R-Tree를 위한 캐시와 디스크 성능 최적화[1)]

박명선° 이석호
서울대학교 전기컴퓨터공학부
mspark@db.snu.ac.kr°, shlee@cse.snu.ac.kr

Optimizing Both Cache and Disk Performance of R-Trees

Myungsun Park° Sukho Lee
School of Electrical Engineering and Computer Science

요 약

R-Tree는 일반적으로 트리 노드의 크기를 디스크 페이지의 크기와 같게 함으로써 I/O 성능에 최적이 되도록 구현한다. 최근에는 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 가능하였다. 그러나, 디스크 최적 R-Tree와 CPU 캐시 최적 R-Tree의 노드 크기 사이에는 수십-수백 바이트와 수-수십 킬로바이트라는 큰 차이가 있으므로, 디스크 최적 R-Tree는 캐시 성능이 나쁘고, CPU 캐시 최적 R-Tree는 나쁜 디스크 성능을 보이는 문제점을 가지고 있다. 이 논문에서는 CPU 캐시와 디스크에 모두 최적인 R-Tree, TR-Tree를 제안한다. 먼저, 디스크 페이지 안에 들어가는 페이지 내부 트리의 높이와 단말, 중간 노드의 크기를 결정하는 방법을 제시한다. 그리고, 이를 이용하여 TR-Tree의 검색 연산에 필요한 캐시 미스 수를 최소화였고, TR-Tree의 검색 성능을 최적화하였다. 또한, 디스크 I/O 성능을 최적화하기 위해 메모리 노드들을 디스크 페이지에 잘 맞게 배치하였다. 여기에서 구현한 TR-Tree는 디스크 최적 R-Tree보다 삽입 연산에서 6에서 28배 정도 빨랐으며, 검색 연산에서는 1.28배에서 2배의 성능 향상을 보였다.

1. 서론

프로세서의 속도 증가와 DRAM의 처리 속도 증가의 차이가 점점 커짐에 따라 데이타베이스 시스템의 성능 향상을 위해 CPU 캐시의 효율적인 이용이 새로운 문제로 떠오르고 있다. 따라서, 기존의 인덱스 구조를 캐시에 효율적으로 개선한 연구 결과가 발표되고 있다. B-Tree에서는 한 노드의 크기를 캐시 라인의 크기와 같게 하거나[1,2] 캐시 선반입을 이용하고 노드를 디스크 페이지에 적절히 배치함으로써 캐시와 디스크 성능 모두를 최적화하려는 연구[3,4]가 있었고, R-Tree에서는 MBR에 저장되는 키를 간단한 방법으로 압축하여 한 노드에 더 많은 엔트리를 저장하고 트리의 높이를 낮춤으로써 성능을 높이는 연구[5]가 있었다. 그러나, 여전히 캐시에 최적화된 R-Tree와 디스크 입출력에 최적화된 전통적인 R-Tree는 각각 디스크 입출력과 캐시 이용에 나쁜 성능을 가지는 문제점을 가지고 있다. 따라서, 이 논문에서는 최근의 프로세서에서 지원하는 캐시 선반입을 이용하여 캐시에 대해 최적화하고 노드를 디스크 페이지에 잘 배치함으로써 캐시와 디스크 입출력 모두에 효율적인 R-Tree인 TR-Tree를 제안한다.

본 논문은 다음과 같이 구성되어 있다. 먼저 2절에서 관련 연구에 대해 기술하고 3절에서 TR-Tree의 구조, 작동 알고리즘에 대해 설명한다. 4절에서는 검색을 할 때 접근하는 캐시 라인의 수를 계산하는 방법으로 TR-Tree의 성능을 분석하고, 5절에서 실험 결과를 보이며, 6절에서 결론을 맺는다.

2. 관련 연구

주메모리 인덱스의 성능을 향상시키기 위해, CPU 캐시를 B-Tree 노드의 크기를 CPU와 메모리 사이의 입출력 단위인 캐시 라인의 크기와 같게 한 연구[1,2]가 있었다. 그리고, 최근의 프로세서가 CPU 캐시의 선반입을 지원하는 기능을 이용하여 B+-Tree의 캐시와 디스크 성능을 모두 최적화하려는 연구[3,4]도 있었다. 또한, 주메모리 R-Tree에서 MBR에 저장되는 키를 압축하는 QRMBR이라는 기법을 이용하여 한 노드의 엔트리 수를 증가시키고 이를 이용하여 성능을 높이는 CR-Tree라는 연구[5]가 있었다. 이 연구는 노드 크기에 따라 다양한 종류의 R-Tree 변형에 대해 성능을 실험하고 있다.

3. TR-Tree

TR-Tree는 CPU 캐시의 선반입 기능을 이용하고 노드들을 디스크에 저장하는 데 필요한 페이지 수가 적은, CPU 캐시와 디스크 성능의 효율적인 R-Tree 인덱스 구조이다.

TR-Tree의 성능 향상은 크게 두 가지로 다음과 같이 요약된다.

첫째, CPU 연산에서 필요한 데이타를 CPU 캐시에 선반입하면 캐시 미스 지연 시간이 줄어드는 성질을 이용하여 노드의 크기를 캐시 라인의 크기보다 크게 함으로써 트리의 높이를 낮춘다. 트리의 높이가 낮아지면 검색할 때 접근하는 노드의 수가 감소하므로 성능이 높아지게 된다.

둘째, 노드를 최소의 디스크 공간을 사용하여 저장하기 위해 메모리 노드로 구성된 트리(페이지 내부 트리)를 각각 하나의 디스크 페이지에 저장한다. 페이지 내부 트리의 높이, 중간 노드의 수, 단말 노드의 수를 구하는 방법은 4절의 내용과 같으며, 이를 이용하여 검색할 때 필요한 캐시 접근 비용을 계산하여 최적화된 값을 찾는다.

3.1. 노드 구조

TR-Tree의 한 노드는 그림 1과 같은 내부 구조 트리를 가진다.

그림 1은 페이지 내부 트리의 높이가 2인 TR-Tree의 한 디스크 페이지이다. 중간 노드와 단말 노드는 크기가 다른데, 이는 노드들이 디스크 페이지 하나에 낭비가 없도록 채워 넣기 위해서이다. w, x, L을 구하는 방법은 4절에서 설명한다.

그림 2는 디스크 페이지 하나의 데이타 구조이다. (b)와 (c)는 각각 중간 노드와 단말 노드의 엔트리인데, 중간 노드의 경우 같은 페이지 내의 다른 노드를 가리키므로 2바이트 offset으로 충분하고 단말 노드의 경우 다른 페이지 내의 루트 노드를 가리키기 때문에 4 바이트의 페이지 ID를 가지고 있다.

1) 이 연구는 두뇌 한국 21의 지원을 받았음

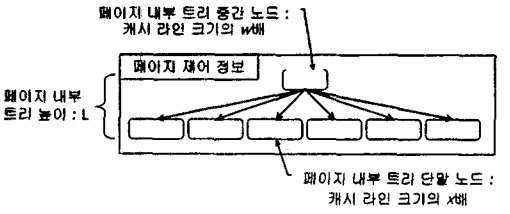

그림 1 노드 구조

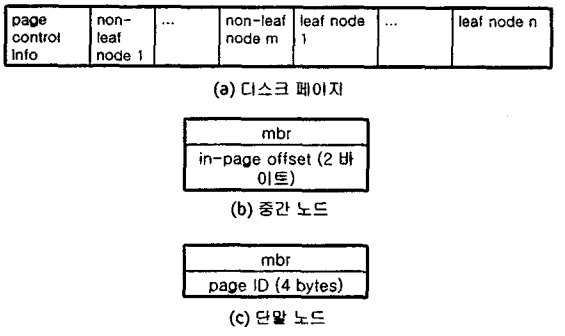

그림 2 데이타 구조

3.2. 작동 알고리즘

TR-Tree에서 삽입, 삭제, 검색 연산은 루트 노드에서 시작한다. 루트 노드가 메모리에 없으면 디스크에서 읽어 온다. 같은 페이지 내의 다른 노드를 방문하는 경우, 페이지 offset을 이용하여 디스크를 읽지 않고 접근할 수 있고, 다른 페이지의 루트 노드를 방문하는 경우(현재 방문하고 있는 노드가 페이지 트리 내의 단말 노드인 경우) 방문할 노드가 메모리에 있는지 여부에 따라 디스크 페이지를 읽기를 결정한다. 즉, 디스크 페이지 내에 페이지 크기보다 작은 노드가 있다는 점과 포인터의 크기가 페이지 내부 트리의 단말, 중간 노드에 따라 다르다는 점이 기존 R-Tree와 차이점이라 할 수 있다.

다음으로, 노드 배치 전략을 보겠다. 노드를 디스크 페이지에 배치하는 기준은 부모 노드와 그 자식 노드를 가능한 한 같은 페이지에 넣어서 검색을 할 때 하나의 디스크 읽기를 하도록 하는 것이다. 따라서 중간 노드와 단말 노드의 구별 없이 분할되어 새로 생성되는 노드가 부모 노드와 같은 디스크 페이지에 들어갈 수 없을 때, 새로운 페이지에 저장한다. 만약, 새로 분할된 노드가 페이지 내부 트리의 루트 노드라면 그 노드의 모든 자손 노드들은 이전 페이지에서 새로운 페이지로 모두 이전하는 작업을 한다.

4. 성능 분석

TR-Tree에서 검색 연산은 각 레벨에서 하나 이상의 노드를 검색하므로 검색 비용을 계산하기 위해 각 높이 별로 방문하는 노드의 개수를 다음과 같이 구한다. 각 데이타 객체가 균등 분포되어 있다고 가정하면 각 질의 MBR은 hyper-square가 된다. 그리고, 같은 높이의 각 노드는 대략 같은 크기를 갖는다고 가정한다.

아래의 분석에서 사용할 용어는 표 1과 같다.

표 1 용어

T_1	캐시 미스 지연 시간
T_{next}	파이프라인 캐시 미스 지연 시간
L	페이지 내부 트리의 최대 레벨
w	중간 노드의 크기(캐시 라인 단위)
x	단말 노드의 크기(캐시 라인 단위)
f	캐시 라인 당 엔트리의 수

h를 노드의 높이 또는 레벨이라 하고 단말 노드의 높이는 1이라고 가정한다. n을 디스크 페이지 트리의 높이라 하고, H_D는 루트 디스크 페이지의 높이이다. 그러면, $h=nL, nL+1, \ldots, nL+(L-1), n = 1, 2, \ldots, H_D$이고, $n=\lfloor(h-1)/L\rfloor+1$이 된다. M_h를 높이 h인 노드의 수라고 하면 위의 가정에 의해

$$M_h = \left\lceil \frac{N}{(xf)^h (wf)^{h-n}} \right\rceil$$

이 된다.

a_h를 높이 h인 노드 하나가 차지하는 평균 면적이라 하면, $a_h = 1/M_h$이 된다. Minkowski sum 기법[6]을 이용하면 크기가 s인 질의 직사각형에 대해 높이 h인 노드 하나가 주어진 질의와 겹칠 확률은 $(\sqrt[d]{s}+\sqrt[d]{a_h})^d$이 된다. 따라서 질의 직사각형과 겹치는 높이가 h인 노드의 수는 $M_h(\sqrt[d]{s}+\sqrt[d]{a_h})^d$ 또는

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

가 된다.

결국, N개의 리프 노드 엔트리를 갖는, L 레벨의 페이지 내부 트리로 구성된 TR-tree를 검색하는 데 소요되는 캐시 지연 비용은 아래와 같다. 즉, TR-tree에서 주어진 질의 직사각형 s를 검색하는데 필요한 캐시 지연 비용은, 높이 h인 노드 하나를 참조하는 비용에 높이 당 검색해야 할 노드의 수를 곱한 값이다.

$$cost = N_w[T_1 + (w-1)T_{next}] + N_x[T_1 + (x-1)T_{next}]$$

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

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

TR-Tree를 캐시에 효율적으로 만들기 위해서 2차원의 데이타에 대해, $L \geq 2, w \geq 64B, x \geq 64B$이고 L, w, x는 한 페이지 내의 공간을 가장 많이 이용하는 조건을 만족하는 가능한 모든 L, w, x의 조합에 대해 위의 캐시 비용을 계산하여 최적인 것을 선택한다.

5. 실험 결과

TR-Tree를 디스크 페이지 크기 4KB, 8KB, 16KB, 32KB에 대해 구현하고, 각 페이지 크기에 대해 가능한 모든 중간, 단말 노드의 크기에 대해 삽입, 검색 성능을 측정하였다. 실험에 사용된 기계는 Intel Pentium III 1GHz이고 L1, L2 캐시 모두 32바이트이다. 하드디스크는 60GB, 메모리는 768MB이고 Redhat Linux가 설치되어 있다. gcc 3.2.1로 최적화 없이 컴파일하였다.

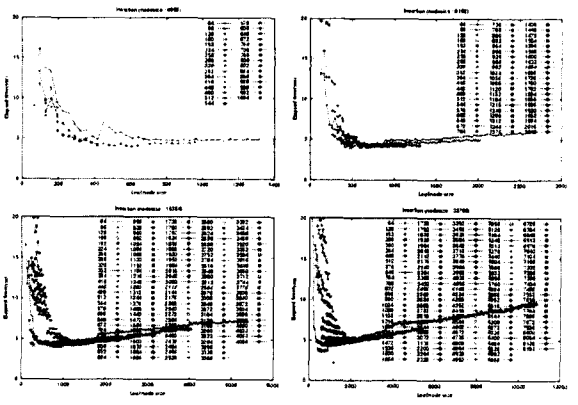

그림 3 TR-Tree에 대한 삽입 연산

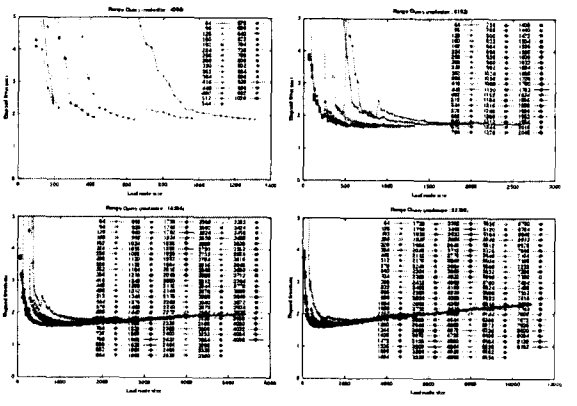

그림 4 TR-Tree에 대한 범위 질의

그림 3은 왼쪽 위부터 각각 4KB, 8KB, 16KB, 32KB의 페이지 크기를 갖는 TR-Tree를 구현하고 0에서 1사이의 float 형 균등 분포 데이타 10만개를 하나씩 삽입하였다. 가로축은 페이지 내부 단말 노드의 크기이며, 그래프의 각 선은 페이지 내부 중간 노드의 크기이다. 그림 3은 각 단말, 중간 노드 크기의 쌍에 대해 삽입 시간을 측정하였다. 노드 크기에 따라 큰 변화는 없지만 단말 노드의 크기가 커짐에 따라 급격히 성능이 좋아졌다가 점점 성능이 나빠지는 모양을 보인다.

그림 4에서 범위 질의는 질의 사각형의 각 변의 길이가 데이타 영역의 30%가 되게 하였고 그림 3과 마찬가지로 각 페이지 내부 단말 노드와 중간 노드의 크기에 대해 1000개의 질의를 수행한 평균 시간을 측정하였다. 대체로 페이지 크기가 클수록 조금씩 나은 성능을 보이고 있으며 단말 노드의 크기가 커짐에 따라 급격히 성능이 좋아졌다가 완만하게 나빠지는 모습을 보인다.

그림 5는 Stefan Berchtold의 R*-Tree를 수정하여 TR-Tree와 같은 조건으로 구현한 다음, 같은 데이타를 삽입하고 같은 크기의 질의 직사각형을 이용하여 범위 질의한 결과를 각 페이지 크기별로 가장 좋은 성능의 TR-Tree와 비교하였다. *R-Tree*의 경우 삽입 연산에서는 노드 크기가 커짐에 따라 성능이 좋아지며, 범위 검색에서는 성능이 나빠지고 있다. 이는 검색 연산의 경우 노드 크기가 커지면 캐시 성능이 나빠지기 때문인 것으로 보인다. 그림 5에 따르면 TR-Tree는 기존의 디스크 최적 R-Tree에 비해 삽입 연산에서는 6에서 28배 정도, 검색 연산에서는 1.28배에서 2배 정도 빠름을 알 수 있다.

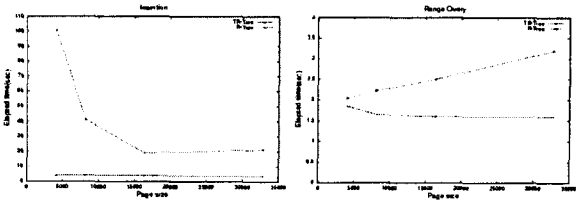

그림 5 TR-Tree와 R-Tree의 삽입, 범위 질의 성능 비교

6. 결론

디스크 최적인 R-Tree와 CPU 캐시 최적인 R-Tree는 노드의 크기가 수십-수백 바이트와 수-수십 킬로바이트라는 큰 차이가 있으므로, 디스크 최적인 경우 CPU 캐시 활용도가 떨어지고 CPU 캐시 최적은 디스크 입출력 성능이 좋지 않다.

이 논문에서는 CPU 캐시 선반입 기법을 이용하여 CPU 캐시 성능을 높이고, 디스크 페이지에 노드를 최적으로 배치함으로써 디스크 입출력에서도 효율적인 TR-Tree를 제안하였다. 이 논문에서는 캐시 효율과 입출력 성능을 높이기 위해 디스크 페이지 안에 들어가는 페이지 내부 트리의 높이와 단말, 중간 노드의 크기를 결정하는 방법을 수식으로 제시하고, 실험을 통해 기존의 디스크 최적 R-Tree에 비해 삽입 연산에서는 6에서 28배 정도, 검색 연산에서는 1.28배에서 2배 정도 빠름을 보였다.

추후로는 MBR 키에 저장되는 키를 압축하는 QRMBR 기법을 TR-Tree에 적용하여 성능 분석을 하고 캐시 효율을 높이는 기법을 연구할 예정이다.

참고문헌

[1] Rao, J., 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.

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

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

[4] Chen, S., Gibbons, P. B., Mowry, T. C., 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.

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

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