멀티코어 프로세서에서의 트리 기반 인덱스 성능 실험 평가

김경화○ 심준호 이익훈
숙명여자대학교, (주)프람트
{kamza81○,jshim}@sookmyung.ac.kr, ihlee@prompt.co.kr

Empirical Performance Evaluation of Tree-based Indexes on Multi-Core Processors

Kyunghwa Kim○ Junho Shim Ig-hoon Lee
Sookmyung Women's University, Promt Corp.

요 약

점차 더 벌어지는 CPU 속도와 메모리 속도의 차이로 인하여 메모리 접근 병목 현상이 발생하였고, 이 현상을 극복하기 위하여 캐시를 고려한 인덱스 구조에 관한 연구가 계속 되었다. 또한 최근 CPU 트렌드가 싱글 코어에서 멀티 코어로 전환점을 맞으면서 캐시메모리의 효율에 대한 중요성이 더욱 부각되었다. 본 논문은 최신 프로세서를 탑재한 시스템에서 메인 메모리 데이터베이스 시스템을 위한 인덱스 구조들의 성능을 비교 평가하고, 그 중 캐시를 고려한 트리 인덱스의 성능이 유용함을 보인다.

1. 서 론

고성능 메인 메모리 데이터베이스 시스템을 위한 다양한 연구 가운데 해결해야 할 대표적 문제가 새로운 성능 병목 현상이다. 지난 십년 동안 프로세서의 속도는 무어의 법칙에 따라 18개월 마다 2배 속도로 빨라지고 있지만 DRAM 의 속도는 그만큼 빨라지지 않고 있다. 즉, CPU와 메모리의 속도 차이로 인해 CPU의 효율을 현저히 떨어뜨리는 결과를 초래하는 메모리 접근 병목현상이 발생하고 있다. 이 현상을 극복하기 위해서는 CPU와 메모리 사이의 속도 차이를 줄이기 위해 만들어진 캐시 메모리를 적절하게 이용하여야 한다. [1]은 L2캐시를 고려한 프로그램 설계 구현으로 메모리의 병목 현상을 완화시킬 수 있음을 보였다. 이에 따라 병목현상의 주요 쟁점이 되는 L2캐시 사용 효율성을 높이기 위해 최신 정보 처리 시스템의 캐시를 효율적으로 활용하는 인덱스 구조에 대한 연구가 계속되고 있다. [2]는 캐시를 고려한 새로운 B-트리인 CSB+-트리를 제시하였고, [3]는 CSB+-트리의 접근방법처럼 캐시를 최대한 잘 이용할 수 있는 T-트리에 대해 연구하고 그 결과로 CST-트리(Cache Sensitive T-트리)를 개발하였으며 CST-트리가 CSB+-트리보다 더 성능이 뛰어남을 보였다.

한편, 최근 컴퓨팅 업계는 컴퓨팅 솔루션에 막대한 영향을 미치게 될 획기적인 전환이 진행되고 있다. 그 중 대표적인 것이 단일 코어에서 멀티 코어 프로세서로의 전환이다. 멀티 코어 기술은 하나의 PC로 여러 가지 작업을 할 수 밖에 없는 멀티미디어 환경에서 뛰어난 성능 향상을 제공하고 있어 다중 처리를 기반으로 하는 홈 멀티미디어를 비롯해 동영상 편집 및 처리, 그리고 서버 등 다양한 분야에서 그 위력을 발휘할 것으로 보인다. 이 멀티 코어 프로세서 중에는 예전보다 훨씬 더 큰 용량의 L2캐시를 포함하거나, 여러 개의 코어가 한 개의 L2캐시를 공유하는 제품도 있다.

본 논문에서는 앞서 말한 메인 메모리 데이터베이스 시스템을 위한 인덱스 구조들이 최신 프로세서를 탑재한 시스템에서 어떠한 성능을 보이는지 의문을 갖고 이를 실험하였다. 그 결과로 T-트리, B+-트리, CSB+-트리, CST-트리의 Insert, Delete, Search 테스트의 실행시간과 Search 테스트의 캐시미스 수를 비교하여 정리하였고, CST-트리가 뛰어난 성능을 나타냄을 보인다.

본문은 다음과 같이 구성되어 있다. 2장에서 몇 가지 인덱스 구조를 소개하고 3장에서 최근 프로세서 기술 동향을 간단히 설명한다. 4장에서는 CST-트리와 기타 다른 트리와의 성능을 비교 분석하며, 5장에서 본 논문의 결론을 내리고 향후 과제에 대하여 기술한다.

2. 배경 연구

2.1 인덱스 구조

T-트리는 AVL 트리의 공간 낭비와 잦은 회전연산을 개선하기 위해 만들어진 인덱스 구조로서 AVL 트리가 하나의 노드에 데이터 한 개만을 가지는 대신 T-트리는 하나의 노드가 n개의 데이터를 가질 수 있도록 자료구조를 변경하였다. AVL 트리와 유사하게 T-트리 노드의 왼쪽 부분 트리에 있는 모든 데이터는 노드의 최소값 보다 작고, T-트리 노드의 오른쪽 부분 트리에 있는 모든 데이터는 노드의 최대값 보다 커야 한다. 또한 노드가 추가되거나 삭제되었을 때 AVL 트리와 동일한 알고리즘을 사용하여 트리의 밸런스를 유지한다.

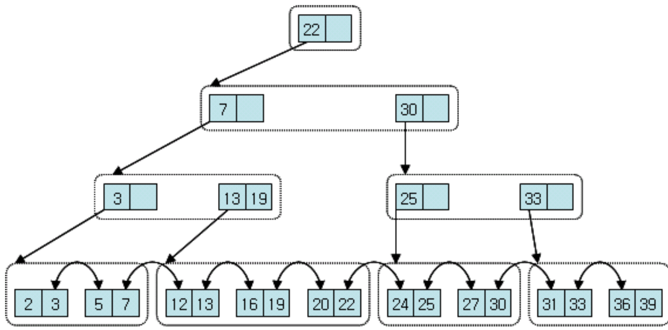


그림 1 CSB+ 트리 구조[2]

이 논문에서 CST-트리와 성능 비교를 위해 실험한 CSB+ 트리[2]는 B+트리와 유사한 구조이다. 그림1과 같이 모든 자식 포인터를 명시적으로 저장하지 않고 배열에 자식 노드들을 인접하게 넣어 한 노드의 같은 자식 노드들을 묶어 노드 그룹을 만들고 각 첫 번째 자식노드의 포인터만 저장한다. 다른 자식노드들은 첫 번째 자식노드 포인터로부터 오프셋을 더하여 구할 수 있다. 이 접근 방법은 포인터 사용을 줄임으로서 키값을 위한 더 많은 공간을 확보할 수 있어 더 좋은 캐시의 성능을 기대할 수 있다.

2.2 CST-트리: Cache-Sensitive T-트리

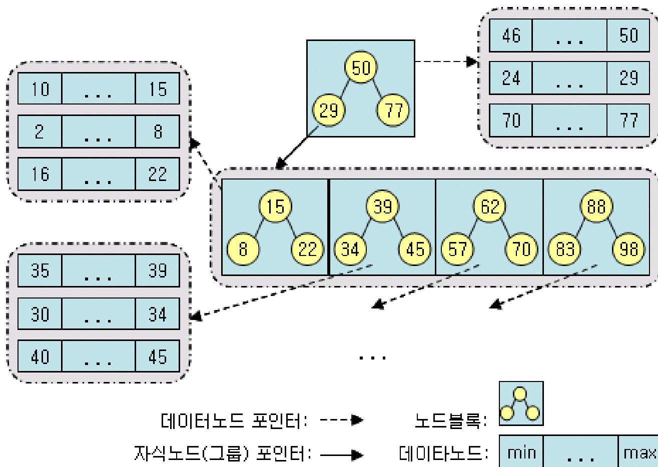


그림 2 CST-트리의 구성 예[3]

T-트리가 캐시를 효율적으로 사용하지 못하는 이유는 데이터 엑세스 패턴에 문제가 있기 때문이며, CST-트리는 이 문제를 고려하여 T-트리를 개선하였다[3].

첫째, 메모리의 특정 주소를 액세스 할 경우 캐시 블록 크기 만큼의 데이터가 메모리에서 L2 캐시로 옮겨지게 되는데 T-트리는 노드의 최소값과 최대값 만을 찾고자 하는 키 값과 비교한 후에 포인터를 따라 다음 노드를 액세스 하게 되므로 최소, 최대를 제외한 값은 사용되지 않는다. 따라서 최대값들만을 취해 트리를 구성하여 캐시 미스가 일어나면 이 최대값들이 최대한 캐시 메모리에 복사되어 있도록 하였다.

둘째, T-트리의 높이는 B-트리보다 높기 때문에 트리의 단말 노드까지 도달하는데 포인터에 의한 메모리 엑세스 횟수가 B-트리보다 많아 많은 캐시 미스를 유발한다. 따라서 CST-트리에서는 포인터를 사용을 최대한 줄이고, T-트리의 최대값들을 모아 배열로 이진트리를 구성하였다. 루트노드의 인덱스를 1이라 하면 부모노드와 자식노드 사이에는 다음과 같은 규칙이 성립한다.

- 부모 노드의 인덱스 = 인덱스/2
- 왼쪽 자식노드의 인덱스 = 인덱스 * 2
- 오른쪽 자식노드의 인덱스 = 인덱스 * 2 + 1

셋째, 인덱스 * 2 + 1 < 캐시 블록 크기 인 경우에는 자식 노드 역시 캐시에 복사되어 있으므로 캐시 미스가 발생하지 않지만 그 반대인 경우에는 항상 캐시 미스가 발생하게 되므로, T-트리 노드의 최대값을 캐시 블록크기 만큼 모아 이진트리를 구성하고 이 블록을 '노드 블록' 이라고 부른다. 노드 블록 내에서는 캐시 미스 없이 자식 노드의 최대값을 읽을 수 있다.

CST-트리 구조의 예를 보인 그림2 에서와 같이 노드 블록과 노드 블록 간의 연결은 포인터를 이용하며, 실제 값들로 연결하기 위해 최대값마다 포인터를 사용하는 대신에 [2]에서 제안한 포인터 제거 방법을 사용하여 상당한 공간을 절약하였다.

CST-트리의 삽입, 삭제 알고리즘은 트리 밸런싱 알고리즘을 제외하면 T-트리와 유사하기 때문에 검색 알고리즘만을 간단히 소개하겠다. 탐색키와 최대값을 비교하고 키가 최대값보다 크면 오른쪽 서브트리에서 search를 진행한다. 키가 최대값 보다 작으면 현재 노드에 찾을 키가 있던지 왼쪽 서브트리에 찾을 키가 있다. 현재 노드를 표시(mark)해 두고 서브트리에서 탐색을 진행한다. 단말 노드까지 도달했는데도 찾지 못했을 경우 마지막으로 표시해둔 노드에서 이진 탐색으로 키를 찾는다.

3. CPU 기술 동향

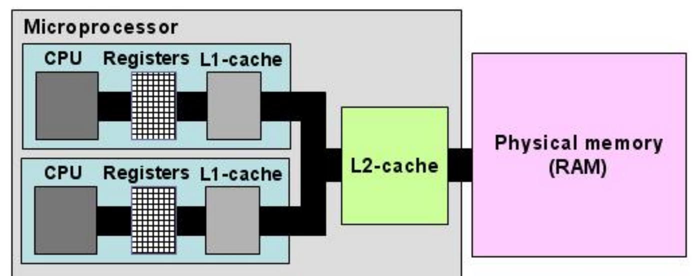


그림 3 듀얼코어 프로세서의 메모리 계층구조

그림 3은 최신 컴퓨터의 메모리 계층구조를 도식화 한 것이다. 최근의 컴퓨터는 프로세서 내에 cpu 클럭과 비슷한 속도로 동작하지만 메모리 크기와 블록 크기가 매우 작은 L1 캐시와 이보다는 약간 느리지만 메모리보다는 빠른 L2 캐시를 포함하고 있다. L2 캐시 미스가 발생하면 CPU에 비해 속도가

매우 느린 메모리로부터 데이터를 읽어 와야 하기 때문에 메모리 접근 병목 현상이 발생한다.

더욱이 요즘, CPU 트렌드는 core를 하나만 넣고 해당 core의 속도를 한 없이 높이는 것 보다는 성능이 적절한 core를 여러 개 박아서 병렬로 처리하는, 즉 멀티코어 CPU가 대세를 이루고 있다. 종전에는 하나의 프로세서가 여러 개의 작업을 도맡아 해 왔지만 멀티코어 플랫폼에서는 여러 개의 CPU가 여러 작업을 각기 나누어 처리하게 되므로 당연히 처리 속도가 빨라 질수밖에 없다. 이러한 뛰어난 성능의 CPU를 뒷받침 해주기 위하여 L2캐시의 활용 효율 또한 중요한 이슈가 되고 있다. 그림 5의 경우도 하나의 칩 안에 코어가 두 개인 듀얼 코어 프로세서를 표현한 것이다. 이 프로세서의 경우는 두 개의 코어가 한 개의 L2 캐시를 공유하여 코어 간의 작업 협력을 용이하게 하여 전체 성능을 향상 시킨 구조이다.

더 많은 수의 코어가 하이엔드 프로세서에 통합됨에 따라 여러 작업을 병렬로 처리할 수 있는 능력은 향후 10년 동안 계속해서 향상될 것으로 전망되고 있으며 이에 따라 병렬 처리 자원을 활용 할 수 있는 최적화 소프트웨어의 중요성도 높아지게 될 것이다. 그러므로 본 논문에서 수행한 하이엔드 프로세서를 사용하여 캐시를 고려한 인덱스 구조의 성능 평가는 중요한 의의를 갖는 다고 할 수 있다.

4. 성능 평가

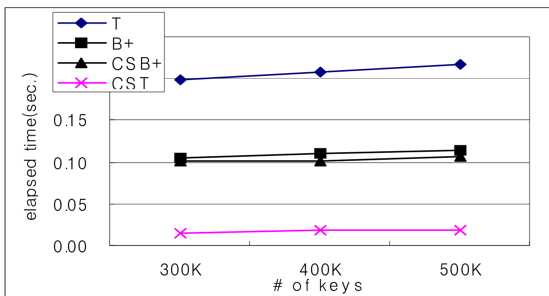
4.1 실험 환경

	Machine-A	Machine-B
멀티코어	No	Yes
CPU 프로세서 수	1	1
CPU 클록 속도	2.40GHz	2.66GHz
L2 캐시 <크기, 라인 크기>	<512K bytes, 64bytes>	<4096K bytes, 64bytes>
메인 메모리 크기	1.5G bytes DDR	2G bytes DDR2
운영체제	Redhat linux Enterprise	Redhat linux Enterprise

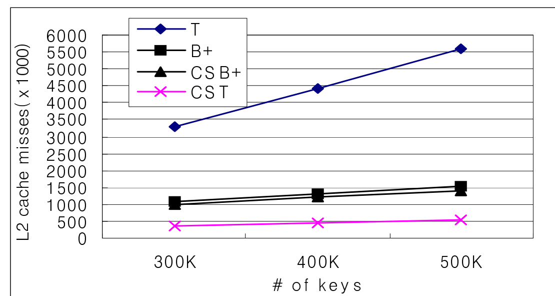
표 1 실험 환경

실험은 표1과 같은 두 종류 하드웨어 환경을 갖춘 컴퓨터에서 각각 진행되었다. Machine A는 2.40GHz의 싱글 코어 프로세서와 512KB의 L2 캐시 메모리를, Machine B는 2.66GHz의 듀얼코어 프로세서와 4096KB의 L2캐시 메모리를 탑재하였다. 실험에 사용된 L2 캐시 메모리의 라인 사이즈는 모두 64 바이트이며, 동일한 회사에서 생산된 제품을 선택하였다. 운영체제는 Redhat linux Enterprise를 사용하였고, L2 캐시 미스 수를 구하기 위해서 Valgrind사에서 개발한 Valgrind의 Cachegrind 툴을 이용하였다[6].

실험에 사용된 인덱스 구조는 CST, B+, CSB+, T 트리이며 이중 CST 트리는 [3]의, B+ 와 CSB+트리는 [2]의, T 트리는 [3]의 저자가 개발한 원 소스를 Machine-A,B의 실험에 사용한 환경에 맞도록 수정하였으며 사용한 컴파일러는 GCC이다. 삽입 시에 CST-트리가 노드 그룹을 위해 선행 메모리 할당을 하므로 공평성을 위하여 CSB+-트리 대신 선행할당을 하는 CSB+_full-트리를 사용하였다[2].

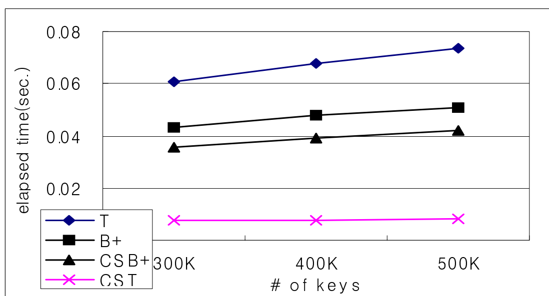


(a) 실행시간

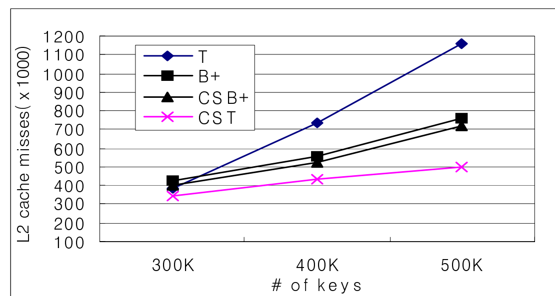


(b) L2캐시 미스 횟수

그림 4 Machine-A의 검색 실험 결과

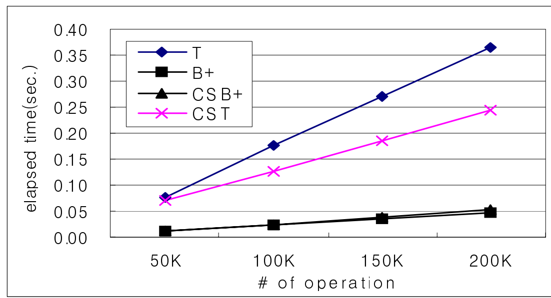


(a) 실행시간

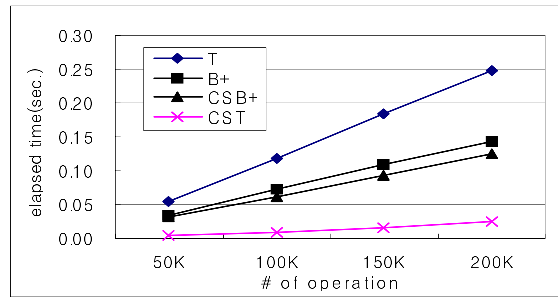


(b) L2캐시 미스 횟수

그림 5 Machine-B의 검색 실험 결과

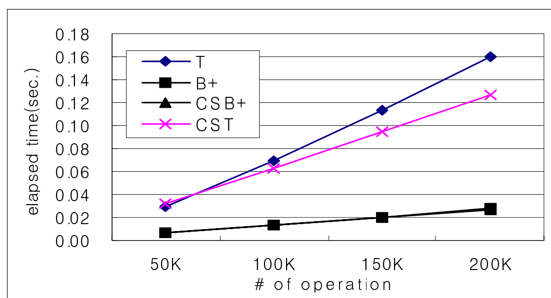


(a) Insertion

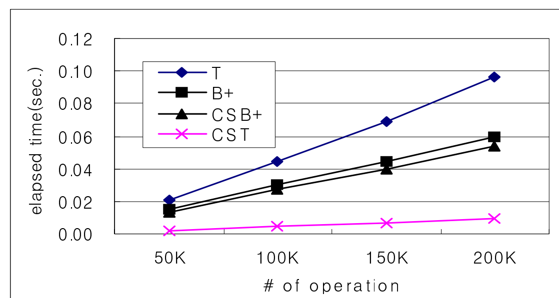


(b) Deletion

그림 6 Machine-A의 실행시간



(a) Insertion



(b) Deletion

그림 7 Machine-B의 실행시간

4.2 실험 결과

그림 4는 Machine-A에서 트리가 각각 300,000, 400,000, 500,000 개의 키를 가지고 있을 때 200,000 번의 임의 검색을 수행하는 동안의 실행시간과 캐시 미스 수를 그래프로 표시한 것이다. 그림 4 (a)의 실행시간 성능은 CST, CSB+, B+, T 트리 순으로 좋은 결과를 보이며, L2캐시 미스 수 또한 유사한 결과를 보이고 있다.

그림 5은 위의 그림 4에서와 같은 실험을 Machine-B에서 수행한 결과를 그래프로 표시한 것이다. 실행시간과 L2캐시 미스 수가 그림 4의 결과에 비하여 매우 작으나, 다른 트리들에 대한 CST-트리의 성능은 유사한 뛰어난 결과를 보이고 있다.

그림 6은 Machine-A에서 트리에 1,000,000개의 키를 벌크로딩하고 50,000, 100,000, 150,000, 200,000 개의 키를 임의로 생성하여 삽입, 삭제했을 때 걸린 시간을 그래프로 표시한 것이다. 그림 6 (a)의 삽입 성능은 B+-트리가 가장 좋은 성능을 보이고 있으며, CSB+-트리가 그 다음으로 좋은 성능을, CST-트리는 CSB+-트리와 가장 성능이 좋지 않은 T-트리의 중간 정도 성능 결과를 보이고 있다. CST-트리의 삽입 시 성능이 좋지 않은 이유는 회전 연산 오버헤드가 CSB+-트리에 비해 크기 때문이다[3]. 그림 6 (b)는 삭제 연산 결과를 보인 것으로서 CST-트리의 성능이 가장 좋은 결과를 보이고 있다. CST-트리의 삭제 연산은 그 키가 존재하는 데이터 노드를 찾기 위한 시간이 대부분이므로 검색 연산과 비슷한 결과를 보이고 있다. 사용된 B+, CSB+, CST-트리 모두 지연 삭제 알고리즘을 사용하였다.

그림 7는 위 그림 8과 같은 실험을 Machine-B에서 수행한 결과를 그래프로 표시한 것이다. 그래프 형태는 그림 6과 굉장히 유사하다.

5. 결론 및 향후 과제

T-트리, B+-트리, CSB+-트리, CST-트리와 같은 메인 메모리 시스템을 위한 인덱스 구조들을 최신 프로세서가 탑재된 시스템에서 검색, 삽입, 삭제 등의 성능을 평가하였다. 그 결과 캐시를 고려한 트리가 일반 트리 구조에 비해서 우월한 성능을 지닌 인덱스 구조임을 보였다. 이 실험은 과거 다른 컴퓨팅 환경에서 제시되고 구현된 인덱스 구조를 최신식 컴퓨팅 시스템에서 실험 평가하여 실질적으로 CST-트리나 CSB+-트리가 메모리 시스템을 위한 인덱스 구조 중에서 유용함을 보였다는 데에 큰 의의가 있다.

본 논문에서 실행한 실험 결과에 대해 추가적인 연구가 필요하다. 트리 구조와 실험 결과와의 관계, 그리고 성능 향상 정도에 대한 분석이 추후에 진행되어야 할 것이다.

참 고 문 헌

[1] S. Manegold, P. A. Boncz and M, L. Kersten, Optimizing database architecture for the new bottleneck: memory access, 2000, VLDB Journal, Vol.0, No.3, pp.231-246, 2000.

[2] Jun Rao, et al, Making B+ Trees Cache Conscious in Main Memory, 2000 SIGMOD.

[3] 이익훈, 김현철, 허재녕, 이상구, 심준호, 장준호, “캐시를 고려한 T-트리 인덱스 구조”, 정보과학회논문지: 데이타베이스, 제 32권 제 1호 (2005.2), p12-23, 한국정보과학회, 2005.

[4] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.-K. Chen, and P. Dubey, “Cache-conscious frequent pattern mining on modern and emerging processors”, The VLDB Journal, Vol. 16, No. 1, pp.77-96, 2006.

[5] S. Manegold, P. A. Boncz and M. L. Kersten, “Optimizing database architecture for the new bottleneck: memory access,” The VLDB Journal, Vol.9, No.3, pp231-246, 2000.

[6] http://www.valgrind.org/