한국산학기술학회논문지
Vol. 9, No. 6, pp. 1696-1704, 2008

하이브리드 하드디스크 시스템을 위한 플래시 노드 캐싱 기법

변시우[1*]

Flash Node Caching Scheme for Hybrid Hard Disk Systems

Siwoo Byun[1*]

요 약 하드 디스크는 25년 이상 절대적인 데이터 저장장치이었다. 하지만, 최근에는 하드디스크에 플래시 메모리를 융합한 하이브리드 시스템이 차세대 주요 저장 시스템으로 각광받고 있다. 하이브리드 디스크는 향상된 데이터 입출력과, 전력 감소와 부팅 시간 단축 등의 장점으로 주요 데이터베이스로 충분한 능력을 보이고 있다. 그러나 기존의 디스크 기반의 B-Tree 계열의 인덱스 관리 방법은 하이브리드 디스크에 적합하지 않으므로, 개선되어야 한다. 본 논문에서는 플래시 메모리에 비하여 낮은 처리 성능을 가지는 하드 디스크의 단점을 보완한 플래시 노드 캐싱 기반의 인덱스 관리 기법을 제안한다. 본 기법은 하드 디스크 접근 부하를 줄이기 위하여, 유휴중인 리프 노드의 여유공간을 이용하여 데이터를 캐싱하여, 입출력 성능을 개선한다. 성능평가 결과로서 본 기법이 기존의 기법보다 개선되었음을 입증하였다.

Abstract The conventional hard disk has been the dominant database storage system for over 25 years. Recently, hybrid systems whichincorporate the advantages of flash memory into the conventional hard disks are considered to be the next dominant storage systems. Their features are satisfying the requirements like enhanced data I/O, energy consumption and reduced boot time, and they are sufficient to hybrid storage systems as major database storages. However, we need to improve traditional index management schemes based on B-Tree due to the relatively slow characteristics of hard disk operations, as compared to flashmemory. In order to achieve this goal, we propose a new index management scheme called FNC-Tree. FNC-Tree-based index management enhanced search and updateperformance by caching data objects in unused free area of flash leaf nodes to reduce slow hard disk I/Os in index access processes. Based on the results of the performance evaluation, we conclude that our scheme outperforms the traditional index management schemes.

Key Words : hybrid hard disk, flash caching, storage system, tree index, flash memory

1. 서 론

지난 25년 동안, 하드 디스크는 대부분의 파일 시스템에서 사용되는 절대적인 저장 미디어였다. 그러나 매년 디스크 저장 용량이 급속도로 증가하고는 있는 반면에, 기계적인 지연시간은 매년 15%정도로만 개선되고 있다. 반면에 메모리와 CPU는 매년 50% 정도의 빠른 속도로 개선 발전되고 있다. 과거 10년 동안 이러한 접근 속도 차이는 5-6배 이상 벌어지게 되었다[1]. 즉, 이러한 기계적 특성에 따른 속도차에 의하여 하드 디스크는 한계 성능에 도달하게 된다.

최근에 각종 이동형 또는 소형 정보기기들이 대중화됨에 따라, 정보 저장용 미디어로 기존의 하드 디스크를 점차 대체하여 플래시 메모리가 보편적으로 활용되게 되었다[4]. 그러나 플래시 메모리와 하드 디스크 모두 보편적인 저장 장치로서 많은 장점을 가지고 있는 반면, 적잖은 단점도 가지고 있다.

먼저, 하드 디스크는 널리 알려진 저장 장치로서 대용량의 저장 능력과 저렴한 저장 비용이 장점이나 기계적인 특성상, 소음, 전력, 속도, 내충격성 등에서 많은 단점

이 논문은 2007년도 정부의 재원으로 한국학술진흥재단의 지원을 받아 수행된 연구임(KRF-2007-313-D00649).
[1]안양대학교 디지털미디어공학과 (부교수)
*교신저자: 변시우(swbyun@anyang.ac.kr)
접수일08년 07년 16일 수정일 08년 10월 13일 게재확정일 08년 12월 16일

을 내포하고 있다[6]. 반면 플래시 메모리는 이러한 하드 디스크에 비하여 작은 용량과 상대적으로 높은 저장비용이 단점이 되고, 기타 측면에서 장점을 가지고 있다.

본 연구에서는 이러한 상반되는 두 저장 장치의 고유한 특성을 상호 보완하여 우수한 저장 및 입출력 성능을 발휘할 수 있는 하이브리드 저장 시스템 구조를 제시하고, 또한 이러한 데이터 저장 시스템에 적합한 인덱스 노드의 빈공간을 재활용한 플래시 노드 캐싱 기법을 제안하고자 한다.

2. 관련 연구

2.1 하이브리드 저장 시스템 구조

기존의 하드 디스크와 최신의 플래시 메모리를 융합한 하이브리드 저장 시스템을 통하여 데이터를 신속하게 저장하고 효율적으로 검색을 위해서는 먼저 플래시 메모리의 특성을 잘 고려하여야 한다. 서로 다른 두 매체간의 융합은 단순히 장치간의 연결로 구현하면 두 매체의 단점이 결합된 최악의 데이터 적체가 나타날 수 있다.

이러한 데이터 적체는 데이터 적체를 유도하여 시스템 성능에 매우 나쁜 영향을 미치게 된다. 따라서 서로 다른 두 매체의 단점을 최대한 보완하고, 우수한 특성은 더 보강할 수 있는 제어 기술이 필요하다.

다음은 보편적으로 사용하는 다양한 저장 장치의 특성을 비교한 표이다.

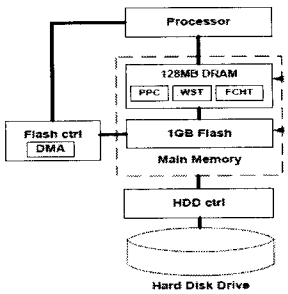
[표 1] 다양한 저장 장치의 성능 비교

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-I Flash	Page	12 μs (512B)	200 μs (512B)	2 ms (16KB)
NAND-MLC Flash	Page	20 μs (512B)	300 μs (512B)	2 ms (16KB)

읽기 연산의 경우에는 플래시 메모리의 연산 처리 속도가 하드 디스크에 비하여 445배 정도로 매우 빠르며, 일반 RAM 메모리에 비해서는 느리지만 12~20μs 정도로 매우 빠르므로 접근 시에 별 문제가 없다[15]. 그리고, 플래시 메모리의 쓰기 연산의 경우에는 속도가 읽기 연산 대비 10배 이상의 시간이 소모되나, 역시 하드 디스크에 비해서는 40배 이상으로 매우 빠르다. 물론 이러한 물성적인 차이는 중간단계의 메모리 캐시를 두게 되면, 그 차이는 줄어들 수 있으나, 전자적인 장치인 플래시 메모리와 기계적인 장치인 하드 디스크의 차이는 기본적으로 매우 크다. 반면에 시장의 하드디스크는 500GB급이 생산되며, 플래시 메모리는 아직은 10GB 대 이므로, 하드 디스크는 저렴한 저장용량이 최대 강점이며, 플래시 메모리는 고속 검색 및 저장 능력이 최대 강점이다[7,12]. 다음은 고속의 플래시 메모리를 디스크와 결합한 구조의 예이다.

(1) 하드 디스크에 추가 장착된 플래시캐시

플래시캐시(FlashCache)[14]는 집적된 1GB NAND 플래시 메모리를 하드 디스크위에 장착하여 시스템의 성능과 전력 사용량을 감축한 저장 시스템 구조이다. 원래 1GB의 DRAM을 128MB로 줄이고, 플래시 메모리가 2차 캐시 역할을 수행한다. 이를 위하여 PPC(Primary Page Cache), FGHT(Flash Cache Hash Table), WST(Wear level Status Table)이 필요하며, 플래시 제어기에 속도 개선을 위하여 DMA가 포함되어 있다. 초기의 기본적인 형태로서 소용량 플래시 메모리가 캐시 역할을 하는 당시에는 고가의 시스템이다.

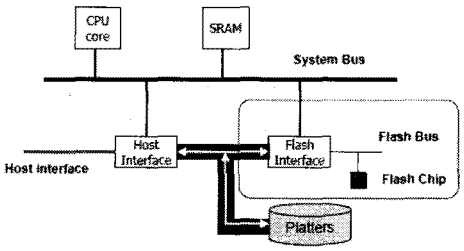


[그림 2] 하드 디스크에 장착된 플래시캐시

(2) 하이브리드 저장 시스템 기본 구조

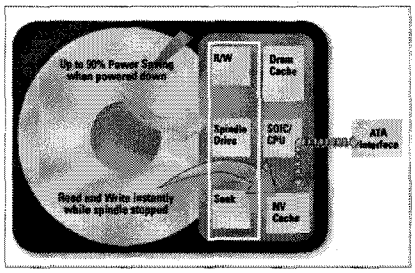
수년 전부터 플래시 메모리의 가격이 내려가고, 용량이 증가함에 따라서 1기가급 용량의 플래시와 하드 디스크와 결합된 직렬형 하이브리드 디스크가 제안되었었다. 다음은 기본적인 하이브리드 디스크 구조이다[13]. 전력 소비 감축과 빠른 스타트업을 위하여 하드 디스크에 플래시 메모리를 추가로 결합하였다. 하이브리드 디스크에서는 쓰기 요청은 하드 디스크의 스핀들 모터를 돌리지 않고, 대신에 플래시 메모리에서 버퍼링된다. 그 후 플래시 쓰기 버퍼가 꽉 차게 되면, 스핀들 모터를 돌리고, 버퍼링된 쓰기 요청이 디스크로 반영한다.

이 쓰기 버퍼링은 128MB의 플래시 메모리를 사용하며, 2.5"하드 디스크의 전력을 80%정도 절감시킨다. 또한, 캐싱된 데이터는 다음 접근시에 응답시간을 단축시켜 준다. 이 구조는 플래시 메모리 캐싱을 효과적으로 하기 위하여 버스와 채널을 개선한 하이브리드 저장 시스템이다. 플래시 메모리와 하드 디스크가 호스트 인터페이스와 연결된 버스는 병렬이며, 데이터는 플래시 메모리에 캐싱된 후, 하드 디스크에 저장되는 저장 구조이다.



[그림 3] 하이브리드 하드 디스크 구조

다음은 작년에 삼성에서 개발한 하이브리드 디스크[11]이다. 플래시 메모리에 캐싱 데이터와 부트 이미지를 포함하여 부트 시간을 반정도로 단축 가능하다. 또한, 플래시 메모리는 하드 디스크의 입출력을 대신 함으로써 하드 디스크 모터 스핀에 의한 전력 소모를 90% 정도 줄이고, 수명을 연장하는 효과가 있다.



[그림 4] 삼성 하이브리드 하드 디스크

2.2 데이터 색인 시스템 구조

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.

개념적으로 디스크 기반 색인 및 저장 기술의 목표는 디스크의 접근 횟수와 디스크 공간을 최소화하는 것이며, 디스크 I/O를 가장 큰 비용으로 고려한다. 그러나 메모리 기반 색인 및 저장 기술은 디스크의 접근이 없으므로, CPU 수행 시간을 줄이고, 최소한의 메모리 공간을 사용하는 것이 중요하다. 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.

(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^+-Tree 계열의 인덱스가 많이 사용되며[3,18], 공간 색인으로는R-Tree, R^*-Tree[2]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 리밸런싱 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 최근에 많이 연구되고 있는 GPS 기반 이동체나 휴대폰의 시공간 색인의 경우에는 이동체의 수가 많거나 위치변화가 많으면 실시간 처리를 어렵게 만드는 디스크 병목현상이 자주 발생한다. 이를 해결하기 위하여 디스크 기반의 한계를 극복한 메인 메모리 색인이 필요하며, 디스크와 메인 메모리를 합친 통합 색인 기법도 제안되었다[16].

(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. 디스크 기반 저장 시스템은 디스크에 데이터를 일일이 저장하므로 장애에 매우 강하다. 반면에 메인 메모리 데이터베이스는 메모리에 저장된 데이터가 사라지므로 디스크 기반 저장 시스템과는 다른 백업, 복

구 방식이 필요하다.

메인 메모리 기반 색인 시스템에서는 일반적으로 T-Tree가 1차원 데이터를 위한 색인으로서 좋은 성능을 보이며, 비교적 적합하다고 알려져 왔다[9]. T-Tree는 이진 검색과 높이 균형을 가지고O(logN)의 트리 순회가 가능한AVL-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[16]. 그러나 [8]에서 T-Tree는 동시성 제어에 대한 고려가 매우 부족하였으며, 이를 고려한다면 B-Tree가 성능을 추월할 수 있음을 밝혔다. 또한 실제로는 T-Tree와 함께 성능 향상을 위하여 노드에 자료 구조를 개선한 B-Tree 계열의 개선된 인덱스가 많이 사용된다.

디스크 기반 색인은 노드의 접근 횟수가 디스크의 I/O의 수와 같으므로, 트리의 깊이는 성능에 큰 영향을 미쳤다. 삽입/검색 시에는 데이터를 삽입/검색할 노드를 찾기 위하여 비교하면서 내려가는 노드의 개수가 성능에 가장 큰 영향을 준다. 따라서 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[16].

3. 플래시 노드 캐싱 기법 제안

3.1 제안 배경

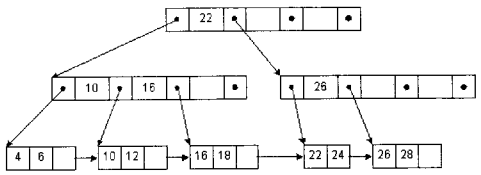
본 논문에서는 디스크 기반 및 메인 메모리 기반 색인 중에서 가장 보편적인 B-Tree 색인을 근간으로 하여 하이브리드 저장 시스템에 적합한 색인 저장 기법을 연구하였다. 메인 메모리 데이터베이스에서 많이 사용되는 T-Tree도 동시성 제어를 고려할 경우 B-Tree 보다 성능이 낮은데, 그 이유는 메모리 보다 CPU의 발전 속도가 더 빠르므로, 상대적으로 접근수가 적은 B-Tree가 유리하기 때문이다[8].

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 기법을B-Tree를 포함한 일반적인 트리 구조의 색인에도 적용이 가능하다.

이러한 관점에서 본 연구에서는 메모리 기반 및 디스크 기반 데이터베이스에서 근간이 되는 B-Tree를 하이브리드 저장 시스템에 적합하게 개선하여, 쓰기 연산과 지우기 연산의 부담을 줄이고, 성능을 개선할 수 있는 효율적인 색인 저장 기법을 제안한다.

3.2 플래시 노드 캐싱 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 5] 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 분

석 결과와 기타 B^+-Tree에 대한 자세한 이론은 [18]에 설명되어 있다.

전술한 바와 같이, 플래시 메모리는 하드 디스크에 비하여 고가의 고속 메모리이므로 상대적으로 저장용량이 충분하지 않다. 만일, 하이브리드 하드 디스크에 작은 양의 데이터를 저장한다면, 플래시 메모리에 데이터베이스를 전부 적재하고 고성능 모드로 운영할 수도 있다. 하지만, 데이터의 건수가 상당히 많아진다면, 일부는 하드 디스크로 다시 내려야만 한다. 즉, 데이터베이스에서 매우 빈번히 접근되는 색인은 루트 노드에서부터 순차적으로 플래시 메모리에 유지하고, 하위 리프 노드에 연결된 실제 데이터는 너무 방대하므로 디스크로 내려야 한다. 또한, 그 중에서도 일부만이 빈번한 시공간적 로컬리티가 존재하므로 이를 고려하여야 한다.

전술한 바와 같이 B^+-Tree 색인은 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 낭비되고 있는 공간이다.

본 연구에서는 이러한 플래시 메모리의 유휴 공간을 활용하여 인덱스 노드에 데이터를 캐싱하는 플래시 노드 캐싱(Flash Node Caching) 기법을 제안한다. 본 기법의 기본 아이디어는 바로 현재 유휴 공간인 31% 저장 공간을 읽기/쓰기 캐시로 활용하자는 것이다. 플래시 메모리는 고속 저장이 가능하므로 충분히 캐시로서의 성능 개선을 얻을 수 있으며, 기존 휘발성 램(RAM) 방식의 캐시에 비하여 비휘발성 영구 저장이 가능하므로, 갑작스런 전원차단에 따른 데이터 손실도 방지할 수 있다. 더우기 하드 디스크에 비하여 매우 고가인 플래시 메모리는 빈 공간의 낭비 없이 더욱더 효율적으로 활용되어야 한다. 또한, 추후에 해당 노드에 신규 엔트리가 삽입되면, 이때는 이 노드는 캐시 역할을 중단하고, 추가 엔트리를 저장하면 되므로 공간 재활용에 따른 오버헤드가 거의 발생하지 않는다. 그리고 추가 엔트리가 저장될 때 그 노드에 대한 플래시 메모리 쓰기가 한번 수행되는데, 이때 해당 데이터도 그 노드안에 같이 저장하여 쓰여진다. 따라서 엔트리에서 포인팅하는 데이터를 캐시에 저장하기 위한 별도의 추가적인 쓰기 연산이 불필요하므로, 플래시 메모리의 쓰기 연산 횟수를 감소시킨다. 읽기 보다 느린 플래시 메모리의 쓰기 속도를 고려할 때 이는 곧 하이브리드 저장 시스템의 성능 향상에 기여한다.

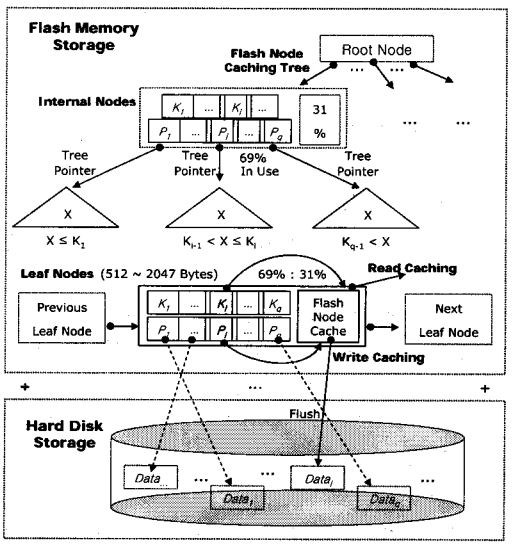
단, 본 기법의 전제 조건으로 한 노드의 빈공간인 31%에 포함되도록, 데이터(저장될 실제 컨텐츠) 크기는 이 크기 이하여야 한다. 예를 들면, 512 바이트의 페이지 크기를 가진 플래시 메모리 구조라면, 데이터 한 건의 크기가 약 150 바이트 이하여야 한다. 예를 들어, 이 보다 큰 플래시 메모리인 2048 바이트인 경우라면 데이터 크기는 약 600 바이트 이하여야 한다. 이 전제는 데이터 한 건의 크기가 매우 큰 대형 데이터베이스가 아닌 중소형 또는 개인형 데이터베이스 크기로 한정됨을 의미한다.

물론 150바이트나 600바이트 이상일 경우에 링크된 포인터를 참조하여 다른 노드의 빈 공간을 활용한다면 충분히 가능하지만, 공간 관리 알고리즘이 복잡하여 지므로, 본 논문에서는 이를 제외시키고, 한 노드 안의 공간 저장만을 대상으로 한다.

여기서150 바이트의 데이터 크기라 하더라도, 10바이트짜리 필드가 15개로 웹상의 기본적인 회원관리가 가능한 크기이다. 이 전제는 데이터 한 건의 크기만을 한정하며, 데이터 건수는 많아도 상관이 없다. 오히려 데이터 건수가 많을수록 캐싱 효과가 중요하므로, 제안 기법의 효용성도는 커진다.

결과적으로 본 기법은 유휴중인 빈 공간을 재활용하므로 큰 오버헤드 없이, 고성능 캐싱 효과와 안정성을 모두 얻을 수 있으므로 저장 시스템의 성능 개선에 기여할 수 있다.

또한, 4장의 시뮬레이션 실험과 분석을 통하여 이 기법의 효용성을 입증하였다. 그림 6은 제안된 노드 캐싱을 지원하는 B^+-Tree 인덱스 구조로서, FNC-Tree(Flash Node Caching Tree)라고 하였다.



[그림 6] 플래시 노드캐싱 트리의 구조

본 기법의FNC-Tree에서 플래시 메모리의 입출력의 효율을 위하여 한 페이지당 하나의 노드를 수용한다. 데이

터 수정 또는 엔트리 삽입이 발생하면 루트 노드에서 중간 노드를 거쳐서 리프 노드에서 도달하며, 리프 노드에서 연결된 하드 디스크의 데이터가 저장되게 된다. 본 기법의 빈 공간을 활용하는 노드는 리프 노드만을 대상으로 하였다. 물론, 루트 노드와 중간 노드를 활용하면 좀 더 효과적이겠지만, 실험결과 리프 노드에 비하여 그 수가 상대적으로 5%이하로 매우 적으면서, 계산과정 너무 복잡하게 만들므로, 본 기법은 데이터에 가장 근접해있는 리프노드만을 대상으로 하였다.

본 기법의 리프 노드 구조는 일반 리프 노드 구조체에서 이 노드가 현재 캐시 모드로 동작하는지에 대한 변수와 몇 바이트부터 캐시 데이터가 저장되는지에 대한 오프셋 변수를 추가로 포함하고 있다. 그리고 리프 노드에서 데이터가 캐싱되는 엔트리 (키+포인터)는 데이터 포인터를 하드 디스크에 있는 데이터 노드가 아닌 바로 자기 자신의 현재 노드를 포인팅하게 된다.

FNC-Tree도 다른 B^+-Tree와 마찬가지로 노드의 엔트리 점유율이 평균 점유율인 69%에서 운영중에 이를 벗어나서 최소 50%에서 최대 100%까지 변동할 수 있다. 이 변동으로 인하여 빈공간이 줄어서 캐시 데이터의 저장이 불가능하게 되면, 캐싱된 데이터를 하드 디스크로 반영한 후 그 캐시영역을 빈 공간으로 모두 반납하게 된다.

다음은 의사코드로 작성된 본 제안 기법의 검색 알고리즘이다.

Algorithm 1. Index Search Operation

```
DataRecord *HBPT-Search (int key) {
    R ←block containing root node of tree;
    N ←FM-Read (R);
// read node R in flash memory
    while ( Type(N)  LEAF_NODE ) {
// N is not a leaf node of tree.
        q ←number of tree pointers in node N
        if ( key ≤ N.K1 )
// N.Ki refers to the i^th search field value in node N.
            N ←N.P1
 // N.Pi refers to the i^thtree pointer in node N.
                        // N is first child node.
        else if ( key > N.Kq-1 )
            N ←N.Pq        // N is last child node.
        else {
            Search node N for an entry i
such that N.Ki-1 < key ≤ N.Ki ;
            N ←N.Pi
        }
        N ←FM-Read (N);
// read internal node N in flash memory
    } // end of while loop
    Search node Nfor entry (Ki,Pi) with key = Ki ;
// search leaf node N.
    if ( entry_found ) {
      if ( N.isCache && N.Pi == N )
// check if data is cached in its leaf node
return ( N.Pi + N.Offset_Cache )
// return cached data
        else          // data is in hard disk
          return HD-Read(N.Pi)
// read data record in hard disk.
    }
    else  return NULL;  // data is not found
} // End of Function
```

4. 실험 및 성능 평가

4.1 시뮬레이션 환경

본 연구에서 제안된 기법의 성능을 검증하기 위하여 시뮬레이션을 수행한 후 그 결과를 분석해 보았다. 본 시뮬레이션 수행시 FNC-Tree(FNCT) 색인 기법과 비교된 대상 색인은 가장 보편적으로 사용하는 B^+-Tree(BPT) 기법이다. FNC-Tree 색인은 다시 FNCT-1과 FNCT-2로 분리하여 총 3가지 종류의 트리 색인에 대하여 비교 실험하였으며, 노드의 평균 점유율 (69%)을 기본으로 하였다. FNCT-1은 리프 노드의 빈 공간에 한 개의 엔트리에 대한 데이터 캐싱이 가능한 크기로서 150 바이트 크기의 사이즈로 실험한 것이다. FNCT-2는 두 개의 엔트리에 대한 데이터 캐싱이 가능한 크기로서 75 바이트 크기로 실험한 것이다.

실험 도구는 CSIM[10] 시뮬레이션 도구를 사용하였다. 실험 환경은 듀얼 펜티엄4-2.1 GHz CPU와 메인 메모리 2G, 하드디스크 160G이며, 윈도우 2003 서버를 사용하였다.

하이브리드 디스크 저장 시스템을 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 20개에서부터 20개 단위로 160개까지 변화시켜 보았으며, 갱신 연산의 경우는 30개에서부터 30개 단위로 240개까지 변화시켜 보았다. 이는 모두 시뮬레이션 시스템에 가해지는 작업 부하를 의미한다. 플래시 메모리의 읽기 연산 수행 시간은 20㎲로 설정하였고, 쓰기 연산 수행 시간은 300㎲로 설정하였으며, 하드 디스크 접근 시간은 8.9㎳로 설정하였다. 각 연산 수행 시간은 기존의 연구[17]와 제품 홈페이지[12]에서 제시한 자료이다. 사용된 샘플의 크기는 150바이트와 75바이트이며, 총 279,841 건이다. 이 데이터의 인덱싱을 위하여 23개의 팬아웃으로 구성하여, 레벨0의 루트 노드 1개, 레벨1의 중간노드 23개, 레벨2의 중간노드 529개, 리프 노드 12,167개로 구성하였다. 여기서, 각 노드를 포인팅하기 위한 포인터의 크기는 5바이트의 크기로 설정하고, 키필드의 크기는 10바이트로 설정하였는데, 일반적인 정수, 실수 등의 데이터 형을 포함하여 키워드 정도의 문자열을 포함할 수 있는 크기이다.

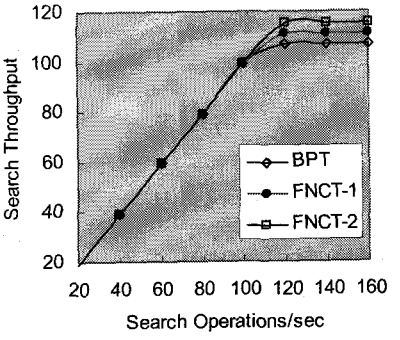
4.2 실험 결과 분석

본 연구의 시뮬레이션은 한 노드에 대한 평균 점유율 69%로 B^+-Tree를 구성한 BPT 색인과 데이터 캐시가 1개인 FNCT-1과 두 개인 FNCT-2 색인을 대상으로 수행하였으며, 실험용 샘플 데이터에 대한 검색 및 갱신 부하에 따른 각 기법의 처리 성능과 처리 시간을 분석하였다.

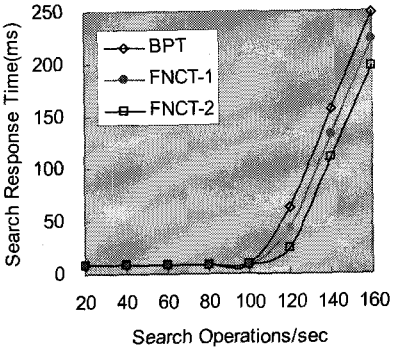
(1) 검색 연산 부하에 따른 성능 비교

그림7은 초당 발생된 검색 연산의 수의 증가에 따른 처리치를 표시한 그래프이다. 발생된 초당 검색 연산의 수가 늘어날수록 점차로 연산의 처리 결과치가 증가함을 알 수 있다. 또한, 전반적인 검색 연산의 처리 성능을 측정한 결과, FNCT-2가 BPT와 FNCT-1 보다 높게 나타났다.

그림7에서 보면, 초당 검색 연산의 수가 대략 100개까지는 기법들간의 별 차이를 보이지 않는다. 이는 검색 연산의 부하를 각 기법들이 충분히 수용 가능함을 의미하며, 그림 8의 응답시간을 분석해 보면, 100개 까지는 데이터 검색에 꼭 필요한 처리 시간이 소요될 뿐, 그 외의 지연 시간이 없어서, 성능 차이가 나타나지 않는다.



[그림 7] 초당 검색 연산 처리치의 비교



[그림 8] 검색 연산의 응답 시간 비교

그러나 검색 연산의 수가 100개를 넘으면서 각 기법의 성능이 점점 낮아진다. 이는 초당 검색 연산수의 증가에 의한 데이터 읽기 적체가 성능에 영향을 크게 미치는 주요 요소임을 의미하며, 이 수치 이상으로 검색 연산을 활성화시키는 것이 성능 향상에 도움이 되지 않음을 의미한다. 여기서 플래시 메모리상의 트리 노드 검색은 고속이므로 성능 저하의 원인이 아니며, 이 색인의 리프노드에서 연결되는 하드 디스크의 느린 데이터 검색 속도가 주요 원인이다. 그래프에서 보면 검색 연산의 부하가 적은 시작부터 중간 구간에서는 느린 하드 디스크 읽기 연산이 시스템 안에서 수용되어 성능 저하를 일으키지 않는다. 하지만, 중간 구간을 지나면서 생성되는 검색 연산의 수가 증가하므로, 느린 하드 디스크에서 처리 되지 못하여 검색 연산들이 작업 큐에 과도하게 쌓이면서, 전체 연산 처리가 지체되며 결과적으로 성능이 서서히 저하되게 된다.

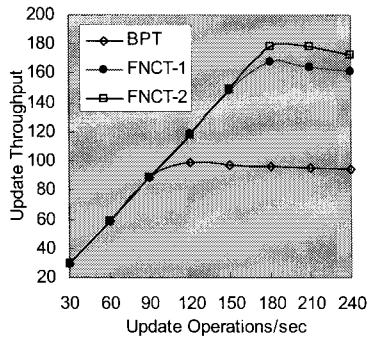
하지만, 검색 연산수 100개 이상의 동일한 조건에서도 제안한 FNCT 색인이 BPT 색인에 비하여 성능이 상대적으로 약 5~9%정도 좀 더 높다. 그 이유는 검색 성능 저하의 주요 원인인 느린 하드 디스크 접근의 부담을 FNCT 색인은 리프 노드내에 데이터 캐싱을 통하여 감소시킬

수 있었기 때문이다. FNCT-2는 FNCT-1에 비하여 평균적으로 하나더 데이터 캐시를 사용할 수 있으므로, FNCT-2의 검색 성능이 약간 더 우수하다. 만일 FNCT 캐싱 기법에 좀 더 정교한 캐시 운영 기법을 사용한다면 hit-ratio를 높일 수 있다. 또한, 캐싱에 의한 디스크 스핀 수 감소에 의하여, 전체적인 전력소모도 줄이는 부수적인 효과도 있다.

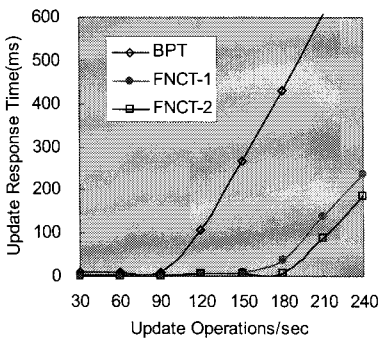
또한, 그림7에서 검색 연산의 응답시간을 비교해보면, FNCT-2, FNCT-1, BPT 순으로 우수하게 나타났다. 특히, 전체 구간에서는 FNCT-1이 BTR69보다 14%정도의 개선된 응답 속도를 나타내었다. 이 결과치로부터 검색 연산에 수반된 플래시 노드 캐싱 효과가 처리 시간 개선 효과를 주고, 이는 다시 초당 검색 연산의 처리 성능을 개선시킴을 확인할 수 있다.

(2) 갱신 연산 부하에 따른 성능 비교

플래시 노드 캐싱 기법은 검색 연산 보다는 갱신 연산에서 큰 효과를 보인다. 이 영향을 분석하기 위하여 갱신 연산의 부하를 초당 30개에서 최대 240개 까지 30개 단위로 변화시켜 보았다. 그림9는 갱신 부하의 증가에 따른 데이터 처리 성능 변화를 표시한 그래프이며, 그림10은 그 응답 시간의 변화를 표시한 그래프이다.



[그림 9] 초당 검색 연산 처리치의 비교



[그림 10] 검색 연산의 응답 시간 비교

그림9과 그림10에서 보면 색인 기법 모두 갱신 부하가 증가할수록 처리 성능이 받쳐주지 못하고 점차로 저하되고, 응답 속도도 매우 느려짐을 알 수 있다. 이유는 검색 연산의 실험에서와 같이 시간이 많이 소모되는 저속의 하드 디스크 쓰기 연산이 점차로 증가할수록 적체 현상이 발생하여, 저장 시스템의 처리 성능과 응답 속도를 크게 저하시키기 때문이다.

그러나 그림9의 그래프를 살펴보면, 전반적으로 FNCT 기법이 BPT에 비하여 갱신 연산 처리 성능이 우수하게 나타났다. 이는 전술한 플래시 노드 캐시에 데이터 쓰기 캐싱 효과 때문이다. 즉, 트리 노드를 갱신한 후 연결된 데이터를 하드 디스크에 쓰지 않고, 리프 노드에 같이 저장하여 신속한 저장이 가능하기 때문이다. 리프 노드의 캐시 영역에 저장된 데이터는 다음 갱신 연산이 요청되는 시간사이의 유휴시간에 디스크로 내려쓰기를 하게 된다.

하지만, 갱신 부하가 초당 180개 이상으로 증가하면서, 플래시 노드 캐싱이 가지는 효과는 점차로 줄어들게 된다. 이유는 그림10에서 보는 바와 같이, 시간을 많이 소모하는 저속의 하드 디스크 쓰기가 적체되어 캐싱과 유휴시간 내려 쓰기의 효과가 급격히 감쇄되기 때문이다. 하지만 전반적인 갱신 부하 구간에서 평가해보면, FNCT-1은 BPT에 비하여 갱신 연산의 응답 시간이 약 80% 단축되었으며, 데이터 저장 성능이 약 42% 높아졌다. 이 효과는 일반 PC의 write-back 캐시와 유사하다.

5. 결 론

본 논문에서는 기존의 대표적 저장 장치인 하드 디스크와 최근 휴대용 고속 데이터 저장 장치로 많이 사용되는 플래시 메모리를 융합한 개선된 하이브리드 스토리지 구조를 분석하였다. 또한, 하이브리드 저장 시스템에서 인덱스 연산의 성능 향상을 위한 새로운 인덱스 관리 기법을 제안하였다.

기존의 하드 디스크 및 메모리를 위한 B-Tree 기반 인덱스 기법을 개선하여, 제안 기법은 플래시 리프 노드의 사용되지 않는 여분의 영역을 활용하여 읽기와 쓰기의 데이터 캐싱 효과를 증대시킴으로써 전반적인 인덱스 연산의 처리치 및 응답 속도를 높였다. 시뮬레이션을 통한 실험 결과 기존 기법에 비하여 읽기 연산의 경우 5%이상, 쓰기 연산의 경우 42%이상의 성능 향상을 얻었는데, 플래시 리프 노드의 캐싱 효과에 의한 결과임을 확인할 수 있었다.

참고문헌

[1] Andy Wang, Geoff Kuenning, Peter Reiher, and Gerald Popek (2006), "The Conquest File System: Better Performance Through a Disk/Persistent -RAM Hybrid Design" ACM Transac. on Storages, vol. 2, no. 3, pp. 309-348.

[2] Beckmann N., H. P. Kriegel, R. Schneider, and B. Seeger, (1990), "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.

[3] B-tree (2008), "B-tree" http://en.wikipedia.org /wiki/B-tree. Accessed 1-Mar-2007.

[4] Byun S., Hur M., and Hwang H. (2007): "An index rewriting scheme using compression for flash memory database systems" Journal of Information Science, 33(4): 398-415.

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

[6] Chang Y. Hsieh J., and Kuo T. (2007), "Endurance Enhancement of Flash-Memory Storage System: An Efficient Static Wear Leveling Design", Proc. 44th conference on Design automation, San Diego, USA, 212-217.

[7] Flash (2008) "What is Flash Memory?" http://www.samsung.com/Products/Semiconductor/Flash/WhatisFlash/FlashStructure.htm.

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

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

[10] Mesquite Software (2008), Getting Started with CSIM, Mesquite Software, 2007. http:// www.mesquite.com/documentation/documents/GettingStartedForCSIMUsers-060208-2.pdf

[11] Samsung (2007), "What is Hybrid HDD?" http://www.samsung.com/Products/HardDiskDrive/whitepapers/WhitePaper_12.htm

[12] Samsung (2007), "SpinPoint T Series" http://www.samsung.com/Products/HardDiskDrive/SpinPointTSeries/index.asp

[13] Sang Lyul Min; Eyee Hyun Nam (Jan. 24-27 2006) "Current trends in flash memory technology; Design Automation" Asia and South Pacific Conference, pp. 332-333.

[14] Taeho Kgil and Trevor Mudge (October 23.25 2006) "FlashCache: A NAND Flash Memory File Cache for Low Power Web Servers", CASES'06, Seoul, Korea. pp. 103-112,

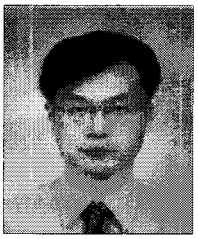
[15] 이옥희, 김진호, 차재혁 (2004.11) "스페어 영역을 활용한 NAND 플래시 메모리 관리", 정보처리학회 추계학술발표대회, 제 11권 2호, pp. 149-152.

[16]이창우, 안경환, 홍봉희 (2003), "이동체 데이터베이스를 위한 메인 메모리 색인의 성능 결정 요소에 관한 연구", 정보처리학회 춘계 학술대회 제10권 1호, 2003년5월, pp. 1575-1578.

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

[18] 황규영, 홍의경, 음두현, 박영철, 김진호 (2000), 데이터베이스 시스템, 생능출판사

변 시 우(Siwoo Byun) **[정회원]**



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

<관심분야>
데이터베이스, 모바일 임베디드 시스템 등