플래쉬 메모리에서 Shadow 버전을 이용한 B-트리 인덱스 관리

온경오° 조행래
영 남 대 학 교 컴 퓨 터 공 학 과
{ondal°, hrcho}@yu.ac.kr

A B-Tree Management Scheme Exploiting Shadow Version on Flash Memory

Kyungoh Ohn° Haengrae Cho
Dept. of Computer Engineering, Yeungnam University

요 약

플래쉬 메모리는 비휘발성, 저전력, 경량, 내구성 등의 장점으로 인해, PDA나 스마트카드, 휴대폰, 휴대용 음악 재생기 등과 같은 이동 컴퓨팅 장치의 저장소로 많이 사용되고 있다. 최근 들어 대용량의 플래쉬 메모리가 출시되고 랩탑 컴퓨터등 이를 탑재한 컴퓨팅 장치들이 증가하면서 대용량의 데이터를 효율적으로 액세스하기 위한 B-트리와 같은 인덱스 기법이 요구되고 있다. 한편, 현재 사용되고 있는 NAND 플래쉬 메모리는 기존의 하드 디스크와는 액세스 특성들이 상이하다. 뿐만 아니라, B-트리 인덱스는 데이터에 비해 빈번히 액세스되고 갱신되기 때문에, 기존의 하드 디스크 기반 B-트리 인덱스 기법을 플래쉬 메모리에 적용할 경우 심각한 성능상의 문제점이 발생한다. 본 논문에서는 shadow 버전을 이용한 플래쉬 메모리 기반의 효율적인 B-트리 인덱스 기법을 제안한다.

1. 서 론

플래쉬 메모리는 비휘발성, 저전력, 경량, 내구성 등의 장점으로 인해 PDA나 스마트카드, 휴대폰, 휴대용 음악 재생기 등과 같은 이동 컴퓨팅 장치의 저장소로 많이 사용되고 있다. 플래쉬 메모리 제조사들이 대용량의 플래쉬 메모리를 출 함에 따라 PMP나 랩탑 컴퓨터등과 같은 대용량의 이동 컴퓨팅 장치들도 하드디스크를 플래쉬 메모리로 대체하는 추세이다.

플래쉬 메모리의 구조는 내부방식에 따라 NOR형과 NAND형으로 나눌 수 있다. NOR형의 경우 데이터를 저장하는 각 셀이 병렬로 연결되어 랜덤 액세스가 가능하다는 장점이 있다. 그러나 병렬로 연결된 각 셀을 개별적으로 접근하기 위한 전극이 필요하기 때문에 이를 위한 면적이 넓어져 단위 면적당 집적도가 낮다. 이에 비해 NAND형은 각 셀이 직렬로 연결되어 있어 블록 액세스 후 순차적으로 데이터를 읽기 때문에 속도는 느리지만 대용량화가 가능하여 대규모의 데이터 저장소로 사용된다[1].

그림 1은 NAND형 플래쉬 메모리의 구조를 나타낸다. 플래쉬 메모리는 다수 개의 블록(erase unit)으로 구성되며 하나의 블록에는 32개의 섹터가 포함된다. 섹터는 플래쉬 메모리의 읽기/쓰기 연산의 단위이며 512 바이트의 데이터를 저장하는 영역과 16 바이트의 헤더영역으로 이루어져 있다. 플래쉬 메모리는 하드디스크와 다른 두 가지 중요한 특성과 그에 따르는 문제점을 가진다[2]. 첫째, 플래쉬 메모리는 데이터가 저장된 섹터에 덮어쓰기(overwrite)를 할 수 없다. 따라서 갱신된 섹터를 플래쉬 메모리에 쓰기위해서는 그 섹터를 포함하고 있는 블록을 RAM으로 가져와서 해당 섹터를 변경하고, 플래쉬 메모리에서 블록을 지운 후 변경된 블록을 저장하는 복잡한 과정을 수행해야 한다. 뿐만 아니라 이 과정에서 고장(failure)이 발생할 경우, 블록 전체 데이터가 손실될 수도 있다.

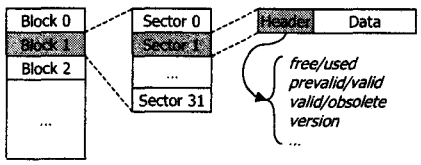


그림 1. NAND형 플래쉬 메모리 구조

둘째, 플래쉬 메모리의 각 블록은 쓰고 지우는 횟수에 한계가 있다. 그러므로 플래쉬 메모리의 특정 블록이 다른 블록에 비해 빈번히 액세스되고 갱신될 경우, 액세스 타임이 길어지고 빨리 손상된다. 이러한 플래쉬 메모리의 문제점을 해결하기 위해 블록 사상(block mapping)에 기반한 플래쉬 변환 계층(flash translation layer: FTL)이 사용된다[2,6]. FTL은 갱신된 섹터만 다 쓰고 논리 섹터 주소(logical sector address)를 새로운 섹터의 주소로 변경함으로써 블록을 지우고 다 쓰는 오버헤드를 없앨 수 있다. 또한 "wear-leveling"[2,4]은 플래쉬 메모리의 모든 블록의 수명을 비슷하게 유지한다.

대용량의 데이터를 효율적으로 액세스하기 위해서는 B-트리와 같은 인덱스가 필요하다. 인덱스는 데이터에 비해 자주 액세스되고 갱신이 빈번하게 발생하는 특징을 가진다. 본 논문에서는 플래쉬 메모리 상에서 제안된 기존의 B-트리 인덱스 기법들의 문제점을 보완한 shadow 버전을 이용한 플래쉬 메모리 기반의 B-트리 인덱스 기법을 제안한다.

본 논문의 구성은 다음과 같다. 2절에서는 기존에 제안된 플래쉬 메모리 기반 B-트리 인덱싱 기법을 살펴보

고, 3절에서는 본 논문에서 제안한 기법을 설명한다. 마지막으로 4절에서 결론 및 앞으로의 연구방향에 대해 논의하기로 한다.

2. 관련 연구

B-트리 인덱스는 대용량의 데이터를 효율적으로 액세스하기 위해 많이 사용되며, 데이터의 삽입/삭제에 의해 B-트리 노드는 빈번하게 변경된다. 일반적으로 전체 B-트리를 RAM에 두는 것은 불가능하기 때문에 갱신된 B-트리 노드들을 플래쉬 메모리로 저장해야할 경우가 발생하고 이를 위해 FTL의 쓰기연산이 수행된다. 플래쉬 메모리에서 섹터 단위의 쓰기연산은 비어있는 섹터(free sector)를 빠르게 소모하므로 블록에서 더 이상 유효하지 않은 섹터들(dead sector)을 찾아서 유효한 섹터(live sector)들만 새로운 블록에 저장하여 빈 섹터를 생성(garbage collection)하는 블록 지우기 연산이 빈번히 발생하게 된다. 결국 플래쉬 메모리의 수명이 단축되고 부가적으로 이동 컴퓨팅 장치의 전력 소모도 증가하게 된다. 이러한 문제점을 해결하기 위해 제안된 플래쉬 메모리를 위한 기존의 B-트리 인덱스 기법에는 BFTL[4]과 BOF(B-Tree on flash memory)[5]가 있다.

그림 2는 BFTL 기법의 구조를 나타낸다. BFTL 기법은 RAM에서 유보버퍼(reservation buffer)와 노드 변환 테이블(node transalation table)을 유지한다. B-트리 액세스가 필요한 서비스가 요청이 되면 BFTL에 의해 플래쉬 메모리에 대한 액세스 요청으로 사상되어 FTL에 전송된다. 이때 요청된 서비스에 의해 삽입, 삭제, 또는 갱신된 레코드들(dirty records)은 유보버퍼에 기록된다. 유보버퍼가 가득 찼을 때, BFTL 기법의 완료정책(commit policy)은 모든 갱신된 레코드에 대해 인덱스 유닛(index unit)을 생성하여 섹터에 저장한다. BFTL 기법의 완료정책은 다른 B-트리 노드에 속하는 인덱스 유닛들을 동일한 섹터에 저장하는 것을 허용하여 플래쉬 메모리에 쓰는 섹터의 수를 최소화한다. 그림 3은 BFTL 기법의 완료정책에 의해 유보버퍼의 데이터와 인덱스 유닛이 관리되는 것을 나타낸다.

BFTL 기법은 하나의 B-트리 노드에 속하는 인덱스 유닛들이 여러 개의 섹터에 흩어져 저장될 수 있기 때문에 B-트리를 순회하기 위해서는 B-트리 노드에 속하는 모든 인덱스 유닛을 유지하기 위해 노드 변환 테이블이 필요하다. 정확한 B-트리 순회를 위해 노드 변환 테이블은 인덱스 유닛을 저장한 섹터들을 연결 리스트로 유지를 하고, 연결된 섹터 수가 미리 정의된 리스트의 최대

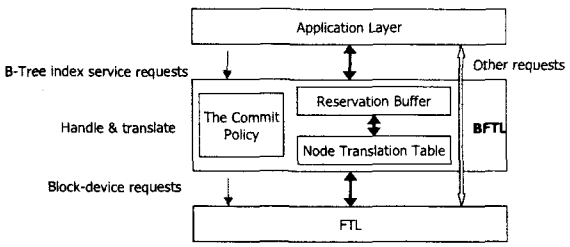


그림 2. BFTL 구조

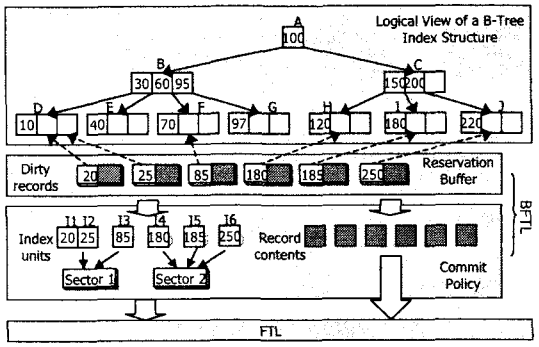


그림 3. BFTL의 완료정책

값보다 클 경우 해당 B-트리 노드와 인덱스 유닛들을 병합하여 B-트리 노드를 재구성한다. BFTL 기법은 섹터의 쓰기연산을 최소화하였지만, 노드 변환 테이블에서의 연결 리스트가 길어지고 이에 따라 액세스 여러 개의 섹터들에 흩어져 있는 인덱스 유닛을 읽어야 하므로 검색비용이 증가한다는 단점을 가진다.

새로운 레코드가 삽입될 때, B-트리 노드에 인덱스 유닛을 추가할 공간이 부족하면 노드 분할(split)이 수행된다. 노드 분할은 새로운 노드를 생성하여 분할이 필요한 노드와 새로운 노드에 인덱스 유닛을 나누어 저장하고 부모 노드에 구분자를 삽입한다. BFTL 기법의 노드 분할은 분할된 두 노드의 인덱스 유닛들을 하나의 섹터에 저장한다. 그림 4는 BFTL 기법에서 노드 분할을 나타낸다. 그림 4의 (a)에서 노드 B에 새로운 인덱스 유닛이 삽입되어 노드 B1, B2로 분할이 수행될 때 노드 B가 저장되어 있던 논리 주소 11인 섹터에 분할된 노드들을 덮어 쓴다. BFTL 기법의 노드 분할은 두개의 노드를 저장하기 위해 두개의 섹터를 이용하는 것을 하나로 줄여 쓰기연산의 수를 감소 킬 수 있다. 하지만, 그림 4의 (b)와 같이 분할된 노드가 다 분할될 경우 두개의 분할된 노드는 새로운 섹터에 같이 쓰고 기존의 섹터에 대해서는 무효화와 같은 처리과정이 필요하다. 즉, 분할 된 노드 B1이 다 분할 될 경우 논리 주소 13의 새로운 섹터에 분할된 두 노드 B11, B12를 저장하고, 논리 주소 11의 섹터에 있는 노드 B1에 대해서는 무효화를 해야 하지만, BFTL 기법에서는 그러한 과정이 존재하지 않는다. 이 경우 노드 B2가 분할되어 새로운 섹터에 저장되더라

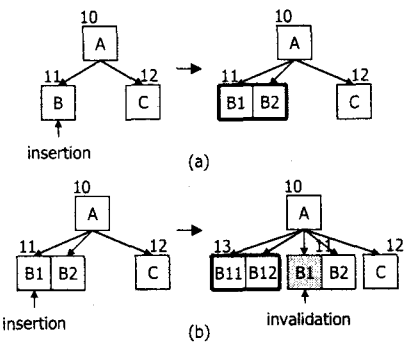


그림 4. BFTL의 노드 분할

도 논리 주소 11의 섹터는 무효화될 수 없기 때문에 쓰레기 값이 저장된 섹터가 늘어나는 문제점이 발생된다.

BOF 기법은 BFTL 기법의 이러한 문제점을 극복하기 위해 제안된 알고리즘으로 BFTL 기법과 같이 유보버퍼를 사용하고 인덱스 유닛을 생성하지만, 동일한 B-트리 노드에 속하는 인덱스 유닛들을 하나의 섹터에 저장함으로써 하나의 B-트리 노드를 액세스하기 위해 하나의 섹터만 읽기연산을 수행하면 된다. 이를 위해 BOF 기법은 유보버퍼에서 생성된 인덱스 유닛을 플래쉬 메모리에 저장할 때 플래쉬 메모리에 저장된 해당 B-트리 노드와 인덱스 유닛의 병합을 수행한다.

BOF 기법은 RAM에서 노드 변환 테이블을 유지하는 오버헤드와 검색비용을 줄일 수 있다. 하지만, 생성된 모든 인덱스 유닛에 대해 플래쉬 메모리에 저장된 B-트리 노드를 읽고 병합한 결과를 다 플래쉬 메모리에 저장함으로써 빈번한 쓰기연산이 발생한다. 최악의 경우, 유보버퍼에 있는 모든 레코드들이 모두 다른 B-트리 노드에 속한다면 갱신된 레코드의 수만큼 섹터 쓰기연산이 발생할 수 있다. 쓰기연산이 증가함에 빈 섹터를 생성하기 위한 블록 지우기 연산이 빈번해지고 전력 소모도 증가하게 되며, 결국 플래쉬 메모리의 수명이 단축된다.

3. Shadow 버전을 이용한 B-트리 관리 기법

BFTL 기법은 생성된 인덱스 유닛을 B-트리 노드에 상관없이 플래쉬 메모리 섹터에 묶어서 저장한다. 이로 인해 쓰기연산을 줄일 수는 있지만, B-트리 노드 재구성과 검색을 위한 오버헤드가 증가하는 단점이 있다. 또한 노드 분할 기존 섹터에 대한 무효화 처리과정이 필요하지만 BFTL 기법에서는 언급되지 않았다. 이러한 BFTL 기법의 단점을 보완한 BOF 기법은 쓰기연산이 증가하는 단점이 존재한다.

본 절에서는 플래쉬 메모리에서 shadow 버전(shadow version)을 이용한 B-트리 인덱스 관리 기법을 제안한다. 제안하는 알고리즘의 기본 개념은 B-트리 노드가 분할될 때 shadow 버전으로 저장하는 것이다. 이때 이전 버전을 저장하고 있는 섹터를 즉 무효화 킬 수 있고, 노드 변환 테이블의 연결 리스트의 길이를 최소화하여 B-트리 액세스 재구성 간을 단축 킬 수 있다. 이를 위해 인덱스 유닛을 저장하고 있는 섹터들에 대해 논리 주소와 해당하는 B-트리 노드의 수로 구성된 "인덱스맵(logical index map)"을 유지한다고 가정한다.

3.1 알고리즘

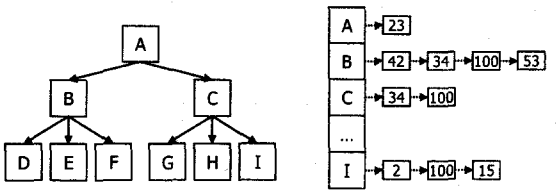
플래쉬 메모리를 위한 기존의 B-트리 인덱스 관리 기법들은 플래쉬 메모리에서 B-트리 인덱스를 위한 인덱스 유닛들을 저장한다. 스탭이 켜질 때 RAM에서 B-트리 노드를 구성하고 레코드의 삽입, 삭제 또는 수정에 의해 생성되는 인덱스 유닛을 각 기법의 완료정책에 따라 플래쉬 메모리에 저장한다.

본 논문에서는 제안하는 기법은 플래쉬 메모리에서 shadow 버전의 B-트리 노드와 인덱스 유닛을 함께 유지하는 것이다. 제안한 기법은 인덱스 유닛의 쓰기연산을 최소화하기 위해 BFTL 기법의 유보버퍼를 위한 완료정책을 사용하고, RAM에 캐싱된 B-트리 노드는 분할될 때 플래쉬 메모리에 shadow 버전으로 저장한다. B-트리 노드가 분할될 때 플래쉬 메모리에 즉 저장함으로써 이전 버전을 저장하고 있는 섹터가 무효화되기 때문에 쓰레기 값이 저장된 섹터를 없앨 수 있다. 뿐만 아니라, B-트리 노드를 액세스하기 위해 가장 최근에 저장된 shadow 버전을 읽어온 후 노드 변환 테이블의 연결된 섹터에서 인덱스 유닛들을 읽어서 B-트리 노드를 구성함으로써 B-트리 재구성 비용을 줄일 수 있다.

RAM에 캐싱된 B-트리 노드가 shadow 버전으로 저장될 때 노드 변환 테이블의 연결 리스트는 삭제된다. 또한 해당 노드의 인덱스 유닛을 저장하고 있는 섹터들에서 부분적인 무효화가 필요하다. 예를 들어, 그림 5의 (b)에서 논리 주소 34의 섹터에는 노드 B와 C의 인덱스 유닛들이 저장되어 있다. 이때, 노드 B가 분할되어 저장되면 노드 B의 인덱스 유닛을 저장하고 있는 연결 리스트는 삭제되고, 논리 주소 42, 34, 100, 53의 섹터들에서 노드 B에 해당하는 인덱스 유닛들은 폐기되어야 한다. 이를 위해 5의 (c)와 같이 인덱스 유닛이 저장된 섹터가 포함하고 있는 B-트리 노드의 수를 표 하는 인덱스맵을 유지한다. 즉, 논리 주소 34에 해당하는 섹터에는 노드 B와 C의 인덱스 유닛들이 저장되어 있으므로 인덱스맵의 34에는 2가 표 된다. 노드 B가 분할되어 플래쉬 메모리에 저장될 경우 노드 변환 테이블에서 노드 B에 연결된 리스트를 찾고 논리 주소에 해당하는 인덱스맵의 값을 1 감소 킨다. 그리고 노드 C가 분할되어 플래쉬 메모리에 저장되면 인덱스맵에서 34의 값이 0이 되므로 논리 주소 34에 해당하는 섹터는 무효화된다. 무효화된 섹터는 빈 섹터를 생성하는 FTL의 블록 지우기 연산에 포함되어야 하기 때문에 섹터의 헤더에 있는 "valid/obsolete" 비트를 '0'으로 변경한다[3].

그리고 유보버퍼에 있는 인덱스 유닛도 shadow 버전에 반영된 것이기 때문에 폐기가 가능하지만, 버퍼에서 폐기할 인덱스 유닛을 찾아야 하는 오버헤드가 존재한다. B-트리 노드의 shadow 버전이 저장된 후 유보버퍼에 남아있던 인덱스 유닛이 완료정책에 의해 플래쉬 메모리에 저장될 수 있지만, B-트리 노드의 타임스탬프와 인덱스 유닛의 타임스탬프를 이용하여 인덱스 유닛이 재반영되는 것을 방지할 수 있다.

그림 6은 B-트리 노드가 분할되어 플래쉬 메모리에



(a) Logical view of a B-Tree (b) Node translation table

...	2	15	23	34	42	53	100	...
...	1	1	1	2	1	1	3	...

(c) Logical index map

그림 5. 자료구조

1. RAM에서 새로운 B-트리 노드를 생성하고 분할되는 노드에서 전체 키들의 절반을 새로운 노드로 옮긴다.
2. 두 노드를 구분할 수 있는 식별자를 부모 노드에 삽입하고 분할된 두 노드를 플래쉬 메모리에 저장한다.
3. 노드 변환 테이블에서 분할된 두 노드의 타임스탬프를 최신의 값으로 변경한다.
4. 노드 변환 테이블에서 연결된 섹터의 논리 주소를 삭제하고, 인덱스맵에서 해당 섹터의 값을 감소 킨다.
5. 부모 노드에서 구분자 삽입으로 노드 분할이 필요할 경우 부모 노드의 노드 분할을 수행한다.

그림 6. B-트리 노드 분할

저장될 때 노드 변환 테이블과 인덱스맵이 수정되는 과정을 나타낸다. 그림 6의 과정 3에서 노드 변환 테이블의 타임스탬프에 관한 것과 유보버퍼에 존재하는 인덱스 유닛에 대한 최적화 방안은 3.2절에서 설명한다.

3.2 오버헤드 분석

본 논문에서 제안한 기법은 플래쉬 메모리에서 B-트리 인덱스를 위하여 shadow 버전을 이용한다. B-트리 노드가 분할될 때 플래쉬 메모리에 저장되면서 플래쉬 메모리에 저장된 해당 B-트리 노드의 인덱스 유닛들은 폐기된다. 이를 위해 노드 변환 테이블과 추가적으로 인덱스맵이 RAM에서 유지된다. 인덱스맵은 인덱스 유닛을 저장하는 섹터의 논리 주소와 그 섹터에 저장된 인덱스 유닛에 해당되는 B-트리 노드의 수를 저장한다. 캐싱된 B-트리 노드를 shadow 버전으로 저장할 때 플래쉬 메모리에 저장된 인덱스 유닛의 섹터를 폐기 킴으로써 노드 변환 테이블에서 유지해야 하는 각 B-트리 노드들의 연결 리스트의 수가 감소한다. 따라서 인덱스맵에서 유지하는 섹터의 수가 작아지므로, 인덱스맵을 위한 공간 오버헤드는 크지 않다.

B-트리 노드가 shadow 버전으로 플래쉬 메모리에 저장될 때, 유보버퍼에 있는 인덱스 유닛도 shadow 버전에 반영된 것이기 때문에 폐기가 가능하지만, 유보버퍼에서 인덱스 유닛을 다 찾아서 삭제해야하는 오버헤드가 존재한다. Shadow 버전에 반영된 B-트리 노드의 인덱스 유닛을 버퍼에서 폐기할 경우 몇 가지 중요한 이익을 얻을 수 있다. 첫째, 인덱스 유닛을 저장하는 섹터의 수를 감소 킴으로써 플래쉬 메모리의 쓰기연산이 감소한다. 둘째, 플래쉬 메모리에 저장되는 shadow 버전 B-트리 노드와 인덱스 유닛의 자료구조에서 타임스탬프를 생략할 수 있다. 이것은 B-트리 노드의 팬-아웃을 증가 킬 수 있으며, 또한 섹터에 저장되는 인덱스 유닛의 수를 증가 킬 수 있다.

Shadow 버전에 반영된 B-트리 노드의 인덱스 유닛을 버퍼에서 폐기하기 위해서는 노드 변환 테이블에서 각 B-트리 노드에 대해 타임스탬프를 유지해야 한다. 즉, 그림 6의 과정 3에서 B-트리 노드가 shadow 버전으로 플래쉬 메모리에 저장될 때 노드 변환 테이블의 타임스탬프를 갱신하고, 유보버퍼의 완료정책에 의해 인덱스 유닛을 플래쉬 메모리에 저장할 때 인덱스 유닛의 타임스탬프와 노드 변환 테이블의 타임스탬프를 비교해서 shadow 버전이 저장된 후 생성된 인덱스 유닛들만 플래쉬 메모리에 저장하는 과정이 필요하다. 유보버퍼의 크기를 크게 할 경우 많은 레코드를 캐싱함으로써 응답 간을 단축 킬 수 있지만, 이동 컴퓨팅 장치의 경우 불안정한 전원 때문에 대부분 유보버퍼의 크기를 작게 설정한다. 따라서 타임스탬프를 비교하기 위한 오버헤드도 크지 않다.

4. 결론

플래쉬 메모리는 데이터가 저장된 섹터에 덮어쓰기가 불가능하기 때문에 갱신된 섹터를 플래쉬 메모리에 쓰기 위해서는 그 섹터를 포함하고 있는 블록을 RAM으로 가져와서 해당 섹터를 변경하고, 플래쉬 메모리에서 블록을 지운 후 RAM의 변경된 블록을 저장해야 한다. 뿐만 아니라, 플래쉬 메모리의 각 블록은 쓰고 지우는 횟수에 한계가 있으므로 쓰기연산을 줄이는 방안이 필요하다. 플래쉬 메모리를 탑재한 이동 컴퓨팅 환경에서 데이터가 대용량화 됨에 따라 효율적으로 액세스하기 위해 B-트리와 같은 인덱스가 필요하다.

본 논문에서는 플래쉬 메모리의 특성을 고려하여 대용량의 데이터를 효율적으로 인덱싱할 수 있는 shadow 버전을 이용한 B-트리 관리 기법을 제안하였다. 제안한 기법은 기본적으로 BFTL 기법의 완료정책을 채용하여 인덱스 유닛을 저장할 때 필요한 섹터의 수를 최소화하여 쓰기연산을 줄일 수 있다. 또한 B-트리 노드의 shadow 버전을 플래쉬 메모리에 저장함으로써 팬-아웃을 증가 킬 수 있으며, B-트리 노드의 shadow 버전을 저장할 때 유보버퍼의 인덱스 유닛을 선택적으로 폐기함으로써 쓰기연산을 더욱 감소 킨다. B-트리 노드를 shadow 버전으로 저장할 때 노드 변환 테이블에서 연결 리스트를 줄일 수 있기 때문에 B-트리 노드 액세스 노드 변환 테이블을 참조한 B-트리 노드 재구성 간도 감소된다.

본 논문의 향후과제는 제안한 알고리즘의 성능을 정량적으로 분석하는 것이다.

5. 참고문헌

[1] D. Woodhouse, Red Hat, Inc. "JFFS: The Journalling Flash File System".

[2] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," *ACM Computing Surveys* 37(2) (2005) 138-163.

[3] Aleph One Company, "Yet Another Flash Filing System".

[4] C. Wu, L.Chang and T. Kuo, "An Efficient B-Tree Layer for Flash-Memory Storage Systems,"LNCS 2968 (2004) 409-430.

[5] 남정현, 박동주 "플래 메모리 상에서 효율적인 B-트리 설계 및 구현," 제32회 정보과학회 추계학술발표회 제32권 제2호 (2005) 55-57.

[6] Intel Corporatin, "Understanding the Flash Translation Layer(FTL) Specification".