접근 빈도에 기반을 둔 SSD에서의 트리 인덱스 개선

최원기, 신민철, 박상현
연세대학교 컴퓨터과학과
e-mail : cwk1412@cs.yonsei.ac.kr

Improving tree index on SSD using access frequency

Won-Ki Choi, Min-Cheol Shin, Sang-Hyun Park
Dept of Computer Science, Yonsei University

요 약

플래시 메모리는 입출력 속도가 빠르고 에너지 효율성이 좋지만, out-place update만 가능하며 쓰기 연산을 위한 IO의 지연시간이 읽기 연산보다 현저히 길다는 단점을 가진다. 이를 보완하기 위해 고안된 FlashSSD(Solid State Drive)는 최근 하드 디스크를 대체하는 저장장치로 주목받고 있다. DBMS(Database Management System)의 성능 개선을 위하여 FlashSSD를 활용한 다양한 연구가 진행되었다. 그중 FD-tree[1]를 사용한 인덱스는 좋은 갱신 성능을 보임과 동시에 검색 성능을 보인다. 하지만 FD-tree의 구성 요소 중 하나인 레벨이 하나의 자료구조로만 이루어져 있어 인덱스로서의 비효율성을 가지고 있기 때문에 이를 인덱스의 접근 빈도를 이용하여 개선하고 검색 성능을 높이고자 한다.

1. 서론

플래시 메모리는 입출력 속도가 빠르고 에너지 효율성이 좋다는 장점을 가지고 있다. 또한, 크기가 작고 전원 공급이 중단되어도 데이터가 손실되지 않는 비휘발성인 특징을 가지고 있다. 하지만 플래시 메모리는 out-place update만 가능하고 비용이 큰 erase operation이 존재한다. 그리고 플래시 메모리에는 수명이 존재하여, erase 할 수 있는 횟수가 제한되어 있다. 또한, 읽기 속도에 비해 쓰기 속도가 느리다는 단점이 있다. FlashSSD는 이러한 플래시 메모리의 단점을 보완하기 위해 고안된 장치이다. FlashSSD는 하드 디스크를 대체하거나 메인 메모리와 하드 디스크의 사이에서 캐시로 사용하는 등 다양한 방법으로 활용된다. FlashSSD의 특징을 활용하여 DBMS(Database Management System) 성능을 극대화하기 위한 다양한 연구가 진행되고 있다.

최근 주목할 만한 연구로 FD-tree[1]와 PIO B-tree[2]가 있다. FD-tree는 쓰기 연산을 최적화하기 위해 기존 B+tree에서 자주 발생하던 random write를 줄이고 빠른 속도의 sequential write를 발생시키는 방향으로 기존 B+tree 구조를 개선하였다. PIO B-tree는 FlashSSD의 내부 병렬성을 활용하기 위해 기존 B+tree를 최적화한 인덱스 구조이다. 기존 IO 연산 단위를 바꾸어, Random IO를 모아서 FlashSSD에 동시에 보내는 방식이다. FlashSSD 내부에서 Random IO를 병렬적으로 처리할 수 있기 때문에 효율성을 극대화할 수 있다.

이 중, FD-tree는 head tree와 정렬된 run들로 구성된 트리 기반의 인덱스 구조이다. Head tree는 B+tree 구조를 사용하고 있으며, 각각의 run들은 head tree 아래로 단계를 이루어 구성되어 있다. 하위 레벨이 될수록 run의 크기는 일정한 비율로 증가한다.

FD-tree에서 각 레벨에 있는 run들의 크기가 제한되어 있기 때문에, 특정 레벨에 있는 run 안 인덱스의 개수가 일정한 값을 초과하게 되면, 해당 레벨에 있는 인덱스들이 하위 레벨로 merge 되는 연산이 수행된다. 이 과정에서 merge의 단위가 run 단위이므로, 인덱스 전체가 하위 레벨로 merge 되어 불필요하게 트리의 depth가 증가하는 경우가 생긴다. 이 때문에 검색의 성능이 저하될 수 있다.

본 논문에서는 위에 언급한 단점을 보완하기 위하여 인덱스 전체를 merge 하는 것이 아니라 자주 접근되지 않는 인덱스만을 merge 하는 방법을 제안한다. merge 할 인덱스를 결정하는 방법은 페이지 안의 인덱스들이 데이터에 접근하는 빈도수를 측정하여 상대적으로 낮은 빈도수를 가진 페이지 안의 인덱스들을 선택하는 방법이다.

위 방법 외에도 추가로, FD-tree에서 인덱스가 하위 레벨로만 merge가 되던 방법을 자주 접근되는 인덱스는 상위 레벨로도 merge가 가능하게 수정해 검색 성능을 높이는 방법도 제안한다. 이때 상위 레벨로 merge 할 일부 인덱스를 결정하는 방법은 첫 번째 방법과 반대로 상대적으로 높은 접근 빈도수를 가진 페이지 안의 인덱스들을 선택하는 방법이다.

이 논문의 구조는 다음과 같다. 2장에서는 FD-tree의 구조적 특성과 문제점을 논하고, 3장에서 인덱스의 접근 빈도를 반영하여 개선한 트리의 구조에 관해 설명한다. 제안

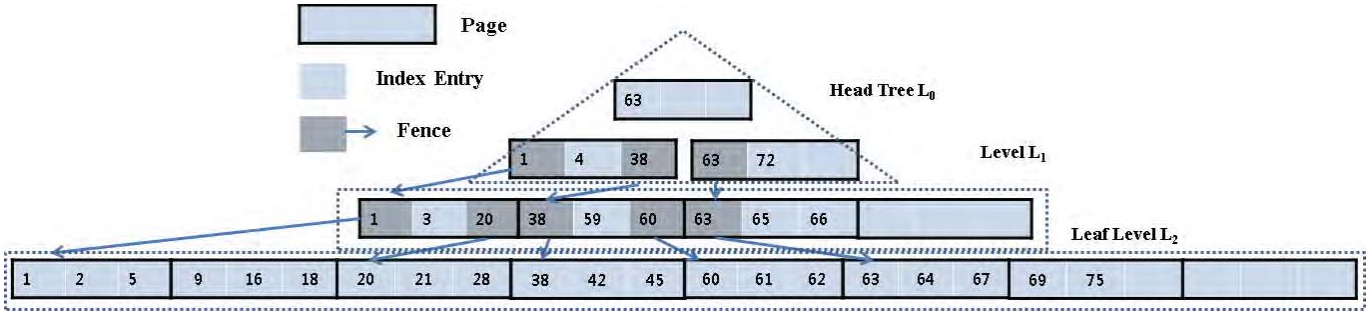

그림 1. FD-tree의 예시

하는 아이디어와 FD-tree의 비용을 4장에서 비교하여 분석한다. 마지막 5장에서는 본 논문의 결론을 내리고 향후 보완되어야 할 내용에 대해서 논한다.

2. 관련 연구 : FD-tree

FD-tree는 FlashSSD에서 느린 random write로 인한 B+tree의 성능 저하를 개선하기 위한 트리 기반의 인덱스이다. 그림 1에 나타나 있듯이 FD-tree는 B+tree 구조와 동일한 head tree와 head tree 아래에 여러 개의 레벨로 이루어져 있으며, 한 레벨은 정렬된 run으로 구성되어 있다. 각 run의 크기는 제한되어 있으며 하위 레벨로 내려갈수록 일정한 비율로 run의 크기 제한이 커진다. FD-tree는 검색 속도를 증가시키기 위하여 각 레벨에 fence라는 유형의 구성 요소를 첨가한다. Fence는 특정 페이지의 첫 인덱스를 가리키는 포인터를 갖고 있다. Fence는 인덱스 검색 시 인덱스의 키 범위를 참조하여, 다음 레벨의 run을 전부 검색하지 않아도 될 수 있도록 하는 구성 요소이다. Fence는 external fence와 internal fence 두 가지 종류가 존재한다. external fence 같은 경우에는 다음 레벨에 있는 페이지들 안의 첫 번째 인덱스들을 가리키며 fence의 키 값이 그 인덱스들의 키 값과 동일하다. internal fence 같은 경우, 인덱스의 키 값이 불균형하게 분포되어 검색 시간이 지연되는 것을 방지하기 위해 다음 레벨에 동일한 키 값을 가진 인덱스의 유무와 상관없이 페이지의 첫 번째 인덱스는 fence로 놓는데 그 fence를 internal fence라고 한다.

FD-tree는 각 레벨에 있는 인덱스가 오름차순으로 정렬된 상태이다. 따라서 정렬된 run 안의 페이지들은 FD-tree의 연산 과정에서 sequential write를 사용할 수 있기 때문에 갱신 성능이 좋다. 그리고 위에서 언급한 fence를 이용한 검색 연산으로 인해 검색 성능 또한 좋다.

다음은 FD-tree의 대표적인 연산인 삽입과 검색에 관해 설명한다. FD-tree의 삽입 연산은 다음과 같이 동작한다. FD-tree에 인덱스가 삽입되면 먼저 head tree에 저장된다. 어느 순간 head tree 안의 인덱스가 일정한 크기를 초과하게 되면, head tree(레벨 L_0) 안의 모든 인덱스가 하위 레벨의 run과 merge가 되어 L_1에 저장된다. 병합된 Run의 인덱스 개수가 일정한 값을 초과하게 되면 다음 레벨의 run에 merge가 된다. Head Tree에 삽입할 때는 random write가 발생하지만, 각 merge 연산에서는 sequential write가 발생한다. merge 연산 같은 경우 그림 2를 보면 알 수 있듯이 인접한 두 레벨을 탐색해나가면서 진행된다. 두 레벨 안의 인덱스의 크기를 비교하면서 정렬된 순서로 인덱스를 하위 레벨에 저장한다. 그 후 페이지의 각 첫 번째 인덱스는 external fence로 상위 레벨(L_{i-1})에 저장한다.

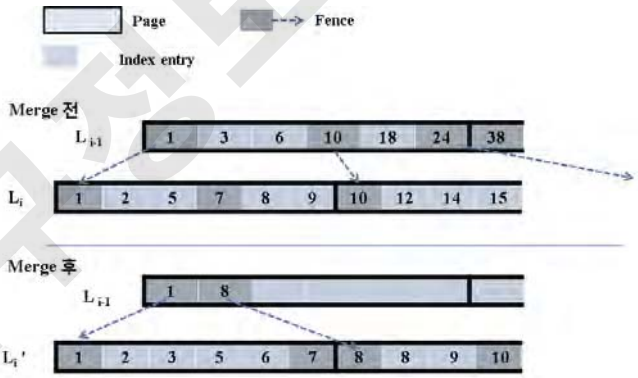

그림 2. FD-tree merge 연산의 예시

검색 연산의 경우, 그림 3에서 보면 알 수 있듯이, 일단 head tree에서 인덱스를 검색하고, 인덱스가 존재하지 않으면 적절한 fence를 찾아 다음 레벨의 특정 페이지로 내려간다. 그 후 페이지 내에서 binary search를 통해 인덱스를 검색을 시도한다. 인덱스가 존재하지 않으면 키 범위에 해당하는 페이지를 가리키는 fence를 찾아 다음 레벨을 내려가면서 위 과정을 반복한다.

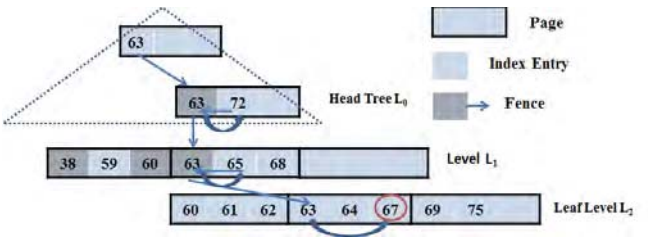

그림 3. FD-tree 검색 연산의 예시

3. 접근 빈도를 이용한 FD-tree 개선

FD-tree는 각 레벨에 있는 run의 크기가 제한되어 있으므로 인덱스의 개수가 일정한 값을 초과하게 되면, 다음 레벨로의 merge 과정이 일어나게 된다. 그러나 merge의 단위가 run이기 때문에, 특정 레벨의 run 안의 인덱스 전체가 다음 레벨로 merge 되기 때문에 불필요하게 트리의 depth가 증가하는 경우가 생긴다. 접근 빈도가 높은 인덱스임에도 불구하고 merge가 일어나게 되면 인덱스가 하위 레벨로 내려가게 되고 검색의 성능이 저하될 수 있다. 그러므로 인덱스의 접근 빈도에 대한 정보를 저장하고, merge 연산을 수정하여 검색의 성능을 높일 필요가 있다. 위와 같은 문제점을 해결하기 위하여 FD-tree를 수정하는 두 가지 방법을 제안한다.

3.1 접근 빈도가 높은 인덱스는 해당 레벨에 유지

첫 번째 방법은 merge가 일어날 때 인덱스가 가리키는 데이터에 접근하는 빈도수를 페이지 단위로 검사하여 자주 이용되는 인덱스들은 다음 레벨로 merge 하지 않고 해당 레벨에 그대로 저장하는 방법이다. 이를 위해 메인 메모리 안에 페이지 단위로 측정한 접근 빈도를 나타내는 변수를 저장하여 관리하는 테이블을 생성한다. 각 페이지의 접근 빈도는 해당 페이지 내의 각 인덱스가 가리키는 데이터에 접근할 때마다 증가한다. 추후 해당 run이 merge 된 이후에 각 접근 빈도는 0으로 초기화된다. 접근 빈도수를 메인 메모리 안에 저장하는 이유는, 페이지 헤더에 저장할 경우 빈도수가 갱신될 때마다 추가적인 IO가 발생하기 때문이다. merge 이후에 접근 빈도를 초기화를 해주는 이유는 merge 이후 페이지 안에 구성된 인덱스가 변화하기 때문이다.

첫 번째 방법의 알고리즘은 그림 4를 보면 알 수 있다. merge가 되는 시점에서 상위 N %의 높은 접근 빈도수를 가진 페이지 안의 인덱스는 하위 레벨로 merge 하지 않도록 해당 레벨(L_{i-1})에 유지한다. 나머지 페이지들 안의 인덱스들은 다음 레벨로 merge가 되어 L_i'를 생성하고, 그로 인해 생성된 external fence 들은 L_{i-1}에 저장한다.

3.2 접근 빈도가 높은 인덱스는 상위레벨로 merge

두 번째 방법은 merge가 일어날 때 접근 빈도수가 높은 인덱스는 선택하여 상위 레벨로 merge 시키는 것이다. 하위 레벨에 저장되어 있던 접근 빈도수가 높은 인덱스의 검색 효율성을 증가시키는 데 필요한 방법이다. 이 방법 또한 인덱스를 검색할 때 필요한 레벨 수가 감소함으로 검색 성능을 증가시킨다. 두 번째 방법의 알고리즘 역시 그림 4를 보면 알 수 있다. 접근 빈도수를 측정하는 법은 3.1과 동일하다. merge가 되는 시점에서 해당 레벨(L_i)에서 상위 N %의 높은 빈도수를 가진 페이지 안의 인덱스를 선택하여 상위 레벨(L_{i-1})로 merge 한다. 이 레벨(L_i)에서의 fence는 다음 레벨(L_{i+1})과 관여되므로 이를 제외한 인덱스들만 상위레벨(L_{i-1})로 merge 한다. L_i와 L_{i-1}의 merge를 통해 생기는 external fence는 L_{i-1}에 저장한다.

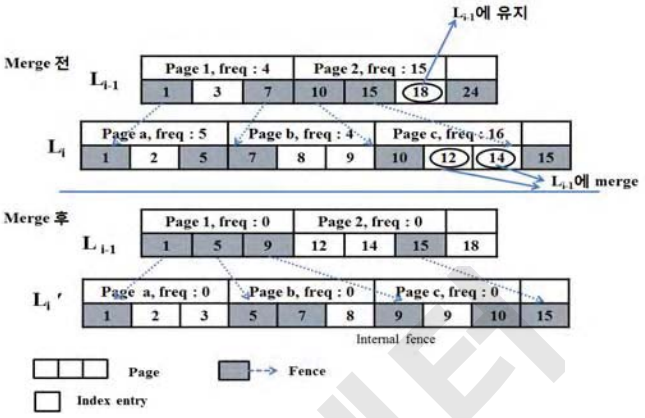

그림 4. 수정된 FD-tree의 merge 알고리즘의 예시

3.3 3.1과 3.2를 동시에 적용한 알고리즘

```
Parameter: Li-1, Li : merge 될 레벨
 ei-1, ei는 각각 Li-1, Li의 첫 인덱스
 접근 빈도수 상위 N %에 해당 하는 페이지 안의 인덱스를 frequentIndexList에 저장(fence는 제외)
while ei-1 ≠ null, ei ≠ null do
  while ei-1.type = Fence do
    ei-1를 Li-1에서 다음 인덱스로 바꾼다.
  end
  while ei.type = Internal Fence do
    ei를 Li에서 다음 인덱스로 바꾼다.
  end
  if ei-1.key ≤ ei.key then
    entryToInsert = ei-1;
    ei-1를 Li-1에서 다음 인덱스로 바꾼다.
    if ei-1 이 in frequentIndexList then
      entryToInsert를 Li-1'에 쓴다.
      continue;  //접근빈도수가 높을수록
                 // 상위 레벨에 유지한다.
  else
    entryToInsert = ei;
    ei를 Li에서 다음 인덱스로 바꾼다.
    if ei 이 in frequentIndexList then
      entryToInsert를 Li-1'에 쓴다.
      continue;
    if entryToInsert.type = Fence then
      lastFence = entryToinsert;
  if Li' 안의 현재 페이지가 비어있다면
    if entryToInsert.type ≠ Fence then
      internalFence.key = entryToInsert.key;
      internalFence.rid = lastFence.rid;
      internalFence를 Li'에 쓴다.
    entryToInsert를 Li'에 쓴다.
    externalFence.key = entryToInsert.key;
    externalFence.rid = ID of current page in Li';
    externalFence를 Li-1'에 쓴다. //연쇄적으로 상위
                                  //레벨에 external fence가
                                  // 생긴다.
  else
    entryToInsert 를 Li'에 쓴다.
end
  Li를 Li'로 교체한다.
  Li-1를 Li-1'로를 교체한다.
```

4. FD-tree와 비교한 비용 모델

S	페이지의 크기
e	페이지 안의 인덱스의 개수
k	인접한 레벨에서 run의 크기 증가율
n	인덱스 처리된 relation의 record 개수
$\|L_i\|$	L_i의 크기
l	FD-tree에서 레벨의 개수
N^i_{write}	레벨 i에서 쓰기 연산이 필요한 인덱스의 수
N^i_{read}	레벨 i에서 읽기 연산이 필요한 인덱스의 수

표 1. 비용 모델에 필요한 notation

검색 연산 : FD-tree 같은 경우 원하는 인덱스를 검색하는 데 필요한 비용은 head tree에서의 검색 비용과 정렬된 run에서의 검색 비용으로 합으로 계산된다. Worst case의 경우, head tree에서의 비용은 $\lceil \log_e |L_0| \rceil$ 로 B+tree에서의 비용과 비슷하며 트리의 depth이다. Run에서의 비용은 각 레벨에서 키 범위에 해당하는 페이지에 접근하는 비용과 하위 레벨을 검색하기 위해 그 페이지 안의 fence에 도달하는 데 필요한 비용의 합이다. Worst case의 경우, 이 비용은 접근해야 레벨의 개수가 되므로, $\lceil \log_k n/|L_0| \rceil$ 으로 표현된다. 따라서 Worst case에서 총 검색 연산의 비용은 아래와 같다.

$$\frac{S(\lceil \log_e |L_0| \rceil + \lceil \log_k n/|L_0| \rceil)}{Bandwidth(S)}$$

개선된 FD-tree의 경우, Worst case는 FD-tree와 동일하다. 그러나 자주 접근되는 인덱스의 경우, 상위 레벨에 유지되는 경향성을 보이기 때문에 FD-tree와 비교하면 접근해야 하는 레벨의 개수가 감소한다. 따라서 접근 빈도수가 높은 인덱스는 검색을 위하여 접근해야 할 페이지 수가 감소하고 IO 비용도 감소하게 된다.

FD-tree의 경우, 인덱스의 접근 빈도수와는 관계없이 먼저 삽입된 인덱스들은 하위 레벨에 존재하는 경향성을 가지게 된다. 접근 빈도수가 높은 인덱스가 검색 비용이 많이 드는 경우가 생기게 되고, 이는 인덱스 구조로서 비효율적이다.

Merge 연산 : FD-tree의 merge 비용은 아래의 식에 비례한다.

$$\frac{1}{n}\sum_{i=0}^{l-1}\left(\frac{N^i_{write}}{Bandwidth_{seqwrite}} + \frac{N^i_{read}}{Bandwidth_{seqread}}\right)$$

각 레벨에서 읽기 연산과 쓰기 연산이 필요한 페이지 수가 merge 비용이 된다. 수정된 FD-tree를 보면, FD-tree와 비교하면 읽고 써야 할 인덱스의 증감이 없고 페이지 수의 변동도 없어서 merge의 비용은 유사하다. FD-tree와 개선된 FD-tree의 읽기 연산이 필요한 페이지 수는 merge의 대상이 되는 두 레벨 안의 페이지 수로 동일하다. 쓰기 연산이 필요한 페이지 수 같은 경우, FD-tree는 하위 레벨에 merge 되는 페이지의 수이다. 개선된 FD-tree 또한 알고리즘 상으로 merge 되어야 할 페이지 수의 총합은 FD-tree와 같다. sequential write의 bandwidth 측면에서도 merge 연산이 일정한 크기의 버퍼를 사용하는 방식이므로 FD-tree와 유사할 것으로 보인다.

상대적으로 FD-tree보다 이전 레벨이 접근 빈도가 높은 인덱스로 채워지기 때문에 실질적인 run의 크기가 제한되어 merge의 횟수가 늘어남에 따라 약간의 성능 저하가 있을 수 있다. 그러나 전체적인 비용 자체는 FD-tree와 큰 차이가 나지 않을 것으로 보인다.

5. 결론

이 논문에서는 FlashSSD에서 쓰기 연산을 최적화하기 위해 random write를 sequential write로 전환한 FD-tree 인덱스를 간략히 소개하였다. 그리고 FD-tree가 merge 연산이 진행될 때 인덱스의 접근 빈도를 고려하지 않아 트리의 depth가 불필요하게 증가하여 검색 성능이 저하된다는 문제점을 발견하였다. 데이터를 참조하는 인덱스가 접근 빈도가 고려되지 않은 채 하위 레벨로 merge가 된다는 것은 비효율적인 방법이다. 자주 접근되는 중요도가 높은 인덱스일 경우라도 먼저 삽입된 인덱스라면 하위 레벨에 존재할 확률이 크게 되고 검색 성능은 매우 저하될 것으로 보인다. 인덱스 구조의 크기가 커질수록 검색 성능에 대한 문제가 심화 될 것으로 보인다. 본 논문에서는 이 문제를 해결하는 방법을 제시하고 있다. 상대적으로 접근 빈도수가 높은 인덱스의 경우 상위 레벨에 유지하도록 하는 방법이다. 이는 추가적인 merge 비용을 발생시키지 않으면서 특정 인덱스에 대해 좋은 검색 성능을 보일 것으로 보인다.

하지만 인덱스가 run 단위로 merge가 되기 때문에 merge 비용이 많이 든다는 문제점은 여전히 남아있다. 검색 성능을 유지함은 물론 위 문제를 해결할 수 있는 새로운 자료구조를 설계하는 것을 고려해보아야 한다. 이를 구현하고 실험을 통한 성능 향상을 증명하는 것이 향후 연구 계획이다.

참고문헌

[1] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, KE Yi “Tree indexing on Solid State Drives” Proc. of VLDB, 2010.

[2] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, Sang-Won Lee “B+-tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives” Proc of VLDB, 2011.