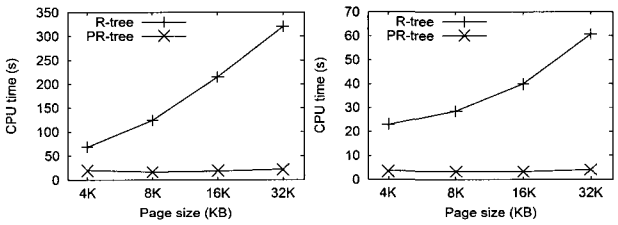
(a) 범위 질의 (b) k-최근접 질의

(그림 17) 캐시 성능

5.3 삽입과 삭제 성능

이 절에서 PR-tree와 R-tree의 삽입과 삭제 성능을 비교한다. 이 실험 역시 삽입과 삭제 연산을 수행하기 전에 트리의 모든 페이지를 버퍼 풀에 적재하였고, 삽입과 삭제는 연속적으로 수행하였다. 그리고 삽입과 삭제에 따라 변화되는 페이지를 디스크에 저장하는 시간은 측정된 시간에서 제외하였으며 삽입을 위해 충분한 버퍼를 미리 확보하여 디스크 접근을 하지 않도록 했다. (그림 18)(a)는 100만개의 객체를 대량적재한 트리에 10만개의 객체를 하나씩 삽입하는데 걸리는 시간을 보여준다. 페이지 크기가 커짐에 따라 R-tree의 성능은 계속 나빠지지만 PR-tree의 성능은 거의 변화가 없다. R-tree의 삽입 알고리즘은 객체를 삽입할 때 노드를 가장 작게 확장시키는 자식 노드를 선택한다. 객체를 삽입할 때 R-tree의 중간 노드(페이지와 같음)의 경우 모든 엔트리를 검사해야 하지만, PR-tree의 경우 페이지 안에 내부 트리가 있으므로, 내부 트리 높이보다 1 작은 갯수의 중간 노드 엔트리와 내부 단말 노드 하나의 엔트리만 검사하면 된다.

(그림 18)(b)에서 100만개의 객체를 대량적재한 후 10만개의 객체를 하나씩 삭제하는데 경과한 시간을 보여주고 있다. (그림 18)(b)는 (그림 18)(a)의 삽입 실험 결과와 유사한 경향을 보이는데 이것은 삭제 알고리즘이 삭제할 객체를 검사하는 부분과 재삽입 집합에 있는 객체를 삽입하는 부분을 포함하고 있기 때문이다.



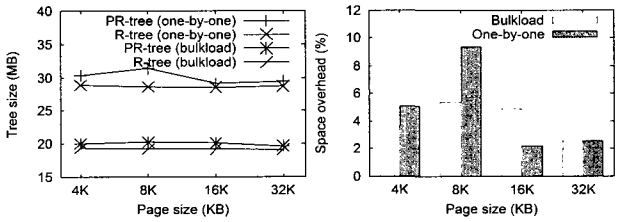
(a) 삽입 (b) 삭제

(그림 18) 삽입과 삭제 성능

5.4 공간 부담(overhead)과 I/O 성능, 총 수행 시간 비교

5.4.1 공간 부담(overhead)

이 절에서는 PR-tree의 공간 부담(overhead)에 대해 보인다. 공간 부담은 PR-tree가 차지하는 페이지 수를 R-tree의 그것으로 나눈 다음 1을 뺀 것이다. (그림 19)는 4KB, 8KB, 16KB, 32KB의 페이지를 갖는 PR-tree와 R-tree의 트리 크기와 R-tree에 대한 PR-tree의 공간 부담을 보여준다. (그림 19)(a)는 100만개의 2차원 점 객체를 각각 대량적재하거



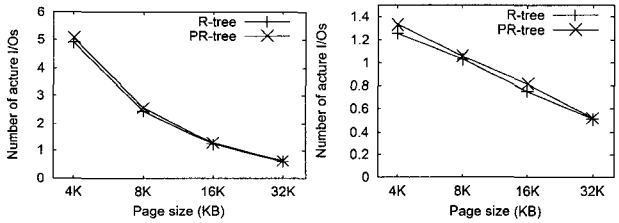
(a) 트리 크기 (b) PR-tree의 공간 부담

(그림 19) 트리 크기와 공간 부담(100만개의 객체)

나 하나씩 삽입한 후 PR-tree와 R-tree의 트리 크기를 비교한 것이다. 두 트리 모두 하나씩 삽입한 것이 대량 적재한 것보다 약 50% 정도 크다. R-tree에 대한 PR-tree의 공간 부담은 (그림 19)(b)에서 보이고 있다. 이 그림에서 PR-tree가 대량적재한 것은 2.6%-5.4%, 하나씩 삽입한 것은 2.2%-9.4%의 공간 부담을 보이고 있다. 결국 이 비율로 PR-tree가 R-tree보다 크다는 의미이다.

5.4.2 I/O 성능

(그림 20)에서 PR-tree의 검색 연산에 대한 I/O 성능을 보이고 있다. I/O 성능은 질의 당 버퍼 풀에 대해 실패하는 I/O 페이지 읽기 요청의 수를 측정하였다. PR-tree와 R-tree 모두 4KB, 8KB, 16KB, 32KB의 페이지 크기에 대해 100만개의 2차원 균등 분포 점 객체를 대량적재한 후 두 가지 질의를 수행하였다. (그림 20)(a)는 크기가 0.01인 범위 질의 1000개를 수행한 결과를 보여주고, (그림 20)(b)는 k 값이 5인 k-최근접 질의 1000개를 수행한 결과이다. 버퍼 풀은 첫 검색 요청 전에 비어 있는 상태이며 각 질의는 계속 이어서 수행된다. PR-tree는 R-tree보다 범위 질의에서 2.4%-5.0%, k-최근접 질의에서 2.1%-8.7% 많은 페이지를 접근한다. 이 결과는 (그림 19)의 공간 부담과 비슷한 경향을 보인다.

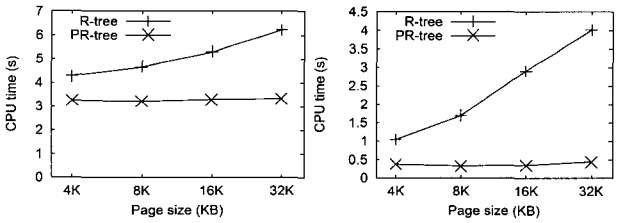


(a) 범위 질의 (b) k-최근접 질의

(그림 20) 총 질의 수행 시간

5.4.3 총 질의 수행 시간 비교

이 절에서는 PR-tree의 성능을 질의 수행에 걸리는 총 시간의 기준으로 평가한다. (그림 21)에서 각 트리에 각각 100만개의 점 객체를 대량적재한 후 페이지 크기에 따라 두 가지 질의를 수행할 때 걸리는 총 소요 시간을 측정한 결과를 보이고 있다. (그림 21)(a)는 크기가 0.01인 범위 질의 1000개를 수행한 결과이고, (그림 21)(b)는 k 값이 5인 k-최근접 질의를 수행한 결과이다. 두 그림 모두 CPU 시간만을



(a) 범위 질의 (b) k-최근접 질의

(그림 21) I/O 성능

측정한 결과(5.2.1절, 5.2.2절)와 비교해 볼 때, 총 소요 시간이 범위 질의의 경우 2.6%-8.6%, k-최근접 질의의 경우 4.6%-88%씩 더 소요되는 것을 제외하고 그래프의 모양은 비슷하다. 이는 PR-tree에서 발생하는 캐시 성능의 향상이 전체 수행 성능에 더 중요한 요소라는 것을 말해준다.

6. 결 론

디스크 I/O에 최적인 R-tree와 CPU 캐시에 최적인 R-tree는 노드의 크기에서 수십-수백 바이트와 수-수십 킬로바이트라는 큰 차이가 있으므로, 디스크 최적인 경우 CPU 캐시 성능이 떨어지고 CPU 캐시 최적인 경우 디스크 I/O 성능이 좋지 않다. 이 논문에서는 CPU 캐시와 디스크 I/O에 모두 효율적인 PR-tree를 제안하였다. PR-tree는 캐시 성능을 높이기 위해 노드 크기를 캐시 라인보다 크게 하고, 노드에 접근하기 전에 선반입 연산으로 캐시에 노드 데이타를 미리 적재하여 캐시 실패 비용을 줄였고, I/O 성능을 높이기 위해 디스크 페이지에 노드를 최적으로 배치하였다. 그리고 범위 질의를 수행할 때 드는 캐시 실패 비용을 분석하였으며, 최적인 성능을 보이는 노드 크기를 결정하기 위해 다양한 디스크 크기에 대해 가능한 노드 크기의 PR-tree를 구현하여 실험하였다. 실험에 의하면 PR-tree는 하나씩 삽입, 삭제, 범위 질의, k-최근접 질의에서 더 좋은 캐시 성능을 보였다.

참 고 문 헌

[1] Boncz, P. A., Manegold, S. and Kersten, M. L., "Database Architecture Optimized for the New Bottleneck: Memory Access," In Proceedings of the 25th VLDB, pp.54-65, 1999.

[2] Rao, J. and Ross, K. A., "Cache Conscious Indexing for Decision-Support in Main Memory," In Proceedings of 25th International Conference on Very Large Data Bases, pp.78-89, 1999.

[3] Rao, J. and Ross, K. A., "Making B+-Trees Cache Conscious in Main Memory," In Proceedings of the 2000 ACM SIGMOD International Conference, pp.475-486, 2000.

[4] Chen, S., Gibbons, P. B. and Mowry, T. C., "Improving Index Performance through Prefetching," In Proceedings of the 2001 ACM SIGMOD International Conference, pp.235-246, 2001.

[5] Chen, S., Gibbons, P. B., Mowry, T. C. and Valentin, G., "Fractal Prefetching B+-Trees: Optimizing Both Cache and Disk Performance," In Proceedings of the 2002 ACM SIGMOD International Conference, pp.157-168, 2002.

[6] Kim, K., Cha, S. and Kwon, K., "Optimizing Multi-dimensional Index Trees for Main Memory Access," In Proceedings of the 2001 ACM SIGMOD International Conference, pp.139-150, 2001.

[7] Leutenegger, S. T, Edgington, J. M. and Lopez, M. A., "STR: A Simple and Efficient Algorithm for R-Tree Packing," In Proceedings of the Thirteenth International Conference on Data Engineering, pp.497-506, 1997.

[8] Guttman, A., "R-Trees: A Dynamic Index Structure for Spatial Searching," SIGMOD'84, Proceedings of Annual Meeting, pp.47-57, 1984.

[9] Roussopoulos, N., Kelley, S. and Vincent, F., "Nearest Neighbor Queries," In Proceedings of the 1995 ACM SIGMOD, pp.71-79, 1995.

[10] Kamel, I. and Faloutsos, C., "On Packing R-trees," In Proceedings of the 1993 ACM CIKM Conference, pp.490-499, 1993.

[11] Leutenegger, S. T. and Lopez, M. A., "The Effect of Buffering on the Performance of R-Trees," In Proceedings of the 14th International Conference on Data Engineering, pp.164-171, 1998.

[12] Beckmann, N., Kriegel, H.-P., Schneider, R. and Seeger, B., "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," In Proceedings of the 1990 ACM SIGMOD, pp.322-331, 1990.

[13] Dongarra, J., London, K., Moore, S., Mucci, P. and Terpstra, D., "Using PAPI for Hardware Performance Monitoring on Linux Systems," In Proceedings of the 2nd LCI International Conference on Linux Clusters: The HPC Revolution 2001.

박 명 선

e-mail : masonbok@kt.co.kr

1995년 서울대학교 컴퓨터공학과(학사)

1997년 서울대학교 컴퓨터공학과(공학석사)

2005년 서울대학교 전기·컴퓨터공학부 (공학박사)

2004년~현재 KT BcN본부 전임연구원

관심분야 : 멀티미디어 데이타베이스, 다차원 인덱스