B^+-Tree에서의 잠금 없는 검색 연산을 위한 연결 리스트 기반의 동시성 제어 기법

어상훈°*, 김명근*, 배해영*
*인하대학교 컴퓨터·정보공학과
e-mail : {eosanghun°, kimmkeun, hybae}@dblab.inha.ac.kr

Linked List Based Concurrency Control Technique of B^+-tree for Non-Locking Retrieval Operation

Sang-Hun Eo°*, Myoung-Keun Kim*, Hae-Young Bae*
*Dept. of Computer Science & Engineering, Inha University

요 약

최근 인터넷 및 이동 통신기기의 사용이 급증하면서 각종 데이터에 대한 사용자들의 검색 요청은 빠른 응답 시간을 요구하는 경우가 늘어나게 되었다. 이를 충족시키기 위하여 주기억 상주 데이터베이스 관리 시스템들이 등장하게 되었고 또한 대량의 데이터들에 대한 색인 구조와 색인에 대한 접근 충돌을 제어하면서도 빠른 응답 시간을 보장하는 색인 동시성 제어 기법들에 관한 연구가 활발히 진행되어 왔다.

현재 대부분의 주기억 상주 데이터베이스 관리 시스템들은 색인에 대한 동시성 제어 기법으로 잠금 기반의 동시성 제어 기법들을 많이 사용하고 있다. 그러나 잠금 기반의 동시성 제어 기법들은 검색 연산을 포함한 모든 연산에 대하여 접근하려고 하는 노드에 잠금을 거는 것을 요구하기 때문에 잠금이 걸려있는 노드를 접근 하려는 연산은 잠금이 풀리기를 기다려야만 한다. 따라서 잠금 기반의 색인 동시성 제어 기법들은 동시성에 제약이 생겨 검색 요청에 대한 응답시간을 지연시킨다.

본 논문에서는 B^+-Tree에서의 잠금 없는 검색 연산을 위하여 엔트리간 연결 리스트 기반의 동시성 제어 기법을 제안한다. 본 논문에서 제안하는 기법은 주기억 상주 데이터베이스 관리 시스템 환경에서 검색 연산이 아무런 잠금 없이 수행되는 것을 보장한다. 특히 본 논문에서 제안하는 기법은 삽입, 삭제 연산이 수행 중인 노드에서의 잠금 없는 검색 연산의 수행이 가능하기 때문에 잠금 기반의 동시성 제어 기법들 보다 빠르고 예측 가능한 응답시간을 보장한다.

1. 서 론

지난 수년간 인터넷 및 이동 통신기기의 사용이 급증하면서 데이터베이스 관리 시스템을 이용하는 수많은 응용 어플리케이션이 등장하였다. 이와 더불어 각종 데이터에 대한 사용자들의 검색 요청은 빠른 응답 시간을 요구하는 경우가 늘어나게 되었고 이를 충족시키기 위하여 주기억 상주 데이터베이스 관리 시스템들이 개발되었다. 또한 대량의 데이터들에 대한 색인 구조 및 색인에 대한 동시성 제어 기법들에 관한 연구가 활발히 진행되어 왔으며 [1,2,3,5,7,8] 특히 이중에서도 B^+-Tree 색인 구조와 이에 대한 여러 가지 잠금 기반의 색인 동시성 제어 기법들이 많이 연구되어왔다 [4]. 일반적으로 주기억 상주 데이터베이스에 대한 색인 구조로 T-tree가 유용하다고 알려져 있지만 T-tree는 동시성 제어 측면에서 성능이 좋지 않다[6]. 따라서 본 논문[1]에서 제안하는 기법은 주기억 상주 데이터베이스에 대한 색인 구조로 B^+-Tree를 사용하였다.

B^+-Tree는 루트(Root) 노드로부터 단말(Leaf) 노드들까지 피라미드 형식의 구조로 이루어져 있고 상위 부모 노드가 여러 개의 하위 자식 노드들을 가리키고 있다. 따라서 상위 노드로 갈수록 접근 빈도수가 많아지고, 특히 루트 노드는 해당 색인을 이용할 때마다 매번 접근된다. 또한 하위 노드에 접근하려면 상위 노드를 반드시 거쳐야 하므로 잠금 기반의 동시성 제어 기법은 상위 노드에 배타적-잠금이 걸려 있으면, 같은 노드 및 그 하위 노드에 대해 다른 연산은 접근이 허용되지 않아 동시성에 심한 제약이 있다. 이러한 제약을 극복하기 위하여 B^+-Tree 대한 적절한 동시성 제어 기법들이 많이 소개되고는 있지만 이들 대부분이 잠금 기반이라는 것에는 큰 차이가 없다.

본 논문에서 제안하는 기법은 검색 연산이 아무런 잠금 없이 수행되는 것을 보장하기 위하여 엔트리간의 연결 리스트를 이용하였다. 노드안의 엔트리들을 연결 리스트로 구성하고 이들을 연결하는 연결 순서를 조정함으로써 검색 연산이 아무런 잠금 없이 수행되는 것을 보장한다.

본 논문의 구성은 다음과 같다. 2장 관련 연구에서는 동시성을 고려하여 B-tree를 변형시킨 B-link 트리와 주기억 상주 데이터베이스에 대한 색인 구조로 널리 알려져 있는 T-tree에 대해서 살펴본다. 3장에서는 본 논문에서 제안하는 연결 리스트 기반의 동시성 제어 기법 및 자료구조들을 기술하고 4장에서는 삭제된 엔트리의 재사용에 대해서 기술한다. 5장에서는 성능평가를 기술하고 마지막 6장에서는 결론을 기술한다.

2. 관련연구

이 장에서는 동시성을 고려하여 B-tree를 변형시킨 B-link 트리 [2]와 주기억 상주 데이터베이스에 대한 색인 구조로 널리 알려져 있는 T-tree[11]에 대해서 살펴본다.

B-link 트리는 아래 방향으로의 색인 탐색 시 한순간에 하나의 노드만 잠금을 소유하도록 할 수 있게 B-tree를 변형 것이다. 동시에 접근/변경이 일어나는 색인에 있어서 어떤 트랜잭션이 목표로 하는 단말 노드를 찾기 위해 탐색해 내려갈 때, 잠금 요청에 의한 대기 상태(Waiting)에 있다가 작업 재개(Wakeup)를 하게 되면 방금 따라 내려온 노드에 분할/합병 등이 일어나 원하는 키가

1) 본 연구는 정보통신부 지원 ITRC 프로그램의 지원을 받아 수행되었음

다른 곳으로 이동했을 가능성이 항상 있게 된다. 따라서 B-link 트리는 색인 구조 변경 시 키 이동이 항상 오른쪽 방향으로만 일어나게 하고, 색인 내 모든 비단말 노드는 최대 키(highest_key)와 오른쪽 형제를 가리키는 링크(Link) 정보를 추가로 관리하는 부분을 포함한다. 따라서 트리 탐색 시 변경된 노드에 접근하여도 가장 큰 값을 갖는 키와 링크를 이용하면 올바른 노드에 도달할 수 있게 된다. B-link 트리는 동시 트리 탐색 시 좋은 성능을 보이는 색인 구조로 여러 문헌상에 소개되어 있으며 실제로도 널리 이용되고 있다[2,3,4].

T-tree는 기존의 AVL-tree와 B-tree로부터 진화되어 나온 색인이다. T-tree는 이진 검색과 높이 균형을 가지는 AVL-tree의 성질을 가지고 있고, 한 노드 안에 여러개의 데이터를 가지는 B-tree의 성질을 가지고 있다. 이러한 성질로 인하여 빠른 처리 속도와 메모리 사용의 최적화라는 주기억 상주 데이터베이스 관리 시스템의 특성에 적합한 구조로 알려져 있다[6].

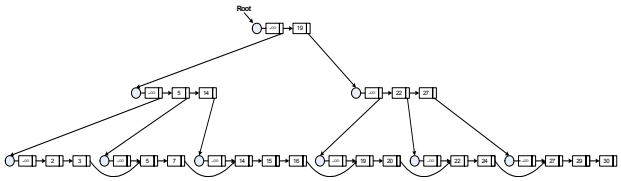
B-link 트리와 T-tree는 색인에 대한 동시성 제어 기법으로 잠금 기반의 알고리즘을 사용한다. 따라서 이들 트리 모두 검색 연산이 아무런 잠금 없이 수행하는 것은 불가능하다.

3. 연결 리스트 기반의 동시성 제어 기법

이 장에서는 검색 연산이 아무런 잠금 없이 수행 될 수 있도록 본 논문에서 제안하는 연결 리스트 기반의 동시성 제어 기법에서 사용하는 노드 구조 및 검색, 삽입, 삭제 등의 알고리즘과 자료구조들에 대해서 설명한다.

3.1 노드구조

제안된 기법에서 각 노드(Node)는 노드의 정보를 관리하는 노드헤더(Node Header)와 연결 리스트(Linked List)로 구성되어 있는 노드 엔트리(Node Entry)들로 이루어져 있다.



[그림 1] 노드구조

[그림 1] 노드구조에서 원으로 표시되어 있는 부분이 노드헤더이며 노드헤더 뒤로 연결 리스트로 구성되어 있는 노드 엔트리들이 존재한다. 이와 같은 구조는 검색 연산의 수행 중에도 새로운 엔트리의 삽입, 삭제뿐만 아니라 노드의 분할까지도 할 수 있는 구조이다.

3.1.1 노드 헤더

제안된 기법에서 각 노드들의 노드헤더는 노드안의 엔트리 개수 및 현재 노드가 단말 노드 인지를 나타내는 플래그 등 노드 관리에 필요한 정보들을 유지한다.

bLeaf	단말 노드인지를 나타내는 플래그
nEntryCnt	노드안의 엔트리 개수
pLstFree	가장 최근에 삭제된 엔트리를 가리키는 포인터
pFstEntry	사용 중인 첫 번째 엔트리를 가리키는 포인터
Lock	갱신 연산에 대한 노드의 잠금
MaxKey	노드에 가질 수 있는 가장 큰 키값

[그림 2] 노드헤더 자료구조

3.1.2. 노드 엔트리

제안된 기법에서 각 노드안의 엔트리들은 키값과 자식 노드 또는 실제 레코드를 가리키는 포인터뿐만 아니라 삭제 타임스탬프 값 및 다음 엔트리를 가리키는 포인터 등을 추가로 가지고 있다.

KeyValue	키값
pChild	자식 노드를 가리키는 포인터 또는 실제 레코드를 가리키는 포인터
DelTimestamp	삭제 타임스탬프
pNext	다음 엔트리를 가리키는 포인터
pPrvFree	이전에 삭제된 엔트리를 가리키는 포인터

[그림 3] 엔트리 자료구조

3.2 검색 알고리즘

검색 연산은 루트 노드에서 시작하여 비단말 노드를 거쳐 단말 노드까지, 각 노드를 탐색 한 다음 대상 키가 있는 비단말 노드로부터는 연결 리스트로 연결 되어있는 엔트리들을 따라가면서 검색 연산을 수행한다.

1. root node header를 current node header로 취급
2. current node header가 leaf이면
   - 2.1 대상 key를 포함한 엔트리를 찾는다.
   - 2.2 엔트리들의 연결 리스트를 따라 검색 진행
3. current node header가 leaf가 아니면
   - 3.1 대상 key가 있는 child node header선정
4. child node header를 current node header로 취급
5. 2단계부터 반복

[그림 4] 검색 알고리즘

[그림 4]는 제안된 기법에서의 검색 알고리즘이다. 잠금 기반의 검색 알고리즘과는 달리 노드 접근 시 아무런 잠금을 요청하지 않는다.

3.3 삽입 알고리즘

제안된 기법에서의 삽입 연산은 먼저 키를 삽입하여야할 노드 헤더를 검색한 다음 해당 노드 헤더에 잠금을 걸고 키를 포함할 엔트리를 할당한다. 그리고 연결 리스트에 해당 엔트리를 연결한 후 노드 헤더에 걸어놓은 잠금을 풀어줌으로써 수행된다.

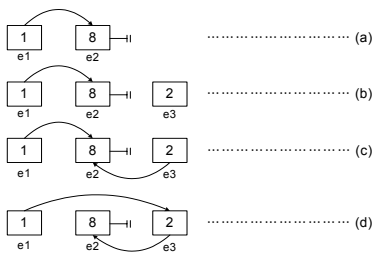
1. current node header에 잠금 요청
2. current node에 삽입 가능 여부 검사
3. 가능하지 않으면 분할 작업 수행
4. 실제 key 삽입
5. 잠금을 해제하고 작업 종료

[그림 5] 삽입 알고리즘

[그림 5]는 키를 삽입하여야할 노드 헤더의 검색을 마친 후 해당 노드에 키를 삽입하는 알고리즘이다. 그러나 검색 연산이 아무런 잠금 없이 검색을 진행할 수 있게 하기 위해서 삽입 연산은 검색 도중의 엔트리 삽입과 노드 분할 문제를 해결해야 한다.

3.3.1 엔트리 삽입

엔트리 삽입은 엔트리간의 연결 리스트로 구성되어 있는 해당 노드에 새로운 엔트리를 추가할 때 링크를 연결하는 순서를 조정함으로써 수행된다.



e1, e2, e3 : 엔트리들

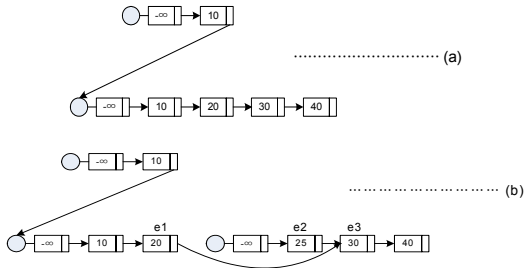
[그림 6] 엔트리 삽입 과정

[그림 6]은 새로운 엔트리 e3를 해당 노드에 삽입하는 과정을 나타낸 것이다. 상태 (a)는 기존에 이미 엔트리 e1과 e2가 노드에 존재한다는 것을 나타낸 것이다. B^+-Tree의 모든 엔트리들은 해당 노드에서 키값으로 정렬되어 있기 때문에 엔트리 e1의 pNext는 엔트리 e2를 가리키고 있다. 엔트리 e3를 노드에 삽입하는 연산과 노드에서 현재 진행 중인 검색 연산이 충돌을 일으키지 않기 위해서는 포인터 변경 연산의 순서가 필요하다. 엔트리 e3를 정렬된 순서로 노드에 삽입하기 위해서 먼저 엔트리 e3를 할당 받는다. 상태 (b)는 엔트리 e3를 할당 받은 모습을 보여준다. 이제 엔트리 e3를 연결 리스트에 연결 시켜주어야 한다. 연결을 하는 것은 2개의 포인터 변경 연산이 필요하다. 엔트리 e1의 pNext를 엔트리 e3를 가리키도록 하는 연산과 엔트리 e3의 pNext를 엔트리 e2를 가리키도록 하는 연산이 그것이다. 만일 엔트리 e1의 pNext를 엔트리 e3를 가리키도록 변경하는 연산을 먼저 수행하게 되면 동시에 수행되고 있는 검색 연산은 엔트리 e1, e3를 읽은 후 더 이상 검색 연산을 지속할 수 없게 된다. 따라서 이러한 문제를 해결하기 위하여 먼저 엔트리 e3의 pNext를 엔트리 e2를 가리키도록 설정한 다음 엔트리 e1의 pNext를 엔트리 e3를 가리키도록 변경하여 준다.

CPU는 워드(Word) 단위의 포인터 변경 연산을 원자적(Atomic)으로 지원 해주기 때문에 (a)→(b)→(c)→(d)의 순서로 엔트리들을 연결함으로써 검색 연산과 삽입 연산은 아무런 충돌 없이 원자적으로 행하여 질 수 있다[10].

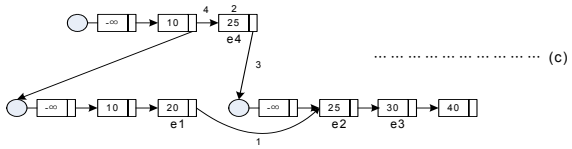
3.3.2 노드분할

노드의 분할은 새로운 노드를 나타내는 노드헤더를 생성하여 분할되는 노드에 연결시켜 주고 상위 노드에 상승되는 키값을 가진 엔트리를 추가시켜 줌으로써 수행된다.



[그림 7] 분할과정1

[그림 7] 분할과정1은 한 노드가 최대 4개의 엔트리를 가질 수 있다는 가정 아래 다섯 번째 엔트리의 삽입으로 노드가 분할되는 과정의 일부를 나타낸 것이다. 먼저 노드 헤더와 엔트리 e2를 생성하여 연결시켜주고 엔트리 e2의 pNext를 엔트리 e1의 pNext가 가리키는 엔트리 e3를 가리키도록 한다. 다음으로 엔트리 e1의 pNext를 새로 할당된 엔트리 e2를 가리키도록 설정한다.



[그림 8] 분할과정2

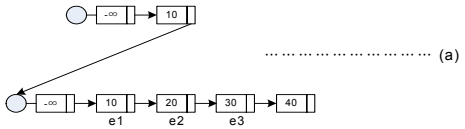
[그림 8] 분할과정2는 엔트리 e1의 pNext를 엔트리 e2를 가리키도록 변경한 후 부모노드에 상승되는 키값 25를 가진 엔트리 e4를 추가 한 모습을 보여준다. 노드의 분할은 (a)→(b)→(c)의 순서로 진행되며 각 과정의 포인터 변경 연산들은 원자적으로 수행되기 때문에 노드 분할 중에도 검색 연산은 아무런 잠금 없이 수행된다.

3.4 삭제 알고리즘

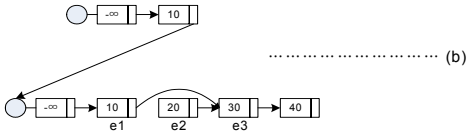
삭제 연산은 해당 엔트리의 이전 엔트리의 pNext를 해당 엔트리의 pNext가 가리키는 엔트리를 가리키도록 변경함으로써 수행된다.

1. current node header에 잠금 요청
2. current node에서 실제 삭제 수행
3. 잠금을 해제하고 작업 종료

[그림 8] 삭제 알고리즘



[그림 9] 삭제 전 노드구조



[그림 10] 삭제 후 노드구조

[그림 9]와 [그림 10]은 키값 20을 가진 엔트리 e2의 삭제 연산 전·후의 변경된 노드구조를 보여준다. 삭제 연산은 엔트리 e1의 pNext를 엔트리 e3를 가리키도록 변경함으로써 수행된다. 삭제 연산 수행 시 현재 진행 중인 검색 연산이 삭제하고 있는 엔트리 e2에 접근 해 있을 수 있기 때문에 엔트리 e2의 pNext를 변경하거나 삭제하여서는 안 된다. 만일 변경하거나 삭제를 한다면 엔트리 e2를 검색한 연산은 더 이상 다음 엔트리를 접근할 수 없게 된다.

4. 엔트리의 재사용

이장에서는 앞장에서 설명한 연결 리스트 기반의 동시성 제어 기법에서의 문제점과 이를 해결하기 위해 본 논문에서 제안하는 삭제 타임스탬프를 이용한 엔트리의 재사용에 대해서 설명한다.

4.1 문제의 정의

앞장에서 설명한 연결 리스트 기반의 동시성 제어 기법에서의 삭제 연산은 삭제되는 엔트리를 링크에서 제외시켜 줌으로써 수행된다. 따라서 삭제 연산 이후에 수행되는 검색 연산들은 더 이상 링크에서 제외된 엔트리를 접근 할 수 없게 된다. 그러나 삭제된 엔트리를 링크에서 제외시켜 주기 전에 삭제되는 엔트리를 접근하고 있는 검색 연산이 검색을 계속 진행 할 수 있도록 하기 위해 삭제되는 엔트리의 pNext를 유지하고 있는 상태이다. 따라서 삭제된 엔트리가 더 이상 사용되지 않는 적절한 시점에서 삭제된 엔트리의 공간을 반환하거나 삭제된 엔트리를 재사용해야 하는 문제점이 있다.

본 논문에서 제안하는 기법은 위에서 언급한 문제를 해결 하기 위해 삭제 타임스탬프 값을 이용한다. 제안된 기법은 글로벌(Global) 삭제 타임스탬프 값을 유지하며 각각의 검색 연산들은 연산을 시작할 때 삭제 타임스탬프 값을 1씩 증가시킨다. 따라서 엔트리를 삭제할 때 삭제 시점을 나타내기 위해 삭제 타임스탬프 값을 현재 삭제 타임스탬프 값+1로 설정한다. 이는 추후에 삭제된 엔트리의 재사용 가능성 유무를 판단 할 때 사용된다.

4.2 연산 테이블(Operation Table)

연산 테이블은 삭제된 엔트리에 설정되어 있는 삭제 타임스탬프 값과 함께 삭제된 엔트리를 재사용 하려고 할 때 재사용 가능성의 유무를 판단하기 위해 사용된다.

연산 식별자	삭제 타임스탬프
4	2
5	4
7	3
8	7
10	6

[그림 11] 연산 테이블

연산 테이블은 현재 진행 중인 검색 연산들에 대해 각각의 연산을 식별하게 해주는 연산 식별자와 그에 대응하는 삭제 타임스탬프 값을 관리한다.

실제로 검색 연산은 두 단계로 나누어져 있다. 첫 번째 단계는 검색 조건을 만족하는 첫 번째 엔트리를 찾는 단계이고, 두 번째 단계는 검색 조건을 만족하는 다음 엔트리를 찾는 단계이다. 따라서 검색 연산은 한번의 첫 번째 단계와 여러번의 두 번째 단계를 통하여 검색을 진행하게 된다.

각각의 검색 연산들은 각 단계를 시작할 때 삭제 타임스탬프 값을 1씩 증가시키고 연산 식별자와 그에 대응하는 삭제 타임스탬프 값은 연산 테이블에 추가된다. 그리고 이들은 각 단계가 종료될 때 연산 테이블에서 삭제된다.

4.3 삭제된 엔트리의 재사용

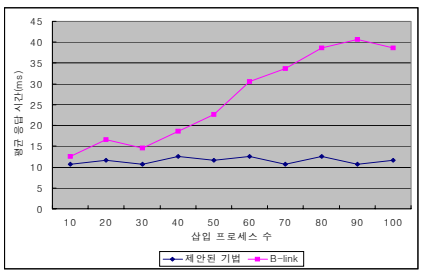
특정 엔트리를 삭제하고 삭제된 엔트리를 재사용하는 것에 있어서 가장 중요한 문제는 엔트리의 삭제 시기이다. 왜냐하면 엔트리의 삭제 이전에 진행 중이었던 검색 연산이 현재 재사용하려는 엔트리를 접근하고 있을 수 있기 때문이다. 따라서 그러한 연산이 존재하는 한 삭제된 엔트리를 재사용 할 수 없다.

삭제된 엔트리의 재사용 가능성 유무는 삭제된 엔트리에 설정되어 있는 삭제 타임스탬프 값과 연산 테이블에 있는 삭제 타임스탬프 값들을 비교하여 판단 할 수 있다. 연산 테이블에 있는 삭제 타임스탬프 값들 중에서 가장 작은 삭제 타임스탬프 값이 삭제된 엔트리에 설정되어 있는 삭제 타임스탬프 값보다 작다면 현재 삭제된 엔트리를 사용하고 있는 검색 연산이 존재 할 수 있으므로 삭제된 엔트리를 재사용 할 수 없다. 그러나 연산 테이블에 있는 삭제 타임스탬프 값들 중에서 가장 작은 삭제 타임스탬프 값이 삭제된 엔트리에 설정되어 있는 삭제 타임스탬프 값보다 크다면 엔트리의 삭제 이전에 수행 중이었던 검색 연산이 더 이상 존재하지 않는다는 의미이므로 삭제된 엔트리를 안전하게 재사용 할 수 있다.

B^+-Tree에서의 검색 연산은 단말 노드까지 탐색하여 내려간 후 해당 단말 노드부터는 순차탐색을 한다. 따라서 비단말 노드에서는 삭제된 엔트리의 재사용이 신속히 이루어 질 수 있다. 그러나 단말 노드에서는 검색 연산이 순차 탐색의 진행 중에 있는 경우가 많기 때문에 삭제된 엔트리의 재사용이 신속히 이루어지지 않을 우려가 있다. 이러한 문제는 검색 연산의 단계를 앞에서 언급한 엔트리 단위가 아닌 노드 단위로 변경하여 해결할 수 있다. 즉, 하나의 노드를 진입하고 빠져나가는 것을 검색 연산의 단계로 설정하고 단말 노드의 경우 조건을 만족하는 해당 엔트리로부터 노드에서 가장 큰 키값을 갖는 엔트리까지 스택에 저장해 놓았다가 다음 엔트리를 요구하는 단계에 대해 스택에서 하나씩 꺼내주는 방식으로 단말 노드에서의 삭제된 엔트리의 재사용이 신속히 이루어지도록 할 수 있다.

5. 성능평가

본 논문에서 제안하는 기법의 성능을 평가하기 위해서 B-link 트리와의 성능을 비교하였다. 실험에 사용된 시스템의 사양은 CPU 펜티엄 4, 2.6MHz이며 주기억 장치 2G, 운영체제는 Windows XP를 사용하였다. 실험의 초기 데이터로 1부터 20000까지의 값의 범위를 갖는 1000개의 엔트리를 각각의 색인에 삽입하였다. 삽입 프로세스는 1부터 20000까지의 값의 범위를 갖는 무작위 데이터를 색인에 삽입한다. 또한 각 검색 프로세스들은 1부터 20000까지의 질의 범위를 검색 한다.



[그림 12] 검색 성능

[그림 12]는 삽입 프로세스 수에 따른 검색 연산의 평균 응답시간을 측정한 결과이다. 검색 연산에 대한 성능을 평가하기 위하여 각각의 색인에 대해 10회의 검색 연산을 수행하여 평균 응답시간을 측정하였다. 이때 검색 연산이 삽입 연산으로 인해 영향을 받도록 삽입 프로세스의 수를 1부터 100까지 10개씩 증가시키면서 삽입 연산을 수행하였다. 각 색인에 대한 검색 성능은 [그림 12]에서 보는 바와 같이 제안된 기법이 실험 환경에서 B-link 트리보다 평균 49% 좋은 성능을 보였다. 제안된 기법이 B-link 트리보다 우수한 성능을 보인 이유는 삽입 프로세스의 수가 많아짐에 따라 B-link 트리는 검색 연산의 블록킹 상태가 빈번하게 발생하였지만 제안된 기법은 검색연산이 아무런 잠금 없이 수행되었기 때문이다. 또한 B-link 트리는 검색연산에 대한 응답시간이 매회 다르게 측정되었던 것과는 달리 제안된 기법은 매회 비슷한 응답시간이 측정되었다.

6. 결 론

본 논문에서는 B^+-Tree에서 각 노드안의 엔트리들을 연결 리스트로 구성함으로써 검색 연산의 잠금 없는 수행을 보장하였다. 또한 삭제 타임스탬프 값을 이용함으로써 엔트리간 연결 리스트 기반의 동시성 제어 기법에서 나타날 수 있는 문제점을 해결하였다. 엔트리가 삽입 될 때 연결 리스트를 연결하는 순서를 조정함으로써 검색 연산이 아무런 잠금 없이 수행 될 수 있도록 보장하였으며, 삭제 타임스탬프를 이용함으로써 삭제된 엔트리의 재사용 문제를 해결하였다.

특히, 제안된 기법은 검색 연산의 잠금 없는 수행을 보장하기 때문에 잠금 기반의 동시성 제어 기법들보다 빠르고 예측 가능한 응답시간을 보장한다.

7. 관련논문

[1] BAYER, R., AND MCCREIGHT, E. Organization and maintenance of large ordered indexes. ActaInf. 1(1972), 173-189.

[2] BAYER, R., SCHKOLNICK, M. Concurrency of operations on B-trees. Acta Inf. 9(1977), 1-21.

[3] 안태호, 이효건 코다/샘터에서의 B-tree 인덱스 동시성 제어 알고리즘의 설계 및 구현. 데이터베이스연구회지, 11권 4호, 1995. 12

[4] V.SRINIVASAN AND MICHAEL J. CAREY Performance of B^+-tree Concurrency Control Algorithms VLDB Journal, 2, 361-406(1993),

[5] C. MOHAN, D. HARDERLE, B. LINDSAY, ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. In Proc. ACM SIGMOD Conf., pages 371-380, June 1992

[6] Lu, H, Ng, YY, Tian, Z. T-Tree or B-Tree: Main Memory Database Index Structure Revisited. 11th Australasian Database Conference 2000, Canberra, Australia, pp 65-73 . IEEE Computer Society. 2000.

[7] MILLER, R., AND SNYDER, L. Multiple access to B-trees. In Proc. Conf. Information Sciences and Systems (preliminary version), Johns Hopkins Univ., Baltimore, March 1978

[8] SAMADI, B. B-trees in a system with multiple users. Inf. Process. Lett. 5, 4 (Oct. 1976), 107-112.

[9] WEDEKINK, H. On the selection of access paths in a data base system. In Data Base Management, J.W. Klimbie and K.L. Koffeman, Eds. North-Holland, Amsterdam, 1974, pp. 385-397.

[10] Philip bohannon, Dennis Leinbaugh, Rajeev Rastogi, S. Seshadri, Avi Silberschatz, S. Sudarshan Logical and Physical Versioning in Main Memory Databases. VLDB. 1997, 86-95.

[11] T.J LEHMAN AND M.J. CAREY. A study of index structures for main memory database management systems. In Proceedings of the Conference on VLDB. pages 294-303, August 1986