한국산학기술학회논문지
Vol. 11, No. 1, pp. 166-174, 2010

플래시메모리-SSD의 인덱스 연산 성능 향상을 위한 압축된 핫-콜드 클러스터링 기법

변시우[1*]
[1]안양대학교 디지털미디어공학과

A Compressed Hot-Cold Clustering to Improve Index Operation Performance of Flash Memory-SSD Systems

Siwoo Byun[1*]
[1]**Dept. of Digital Media, Anyang University**

요 약 SSD는 데스크탑 및 이동형 컴퓨터의 저장 장치를 지원하는 우수한 미디어이다. SSD는 비휘발성, 낮은 전력 소모, 빠른 데이터 접근 속도 등의 특징으로 데스크탑 및 서버용 데이터베이스의 핵심 저장 요소가 되었다. 하지만, 일반 RAM 메모리에 비하여 상대적으로 느린 연산 특성을 고려하여 기존의 전통적인 인덱스 관리 기법을 개선할 필요가 있다. 이를 위하여, 본 논문은 CHC-Tree 라고 하는 압축된 핫-콜드 클러스터링에 기반하는 새로운 인덱스 관리 기법을 제안한다. CHC-Tree는 인덱스 노드를 핫-콜드 세그먼트로 분류하며, 인덱스 노드의 키와 포인터를 압축한다. 콜드 세그먼트의 비활용노드의 오프셋 압축으로 느린 쓰기연산의 부담을 줄인다. 또한, 실험 결과를 통하여 기존의 B-Tree 기반의 인덱스 관리 기법보다 인덱스 검색 연산에서 26%, 인덱스 수정 연산에서 23% 이상 우수함을 확인하였다.

Abstract SSDs are one of the best media to support portable and desktop computers' storage devices. Their features include non-volatility, low power consumption, and fast access time for read operations, which are sufficient to present flash memories as major database storage components for desktop and server computers. However, we need to improve traditional index management schemes based on B-Tree due to the relatively slow characteristics of flash memory operations, as compared to RAM memory. In order to achieve this goal, we propose a new index management scheme based on a compressed hot-cold clustering called CHC-Tree. CHC-Tree-based index management improves index operation performance by dividing index nodes into hot or cold segments and compressing pointers and keys in the index nodes and clustering the hot or cold segments. The offset compression techniques using unused free area in cold index node lead to reduce the number of slow erase operations in index node insert/delete processes. Simulation results show that our scheme significantly reduces the write and erase operation overheads, improving the index search performance of B-Tree by up to 26 percent, and the index update performance by up to 23 percent.

Key Words : Hot-cold segment clustering, Tree indexing, Index compression, Flash memory

1. 서론

최근 각종 소형 정보기기들이 대중화됨에 따라, 정보 저장용 미디어로 플래시 메모리가 보편적으로 활용되게 되었다. 또한, 플래시 메모리의 지속적인 가격 하락으로 일반 사용자용 데스크톱이나 전산 센터내의 데이터 서버에서도 플래시 메모리 기반의SSD(Solid State Disk)가 활용되기 시작하였다. 출시 당시 겨우8GB 정도의 저 용량

이 논문은 2008년 정부(교육과학기술부)의 재원으로 한국학술진흥재단의 지원을 받아 수행된 연구임 (KRF-2008-313-D00851).
*교신저자 : 변시우(swbyun@anyang.ac.kr)
접수일 09년 09월 10일 수정일 09년 12월 11일 게재확정일 10년 01월 20일

의 SSD가 16→32→64→128→256GB로 불과 2년 사이에 매우 빠르게 대용량화 되고 있다는 사실이다. 아래 그림은 서버에 내장되는 대표적인 저장장치인 하드디스크와 이를 대체하고 있는 플래시 메모리 저장 장치(SSD)의 예이다.

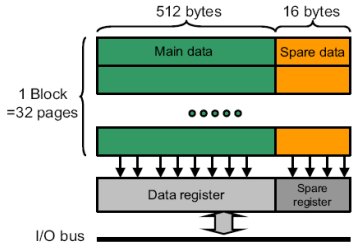
[그림 1] SSD와 기존 하드 디스크 저장장치

따라서 범용 정보기기에서의 기존의 하드 디스크나 메모리 저장 장치와 더불어, 플래시 메모리 기반 데이터 저장 시스템은 향후 상당히 유망하며, 이와 관련된 데이터 관리 기술의 개발이 매우 시급하다고 할 수 있다. 이러한 저장 미디어에 대한 성능상의 차이는 다음 표1.1 과 같다.

[표 1] 다양한 저장 장치의 성능 비교

저장장치	I/O	읽기	쓰기	소거
DRAM	Byte	60 ns (1B)	60 ns (1B)	-
Hard Disk	Page	8.9 ms (512B)	8.9 ms (512B)	-
NOR Flash	Byte	150 ns (1B)	200 ns (1B)	1 s (128KB)
NAND-MLC Flash	Page	20 ㎲ (512B)	300 ㎲ (512B)	2 ㎳ (16KB)

플래시 메모리는 영구 저장이 가능하다는 측면에서 하드 디스크와 유사하지만, 내충격성, 휴대성, 접근속도, 무진동성, 무소음성, 부품크기 측면에서 비교할 수 없을 정도로 매우 우수하다. 전력 소모량도 10㎃ 정도인데, 이는 저 전력 메인 메모리의 8분의 1정도에 불과하다. 또한, 메인 메모리와 비교하면, 메인 메모리의 최대 약점인 영구저장 불가능성(휘발성)을 해결했다는 측면에서 더 우수하므로, 차세대 기억장치라고 한다. 대부분의 소형 정보기기에서는 공간제약, 전력소모, 중량 및 내충격성 문제로 하드 디스크는 사용할 수 없기 때문에, 비휘발성 저장장치로서 플래시 메모리를 반드시 사용하여야만 한다. 다만, 일반 메인 메모리와는 달리, 쓰기와 소거 연산에 상당히 많은 시간이 소요되며, 쓰기 횟수가 최대 100,000에서 1,000,000번 정도로 제한되는 고유한 특성(약점)이 있다[1].



[그림 2] 플래시메모리의 내부 구조

플래시 메모리는 일반 메인 메모리와 달리 바로 쓰기 연산을 수행할 수 없고, 이전에 미리 블록 단위로 포맷 개념의 소거 연산을 수행한 후 쓰기 연산을 수행할 수 있다. 읽기 및 쓰기 연산도 전통적인 메인 메모리처럼 워드나 바이트 단위가 아니고, 위 그림의 예[2]와 같이 페이지 단위로 수행된다.

이러한 플래시 메모리 기반의 데이터 저장 장치를 통하여 신속하게 저장하고 효율적으로 검색을 위해서는 플래시 메모리의 특성을 고려한 효율적인 색인 구조와 저장 기법이 필요하다. 그 이유는 플래시 메모리에는 기존의 하드 디스크나 메인 메모리와는 전혀 다른 다음의 특성(약점)들이 존재하기 때문이다.

첫째, 읽기 연산의 경우에는 플래시 메모리의 연산 처리 속도가 하드 디스크에 비하여 800배 정도로 매우 빠르며, 일반 RAM 메모리에 비해서는 좀 느리지만 10㎲ 정도로 빠르므로 접근 시에 별 문제가 없다[3, 4]. 하지만, 플래시 메모리의 쓰기 연산의 경우에는 속도가 읽기 연산 대비 20배 정도의 많은 시간이 소모되어 매우 느리며, 메인 메모리처럼 Update-In-Place(제자리 덮어 쓰기)가 불가능한 Update-Out-Place 구조이다.

둘째, 하드 디스크나 메인 메모리는 쓰기 횟수가 거의 제한이 없는 반영구적 수명을 가지고 있는 반면에, 플래시 메모리는 쓰기 횟수가 최대 1,000,000번 정도로 수명이 제한된다.

이러한 단점들로 인하여, 기존의 인덱스 관리 기법들은 플래시 메모리에 쉽게 적용이 되지 못하였다. 따라서 플래시 메모리의 제약점을 효과적으로 극복할 뿐 아니라, 그 고유의 장점을 효과적으로 활용하는 새로운 인덱스 관리 구조와 저장 시스템의 연구가 필요하다.

2. 연구 배경

현재의 데이터 저장 시스템에서 사용되는 일반적인 색인 기법에 대한 기존의 연구를 분류해 보면, 크게 디스크 기반 색인 시스템과 메인 메모리 기반 색인 시스템으로 접근 방향을 나눌 수 있다. 저렴하고 대용량인 디스크 기반 저장 방식이 저장 비용 측면에서는 유리하지만, 아무리 I/O 버퍼를 많이 할당하더라도 속도 측면에서는 메모리 기반 저장 방식보다는 매우 느리다. 그러나 두 방식 모두 플래시 메모리 기반 저장 환경에는 부적합하므로 플래시 메모리의 고유한 특성을 고려하여 개발하여야 한다.

2.1 디스크 기반 색인 시스템:

디스크 기반 색인 및 저장 시스템에서는 검색 시에 디스크 접근을 최소화하기 위하여 노드의 크기를 디스크 페이지와 같은 크기나 같은 배수로 설정하고, 되도록이면 많은 엔트리를 한 노드에 넣어야 유리하다. 한 노드에 많은 엔트리가 들어갈 경우 모든 엔트리를 검색해야 하기 때문에 검색 시에 연산 처리 성능은 당연히 저하된다. 그러나 디스크 기반 저장 시스템에서는 이러한 검색 성능 저하보다는 디스크 접근에 의한 성능 저하가 훨씬 더 크기 때문에 한 노드에 많은 엔트리를 넣는 방향으로 설계한다[5]. 주로 B-Tree, B^{+}-Tree계열의 인덱스가 많이 사용되며[6, 7], 공간 색인으로는 R-Tree, R^{*}-Tree[8]계열이 사용되고 있다.

R-Tree계열의 인덱스는 일반적으로 디스크 기반 색인으로서 메인 메모리를 사용하지 않는 공간 인덱스로 구현되었다. R-Tree는 삽입 연산, 삭제 연산, 분할이나 병합과 같은 리벨런싱 연산이 수행될 경우, 동일한 위치로 많은 섹터가 판독 또는 재기록 되는 부담이 있다. 디스크 기반 시스템에서는 이러한 연산들의 검색 효율을 위하여 디스크의 연속 섹터에 그룹핑되어 있으며, 디스크 특성을 잘 고려한 R-Tree는 디스크 기반 시스템에서 실제로 매우 효과적이다. 그러나 디스크 병목현상이 자주 발생한다면, 메인 메모리 색인이 필요하며, 디스크와 메인 메모리를 합친 통합 기법도 필요하다[9].

2.2 메인 메모리 기반 색인 시스템:

메인 메모리 기반 색인 및 저장 시스템은 디스크 기반 시스템에 비하여 디스크에 접근하는 시간이 대폭 줄어들기 때문에 훨씬 더 빠른 성능을 보여줄 수 있다. 그러나 문제는 시스템 전원이 차단되는 등의 치명적인 장애가 발생할 경우이다.

메인 메모리 기반 색인 시스템에서는 일반적으로 T-Tree가 1차원 데이터를 위한 색인으로서 좋은 성능을 보이며, 비교적 적합하다고 알려져 왔다[10]. T-Tree는 이진 검색과 높이 균형을 가지고 O(logN)의 트리 순회가 가능한 AVL-Tree의 빠른 검색 특성을 가지고 있으며, 한 노드 안에 여러 개의 데이터를 가지고 저장효율이 좋은 B-Tree의 성질도 함께 가지고 있다. T-Tree는 빠른 처리 속도와 메모리 사용의 최적화라는 메인 메모리의 특성에 적합한 구조로 알려져 있다[9]. 그러나 [11]에서 T-Tree는 동시성 제어에 대한 고려가 매우 부족하였으며, 이를 고려한다면 B-Tree가 성능을 추월할 수 있음을 밝혔다.

디스크 기반 색인은 깊이가 얕고 넓게 퍼진 트리를 써서 삽입/검색 시에 I/O 비용을 최소화 하였다. 반면에, 메모리 기반 색인의 접근 비용은 포인터로 노드의 메모리 주소를 획득하는 비용이므로 크지 않다. 따라서 디스크 기반 색인에서 선호하는 얕고 넓게 퍼진 트리 구조는 더 이상 유용하지 않다. 메모리 기반 색인에서는 디스크 기반 색인과는 달리 노드의 용량을 변화시켜 트리의 깊이와 비교횟수를 조절하여 성능 향상이 가능하다[9].

3. 플래시 메모리 저장 시스템을 위한 효율적인 인덱스 저장 및 관리

본 논문에서는 디스크 기반 및 메인 메모리 기반 색인 중에서 가장 보편적인 B-Tree 색인을 근간으로 하여 플래시 메모리 저장 시스템에 적합한 색인 저장 기법을 연구하였다. 메인 메모리 데이터베이스에서 많이 사용되는 T-Tree도 가능하지만, 동시성 제어를 고려할 경우 B-Tree보다 성능이 낮은데, 그 이유는 메모리 보다 CPU의 발전 속도가 더 빠르므로, 상대적으로 접근수가 적은 B-Tree가 유리하기 때문이다[11].

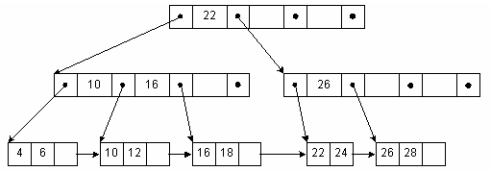
또한, T-Tree도 B-Tree에서 파생되어 B-Tree의 속성을 가지고 있으며, 실제 메모리 데이터베이스에서 B-Tree도 많이 사용된다. 또한, 기존의 디스크 기반 데이터베이스에서도 B-Tree가 대부분 활용된다는 관점에서 본 논문에서는 B-Tree에 기초하였고, B-Tree 중에서도 더 진보되어 보편적인 B^{+}-Tree를 대상으로 하였다. 그 이유는 B^{+}-Tree는 B-Tree와는 달리 중간 노드에 데이터 관련 정보를 넣지 않고 리프 노드에서 저장하므로, 상대적으로 색인 자체의 저장 효율이 높아지기 때문이다. 또한, 우선적으로는 B^{+}-Tree에 적용하였지만, 제안 기법을 B-Tree를 포함한 일반적인 트리 구조의 색인에도 적용이 가능하다.

이러한 관점에서 본 연구에서는 메모리 기반 및 디스크 기반 데이터베이스에서 근간이 되는 B^{+}-Tree를 본 플

래시 메모리 저장 시스템에 적합하게 개선하여, RAM 메모리에 비하여 매우 느린 쓰기 연산과 지우기 연산의 부담을 줄이고, 성능을 개선할 수 있는 효율적인 색인 저장 기법인 CHC-Tree (Compressed Hot-cold Cluster Tree)를 제안한다. CHC-Tree 기법의 핵심은 핫-콜드 분리 기술과 콜드 세그먼트의 비활용 영역의 압축 기술로 구성된 압축된 핫-콜드 클러스터링 기법이다.

3.1 압축된 핫–콜드 클러스터링 기법

B-Tree 계열의 색인은 데이터의 삽입, 삭제, 검색을 효율적으로 처리하기 위하여 가장 널리 사용되는 색인 구조이다. B-Tree의 삽입, 삭제, 리벨런싱은 많은 노드들이 읽혀지고 쓰여 지게 한다. 기본적으로 B-Tree는 하위 노드에 대한 포인터 정보와 함께 데이터 관련 정보를 한 노드 안에 보관한다. 그러나 아래 그림에서 보는 바와 같이 B^+-Tree는 중간 노드에 데이터 관련 정보를 넣지 않고 리프 노드에 이를 저장한다. 즉, 중간 노드에서는 순수한 색인 정보만을 저장하므로, 색인 자체의 저장 효율이 높아진다. 또한, 리프 노드에서는 색인 검색의 도움 없이 바로 순차적인 접근이 가능한 장점도 있다. 따라서 플래시 메모리의 색인으로서는 기본적인 B-Tree 보다도 B^+-Tree가 더 적합하다.



[그림 3] B^+-Tree 색인의 예

그리고 B^+-Tree에서 수많은 임의의 값들을 삽입하고 삭제하는 시뮬레이션을 수행하여 분석한 결과 69%정도 차 있을 때가 가장 효율적이다. 한 노드에 최대한 많이 저장하면 검색 노드수도 줄어들고 저장 효율은 향상되지만, 삽입, 삭제시 노드의 변동이 너무 빈번하여 많은 수의 쓰기 연산을 유도하여 결과적으로 더 손실이 크다.

즉, 평균점유율(average fill factor)이 69%일 때 가장 안정되어 인덱스 리벨런싱과 재구성을 하지 않고도 인덱스 연산을 수행할 수 있다. 따라서 인덱스 구성시 이 평균점유율에 맞추어 B^+-Tree를 조직하게 된다. 이러한 분석 결과와 기타 B^+-Tree에 대한 자세한 이론은 [6]에 설명되어 있다.

그러나 전술한 바와 같이, 플래시 메모리에 기존 B^+-tree를 그대로 적용하게 되면, 느린 쓰기 연산으로 인하여 심각한 성능저하를 야기할 수 있다. 따라서 쓰기와 지우기 연산의 횟수를 최대한 줄이는 것이 매우 중요하다. 이러한 최소 쓰기 목적을 위하여, 본 연구에서는 핫-콜드 분류 클러스터링 및 콜드 세그먼트 압축 기법을 사용한다.

또한, 전술한 바와 같이 B^+-Tree색인은 효율을 위하여 한 노드 당 평균 69%정도만을 엔트리(키와 포인터)공간으로 점유하고 있다. 이때, 그 노드의 나머지 31%의 저장 공간은 추후 발생 가능한 엔트리 삽입을 위하여 빈 공간으로 계속 유지하고 있다. 즉, 추후 삽입을 위하여 계속 낭비되고 있는 공간이다.

본 연구에서는 인덱스 노드를 접근 패턴에 따라서 자주 수정되는 “핫-노드”와 드물게 수정되는 “콜드 노드”로 두 가지로 분류한다. 각 인덱스 노드는 이러한 핫-콜드 분류를 위하여 수정빈도시간을 저장하고 있다. 또한, 인덱스 영역을 핫 클러스터와 콜드 클러스터로 분류한다. 핫-클러스터는 핫-노드만을 포함하는 핫-세그먼트들로 구성되어 있다. 마찬가지로 콜드-클러스터는 콜드-노드를 포함하는 여러 개의 콜드-세그먼트로 클러스터링 되어 있다. 본 연구에서는 핫-클러스터와 콜드-클러스터의 비율은 1:2로 설정하였는데, 이는 핫-클러스터의 크기는 콜드-클러스터의 반이면서, 두 배로 빈번히 수정된다는 의미이다. 개별 사용자 환경에 따라서 성능최적화를 위하여 두 클러스터간의 비율은 다르게 조정 가능하다.

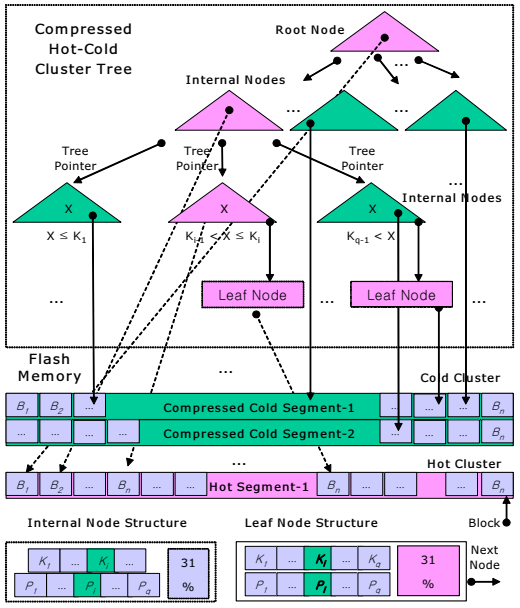
플래시 메모리는 쓰기 연산 전에 지우기 연산을 수행하여, 빈 세그먼트를 확보한 후 실제 쓰기 연산을 수행된다. 이때, 이러한 빈 세그먼트를 확보하기 위하여 세그먼트 단위로 클리닝이 수행된다. 만일, 세그먼트 클리닝을 수행하지 않으면, 플래시 메모리 상에서 조각난 데이터들이 여기 저기 불필요하게 떠다니면서 그 데이터가 속한 세그먼트들의 업데이트 횟수를 증가시키기 때문이다.

본 연구에서는 이러한 데이터의 접근 패턴을 분석한 후, 정적인 속성의 데이터는 정적인 콜드 세그먼트에 클러스터링하고, 빈번히 수정되는 동적인 속성의 데이터는 동적인 핫 세그먼트에 저장하여, 전체적으로 세그먼트 업데이트 횟수를 줄이고자 한다. 보통 데이터가 수정된 블록을 오염된(dirty) 블록이라고 하는데, 세그먼트 내에 더티 블록이 모이면 클리닝을 수행하게 된다. 비유를 하자면, 활동이 심한 개구쟁이들을 한 방에 넣고, 정적인 아이들을 다른 방에 분류하여 배치하면, 방 청소횟수를 줄일 수 있는 것과 같다.

여기서, 콜드 세그먼트내의 데이터는 내부적으로는 압축되어, 고가의 플래시 메모리의 저장 공간을 늘려줌과 동시에 세그먼트의 클리닝시 압축효과로 한 번에 더 많은 더티 데이터를 클리닝하게 된다. 다만, 핫 세그먼트는 매우 빈번히 수정되어 콜드 세그먼트에 비하여 압축의

효용성이 낮으므로, 압축하지 않고 그대로 저장된다. 이러한 핫-콜드 압축 클러스터링을 통한 세그먼트 클리닝 감소로 인하여, 결과적으로 인덱스 연산의 성능이 향상되고, 더불어 플래시 메모리의 수명도 연장된다.

콜드 세그먼트내의 각 인덱스 노드에 대하여 각 키와 포인터 필드의 배열을 오프셋 압축하여, 52%의 압축이 가능하였다. 압축률과 압축속도는 레벨에 따라 효율적으로 조절이 가능하며, CHC-Tree의 인덱스 구조는 다음과 같다.



[그림 4] CHC-Tree의 인덱스 저장 구조

본 실험에 사용된 압축 알고리즘은 명료하고 공개적으로 쉽게 구할 수 있는 LZO1X 압축 기법이다. 이 알고리즘의 성능은 웹에 공개되어 있다. 본 실험에서 트리 샘플로서 압축률을 측정해보니 중간 노드는 55.6%이고, 리프 노드는 54.7%로 나타났다. 물론 이 보다 더 압축률이 좋은 알고리즘이 많이 있으므로, 추후에 더 성능을 개선할 수 있으나, 본 연구의 편의상 프로그램 코드가 공개되어 있는 LZO1X 압축 기법을 사용하였으며, 압축레벨을 조절함으로써 압축률과 압축속도를 제어 가능하다. 여기에 키와 포인터를 분리하여, 각각 재 정렬하고, 오프셋을 압축함으로써 압축 성능을 LZW대비 약 8% 더 개선할 수 있었는데, 결과적으로 인덱스 접근성능이 향상되게 된다.

[표 2] LZO 기법의 레벨별 압축 성능

압축 레벨	압축 비율 %	압축 속도 (KB/s)	복원 속도 (KB/s)
LZO-Low	50.4	4565.53	15438.34
LZO-2	51.6	4297.33	15492.79
LZO-3	52.2	4018.21	15373.52
…	…	…	…
LZO-9	55.0	1677.06	15069.60
LZO-High	56.4	1286.87	15656.11
LZO-Optimal	60.9	232.40	16445.05

실제 시스템의 CPU와 RAM의 접근 속도는 플래시 메모리보다 훨씬 빠르므로, 압축/복원의 부담은 성능향상효과에 비하면 상대적으로 매우 작다. 이 이유는 전술한 바와 같이, 주요 병목 원인은 CPU 와 RAM의 속도가 아니고, 플래시 메모리의 매우 느린 지우기와 쓰기 속도에 있기 때문이다.

또한, CHC-Tree에서 압축을 하는 이유는 한 노드에 더 많은 엔트리(키값과 포인터)들이 저장시켜서 비싼 플래시 메모리의 저장 효율을 높이는 이유도 있지만, 더 중요한 이유는 압축 저장을 함으로써, 결과적으로 플래시 메모리의 지우기 및 쓰기 연산 횟수를 절반으로 가까이 줄일 수 있기 때문이다. 이는 결과적으로 전체적인 트리 연산의 입출력 처리 성능을 높이게 된다.

3.2 CHC-Tree의 관리

CHC-Tree에서 검색, 삽입, 삭제 연산은 인덱스 관리자(Index Manager:IM)에 의하여 수행된다. IM 은 트리 검색을 위하여 CHCT_Search()함수를, 삽입을 위하여 CHCT_Insert()를, 삭제를 위하여 CHCT_Delete()를 수행시킨다. 또한, 이러한 함수의 수행중에 수반되는 플래시 메모리상에 압축 인덱스의 쓰기연산을 위하여 FM_Write()를, 압축 인덱스의 읽기 연산을 위해서는 FM_Read()를 수행하게 된다. 트리 노드에 대한 압축 쓰기는 중간 노드와 리프 노드에서만 수행되며, 루트 노드는 너무 빈번히 접근되어 병목현상이 발생할 수 있으므로, 압축 저장하지 않는다.

4. 실험 및 성능 평가

본 연구에서 제안된 기법의 성능을 검증하기 위하여 시뮬레이션을 수행한 후 그 결과를 분석해 보았다. 본 시뮬레이션 수행시 CHC-Tree색인 기법과 비교된 대상 색

인은 가장 보편적으로 사용하는 B^+-Tree(BTR)기법이다. 실제 실험을 대신하여 컴퓨터 시뮬레이션 방식을 통한 모의실험을 선택한 이유는 다음과 같다.

플래시 메모리는 거의 표준화 되어 있는 디스크나 메모리와는 달리 다양한 방식과 종류가 존재한다. 생산 방식에 따라서 NAND 방식, NOR 방식 등의 유형으로 나뉘며, 동일한 방식이라도 삼성, 도시바 등 제조회사에 따라서 특징이 다르다. 또한, 메모리 용량 변화와 페이지 크기, 블록 크기 등의 변화와 같은 다양한 실험을 수행할 수 없다. 그리고 시스템 중간 과정에서 발생하는 페이징, 클리닝, 가비지 수집 부하, 캐싱, 프리패칭, 버퍼링, 카피 오버헤드 등의 다양한 간섭 요소가 존재하여 실제 알고리즘의 효과를 일관성 있게 측정하기가 어렵다. 그리고 플래시 메모리의 속도 개선을 위하여 고속고가의 저장 장치인 SRAM을 결합하거나 버스, 컨트롤러 등의 결합으로 성능이 원래보다 상향된 형태도 많으므로 역시 일관된 측정에 어려움이 있다.

또한, 프로세서 수행 시간에 비하여 상대적으로 느린 하드 디스크와는 달리 플래시 메모리는 빠른 수행 시간을 가지므로, 실제 시스템에서 실험 도중에 간섭되는 빈번한 이벤트의 발생, 커널의 다양한 인터럽트 처리, 멀티태스킹에 의한 프로세서의 문맥 교환(context switching) 때문에 정확한 결과 값을 얻기가 어렵게 된다. 이와 같은 이유로 시뮬레이션을 사용하는 것이 더 융통성이 있으며 경제적이다[12].

4.1 시뮬레이션 환경 구성

본 시뮬레이션 수행시 CHC-Tree 색인 기법(CHCTR)과 비교된 대상 색인은 가장 보편적으로 사용하는 B^+-Tree기법(BTR_N)이며, 전술한 바와 같이 최적성능이 나오는 트리 노드의 평균 점유율(69%)을 기본으로 하였다. 또한 트리가 거의 차 있는 최고밀도 저장의 경우(97%)의 B^+-Tree기법(BTR_F)도 포함하여, 총 세 가지 색인을 비교 시험하였다. 실험 도구는 CSIM discrete- event simulation software [13] 시뮬레이션 언어와 Microsoft Visual C++를 사용하였다. 실험이 수행된 하드웨어 환경은 펜티엄 쿼드CPU와 메인 메모리 2G, 하드디스크 500G이며, 운영체제는 윈도우 2003 서버를 사용하였다.

다음으로 플래시 메모리 데이터베이스를 위한 시뮬레이션 모듈이 필요한데, 사용된 모델은 CSIM에서 제공되는 폐쇄형 큐잉 모델(Closed Queuing Model)이다. 이 모델을 통하여 초당 일정한 수의 읽기 쓰기의 인덱스 연산을 생성하고, 이와 연관된 인덱스 트리가 검색 또는 수정되면서 발생한 시스템의 처리 시간과 성능을 측정하면 된다.

플래시 메모리 데이터베이스 운영 환경을 위한 시뮬레이션의 주요 성능 평가 지표는 시스템의 인덱스 연산에 대한 연산 처리치(throughput)와 응답시간(response time)이다. 이러한 연산 처리치는 초당 몇 개의 인덱스 연산이 처리되었는지를 의미하고, 응답시간은 인덱스 연산을 요청한 후 수행완료까지의 걸린 시간을 의미한다.

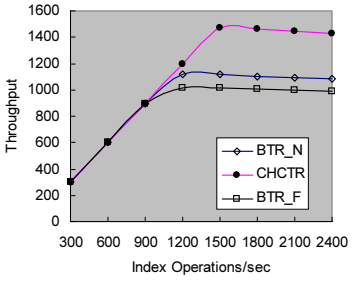
주요 시뮬레이션 파라미터는 초당 생성된 인덱스 연산의 수, 읽기 연산 수행시간, 쓰기 연산 수행 시간, 소거 연산 수행시간 등이다. 초당 생성된 인덱스 연산의 수는 검색연산의 경우는 1500개에서부터 500개 단위로 5,000개까지 변화시켜 보았으며, 갱신 연산의 경우는 300개에서부터 300개 단위로 2,400개까지 변화시켜 보았다. 이는 모두 시뮬레이션 시스템에 가해지는 작업 부하를 의미한다. 플래시 메모리의 읽기 연산 수행 시간은 16μs로 설정하였고, 쓰기 연산 수행 시간은 250μs로 설정하였으며, 소거 시간은 세그먼트당 2㎳로 설정하였다. 각 연산 수행 시간은 기존의 연구[1]와 제품 홈페이지[14]에서 제시한 자료이다.

본 연구에서 핫 클러스터와 콜드 클러스터의 설정 비율은 전술한 데로 1:2이므로, 상대적인 hot-ratio 값은 33%로, hot-ratio값은 67%로 설정하였다. 또한, 전체 인덱스 접근 연산에서 검색 연산이 50%를 차지하고, 삽입 연산이 25%, 삭제연산이 25%로 하여 구성된다.

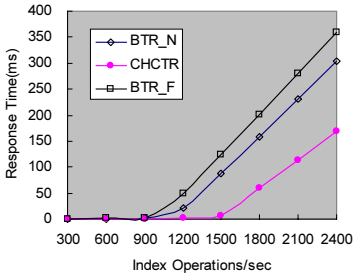
4.2 실험 결과 분석

본 연구의 시뮬레이션에서는 제안한 CHCTR 색인을 기존의 BTR_N 및 BTR_F 색인과 비교 분석하였으며, 검색 및 갱신 부하에 따른 각 기법의 처리 성능과 응답 시간을 분석하였다. 또한, 전체 인덱스 연산 중 검색 연산은 50%, 삽입 및 삭제 연산은 각각 25%로 기본 구성하여 실험하였다.

그림 5는 초당 발생된 인덱스 연산의 수(num_OPs)의 증가에 따른 각 기법의 처리치의 그래프이며, 그림 6은 평균 응답시간 그래프이다. 그림 5에서 보면, 발생된 초당 인덱스 연산의 수가 늘어날수록 점차로 응답시간이 점차로 증가함을 알 수 있다. 이는 주로 작업 부하 증가에 따라서, 인덱스 연산의 집중에 의한 것이다. 또한, 전반적인 인덱스 연산의 처리 성능을 측정한 결과, CHCTR이 BTR_N 과 BTR_F 보다 더 높게 나타났다.



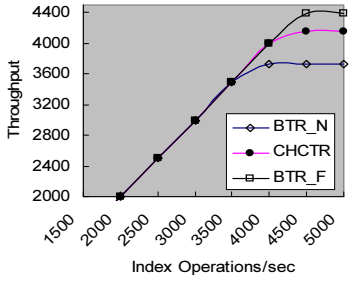
[그림 5] Index Operation Throughput



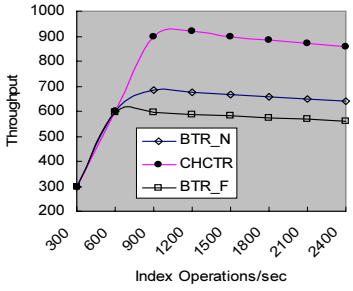
[그림 6] Average Response Time

그림 5에서 보면, 초당 인덱스 연산의 수가 대략 900개까지는 기법들 간의 별 성능 차이를 보이지 않는다. 이는 인덱스 연산의 부하를 각 기법들이 충분히 수용 가능함을 의미하며, 그림 6의 응답시간을 분석해 보면, 900개까지는 데이터 검색에 꼭 필요한 처리 시간이 소요될 뿐, 그 외의 지연 시간이 없어서, 성능 차이가 나타나지 않는다. 그러나 인덱스 연산의 수가 1,200~1,500개를 넘으면서 각 기법의 성능은 낮아지면서, 각 기법 별로 성능 차이가 나기 시작한다. 이는 초당 인덱스 연산수의 증가에 의한 인덱스 접근 적체가 성능에 영향을 크게 미치는 주요 요소임을 의미하며, 이 수치 이상으로 인덱스 접근 연산을 활성화시키는 것이 성능 향상에 도움이 되지 않음을 의미한다. 여기서 플래시 메모리상의 인덱스 노드 검색은 상대적으로 고속이므로 아직 이 구간에서는 성능 저하의 원인이 아니며, 인덱스의 삽입 및 삭제 연산에 의하여 야기되는 플래시 메모리의 느린 저장(지우기, 쓰기) 속도가 주요 원인이다.

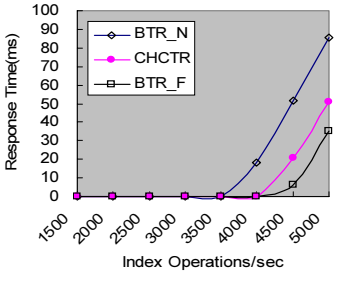
그림 5의 그래프에서 보면 본 CHCTR 기법이 느린 저장 연산을 최소화하여 기존 기법보다 처리 성능이 우세하다고는 하지만, 인덱스 연산 부하가1,500을 넘으면, 과도한 부하로 인하여, 서서히 성능이 저하됨을 알 수 있었다. 그림 6에서 인덱스 접근 부하가 증가하여도, 900이하의 구간에서는 별로 응답지연이 발생하지 않으나, 이 구간을 넘게 되면 적체현상이 발생하여 평균응답시간이 빠르게 증가함을 알 수 있었다. 전체구간에서 CHCTR 의 커브가 기존 기법에 비하여 완만하면서도 더 늦게 증가하면서 상대적으로 빠른 응답속도를 보였다. CHCTR은 인덱스 노드의 부하가 초당 2,400일 때, 기존 기법대비 23%의 개선된 처리 결과를 보였는데, 이러한 성능차이로 압축 핫-콜드 클러스터링의 효과를 확인할 수 있었다.



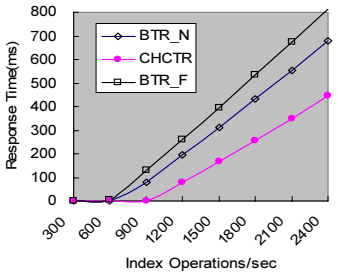
[그림 7] Index Search Throughput



[그림 8] Index Update Throughput



[그림 9] Index Search Response Time



[그림 10] Index Update Response Time

보다 세부적으로 CHCTR 기법의 효과를 분석하기 위하여, 혼합된 인덱스 연산을 검색 연산과 업데이트 연산(insert and delete)으로 다시 분리하여 실험해 보았다. 인덱스 검색 연산에 대한 처리 성능은 그림 7에 그려져 있고, 그 평균 응답 시간은 그림 9에 나타나 있다. 실험결과 BTR_F 가 가장 우수한 성능을 나타내었는데, 그 이유는 노드당 평균 점유율이 97%로서 세 기법 중 가장 높으므로, 한 번에 가장 많이 읽을 수 있기 때문이다. 즉, BTR_F는 최고의 저장효율을 가지고 있는데, 이는 결과적으로 인덱스 트리의 깊이를 짧게 만들어서 노드 접근 횟수가 줄었기 때문으로 분석된다. BTR_F는 5000개의 인덱스 검색 부하 시에 본 CHCTR 기법의 보다 6% 더 우수한 성능을 보였다.

인덱스 업데이트 연산에 대한 처리 성능은 그림 8에 나타나 있고, 그 평균 응답 시간은 그림 10에 나타나 있다. 업데이트 연산은 인덱스 검색 연산에 비하여 상대적으로 매우 인텐시브한 접근 특성을 가진다. 이유는 업데이트 연산에는 매우 느린 지우기와 쓰기를 동반하는 삽입 연산이 다수 포함되어 있기 때문이다. 이러한 인텐시브 저장 연산이 수행되는 실험 환경에서는 당연히 본 CHCTR이 전체 구간에서 더 높은 성능을 보인다. 실험결과 특히, 900개 이상의 초당 인덱스 부하 상태에서 기존 기법에 비하여 26%정도의 훨씬 더 높은 성능을 보였다. 이러한 성능의 우위는 압축된 콜드 클러스터에서 지우기 및 쓰기 연산이 줄어들어서, 결과적으로 전체적인 인덱스 노드 업데이트 연산수가 감소하였기 때문으로 분석된다. 이 사실은 그림 10에서 압축 핫-콜드 클러스터링으로 인하여 역시 응답시간도 더 개선되었음을 확인할 수 있었다. 실제로, CHCTR의 평균응답시간은 기존 기법에 비하여 최대 38%가 개선되었다.

5. 결론

본 논문에서는 SSD 저장 시스템의 인덱스 연산 성능을 높이기 위하여 CHC-Tree 기반의 새로운 인덱스 관리 기법을 제안하였다. 기존의 하드 디스크 및 메모리를 위한 B-Tree 기반 인덱스 기법을 개선하여, 제안 기법은 트리의 인덱스 노드를 핫-콜드 클러스터로 분리하여 다른 클러스터로 저장하였고, 키와 포인터를 재배열하여 오프셋 압축함으로써 플래시 메모리의 저장 연산 수를 대폭 줄여서, 전반적인 저장성능을 높일 수 있었다. 또한, 성능 확인을 위하여 큐잉방식의 시뮬레이션 모델을 제시하였다. 특히, 부하가 심한 저장 환경에서 실험한 결과, 전체적인 인덱스 처리 성능이 기존 기법에 비하여 23% 이상 개선됨을 확인하였다. 본 CHC-트리 기반 인덱스 관리 기법은 플래시 메모리나 SSD가 장착된 최신 데스크톱 컴퓨터 및 중대형 데이터 서버 등에서 폭넓게 활용 가능하다.

참고문헌

[1] 임근수, 고건, "플래시 메모리 기반 저장장치의 설계 기법", 정보과학회 추계 학술대회, 제30권 2-1호, pp. 274-276, 2003.10.

[2] Chanik Park, Jaeyu Seo,Dongyoung Seo, Shinhan Kim, and Bumsoo Kim, "Cost-Efficient Memory Architecture Design of NAND Flash Memory Embedded Systems", 21st International Conference on Computer Design, San Jose, California, pp. 474-479, 2003 October 13-15.

[3] 변시우, "하이브리드 하드디스크 시스템을 위한 플래시 노드 캐싱 기법", 한국산학기술학회논문지, 제 9권 6호, pp. 1696-1704, 2008.

[4] 성민영, “플래시 메모리 기반의 파일 저장 장치에 대한 성능분석”, 제 9권 3호, 제 9권 6호, pp. 1696-1704, 2008.

[5] Cha S. K., J. H. Park, and B.D.Park, "Xmas: An Extensible Main-Memory Storage System," Proc. of 6th ACM Int'l Conference on Information and Knowledge Management, 1997.

[6] 황규영, 홍의경, 음두현, 박영철, 김진호, 데이터베이스 시스템, 생능출판사, 2000.

[7] B-tree, “B-tree”, http://en.wikipedia.org/wiki /B-tree, 2009.

[8] Beckmann N., H. P. Kriegel, R. Schneider, and B. Seeger, "The R*Tree: An Efficient and Robust Access Method for Points and Rectangles," Proc. of ACM SIGMOD Intl. Symp. on the Management of Data, pp. 322-331. 1990.

[9] 이창우, 안경환, 홍봉희 (2003), "이동체 데이터베이스를 위한 메인 메모리 색인의 성능 결정 요소에 관한

연구", 정보처리학회 춘계 학술대회 제10권 1호, pp. 1575-1578, 2003.5.

[10] Lehman T. J. and M. J. Carey, "A Study of Index Structures for Main Memory Database Management Systems", Proc. of 12th Intl. Conf. on Very Large Database, pp. 294-303, 1986.

[11] Hongjun Lu, Yuet Yeung Ng, and Zengping Tang, "T-Tree or B-Tree: Main Memory Database Index Structure Revisited", Proc. of 11th Australasian Database Conference, 2000.

[12] 정재용, 노삼혁, 민상렬, 조유근, 플래시 메모리 시뮬레이터의 설계 및 구현, 한국정보과학회 논문지 C-컴퓨팅의 실제, 제 8권 1호, pp. 36-45, 2002.

[13] Mesquite, “CSIM2.0”, http://www.mesquite.com/documentation/documents/CSIM20_User_Guide-C.pdf, 2008.

[14] Samsung, “SpinPoint T Series”, http://www.samsung.com/Products/HardDiskDrive/SpinPointTSeries/index.asp, 2009.

변 시 우(Siwoo Byun) **[정회원]**

- 1989년 2월 : 연세대학교 이과대학 전산과학과(공학사)
- 1991년 2월 : 한국과학기술원 전산학과(공학석사)
- 1999년 2월 : 한국과학기술원 전산학과(공학박사)
- 2000년 3월 ~ 현재 : 안양대학교 디지털미디어학부 부교수

<관심분야>
데이터베이스, 저장장치, 모바일 임베디드 시스템 등