Journal of the Korea Academia-Industrial cooperation Society
Vol. 14, No. 1 pp. 393-401, 2013

http://dx.doi.org/10.5762/KAIS.2013.14.1.393

분할된 압축 인덱스를 이용한 컬럼-지향 플래시 스토리지의 검색 성능 개선

변시우[1*]
[1]안양대학교 디지털미디어학과

Search Performance Improvement of Column-oriented Flash Storages using Segmented Compression Index

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

요 약 대부분의 기존 데이터베이스들은 빠른 저장 성능을 얻기 위하여 한 레코드의 속성들을 하드 디스크에 연속적으로 배치하는 레코드-지향 저장 모델을 사용하였다. 하지만 검색이 대부분인 데이터웨어하우스 시스템에는 월등한 읽기 성능 때문에 컬럼-지향 저장 방식이 적합한 모델이 되고 있다. 또한, 현재 플래시 메모리가 고속 데이터베이스 시스템을 위한 선호 저장 매체로 인정되고 있다.
본 논문에서는 고속 컬럼-지향 데이터베이스 모델을 도입하고, 고속 컬럼-지향 데이터웨어하우스 시스템을 위한 컬럼-인지 인덱스 관리 기법을 제안한다. 본 인덱스 관리 기법은 개선된 B트리에 기반하며, 중간 노드와 리프노드에서 내장 플래시 인덱스와 빈공간 압축을 통하여 높은 검색 성능을 얻는다. 성능 평가 결과를 기반으로 본 인덱스 관리 기법이 기존 기법보다 검색 처리 및 응답 시간 측면에서 더 우수함을 확인하였다.

Abstract Most traditional databases exploit record-oriented storage model where the attributes of a record are placed contiguously in hard disk to achieve high performance writes. However, for search-mostly datawarehouse systems, column-oriented storage has become a proper model because of its superior read performance. Today, flash memory is largely recognized as the preferred storage media for high-speed database systems.
In this paper, we introduce fast column-oriented database model and then propose a new column-aware index management scheme for the high-speed column-oriented datawarehouse system. Our index management scheme which is based on enhanced B^+-Tree achieves high search performance by embedded flash index and unused space compression in internal and leaf nodes. Based on the results of the performance evaluation, we conclude that our index management scheme outperforms the traditional scheme in the respect of the search throughput and response time.

Key Words : Column-oriented database, Column-aware index management, B^+-Tree index, Flash memory device

1. 서론

일반적으로 기존의 데이터베이스 시스템은 하드 디스크를 기반으로 하면서, 데이터를 한 레코드씩 연속적으로 기록하는 전통적인 레코드(가로)-지향 저장 구조를 고수하였다. 그러나 이러한 전통적인 가로-지향 데이터 저장 및 검색 방식은, 하드 디스크 대체 미디어로 부상한 플래시 메모리 기반 고속 저장 장치와 대용량 고속 검색을 위한 컬럼(세로)-지향 저장 기술의 등장으로 인하여 더욱 개선되고 있다.

이논문은 2012년도 정부(교육과학기술부)의재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임 (No.20120002767)
*Corresponding Author : Siwoo Byun (Anyang University)
Tel: +82-10-2776-0922 email: swbyun@anyang.ac.kr
Received July 24, 2012 Revised (1st September 11, 2012, 2nd October 13, 2012) Accepted January 10, 2013

회전형 하드 디스크는 오랫동안 대부분의 파일 시스템에서 사용되는 절대적인 저장 미디어였다. 그러나 기계적인 지연시간은 개선이 어려워 매년 15%정도로만 개선되고 있다. 반면에 램이나 플래시 메모리 부품과 CPU는 매년 더 빠른 속도로 개선 발전되고 있다. 과거 10년 동안 하드 디스크와 메모리의 접근 속도 차이는 수십 배 이상 벌어지게 되었다.[1]

반면, 이러한 기존의 하드 디스크를 점차 대체되고, 고속 저장 장치나 모바일 정보기기들이 대중화됨에 따라 새로운 저장 미디어로 플래시 메모리가 점차 활용되고 있다[2]. 그러나 플래시 메모리와 하드 디스크 모두 대표적인 저장 장치로서 많은 장점을 가지고 있는 반면, 적잖은 단점도 가지고 있다. 먼저, 하드 디스크는 널리 알려진 저장 장치로서 매우 저렴한 저장 비용이 장점이나 기계적인 특성상, 소음, 전력, 속도, 내충격성 등에서 많은 단점을 내포하고 있다[3]. 반면 플래시 메모리는 이러한 하드 디스크에 비하여 상대적으로 높은 저장비용이 단점이 되고, 다른 측면에서는 월등한 장점을 가지고 있으므로, 향후 고성능 저장 시스템으로 활용가치가 매우 높다.

2. 제안 연구의 배경

최근 컴퓨팅 환경이 빅 데이터 환경이나 클라우드 환경으로 진화됨에 따라서, 기존의 일반적인 데이터베이스에서 대용량 콘텐츠에 대한 고속 검색의 필요성이 더욱 더 커지고 있다. 이러한 고속 데이터베이스 검색을 위하여, 컬럼-지향(Column-Oriented) 데이터베이스[4,5]가 제안되었다.

이는 기존의 한 레코드 단위로 연속 저장하는 가로방향-저장식인 일반 데이터베이스와는 달리, 세로의 필드단위(컬럼)로 분리, 저장, 검색하는 새로운 데이터베이스 모델이다. 컬럼-지향 데이터베이스는 같은 필드인 컬럼 단위로 모아서 저장하므로, 유사성이 높은 데이터들이 서로 군집되므로, 압축, 저장, 검색에 매우 효과적인 구조이다.[6] 특히, 읽기 위주의 고속 검색 환경에서는 아래와 같이 특정 컬럼에 데이터가 모여 있는 컬럼 저장 구조가 효율적이다. 기존 사례에서 MonetDB[7]이 고속화를 위한 기본적인 시도를 하였고, C-Store[8]가 잘 알려져 있다.

과거에는 하드 디스크가 대세였고 운영하는 데이터베이스도 이에 적응하였으므로, 하드 디스크의 접근 특성을 고려하였고, 당연히 일반 데이터베이스의 트랜잭션 처리에 가장 효율적인 저장 방식인 가로-지향 저장 방식으로 설계되었다. 그러나 최근에는 전통적인 하드디스크 드라이브(HDD) 저장 시스템에서 초고속 플래시 기반의 저장 시스템(SSD) [9,10]으로의 대체이동이 빠르게 진행되고 있다.

컬럼-지향 데이터베이스의 효과는 대표적으로 Star Schema Benchmark[6]에서 공표한 성능 결과를 보면, 일반 데이터베이스에 비하여 3배 이상의 압도적인 처리 성능을 보여준다. 또한, 컬럼-저장 스토리지는 다량의 레코드에 한 컬럼의 수치를 수정 반영할 때도 상당히 효율적이다. 실제 이런 업데이트 연산이 일상적으로도 자주 발생하는데, 일반 스토리지는 업데이트와 관련 없는 다른 컬럼들까지도 스토리지로부터 읽고 메모리상에도 할당시키느라 엄청난 시간과 공간을 낭비한다. 더욱이 이제 고가의 플래시 메모리 SSD가 속도와 더불어 충분한 가격 경쟁력을 갖춤에 따라서, 차세대 컬럼-기반 데이터베이스를 위한 효과적인 압축 저장 기법과 대용량 데이터 검색의 고속화 기법이 더욱더 필요하게 되었다.

3. 고속 DB 검색을 위한 CaF 색인

3.1 기존의 가로-지향 DB의 색인 분석

기존의 가로-지향 방식인 일반 데이터베이스에 사용되는 하드 디스크 기반 색인 및 저장 기법은 검색 시에 디스크 접근을 최소화하기 위하여 노드의 크기를 디스크 페이지와 같은 크기나 그 배수로 설정하고, 되도록이면 많은 엔트리를 한 노드에 넣어야 유리하였다. 디스크 기반 저장 시스템에서는 검색 성능 저하보다는 헤드 무빙이나 회전 지연에 의한 성능 저하가 훨씬 더 크기 때문에 한 노드에 많은 엔트리를 넣는 방향으로 설계한다[5]. 이런 용도로 B-Tree 인덱스가 대표적으로 사용되어 왔다.

또한, 고속처리를 위한 램 메모리 기반 색인 시스템에서는 일반적으로 T-Tree가 데이터를 위한 색인으로서 좋은 성능을 보이며, 비교적 적합하다고 알려져 있다. 그리고 디스크 기반 색인은 깊이가 얕고 넓게 퍼진 트리를 써서 삽입/검색 시에 I/O 비용을 최소화 하였다. 반면, 메모리 기반 색인의 접근 비용은 포인터로 노드의 메모리 주소를 획득하는 비용이므로 크지 않다. 따라서 디스크 기반 색인에서 선호하는 얕고 넓게 퍼진 트리 구조는 메모리 기반 색인에서는 유용하지 않다. 메모리 기반 색인에서는 디스크 기반 색인과는 달리 노드의 용량을 변화시켜 깊이와 비교횟수를 조절하여 향상이 가능하다[9].

3.2 컬럼-지향 DB를 위한 CaF 색인의 제안

본 연구에서는 가장 보편적이며 공통적인 특성의 B-Tree 색인을 근간으로 하여 컬럼-지향 데이터베이스

및 고속 플래시 메모리 스토리지에 효과적인 색인 저장 관리 및 고속 검색 기법을 제안하였다. 메인 메모리 데이터베이스에서 많이 사용되는 T-Tree도 가능하겠지만, T-Tree도 B-Tree에서 파생되어 B-Tree의 기본 속성을 가지고 있으며, 실제 메모리 데이터베이스에서 B-Tree도 많이 사용된다.[9]

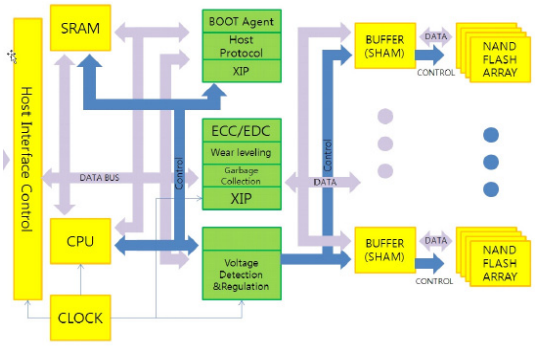
또한, 기본 B-Tree 중에서도 더 개선된 B^+-Tree를 대상으로 하였다. 그 이유는 B^+-Tree는 B-Tree와는 달리 중간 노드에 데이터 관련 정보를 넣지 않고 리프 노드에서 저장하므로, 상대적으로 색인 자체의 저장 효율이 높아지기 때문이다. 또한, 우선적으로는 B^+-Tree에 적용하였지만, 제안 기법을 일반적인 색인에도 적용가능하다.

이러한 관점에서 본 연구에서는 일반 데이터베이스의 기본 색인인 B^+-Tree를 본 컬럼-지향 플래시 메모리 저장 환경에 적합하게 개선하여, 검색 속도를 높이고, 읽기에 비하여 느린 쓰기 연산의 부담을 줄이고, 저장 성능을 개선할 수 있는 효율적인 색인 저장 기법인 CaF-Tree (Column-aware Flash Tree) 및 검색 기법을 제안한다. CaF-Tree 기법의 핵심은 B-Tree의 비활용 영역을 제거한 압축과 내장된 압축 인덱스 기반의 고속 검색이다.

일반적으로 B-Tree 계열의 색인은 데이터의 삽입, 삭제, 검색을 효율적으로 처리하기 위하여 가장 널리 사용되는 색인 구조이다. 기본적으로 B-Tree는 하위 노드에 대한 포인터 정보와 함께 데이터 관련 정보를 한 노드 안에 보관한다. B^+-Tree는 중간 노드에 실제 데이터 관련 정보를 넣지 않고 리프 노드에 이 데이터를 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. 특히, 컬럼-지향 데이터베이스에서는 데이터가 압축되어 저장되므로, 데이터의 크기를 고정적으로 예측하여 저장이 어려우므로, 리프 노드에서 순차적인 압축 저장이 효율적이다. 따라서 컬럼-지향 데이터베이스를 위한 플래시 메모리의 색인으로서는 기본적인 B-Tree 보다도 B^+-Tree가 더 적합하다.

그리고 B^+-Tree에서 수많은 임의의 값들을 삽입하고 삭제하는 시뮬레이션을 수행하여 분석한 결과 70%정도 차 있을 때가 성능상 최적이다.[2] 한 노드에 최대한 많이 저장하면 검색 노드수도 줄어들고 공간 효율은 향상되지만, 삽입, 삭제시 노드의 변동이 너무 빈번하여 많은 수의 쓰기 연산을 유도하여 결과적으로 더 성능 손실이 크다. 즉, 평균점유율(average fill factor)이 70%일 때 가장 안정되어 인덱스 리벨런싱과 재구성을 하지 않고도 인덱스 연산을 수행할 수 있다. 따라서 인덱스 구성시 이 평균점유율에 맞추어 B^+-Tree를 조직하게 된다. 이때, 그 노드의 나머지 30%의 저장 공간은 추후 발생 가능한 엔트리 삽입을 위하여 빈공간으로 유지하고 있다.

그러나 플래시 메모리는 물리적인 특성상 제자리 쓰기(overwrite)가 불가능하다. 즉, 한 노드 블록안의 일부 수정이 요청되면 새로운 클린 블록을 할당받아 쓰기 연산이 진행된다. 즉, 기존 B^+-tree를 그대로 적용하게 되면, 30%의 공간이 낭비되며, 특히, 빈공간에 대고 무의미한 쓰기 연산을 계속 수행하면, 쓰기 적체로 인하여 성능저하를 야기할 수 있다.



[Fig. 1] Architecture of Flash Memory Storage

그림1을 통하여 플래시 메모리의 쓰기 적체의 이유를 설명하면, NAND Flash Array의 블록 크기가 작아서 성능에 영향을 주기도 하지만, 주로 쓰기 연산의 과도한 집중으로 플래시 저장 시스템내의 SRAM 버퍼가 포화되어, 더 이상 버퍼에서 지연시키지 못하며, 지연되었던 셀 소거 작업과 Garbage Collection 작업, 마모 평준화(Wear Leveling) 연산, 저성능 Bus, CPU, Controller, 카피 오버헤드 등의 복합적인 요인 때문이다.

또한, 전술한 바와 같이, 컬럼-데이터베이스에서 컬럼별로 데이터를 모아 붙인 적당한 길이의 클러스터별로 압축해야 하는데, 1만 건 이상의 단위로 모아서 압축한다. 보통 1만 건은 일반 가로-지향 데이터베이스에서는 압축이 잘 안되어 큰 용량일 수 있으나, 컬럼-데이터베이스에서 컬럼별로 저장하므로, 압축율이 매우 높다. 예를 들면, 성별 및 마크 같은 단순 코드나 패턴의 데이터인 경우나, 나이와 같은 수치형이나, 부서명과 같은 텍스트가 많아서 압축이 매우 잘 된다. 물론, 이러한 고효율 압축을 잘 활용한 것이 컬럼-데이터베이스의 기본 발상이며 큰 장점이기도 하다. 본 연구에서는 플래시 메모리를 통한 검색 고속화와 공간효율을 위하여 중간 노드는 빈공간 압축하며, 실제 데이터가 있는 리프 노드는 빈공간 압축과 더불어 세부 세그먼트로 나누어 압축한다.

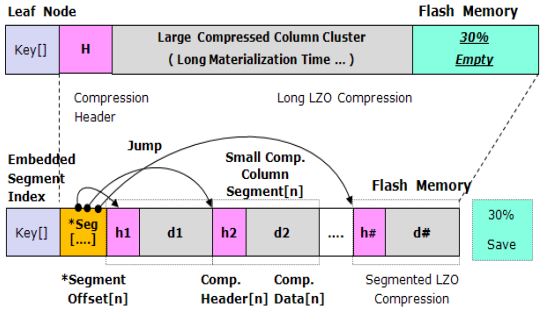
본 실험에 사용된 압축 알고리즘은 단순하고 공개적으

로 쉽게 구할 수 있으며, Lha나 Gzip 유틸리티의 기반이 되는 lzo 압축 기법[11]이다. 물론 이 보다 더 압축률이 좋은 알고리즘이 더 있으므로, 추후에 더 성능을 개선할 수 있으나, 본 연구의 효율상 프로그램 코드가 공개되어 있고 활용 경험이 있는 이 압축 기법을 사용하였다.

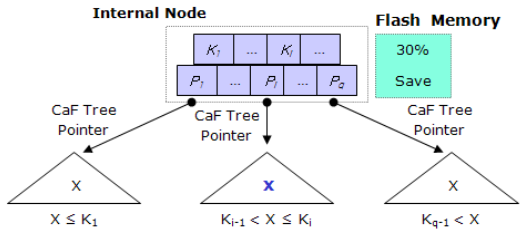
컬럼-지향 데이터베이스에서 인덱스와 데이터 저장시 플래시 메모리 불록상의 어느 정도의 압축 시간이 요구되며, 압축된 노드를 검색할 경우에도 복원 시간이 필요한데, 특히 컬럼-데이터베이스의 경우에는 일반-데이터베이스와는 달리 압축 및 복원 작업이 훨씬 더 빈번하다. 따라서 압축 대상인 컬럼-데이터가 길면 길수록 그 압축 및 복원 부담은 상대적으로 더욱더 커진다.

이러한 압축 복원 부담을 줄이기 위하여, 본 연구에서는 리프 노드 저장시, 압축 및 복원 부담이 매우 큰 긴 길이의 클러스터를 통째로 압축하지 않고, 여러 개의 세그먼트로 나누어 부하를 가볍게 만들어 압축한 후, 이 세그먼트들을 서로 연결하여 필요시 해당 위치로 바로 점프가 가능한 내장 인덱스로 구성하여 저장한다. 이러한 내장된 점프 인덱스로 구성하면 저장시에 내장 인덱스 헤더가 하나 추가되고, 압축시간이 약간 더 걸릴 수는 있다. 그러나 리프 노드의 데이터 검색 시에는 원하는 컬럼의 데이터를 맨 앞에서부터 길게 압축을 풀면서 찾아가느라 시간을 낭비하지 않고, 해당 세그먼트로 바로 점프하여 해당 세그먼트만을 대상으로 압축을 풀 수 있으므로, 결과적으로 컬럼-데이터베이스의 검색 성능을 매우 높일 수 있다.

다음 그림 2와 그림 3은 본 연구에서 컬럼-지향데이터베이스를 위하여 검색 성능을 개선 제안한 "컬럼-인지 플래시 인덱스(Column-aware Flash: CaF index)" 기법의 리프 노드와 중간 노드 구조이다.



[Fig. 2] Leaf Node Structure of CaF Index



[Fig. 3] Internal Node Structure of CaF Index

그림 2에서 상단의 기본 노드 구조와 제안한 내장 세그먼트 인덱스 구조의 검색 효율성을 분석하면 다음과 같다. 먼저 본 CaF 기법을 적용하기 전의 원래 리프 노드 구조에 대한 총 검색 시간을 S_{org}라 하면, $S_{org} = T_{org}(H) + T_{org}(D)/2$ 로 계산된다.

즉, 원래의 압축 헤더를 읽는 시간인 $T_{org}(H)$와 저장된 블록에서 원하는 데이터가 있는 평균 거리는 중간점인 $T_{org}(D)/2$가 된다. 이제 CaF 기법의 리프 노드 검색 시간인 S_{CaF}을 계산하면 다음과 같다.

$S_{CaF} = T_{CaF}(segoff) + T_{CaF}(h) + [T_{CaF}(d)/2]/dg$ 즉, 세그먼트 오프셋 리스트인 *segoff*의 접근시간과 해당 세그먼트에 점프한 후 데이터를 읽어 가는 시간의 합이다. 여기서 dg는 그림2에서와 같이 하나의 큰 블록을 몇 개의 세그먼트로 나눔을 뜻하는 degree를 의미한다. 예를 들어 3등분 하였다면 dg는 3이 된다. h는 원래 큰 H 헤더가 나누어진 작은 헤더를 의미하며, d는 원래 큰 D 데이터가 나누어진 작은 크기를 의미한다.

이 식에서 값을 대입해 보면, 원래 큰 압축 스트림을 dg로 분할한 것이므로, h는 H를 dg로 나눈 근삿값이 되며, d는 D를 dg로 나눈 근삿값이 된다. *segoff*의 크기는 포인터 타입으로 4바이트이며, degree를 10이하로 설정되어 총 40바이트 이하이다. 플래시 메모리의 세그먼트는 최소 블록 크기로 해도 1024 바이트이다. 헤더의 크기는 데이터의 압축 복잡도에 따라 변화되며 크기가 작으므로, 대략 데이터 위주로 검색 범위를 계산하면, 원래 방식은 약 500 바이트이다. 반면 본 제안 기법은 dg값이 2인 경우에도 40+1024/2/2로서 약 300바이트로 검색에 필요한 읽기가 60%에 불과하므로 세그먼트 오프셋 관리 오버헤드를 가만하더라도 접근 성능이 향상된다.

그림 3에서 중간노드는 루트 노드에서 포인팅 되어서 다시 다수의 중간노드로 분기될 수 있으며, 최종적으로 리프 노드를 포인팅하게 된다. 본 기법의 중간노드는 기존 플래시 노드 구조[9]를 참조하였다.

이 플래시 노드 구조의 핵심은 유휴 공간인 30% 저장 공간을 읽기/쓰기 캐시로 임시적으로 활용하는 것이다. 이 구조는 하드 디스크와 플래시 메모리가 복합된 하이

브리드 저장 시스템을 전제로 하며, 플래시 메모리는 고속 저장이 가능하므로 충분히 캐시로서의 성능 개선을 얻을 수 있으며, 기존 휘발성 RAM 방식의 캐시에 비하여 비휘발성 영구 저장이 가능하므로, 갑작스런 전원차단에 따른 데이터 손실도 방지할 수 있다.

이 플래시 노드 구조는 하드 디스크에 비하여 매우 고가인 플래시 메모리를 빈공간의 낭비 없이 더 효율적으로 활용한다. 그리고 플래시 노드에 대한 쓰기 연산이 수행될 때, 이때 해당 데이터도 그 노드 안에 같이 저장된다. 따라서 엔트리에서 포인팅 하는 데이터를 캐시에 저장하기 위한 별도의 추가적인 쓰기 연산이 불필요하므로, 플래시 메모리의 쓰기 연산 횟수를 감소된다. 읽기보다 느린 플래시 메모리의 쓰기 속도를 고려할 때 이는 곧 하이브리드 저장 시스템의 성능 향상에 기여한다.

또한 사용 대상 측면에서 보면, 이 플래시 노드 구조의 전제는 플래시 메모리를 임시 캐시로 사용하여 데이터도 같이 저장해야하므로, 데이터 한 건의 크기가 매우 제한된다. 즉, 데이터 레코드의 크기가 큰 일반 데이터베이스가 아닌 수백 바이트 이하의 소형 데이터베이스로서 그 크기가 한정된다.

요약하면, CaF 기법은 이 플래시 노드의 유휴중인 빈 공간을 재활용한다는 측면은 참조하였지만, CaF 기법은 빈공간 제거후 패킹 압축이 핵심인 반면, 플래시 노드 기법은 비압축이며 임시 데이터 캐싱이 핵심이다. 또한, CaF는 대용량의 컬럼-지향 압축 데이터베이스가 대상이며, 플래시 노드 구조는 소형 데이터베이스가 대상이다. 저장 매체측면에서도 CaF는 플래시 메모리만을 대상으로 하지만, 플래시 노드 구조는 하드 디스크에 저장되기 전의 임시 캐시 역할로 하이브리드 저장 시스템이 대상이다.

본 기법의 중간 노드 탐색 방법은 기존 B^+-Tree와 같이 루트 노드로부터 킷값을 좌우 범위를 비교하여 킷값의 범위를 아래로 좁혀가며 최종적으로 리프 노드까지 인덱스 트리를 찾아 가는 방식이다. 다음은 의사코드로 작성된 본 제안 기법의 검색 알고리즘이다.

```
Algorithm: Search Operation of CaF-Index
 ColumnDataRecord *CaF-Search (int key) {
  R ← node containing root 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^th tree
                      // 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
   }  // end of while loop

   if ( Leaf_Node_found ) {
        Create a temp node T in RAM
                        for fast uncompression;
        S ← FM_Read( N + N.SegmentOffset[N.Ki] )
        // jump uncompression pointer to Segment S.
     T ← Lzo_Uncompress_Segment(S);
              // uncompress only segment S
        Search Data Di such that key = Ki in T;
                  // search target data Di.
        if ( Data_found )
            Return Di  // Success: return data record
        else
            Return NO_DATA;  // Fail: data not found
   }
   else  // Leaf node not found
        return NO_LEAF_NODE;
 } // End of Search Operation
```

알고리즘을 간단히 살펴보면, 먼저 루트 노드 *R*에서부터 전술한 방식으로 아래 방향으로 좌우로 킷값을 비교하며 목표 key를 찾아 비교 탐색한다. 최종적으로 key가 포함된 해당 리프 노드를 찾게 되면, 기존의 가로-지향 데이터베이스 방식처럼 해당 리프 노드의 긴 클러스터를 한꺼번에 압축을 풀면서 시간을 낭비하지 않는다. 대신에 먼저 목표 key값을 비교하여, 목적 데이터가 있는 세그먼트 *S*의 위치를 찾는다. 즉, SegmentOffset의 값을 판독한 후, 세그먼트 *S*의 시작위치까지 포인터를 이동한 후, 이 위치부터 세그먼트 *S*의 압축을 푼다. 이 세그먼트 *S*의 압축 풀기 작업은 고속의 RAM 메모리상에서 수행되는데, *T*라는 임시 메모리 영역에 신속히 저장된다.

이제 이 *T* 메모리 영역 내에서 key 값이 일치하는 실제 컬럼-데이터 아이템 D_i를 검색후 판독하여 그 판독 값

을 리턴 한다. 결과적으로 본 기법은 리프 노드의 긴 클러스터가 아닌, 그 일부인 하나의 세그먼트 *S*만 압축을 풀면 된다. 따라서 플래시 메모리 접근 I/O가 대폭 감소함과 동시에 압축 부하도 크게 감소하여 컬럼-지향 데이터베이스의 검색 속도와 응답시간이 자연적으로 개선되게 된다.

4. CaF 색인 기법의 성능 분석

4.1 성능 실험 환경

성능 테스트에서는 제안한 컬럼-인지 플래시 색인 기법을 축약하여 CT(Column-aware Tree)라 하였다. 이 CT와 비교된 대상 색인은 데이터베이스 시스템에서 가장 보편적이며 기본적으로 사용하는 B^+-Tree(BT)기법이다. BT는 전술한 바와 같이 최적성능이 나오는 트리 노드의 평균 점유율(70%)을 기본으로 하여 BT-BAS로 축약하여 표기하였다. 또한 본 연구에서는 이 B^+-Tree를 컬럼-지향 데이터베이스 환경에 맞게 개선하여 최고 밀도로 높여 꽉 찬 트리로 플래시 인덱스를 패킹화 한 CT 버전을 CT-PACK 으로 표기하였으며, CT-EMB는 제안한 컬럼-인지 플래시 기법에서 내장 인덱스만을 적용한 것이다. 마지막으로 CP-EaP는 플래시 인덱스 팩킹과 내장 플래시 인덱스 기법을 모두 반영한 버전이다.

실험이 수행된 하드웨어 환경은 인텔 i7 CPU에 RAM 8G를 사용하였고, 비주얼 스튜디오 2008을 사용하였다. 또한, 실험용 샘플인 대용량 데이터로서 총 백만 건의 데이터베이스를 구축하였으며, 부수적으로 작업 부하 생성 모듈과 통계 자료 수집 및 성능 수치 분석용으로 CSim API[12]가 사용되었다.

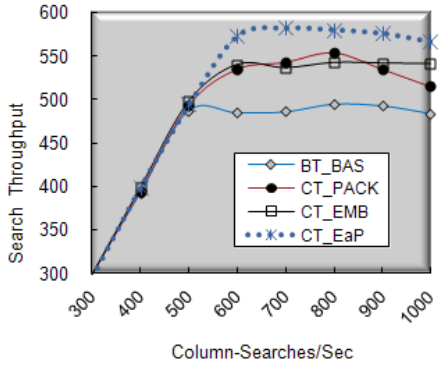
본 실험을 위한 저장 Workload 생성 및 결과 분석 모델이 필요한데, 이는 CSim에서 제공되는 통계 라이브러리를 사용하였다. 이 모듈들을 통하여 원하는 만큼이 일정한 부하가 생성되므로, 초당 일정한 수의 검색 및 업데이트 오퍼레이션을 발생시킨 후, 이 요청을 수행한 저장 시스템의 처리 시간과 성능을 측정하면 된다. 성능 지표는 검색과 업데이트 연산 처리치(search or update operation throughput)와 그 응답 시간(response time)이다. 이 operation throughput은 초당 몇 개의 검색과 업데이트 오퍼레이션이 처리 되었는지를 의미하고, response time은 해당 오퍼레이션이 발생한 후 수행까지의 소요된 시간을 의미한다. 초당 부가된 컬럼-검색 오퍼레이션의 수는 300개에서부터 10,00개 단위로 100개까지 8단계로 변화시켜 보았으며, 이는 컬럼-지향 데이터베이스 시스템에 가해지는 작업 부하를 의미한다. 그 다음으로 중요한 부하 파라미터인 초당 생성된 컬럼-업데이트 오퍼레이션 수는 50개에서부터 400개 단위로 50개까지 8단계로 변화시킨 후 분석하였다.

4.2 실험 결과 분석

본 성능 실험은 한 노드에 대한 평균 점유율 70%로 B^+-Tree를 구성한 기본형인 BT_BAS 색인과 제안한 컬럼-인지 트리 색인에서 빈공간을 제거 압축한 CT_PACK 및 제안한 내장형 플래시 색인인 CT_EMB 기법을 기본 비교 대상으로 하였다. 또한 최고 성능을 위하여 CT-PACK 기법과 CT-EMB기법을 합한 CT-EaP를 포함하여, 대용량 데이터에 대한 컬럼 검색 및 컬럼 업데이트 부하에 따른 각 기법의 처리 성능과 처리 응답 시간을 분석하였다.

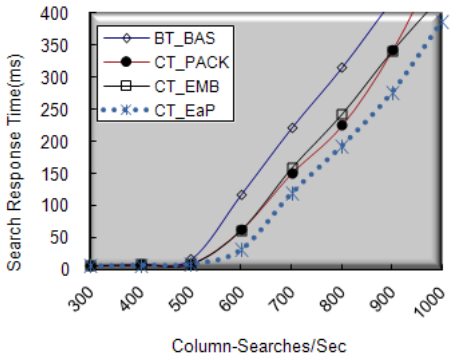
(1) 컬럼의 검색 연산 부하에 따른 성능 분석

그림 4는 초당 발생된 컬럼 검색 연산의 수의 증가에 따른 처리치를 표시한 그래프이다. 그래프에서 보면, 발생된 초당 검색 연산의 수가 늘어날수록 점차로 그 연산의 처리 결과치가 순조롭게 증가함을 알 수 있다. 전반적인 검색 연산의 처리 성능을 측정한 결과, CT-EaP, CT-EMB, CT-PACK, BT_BAS 순으로 우수하게 나타났다. 즉 CT_EaP가 가장 높으며, CT_EMB와 CT_PACK 보다 더 높게 나타났다. 다만, CT_EMB와 CT_PACK은 거의 유사한데, 작은 부하의 워크로드나 중간 정도의 워크로드에서는 미세하게 CT_PACK 보다 우세하지만, 높은 워크 로드부터는 CT_EMB가 그래프의 방향상 좀 더 우세하게 나타났다. 즉, 부하 감내 능력에서 CT_EMB가 CT_ PACK 보다 약간 더 우월하므로, 성능은 좀 더 높은 것으로 평가한다.

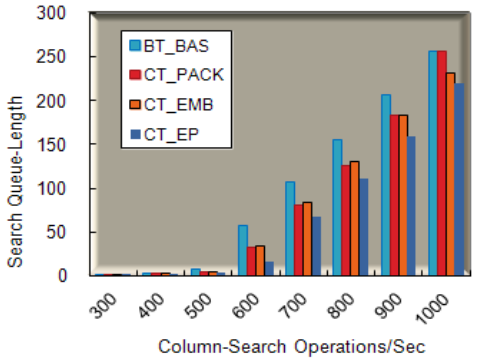


[Fig. 4] Throughput of Column Searches

그림 4에서 보면, 초당 컬럼 검색 연산의 수가 대략 500개까지는 각 기법간의 별 차이를 보이지 않는다. 이는 컬럼 검색 연산의 이 부하까지는 각 기법들이 충분히 수용 가능함을 의미하며, 그림5의 응답시간을 분석해 보면, 500개 까지는 컬럼 데이터 검색에 꼭 필요한 처리 시간이 소요될 뿐, 그 외의 지연 시간이 없어서, 성능 차이가 거의 나타나지 않는다.



[Fig. 5] Response Time of Column Searches



[Fig. 6] Queue Length of Column Searches

그러나 컬럼 검색 연산의 수가 500개를 넘으면서 각 기법의 성능이 점점 낮아지며 서서히 차이가 발생하기 시작한다. 이는 초당 약 500~600 개 정도의 컬럼 검색 연산이 최고 부하임을 의미한다.

여러 기법간의 성능 차이의 원인은 플래시 메모리상에 트리 노드의 저장 방식과 검색 방식의 효율성에 달려 있다. 그래프에서 보면 검색 연산의 부하가 적은 처음 1/3 구간에서는 부하가 시스템 안에서 수용되어 성능 저하를 일으키지 않는다. 하지만, 이 구간을 지나면서 발생되는 검색 연산의 수가 증가하므로, 바로 처리 되지 못하여 누적되어 작업 큐에 과도하게 쌓이면서, 전체 연산 처리가 지체되며 결과적으로 성능저하가 시작된다.

하지만, 부하가 심할수록 다른 기법에 비하여 CT_EaP 기법이 컬럼 검색 연산 처리 효율이 더 높다. 그림 4를 살펴보면, 컬럼 검색 연산수가 초당 700개 이상의 고부하 조건에서 제안한 CT_EaP 기법이 기존의 BT_BAS 기법에 비하여 성능이 상대적으로 약 17%정도 더 높게 나타났다. 그 이유는 CT_EaP는 플래시 메모리 특성상 불필요한 빈공간을 제거 압축함으로써 검색 블록수를 줄이고, 해당 검색 영역에 세그먼트 인덱스를 내장하여 군집 압축된 대용량 컬럼 데이터를 건너띄기 방식으로 목표한 레코드에 신속하게 접근하고, 세그먼트만 압축을 풀어서 부하를 줄일 수 있었기 때문이다. CT_EaP의 내장된 플래시 인덱스는 비인접 메모리 영역에도 신속한 랜덤 접근 가능한 플래시 메모리 스토리지에서 더 유용하고 빠르다.

이는 그림 5의 컬럼 검색 연산 응답시간 비교 및 그림 6의 작업 처리큐 길이를 비교하여 재확인할 수 있다. 그림 5의 검색 연산 응답시간 그래프에서 CT_EaP, CT_EMB, CT_PACK, BT_BAS 순으로 우수하게 나타났다. 또한, 그림6을 보면, 일단 네 가지 색인 저장 기법 모두가 발생된 워크로드, 즉 초당 컬럼-검색 연산수가 늘어날수록 점차로 처리큐의 대기 길이가 증가함을 알 수 있다. 즉, 저 부하 구간에서는 큐의 길이, 즉 적체된 검색 연산의 수가 거의 없다가, 특정 구간을 지나면서 갑자기 증가함을 알 수 있다. 이는 원활하게 감내할 임계 작업치를 넘기면, 적체가 적체를 부르는 악순환을 의미한다.

결과적으로, 전체 워크로드 구간에서는 CT_EaP이 BT_BAS보다 34%정도의 개선된 응답 속도를 나타냈다. 이 비교 결과로부터 검색 연산에 수반된 컬럼-인지 플래시 인덱스 효과가 검색 연산의 처리 시간에 개선 효과를 주고, 이는 다시 초당 검색 연산의 처리 성능치을 개선시킴을 확인할 수 있었다.

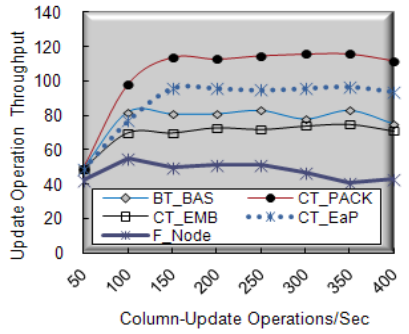
(2) 컬럼의 업데이트 부하에 따른 성능 분석

본 실험에서는 여러 색인 관리 기법들의 업데이트 연산 성능을 분석하기 위하여 연산 부하를 초당 50개에서 최대 400개 까지 50개 단위로 변화시켜 보았다. 그림 7은 업데이트 부하의 증가에 따른 실제 처리 성능 변화를 표시한 그래프이며, 그림8은 그 응답 시간의 변화를 표시한 그래프이다.

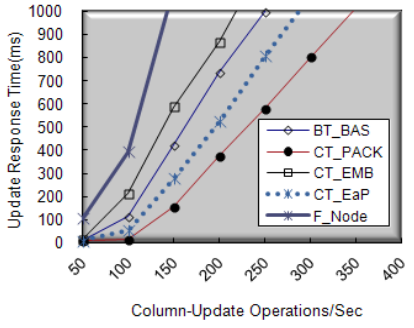
이 실험에서는 3장에서 앞서 비교한 플래시 노드 기법을 F_Node로 명하고, 그래프에 추가하여 분석하였다. 검색 성능은 같은 플래시 메모리에서 읽으므로 비슷하나, 업데이트 성능은 데이터 압축과 하이브리드 디스크의 캐싱에 영향을 받는다.

그림 7과 그림 8에서 살펴보면, 여러 색인 기법 모두

컬럼 업데이트 부하가 증가할수록 업데이트처리 성능이 받쳐주지 못하고 점차로 저하되고, 업데이트 응답 속도도 점점 느려짐을 알 수 있다. 이유는 검색 연산의 실험에서와 같이 플래시 메모리가 빠르기는 하지만, 대용량의 업데이트 연산 부하가 집중되어 적체가 시작되었기 때문이다. 즉, 읽기에 비하여 상대적으로 업데이트 작업에 시간이 많이 소모되는 플래시 메모리의 쓰기 연산이 점차로 증가할수록 적체 현상이 더 심하게 발생하여, 저장 시스템의 처리 성능과 응답 속도를 크게 저하시키기 때문이다.



[Fig. 7] Throughput of Column Updates



[Fig. 8] Response Time of Column Updates

그러나 그림 7의 그래프를 살펴보면, CT_PACK 기법이 기존의 BT_BAS뿐 아니라 다른 기법에 비하여 컬럼 업데이트 연산 처리 성능이 더 우수하게 나타났다. 이는 CT_PACK 기법은 플래시 노드에 빈공간을 압축하여 리프 노드에 같이 저장하여 신속한 저장이 가능하기 때문이다. 다음으로 CT_EaP가 우수한데, 이는 CT_PACK처럼 빈공간 압축효과가 있지만, 빈번한 검색에 대비하여 속도를 높이고자 내장 인덱스를 만들어 압축 저장하는 부하가 더 수반되기 때문이다.

그림 7의 그래프를 다시 보면, 플래시 노드 기법이 본 제안 기법에 비하여 갱신 연산 처리 성능이 초당 50 연산 이하로 낮게 나타났다. 물론 플래시 노드 캐시에 데이터 쓰기 캐싱 효과 즉, 트리 노드를 갱신한 후 연결된 데이터를 하드 디스크에 쓰지 않고, 리프 노드에 같이 저장하여 신속한 저장의 효과가 있다. 이 효과는 일반 PC의 write-back 캐시와 유사하다.

하지만, 리프 노드의 캐시 영역에 저장된 데이터는 결국, 다음 갱신 연산이 요청되는 시간사이의 유휴시간에 디스크로 내려쓰기를 해야만 한다. 이 경우 갱신 연산이 과도하게 집중되면, 플래시 노드 캐싱이 가지는 효과는 점차로 줄어들게 된다. 즉, 그림 8에서 보는 바와 같이, 시간을 많이 소모하는 저속의 하드 디스크 쓰기가 적체되어 캐싱과 유휴시간 내려 쓰기의 효과가 급격히 감쇄되며 결과적으로 갱신 연산의 응답시간도 길어지게 된다.

이상의 실험에서 전체 업데이트 부하 구간에서 평가해 보면, 제안된 CT_EaP, CT_PACK, CT_EMB는 평균하여 기존 BT_BAS에 비하여 컬럼 업데이트 연산의 응답 시간이 약 21% 단축되었으며, 컬럼 업데이트 성능이 약 18% 높아졌다. 또한, 업데이트 성능 개선 효과에 의하여, 전체적인 플래시 메모리 저장 시스템의 전력소모도 줄이는 부수적인 효과도 기대할 수 있다.

5. 결 론

본 논문에서는 최근 데이터웨어하우스 및 대용량 데이터베이스로 부상하고 있는 고속 컬럼-지향 데이터 저장 관리 기술을 도입하고, 기존의 가로-지향형 일반 데이터베이스와 특성을 비교하였다.

또한, 컬럼-지향 데이터웨어하우스의 고속 검색을 위한 컬럼-인지 인덱스 관리 기법을 제안하였다. 제안한 컬럼-인지 인덱스 관리 기법은 중간 노드와 리프노드에서 내장 플래시 인덱스와 빈공간 압축을 통하여 더 높은 검색 성능을 얻는다.

성능 평가 결과, 전체 워크로드를 평균하여 본 인덱스 관리 기법이 기존 기법보다 컬럼 검색 처리치 에서 17% 개선되었고, 응답 시간 측면에서 34% 더 우수함을 확인하였다. 컬럼 업데이트 처리치는 21% 개선되었고, 업데이트 응답 시간은 18% 단축되었다.

References

[1] A. Wang, G. Kuenning, P. Reiher, and G. Popek "The Conquest File System: Better Performance Through a

Disk/Persistent -RAM Hybrid Design", ACM Transac. on Storages, Vol. 2, No. 3, pp. 309-348, 2006. DOI: http://dx.doi.org/10.1145/1168910.1168914

[2] S. Byun, M. Hur, and H. Hwang, "An index rewriting scheme using compression for flash memory database systems" Journal of Information Science, Vol. 33, No.4, pp. 398 – 415, 2007. DOI: http://dx.doi.org/10.1177/0165551506076331

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

[4] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems", In SIGMOD, pp. 671 – 682, 2006.

[5] D. Abadi, D. Myers, D. DeWitt, and S. Madden, "Materialization strategies in a column-oriented dbms", MIT CSAIL Technical Report. MIT-CSAIL-TR-2006-078, 2006.

[6] A. Halverson, J. Beckmann, and J. Naughton, "A comparison of c-store and row-store in a common framework", Technical Report, UW Madison Department of CS, TR1566, 2006.

[7] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden, "Performance tradeoffs in read-optimized databases", In VLDB, pp. 487 – 498, 2006.

[8] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. and S. B. Zdonik. "C-store: A column-oriented dbms" In VLDB, 2005, pp. 553 – 564.

[9] S. Byun, "Flash Node Caching Scheme for Hybrid Hard Disk Systems", Journal of the Korea Academia-Industrial cooperation Society, Vol. 9, No. 6, pp. 1696-1704, 2008.

[10] A. Roberts, T. Kgil, and T. Mudge, "Integrating NAMD Flash Devices onto Servers", Communications of the ACM, Vol.52, No.4, pp. 98-106, 2009. DOI: http://dx.doi.org/10.1145/1498765.1498791

[11] Oberhumer, "LZO a real-time data compression library", http://www.oberhumer.com/opensource/ lzo/lzodoc.php

[12] Mesquite, "CSIM2.0 Development Toolkit for Simulation&Modeling", http://www.mesquite.com/documentation/documents/CSIM20_User_Guide-C. pdf, 2012.

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 디지털미디어학과 교수

<관심분야>
데이터베이스, 저장장치, 임베디드 시스템 등