SSD 상에서 B-tree 삽입 성능 향상*

김성호, 노홍찬, 박상현
연세대학교 컴퓨터과학과
e-mail : runtodream@gmail.com, fallsmal@cs.yonsei.ac.kr, sanghyun@cs.yonsei.ac.kr

Enhancement of B-tree insertion performance on SSD

Sungho Kim, Hongchan Roh, Sanghyun Park
Dept. of Computer Science, Yonsei University

요 약

최근 플래시 메모리뿐만 아니라 SSD 를 활용한 데이터베이스의 사용이 점차 늘어나고 있다. 대용량의 데이터를 처리하는 데이터베이스에서는 삽입, 삭제, 검색을 빠르게 하기 위해 다양한 색인 기법을 사용하는데 그 중 B-트리 구조가 대표적인 기법이다. B-트리는 삽입, 삭제, 검색을 할 때 더 나은 성능을 갖도록 도와주지만 그 구조를 유지하기 위한 비용이 많이 들어간다는 단점이 있다. 그 중 하나로 삽입 시 키가 삽입된 단말노드뿐만 아니라 그 부모노드까지 수정이 되어 한 번의 삽입에 여러 노드가 여러 페이지에 씌어져서 삽입시간이 길어지는 단점이 있다. 본 논문에서는 이러한 단점을 개선하기 위하여 SSD 에서 데이터베이스를 사용할 때 SSD 의 병렬 접근(parallel access) 방식을 사용해서 수정된 단말노드부터 루트노드까지의 경로에 있는 모든 노드들을 연속한 논리 주소 공간에 쓰는 방식을 적용하였다.

키워드: SSD, 플래시 메모리, B-트리, 병렬 접근

1. 서론

최근 플래시 메모리는 PMP, 카메라, 캠코더, 휴대폰 등의 저장 매체로써 널리 사용되고 있다. 플래시 메모리에서 데이터베이스를 사용할 경우 성능향상을 위해 색인 구조를 구성하는 경우가 많은데 이 때 플래시 메모리의 속성을 고려하여 최적화된 색인 구조를 많이 제안하고 있다. B-트리, 해시 테이블(hash table)등이 그 예이다.

최근에는 NAND 플래시 메모리를 기반으로 하는 SSD 가 나와서 그 활용이 점차 많아지고 있는 추세이다. SSD 는 여러 NAND 플래시 메모리가 결합한 형태로 페이지 단위로 읽기와 쓰기가 이루어지는 NAND 플래시 메모리와 다르게 하드디스크처럼 연속된 읽기와 쓰기가 가능하도록 패럴렐리즘(parallelism)과 인터리빙(interleaving) 등의 기능을 지원한다.

B-트리는 삽입, 삭제, 검색을 할 때 더 나은 성능을 갖도록 도와주지만 그 구조를 유지하기 위한 비용이 많이 들어간다는 단점이 있다. 그 중 하나로 삽입 시 키가 삽입된 단말노드(리프노드) 뿐만 아니라 그 부모노드까지 수정이 되어 한 번의 삽입에 여러 노드가 여러 페이지에 씌어져서 삽입시간이 길어지는 단점이 있다. 본 논문에서는 이러한 단점을 개선하기 위하여 SSD 의 병렬 접근(parallel access) 방식을 사용하였는데 이를 활용하기 위해서는 수정된 노드들이 아웃플레이스(out-place) 업데이트가 되도록 해야 한다. 따라서 수정된 단말노드부터 루트노드까지의 경로에 있는 모든 노드들을 연속한 논리 주소 공간에 쓰는 방식을 사용하였는데 이 방법을 통하여 SSD 를 사용한 B-트리에서 삽입 성능이 얼마나 향상되는지 확인하도록 한다.

2. 관련 기술

2.1 NAND 플래시 메모리 속성

하나의 NAND 플래시 메모리 칩은 여러 개의 블록(block)으로 구성되어 있고, 그 블록은 여러 개의 페이지로 구성되어 있다. 각 페이지는 main data area 와 spare area 로 구성되어 있으며 spare area 는 보통 오류 정정 코드(ECC) 및 그 외 부가 정보들이 저장되어 있다.

NAND 플래시 메모리는 페이지 단위로 읽기와 쓰기가 가능하며 블록 단위로 삭제가 가능하다. 그리고 새로운 데이터가 같은 페이지에 저장되기 위해서는 먼저 해당 블록이 삭제되어야 한다.

NAND 플래시 메모리를 저장 매체로 사용하기 위해서는 FTL(Flash Translation Layer)라는 소프트웨어 계층을 사용한다. FTL 은 디스크 기반으로 설계된 파일 시스템이나 DBMS 가 별도의 수정없이 NAND 플래시를 사용할 수 있도록 도와준다[4].

*본 연구는 '서울시 산학연 협력사업(PA090903)'의 지원을 받아 수행되었음

2.2 B-tree

B-트리는 대용량의 데이터를 효율적으로 관리하는데 널리 사용되는 색인 구조중 하나이다.

B-트리에서 한 노드는 d 개의 키 값(K_1, K_2, …, K_d)을 가지고 있고 d+1 개의 포인터(P_1, P_2, …,P_{d+1})를 가지고 있으며 이 d 개의 키 값들은 모두 정렬되어 있다.

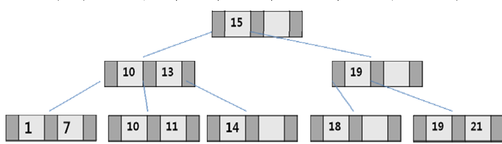
B-트리는 삽입과 삭제가 발생할 때마다 항상 모든 노드들이 골고루 분포되도록 균형을 유지하고 있다. 따라서 B-트리에서는 모든 단말노드에 대한 접근에서 균일한 깊이를 유지하고 있다.

B-트리에서 K 라는 키 값을 가지고 있는 레코드를 찾으려고 할 때, 루트노드부터 단말노드까지의 경로에 있는 여러 노드를 접근한다.

K 값을 가진 키를 삽입하려고 할 때, 루트노드부터 단말노드까지 키 값을 비교하여 삽입할 위치를 찾는다. 그 다음에 그 위치에 키를 삽입한다. 이 때 삽입할 노드가 키 값들로 꽉 찬 상태라면 분할(split)이 발생한다. 일반적으로 d+1 개의 키들의 경우, 앞에 있는 (d+1)/2 개의 키들은 현재 노드에 그대로 두고 나머지 키들은 새 노드에 저장한다. 분할 후에는 그 노드의 부모노드에 새 노드의 가장 작은 키 값을 삽입한다.

K 값을 가지 키를 삭제하려고 할 때, 마찬가지로 해당 키 값을 가지고 있는 키를 찾은 후에 그 키를 가지고 있는 노드에서 그 키를 제거한다. 이 때 노드의 키의 개수가 d/2 보다 작다면 노드의 균형을 맞추기 위해 재분배(redistribution) 및 연결(concatenation)이 일어난다.

d 개의 키를 가지고 있고 전체 레코드 수가 n 인 B-트리의 경우 검색, 삽입 또는 삭제에 드는 비용이 $\log_{d/2}n$ 에 비례한다. 따라서 d 가 클수록, 즉 노드의 크기가 클수록 비용이 줄어들게 된다.

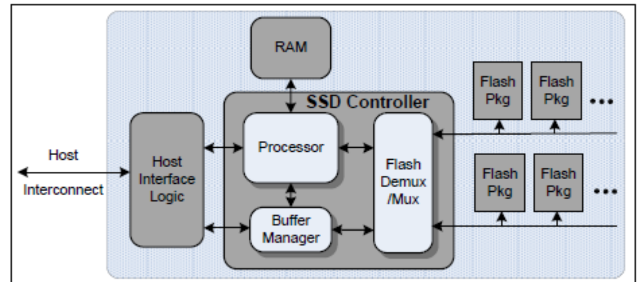


(그림 1) B-트리 구조

2.3 SSD 의 속성

(그림 2)는 SSD 의 논리 구조를 나타내고 있다. NAND 플래시 메모리 패키지는 각각 제한된 대역폭(약 40MB/sec)을 갖고 있는데, SSD 는 배열 형태로 연결된 여러 플래시 메모리 칩들에 연속된 논리 페이지를 구성하도록 하여 병렬 접근(parallel access)을 통하여 더 넓은 대역폭을 갖도록 한 것이다[1]. 시리얼 I/O 버스는 플래시 메모리 패키지를 SSD 컨트롤러에 연결시켜 준다. SSD 컨트롤러는 프로세서로부터 요청을 받으면 연결된 인터페이스를 통하여 플래시 패키지로부터 데이터를 읽어 오거나 플래시 패키지에 데이터를 쓰게 된다.

SSD 에서 데이터를 읽어오는 과정은, 먼저 첫 번째 플래시 메모리의 페이지로부터 데이터를 읽어와서 그 플레인(plane)의 레지스터에 저장하고 시리얼 버스를 통하여 데이터가 컨트롤러로 이동한다. 데이터를 쓰는 과정은 읽어오는 과정과 반대로 진행된다. 이런 방식으로 SSD 는 여러 플래시 메모리 칩들이 배열형태로 구조를 이루고 있으면서 SSD 컨트롤러로 하여금 병렬 접근 방식을 통하여 한 개 이상의 연속된 논리 페이지를 동시에 각각 접근할 수 있다.



(그림 2) SSD 논리 구조[3]

<표 2>는 Linux OS, 8 Core CPU, 16GB RAM, IOMeter 워크로드 환경에서 SSD 의 성능을 나타내고 있다. 표에서 SR(Sequential Read)는 연속된 페이지를 읽는 경우를, SW(Sequential Write)는 연속된 페이지를 쓰는 경우, RR(Random Read)는 연속되지 않은 페이지를 읽는 경우, RW(Random Write)는 연속되지 않은 페이지를 쓰는 경우를 나타낸다. 표에서 I/O 단위가 8KB 인 경우를 보면 SW 가 188Mbps 이고 RW 가 22Mbps 로 대략 9 배 빠른 성능을 보이고 있음을 알 수 있다.

(Mbps)

I/O 단위	SR	SW	RR	RW
4KB	193	158	149	21
8KB	206	188	192	22
16KB	206	186	227	32
32KB	206	175	256	40

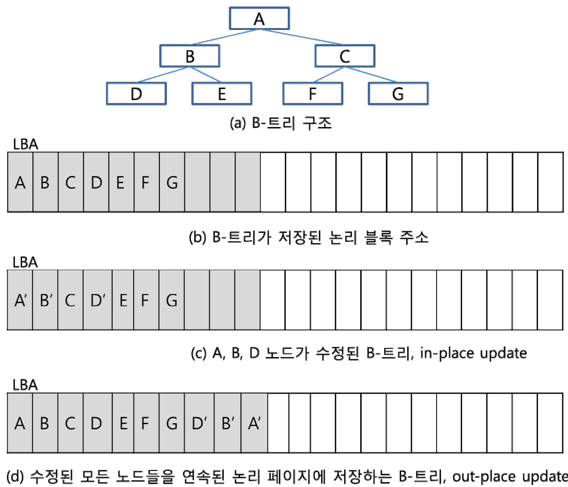
<표 1> SSD 읽기/쓰기 성능(Linux OS, Intel 8 Core CPU, 16GB RAM, IOMeter 워크로드, SSD 사용)

3. 제안 기술

B-tree 는 데이터를 수정하기 위해서는 먼저 삭제를 해야 하는 플래시 메모리 속성(write-after-erase)때문에 수정된 단말노드 및 그 부모노드들은 다른 페이지에 저장이 된다. 그리고 단말노드로 부터 노드의 분할 또는 병합(merge) 그리고 노드 내부의 키 값의 영역이 변하게 되면 단말노드가 먼저 수정되어 저장된 후 부모노드도 수정이 되어 저장이 된다. 이런 식으로 루트노드까지 수정이 반복되는데 이렇게 하나씩 순차적으로 노드를 페이지에 쓰게 됨으로써 각 노드는 연속적이지 않은 논리 페이지에 저장이 된다. 여기서 제안하는 기술은 이렇게 연속적이지 않고 논리적으로 떨어져있는 각 페이지들을 한 번에 쓰게 함으로써 SSD 의 병렬 접근 방식을 활용하는 것이다. 그렇게 하기 위해서 여기서는 수정된 단말노드부터 그의 루트노드까지의 경로에 있는 모든 노드를 한번에 연속된 논리 페이지에 쓰도록 한다. 즉 단말노드부터 그의 루트노드까지의 수정을 모두 메인 메모리에서 수행하고 수정이 완료된 후에 한꺼번에 SSD 에 저장한다. 그렇게

함으로써 여러 페이지를 SSD 에서 한 번에 쓰는 시간 동안 전체 경로에 있는 노드들을 모두 쓸 수 있다.

(그림 3)에서 (a)와 (b)는 B-tree 구조와 그 B-트리가 저장된 논리 블록주소(LBA, Logical Block Address)공간을 나타내고 있다. (c)는 B-트리에서 삽입 또는 삭제 등의 이유로 단말노드(D)가 수정되었을 때, 단말노드에서 변경된 키 값으로 인해 부모 노드(B,A)까지 수정이 되어 루트 노드부터 단말노드까지의 경로에 있는 모든 노드가 수정되어 씌어진 경우를 나타내고 있다. 이 때 단말노드인 D 가 먼저 수정이 되어 저장되고 그 다음에 B 가 수정되어 저장되고 마지막으로 A 가 저장되는데, 이 때 수정된 D, B, A 노드들은 논리 주소 공간에서는 인플레이스(in-place) 업데이트를 한다[5]. 이렇게 되면 각 노드 별로 쓰기 지연시간이 소요되고 다음에 다시 읽을 때에서 각각 읽기 지연시간이 소요된다.



(그림 3) B-트리와 각 노드가 저장된 논리 블록 주소 공간

(d)는 단말노드가 수정되었을 때 단말노드부터 루트 노드까지의 경로에 있는 모든 노드를 인플레이스 업데이트를 하는 것이 아니라 논리적으로 연속된 페이지에 이어서 쓴 것이다. 여기서 루트노드인 A' 노드는 계속 쓰여지게 되어 저장되는 주소가 계속 바뀌게 된다. 따라서 최초에 RAM 에 루트노드에 대한 포인터를 저장하고 계속 업데이트를 하여야 한다.

이 때 앞에서 언급한 SSD 의 병렬 접근 방식을 사용하면 연속된 세 개의 논리 페이지를 동시에 쓸 수 있다. 즉, 연속된 세 개의 논리 페이지를 동시에 쓰게 되면 SSD 컨트롤러는 병렬로 연결되어 있는 플래시 메모리 패키지에 연속된 논리 페이지들을 각각 분산시켜서 동시에 쓰게 된다. 따라서 기존에 SSD 의 플래시 메모리에 3 번에 걸쳐 각각의 페이지에 쓰게 되면 쓰기 지연시간의 세 배 가 소요된 것에 반해 연속된 논리 페이지를 동시에 쓰게 되면 한 번의 쓰기 지연 만 소요되어 쓰기 시간을 절약할 수 있게 된다. 연속한 논리 페이지를 한 번에 쓰는 경우와 각각 페이지를 쓰는 경우에서의 쓰기 지연 시간 차이는 <표 2>에서 확인할 수 있다.

4. 성능 분석

높이가 3 이고 단말노드의 개수가 N 이며 한 노드에 들어갈 수 있는 키들의 최대 개수가 d 인 B-트리를 생각해보자. 이 B-트리에는 랜덤한 키 값들이 균일하게 분포되었다고 가정한다.

B-트리의 특성상 한 노드에 들어갈 수 있는 키들의 개수 n 은 $(d+1)/2 \le n \le d$ 이다. 따라서 위 범위에서 특정 키들의 개수가 정해질 확률은 1/(d/2) 이다. N 이 d 일 경우 1 개의 키만 삽입되어도 분할이 발생하여 부모 노드가 수정된다. N 이 (d+1)/2 일 경우에는 d/2 개의 키가 수정되어야 분할이 발생한다. 한 번의 삽입에서 해당 노드에 키가 추가될 확률은 1/N 이다. 따라서 한 노드가 분할되기 위한 확률 P_{split} 은 아래와 같이 얻을 수 있다.

$$P_{split} = \frac{1}{(d/2)}\left\{\frac{1}{N} + \left(\frac{1}{N}\right)^2 + \dots + \left(\frac{1}{N}\right)^{(d/2)}\right\} = \frac{2}{d}\sum_{x=1}^{\left(\frac{d}{2}\right)}\left(\frac{1}{N}\right)^x \quad (식\ 1)$$

N 이 아주 큰 수라고 가정하면 아래와 같이 간단하게 줄일 수 있다.

$$P_{split} \fallingdotseq \frac{2}{d} * \frac{1}{N} = \frac{2}{d*N} \quad (식\ 2)$$

단말노드의 개수가 N 이고 각 노드마다 평균적으로 3*d/4 개의 키들이 존재한다고 가정했을 때 새 키가 삽입될 때 들어갈 수 있는 경우의 수는 (N+1) + (N *(3*d/4)) 가 된다. (N+1)는 각 노드 사이에 들어가는 경우의 수이고 (N *(3*d/4))는 각 노드 안에 있는 키들 사이에 들어갈 수 있는 경우의 수이다. 따라서 각 노드에 있는 키의 최대값보다 크거나 최소값보다 작은 키가 삽입되어 부모노드가 수정되는 확률은 아래와 같이 얻을 수 있다.

$$P_{out\ of\ range} = \frac{N+1}{(N+1)+\left(\frac{3}{4}d*N\right)} \quad (식\ 3)$$

따라서 한 개의 키가 삽입될 때 부모 노드가 수정될 확률은 아래와 같이 나온다.

$$P = P_{split} + P_{out\ of\ range} = \frac{2}{d*N} + \frac{N+1}{(N+1)+\left(\frac{3}{4}d*N\right)} \quad (식\ 4)$$

d 가 128 이고 N 이 1024 인 경우 P 는 1/383 이 된다. 즉 383 개의 삽입이 발생할 경우 부모노드가 한 번 수정이 된다. 이때 수정된 단말노드와 부모노드를 각각 플래시 메모리에 쓰지 않고 한 번에 연속된 페이지로 쓰게 되면 거의 동시에 두 개의 페이지를 쓸 수 있어 한 페이지 쓰기에 필요한 시간만큼을 절약하게 된다.

단말노드, 중간노드 그리고 루트노드로 구성된, 즉 높이가 3 인, B-트리가 있을 때 제안 기술로 인한 쓰기 성능 개선 효과는 아래와 같이 얻을 수 있다. R 은 기존 B-트리 구조에서 i 번 삽입을 한 경우 쓰기 지연시간을 나타내고 S 는 제안 기술 적용 시 i 번 삽입을 한 경우 쓰기 지연 시간을 나타낸다. B-트리

가 랜덤 변수에 의해 고른 분포로 키가 삽입된다면 중간노드 개수 N' 는 대략 $\left(\frac{3}{2d}*N\right)$ 이 된다. 따라서 루트노드가 수정이 될 확률($P_{중간노드}$)은 (식 4)에서 N 대신에 $\left(\frac{3}{2d}*N\right)$를 대입하면 얻을 수 있다. 쓰기 지연 시간은 성능에 반비례하므로 (식 8)에서 T2/T2 는 <표 2>의 두번째 데이터(I/O 단위:8KB)에서 188/22 로 얻을 수 있다. 제안 기술로 인한 쓰기 성능 개선 효과는 아래와 같이 얻을 수 있다.

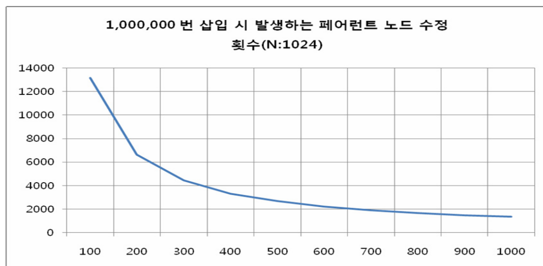
$$P_{중간노드} = \frac{4}{3N} + \frac{3N+2d}{(3N+2d)+(9d*N)}$$ (식 5)

$$R = (1 + P_{리프노드} + P_{리프노드} * P_{중간노드}) * i * T_{RW}$$ (식 6)

$$S = 3 * i * T_{RW}$$ (식 7)

$$\rho = \frac{R}{S} = \frac{(1 + P_{리프노드} + P_{리프노드} * P_{중간노드}) * i}{3 * i} * \frac{T_{RW}}{T_{SW}}$$
$$= \frac{(1 + P_{리프노드} + P_{리프노드} * P_{중간노드})}{3} * \frac{188}{22}$$
$$= 2.85 * (1 + P_{리프노드} + P_{리프노드} * P_{중간노드})$$ (식 8)

(그림 4)에서 d 가 100 이고 N 이 1024 인 경우, $P_{단말노드}$는 0.013170 이고 $P_{중간노드}$는 0.0048 이다. 이 때 (식 8)을 이용하면 ρ 가 2.85*(1+0.01317 + 0.01317*0.0048) = 2.89 로 나온다. 즉 2.89 배 만큼의 시간이 줄어든 것을 알 수 있다.



(그림 4) 1,000,000 번 삽입 시 부모 노드 수정 횟수, x 축:d

(그림 4)는 1,000,000 번 삽입 시 부모 노드 수정 횟수를 확률적으로 표현한 것이다. d 의 값에 따라 차이는 있지만 삽입이 많이 발생하는 데이터베이스에서는 빈번히 부모노드의 수정이 발생하는 것을 알 수 있다. 게다가 위의 데이터는 랜덤 변수를 균일하게 삽입하여 얻은 결과로 특정 범위의 변수가 집중적으로 삽입이 되어 특정 노드에 삽입이 반복된다면 부모노드의 수정 확률이 훨씬 높아질 수 있다.

비록 단말노드를 수정할 때마다 경로에 있는 전체 노드를 쓰게 되어 기존보다 많은 저장 용량이 필요하지만 삽입이 많이 발생하는 워크로드를 가진 데이터베이스를 사용할 경우, 특히 저장 용량이 충분히 보장되어 있는 경우, 에는 여기서 제안한 방법을 사용하면 많은 쓰기 지연 시간을 줄일 수 있다.

5. 결론

본 논문에서는 SSD 의 병렬 접근 방식을 이용한 B-트리 구조에서 수정된 노드에 대해서 인플레이스 업데이트를 하는 것이 아니라 수정된 단말노드부터 루트노드까지의 경로에 있는 모든 노드들을 연속한 논리 주소 공간에 쓰는 방식을 적용하여 삽입 성능을 개선하였다.

테스트는 한 노드에 들어갈 수 있는 키의 수가 100 이고 단말노드의 개수가 1024 인 B-tree 에 1,000,000 번 삽입을 시도하는 방식으로 진행하여 그 결과 쓰기 시간 지연이 2.89 배 개선된 것을 확인하였다.

테스트한 B-트리는 랜덤 변수를 고른 분포로 추가하여 전체 노드들로 하여금 고르게 키가 삽입되도록 하였다. 하지만 실제 환경에서는 특정 범위의 노드에 삽입이 집중되는 경우가 많을 수 있으며 이 경우에는 더 많은 시간을 절약할 수 있을 것이다. 향후 B-트리 구조에서 수정된 노드를 아웃플레이스 업데이트하도록 구현하여 좀더 정확한 데이터를 도출하고, 또 SSD 의 병렬 접근 방식을 활용하여 성능을 더욱 향상시킬 수 있는 방법도 도출할 예정이다.

6. 참고 문헌

[1] Feng Chen, David A. Koufaty, and Xiaodong Zhang, "Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives", Joint International Conference on Measurement and Modeling of Computer Systems, Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems

[2] Sang-Won Lee, Bongki Moon, Chanik Park, "Advances in flash memory SSD technology for enterprise database applications", International Conference on Management of Data, Proceedings of the 35th SIGMOD international conference on Management of data

[3] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, Rina Panigrahy Microsoft Research, Silicon Valley, University of Wisconsin-Madison, "Design tradeoffs for SSD performance", USENIX 2008 Annual Technical Conference on Annual Technical Conference

[4] Dongwon Kang, Dawoon Jung, Jeong-Uk Kang, Jin-Soo Kim, Computer Science Division Korea Advanced Institute of Science and Technology (KAIST), " μ-tree: an ordered index structure for NAND flash memory", International Conference On Embedded Software, Proceedings of the 7th ACM & IEEE international conference on Embedded software

[5] Goetz Graefe, "Write-optimized B-trees", Very Large Data Bases, Proceedings of the Thirtieth international conference on Very large data bases - Volume 30